Calculating compilers categorically

(early draft—comments invited)

Conal Elliott

July 26, 2018

Abstract

This note revisits the classic exercise of compiling a programming language to a stack-based virtual machine. The main innovation is to factor the exercise into two phases: translation into standard algebraic vocabulary, and a stack-oriented interpretation of that vocabulary. The first phase is independent of stack machines and has already been justified and implemented in a much more general setting. The second phase captures the essential nature of stack-based computation, is independent of the source language, and is calculated from a very simple specification.

The first translation phase converts a typed functional language (here, Haskell) to the vocabulary of categories [Elliott, 2017]. All that remains is to specify and calculate a category of stack computations, which is quite easily done as demonstrated below. Other examples of this compiling-to-categories technique include generation of massively parallel implementations on GPUs and FPGAs, incremental evaluation, interval analysis, and automatic differentiation [Elliott, 2017, 2018].

1 Stack functions

A stack machine for functional computation is like a mathematical function $f :: a \to b$, but it can also use additional storage to help compute f, as long as it does so in a stack discipline.[1] A simple formalization of this informal description is that the machine computes *first* f, where[2]

$$\begin{array}{l} \mathit{first} :: (a \to b) \to \forall z.(a \times z \to b \times z) \\ \mathit{first}\ f\ (a, z) = (f\ a, z) \end{array}$$

We are representing the stack as a pair, with a on top at the start of the computation, $f\ a$ on top at the end of the computation, and z as the rest of the stack at the start and finish. In-between the start and end, the stack may grow and shrink, but in the end the *only* stack change is on top. Note also that *first* f can do nothing with z other than preserve it.[3]

The purpose of a stack in language implementation is as a place to save intermediate results until they are ready to be consumed, after a given sub-computation completes. For instance, suppose we want to apply the function $\lambda x \to (x + 2) * (x - 3)$. Assuming right-to-left evaluation, a stack machine would evaluate $x - 3$, leaving the result v on the stack, then $x + 2$, leaving its result u on the stack above v, and then replace the top two stack elements u and v with $u * v$.

Let's now further formalize this notion of stack computation as a data type of "stack functions", having a simple relationship with regular functions:[4]

$$\mathbf{newtype}\ \mathit{StackFun}\ a\ b = \mathit{SF}\ (\forall z.a \times z \to b \times z)$$

$$\begin{array}{l} \mathit{stackFun} :: (a \to b) \to \mathit{StackFun}\ a\ b \\ \mathit{stackFun}\ f = \mathit{SF}\ (\mathit{first}\ f) \end{array}$$

[1] This paper uses "stack machine" to refers to data stacks, not control stacks.

[2] In this paper, $(\times)$ (cartesian product) has higher syntactic precedence than $(\to)$ (functions), so the type of *first* is equivalent to $(a \to b) \to \forall z.((a \times z) \to (b \times z))$.

[3] [Find and cite some reasonably clear descriptions of this stack discipline.]

[4] [Is there a free theorem saying that any function of type $\forall z.a \times z \to b \times z$ *must* be equivalent to *first* f for some $f :: a \to b$? If so, then *stackFun* is an isomorphism, which may be useful.]

Conversely, we can evaluate a stack function into a regular function, initializing the stack to contain a and $()$, evaluating the contained stack operations, and discarding the final $()$:[5]

$$\begin{array}{l} \mathit{evalStackFun} :: \mathit{StackFun}\ a\ b \to (a \to b) \\ \mathit{evalStackFun}\ (\mathit{SF}\ f)\ a = b\ \mathbf{where}\ (b, ()) = f\ (a, ()) \end{array}$$

We can also formulate *evalStackFun* in more general terms:

$$\mathit{evalStackFun}\ (\mathit{SF}\ f) = \mathit{rcounit} \circ f \circ \mathit{runit}$$

The new operations belong to a categorical interface:

$$\begin{array}{l} \mathbf{class}\ \mathit{UnitCat}\ k\ \mathbf{where} \\ \quad \mathit{lunit}\ \ \ :: a \mathbin{`k`} (() \times a) \\ \quad \mathit{lcounit} :: (() \times a) \mathbin{`k`} a \\ \quad \mathit{runit}\ \ \ :: a \mathbin{`k`} (a \times ()) \\ \quad \mathit{rcounit} :: (a \times ()) \mathbin{`k`} a \end{array}
\qquad
\begin{array}{l} \mathbf{instance}\ \mathit{UnitCat}\ (\to)\ \mathbf{where} \\ \quad \mathit{lunit}\ a = ((), a) \\ \quad \mathit{lcounit}\ ((), a) = a \\ \quad \mathit{runit}\ a = (a, ()) \\ \quad \mathit{rcounit}\ (a, ()) = a \end{array}$$

Lemma 1 (Proved in Appendix A.1). *evalStackFun* is a left inverse for *stackFun*, i.e., $\mathit{evalStackFun} \circ \mathit{stackFun} = \mathit{id}$.

Lemma 2 (Proved in Appendix A.2). *stackFun* is surjective, i.e., *every* $h :: \mathit{StackFun}\ a\ b$ has the form $\mathit{SF}\ (\mathit{first}\ f)$ for some $f :: a \to b$.

Lemma 3 (Proved in Appendix A.3). *stackFun* is injective, i.e., *every* $\mathit{stackFun}\ f = \mathit{stackFun}\ f' \Longrightarrow f = f'$ for all $f, f' :: a \to b$.

Corollary 3.1. *evalStackFun* is the full (*left* and right) inverse for *stackFun*.

Proof. Since *stackFun* is surjective and injective, it has a full (two-sided) inverse, which is necessarily unique. Moreover, whenever a category morphism has both left and right inverses, those inverses must be equal [nLab, 2009–2018, Lemma 2.1]. □

The definition of *stackFun* above serves as a simple *specification*. Instead of starting with a *function* f as suggested by *stackFun*, we will start with a recipe for f and systematically construct an analogous recipe for $\mathit{stackFun}\ f$. Specifically, start with a formulation of f in the vocabulary of categories [Mac Lane, 1998; Lawvere and Schanuel, 2009; Awodey, 2006], and require that *stackFun* preserves the algebraic structure of that vocabulary. While inconvenient to program in this vocabulary directly, we can instead automatically convert from Haskell programs [Elliott, 2017]. This approach to calculating correct implementations has also been used for automatic differentiation [Elliott, 2018]. A benefit is that we need only implement a few type class instances rather than manipulate any syntactic representation.

1.1 Sequential composition

The first requirement is that *stackFun* preserve the structure of *Category*, which is to say that it a category homomorphism (also called a "functor"). The *Category* interface:

$$\begin{array}{l} \mathbf{class}\ \mathit{Category}\ k\ \mathbf{where} \\ \quad \mathit{id}\ \ :: a \mathbin{`k`} a \\ \quad (\circ) :: (b \mathbin{`k`} c) \to (a \mathbin{`k`} b) \to (a \mathbin{`k`} c) \end{array}$$

The corresponding structure preservation (homomorphism) properties:

$$\begin{array}{l} \mathit{id} = \mathit{stackFun}\ \mathit{id} \\ \mathit{stackFun}\ g \circ \mathit{stackFun}\ f = \mathit{stackFun}\ (g \circ f) \end{array}$$

The identity and composition operations on the LHS are for *StackFun*, while the ones on the right are for $(\to)$ (i.e., regular functions). Solving these equations for the LHS operations results in a correct instance of *Category* for *StackFun*.

[5]The $()$ type contains only a single value (other than $\bot$), which is also called "$()$". As such, it takes no space to represent.

The id equation is trivial to satisfy, since it is already in solved form, so we can use it directly as an implementation. Instead, simplify the equation as follows:

$$
\begin{aligned}
& \mathit{stackFun}\ \mathit{id} \\
=\ & \{ \text{ definition of } \mathit{stackFun} \ \} \\
& \mathit{SF}\ (\mathit{first}\ \mathit{id}) \\
=\ & \{ \text{ property of } \mathit{first} \text{ and } \mathit{id} \ \} \\
& \mathit{id} = \mathit{SF}\ \mathit{id}
\end{aligned}
$$

The $(\circ)$ equation requires a little more work. First simplify the LHS:

$$
\begin{aligned}
& \mathit{stackFun}\ g \circ \mathit{stackFun}\ f \\
=\ & \{ \text{ definition of } \mathit{stackFun} \ \} \\
& \mathit{SF}\ (\mathit{first}\ g) \circ \mathit{SF}\ (\mathit{first}\ f)
\end{aligned}
$$

Then the RHS:

$$
\begin{aligned}
& \mathit{stackFun}\ (g \circ f) \\
=\ & \{ \text{ definition of } \mathit{stackFun} \ \} \\
& \mathit{SF}\ (\mathit{first}\ (g \circ f)) \\
=\ & \{ \text{ property of } \mathit{first} \text{ and } (\circ) \ \} \\
& \mathit{SF}\ (\mathit{first}\ g \circ \mathit{first}\ f)
\end{aligned}
$$

The simplified specification:

$$\mathit{SF}\ (\mathit{first}\ g) \circ \mathit{SF}\ (\mathit{first}\ f) = \mathit{SF}\ (\mathit{first}\ g \circ \mathit{first}\ f)$$

Strengthen this equation by generalizing from $\mathit{first}\ g$ and $\mathit{first}\ f$ to arbitrary functions (also called "g" and "f" and having the same types as $\mathit{first}\ g$ and $\mathit{first}\ f$):

$$\mathit{SF}\ g \circ \mathit{SF}\ f = \mathit{SF}\ (g \circ f)$$

This generalized/strengthened condition is in solved form, so we can satisfy it simply by definition, yielding sufficient definitions for both category operations:

$$
\begin{aligned}
& \textbf{instance}\ \mathit{Category}\ \mathit{StackFun}\ \textbf{where} \\
& \quad \mathit{id} = \mathit{SF}\ \mathit{id} \\
& \quad \mathit{SF}\ g \circ \mathit{SF}\ f = \mathit{SF}\ (g \circ f)
\end{aligned}
$$

In words, the identity stack function is the identity function on stacks, and the composition of stack functions is the composition of functions on stacks.

Two other categorical classes can be trivially handled in the same manner as id above:[6]

$$
\begin{aligned}
& \textbf{class}\ \mathit{AssociativeCat}\ k\ \textbf{where} \\
& \quad \mathit{rassoc} :: ((a \times b) \times c)\ \text{`}k\text{`}\ (a \times (b \times c)) \\
& \quad \mathit{lassoc} :: (a \times (b \times c))\ \text{`}k\text{`}\ ((a \times b) \times c)
\end{aligned}
$$

$$
\begin{aligned}
& \textbf{class}\ \mathit{BraidedCat}\ k\ \textbf{where} \\
& \quad \mathit{swap} :: (a \times b)\ \text{`}k\text{`}\ (b \times a)
\end{aligned}
$$

The associated homomorphism equations are in solved form and can serve as definitions:

$$
\begin{aligned}
& \textbf{instance}\ \mathit{AssociativeCat}\ \mathit{StackFun}\ \textbf{where} \\
& \quad \mathit{rassoc} = \mathit{stackFun}\ \mathit{rassoc} \\
& \quad \mathit{lassoc} = \mathit{stackFun}\ \mathit{lassoc}
\end{aligned}
$$

$$
\begin{aligned}
& \textbf{instance}\ \mathit{BraidedCat}\ \mathit{StackFun}\ \textbf{where} \\
& \quad \mathit{swap} = \mathit{stackFun}\ \mathit{swap}
\end{aligned}
$$

[6] [Maybe drop these two.]

1.2 Parallel composition (products)

In general, the purpose of a stack is to sequentialize computations. Since we've only considered sequential composition so far, we've done nothing interesting with the stack. Nonsequential computation comes from parallel composition, as embodied in the "cross" operation in the *MonoidalP* interface:

$$
\begin{array}{l}
\textbf{class}\ \mathit{MonoidalP}\ k\ \textbf{where} \\
\quad (\times) :: (a\ `k`\ c) \rightarrow (b\ `k`\ d) \rightarrow ((a \times b)\ `k`\ (c \times d))
\end{array}
$$

There are two special forms that are sometimes more convenient (one of which we've already seen in a more specialized context):

$$
\begin{array}{l}
\mathit{first} :: \mathit{MonoidalP}\ k \Rightarrow (a\ `k`\ c) \rightarrow ((a \times b)\ `k`\ (c \times b)) \\
\mathit{first}\ f = f \times \mathit{id}
\end{array}
$$

$$
\begin{array}{l}
\mathit{second} :: \mathit{MonoidalP}\ k \Rightarrow (b\ `k`\ d) \rightarrow ((a \times b)\ `k`\ (a \times d)) \\
\mathit{second}\ g = \mathit{id} \times g
\end{array}
$$

The following law holds for all monoidal categories [Gibbons, 2002, Section 1.5.1]:

$$(f \times g) \circ (p \times q) = (f \circ p) \times (g \circ q)$$

Taking $g = \mathit{id}$ and $p = \mathit{id}$, and renaming q to "g", we get

$$\mathit{first}\ f \circ \mathit{second}\ g = f \times g$$

Similarly,

$$\mathit{second}\ g \circ \mathit{first}\ f = f \times g$$

We can also define *second* in terms of *first* (or vice versa):[7]

$$\mathit{second}\ g = \mathit{swap} \circ \mathit{first}\ g \circ \mathit{swap}$$

Thanks to these relationships, any two of $(\times)$, *first*, and *second* can be defined in terms of the other. For our purpose, it will be convenient to calculate a definition of *first* on *StackFun*, and then define $(\times)$ as follows:

$$
\begin{array}{ll}
f \times g & = \mathit{first}\ f \circ \mathit{second}\ g \\
 & = \mathit{first}\ f \circ \mathit{swap} \circ \mathit{first}\ g \circ \mathit{swap}
\end{array}
$$

We thus need only define *first*, which we can do by solving the corresponding homomorphism property, i.e.,

$$\mathit{first}\ (\mathit{stackFun}\ f) = \mathit{stackFun}\ (\mathit{first}\ f)$$

Equivalently (filling in the definition of *stackFun*),

$$\mathit{first}\ (\mathit{SF}\ (\mathit{first}\ f)) = \mathit{SF}\ (\mathit{first}\ (\mathit{first}\ f))$$

What do we do with $\mathit{first}\ (\mathit{first}\ f)$?[8] Let's examine the types involved:

$$
\begin{array}{rl}
f & :: a \rightarrow c \\
\mathit{first}\ f & :: a \times b \rightarrow c \times b \\
\mathit{first}\ (\mathit{first}\ f) & :: \forall z.(a \times b) \times z \rightarrow (c \times b) \times z
\end{array}
$$

To reshape this computation into a stack function, temporarily move b aside by re-associating:

$$
\begin{array}{ll}
 & \mathit{first}\ (\mathit{first}\ f) \\
= & \{\ \text{definition of}\ \mathit{first}\ \text{on}\ (\rightarrow)\ \}
\end{array}
$$

---

[7] [What would it take to prove this claim in general?]

[8] [Also noted by Paterson [2003, Section 1.1] and by Alimarine et al. [2006, Definition 2]. [Is there a category theory reference for this property in, say, monoidal categories?]]

$$\begin{aligned}
&\quad \lambda((a,b),z) \to ((f\ a,b),z) \\
&= \{\ \text{definition of } \mathit{lassoc}, \mathit{rassoc}, \text{ and } \mathit{first} \text{ on } (\to)\ \} \\
&\quad \mathit{lassoc} \circ \mathit{first}\ f \circ \mathit{rassoc}
\end{aligned}$$

Our required homomorphism equation for *first* is thus equivalent to the following:[9]

$$\mathit{first}\ (\mathit{SF}\ (\mathit{first}\ f)) = \mathit{SF}\ (\mathit{lassoc} \circ \mathit{first}\ f \circ \mathit{rassoc})$$

Generalizing from $\mathit{first}\ f$, we get the following sufficient condition:

$$\mathit{first}\ (\mathit{SF}\ f) = \mathit{SF}\ (\mathit{lassoc} \circ f \circ \mathit{rassoc})$$

Since this generalized equation is in solved form, we can use it as a definition, expressing *second* and $(\times)$ in terms of it:

$$\begin{aligned}
&\textbf{instance}\ \mathit{MonoidalP}\ \mathit{StackFun}\ \textbf{where} \\
&\quad \mathit{first}\ (\mathit{SF}\ f) = \mathit{SF}\ (\mathit{lassoc} \circ f \circ \mathit{rassoc}) \\
&\quad \mathit{second}\ g = \mathit{swap} \circ \mathit{first}\ g \circ \mathit{swap} \\
&\quad f \times g = \mathit{first}\ f \circ \mathit{second}\ g
\end{aligned}$$

This sequentialized computation corresponds to right-to-left evaluation of arguments. We can get left-to-right evaluation by reformulating parallel composition as $f \times g = \mathit{first}\ f \circ \mathit{second}\ g$.

To understand the operational implications of this *MonoidalP* instance, let's see how parallel composition unfolds on a stack machine:

$$\begin{aligned}
&\quad \mathit{stackFun}\ f \times \mathit{stackFun}\ g \\
&= \{\ \text{definition of } \mathit{stackFun}\ \} \\
&\quad \mathit{SF}\ (\mathit{first}\ f) \times \mathit{SF}\ (\mathit{first}\ g) \\
&= \{\ \text{definition of } (\times) \text{ on } \mathit{StackFun}\ \} \\
&\quad \mathit{first}\ (\mathit{SF}\ (\mathit{first}\ f)) \circ \mathit{second}\ (\mathit{SF}\ (\mathit{first}\ g)) \\
&= \{\ \text{definition of } \mathit{second} \text{ on } \mathit{StackFun}\ \} \\
&\quad \mathit{first}\ (\mathit{SF}\ (\mathit{first}\ f)) \circ \mathit{swap} \circ \mathit{first}\ (\mathit{SF}\ (\mathit{first}\ g)) \circ \mathit{swap} \\
&= \{\ \text{definitions of } \mathit{first} \text{ and } \mathit{swap} \text{ on } \mathit{StackFun}\ \} \\
&\quad \mathit{SF}\ (\mathit{lassoc} \circ \mathit{first}\ f \circ \mathit{rassoc}) \circ \mathit{stackFun}\ \mathit{swap} \circ \mathit{SF}\ (\mathit{lassoc} \circ \mathit{first}\ g \circ \mathit{rassoc}) \circ \mathit{stackFun}\ \mathit{swap} \\
&= \{\ \text{definition of } \mathit{stackFun}\ \} \\
&\quad \mathit{SF}\ (\mathit{lassoc} \circ \mathit{first}\ f \circ \mathit{rassoc}) \circ \mathit{SF}\ (\mathit{first}\ \mathit{swap}) \circ \mathit{SF}\ (\mathit{lassoc} \circ \mathit{first}\ g \circ \mathit{rassoc}) \circ \mathit{SF}\ (\mathit{first}\ \mathit{swap}) \\
&= \{\ \text{definition of } (\circ) \text{ on } \mathit{StackFun}\ \} \\
&\quad \mathit{SF}\ (\mathit{lassoc} \circ \mathit{first}\ f \circ \mathit{rassoc} \circ \mathit{first}\ \mathit{swap} \circ \mathit{lassoc} \circ \mathit{first}\ g \circ \mathit{rassoc} \circ \mathit{first}\ \mathit{swap})
\end{aligned}$$

Step-by-step, the stack evolves as follows:

$$\begin{array}{lcll}
 & & ((a,b) & ,z) \\
\mathit{first}\ \mathit{swap} & \longmapsto & ((b,a) & ,z) \\
\mathit{rassoc} & \longmapsto & (b & ,(a,z)) \\
\mathit{first}\ g & \longmapsto & (g\ b & ,(a,z)) \\
\mathit{lassoc} & \longmapsto & ((g\ b,a) & ,z) \\
\mathit{first}\ \mathit{swap} & \longmapsto & ((a,g\ b) & ,z) \\
\mathit{rassoc} & \longmapsto & (a & ,(g\ b,z)) \\
\mathit{first}\ f & \longmapsto & (f\ a & ,(g\ b,z)) \\
\mathit{lassoc} & \longmapsto & ((f\ a,g\ b),z) &
\end{array}$$

Operationally, $\mathit{first}\ g$ and $\mathit{first}\ f$ stand for stack-transformation sub-sequences. Note that this final stack state is equal to $\mathit{first}\ (f \times g)\ ((a,b),z)$ as needed. We have, however, flattened (under the *SF* constructor) into *purely sequential* compositions of functions of three forms:

- $\mathit{first}\ p$ for simple functions p,
- *rassoc*, and
- *lassoc*.

Moreover, the latter two always come in balanced pairs.

[9] It may be tempting to invoke the definition of of $(\circ)$ on *StackFun*, and rewrite the RHS to $\mathit{SF}\ \mathit{lassoc} \circ \mathit{SF}\ (\mathit{first}\ f) \circ \mathit{SF}\ \mathit{rassoc}$. Exercise: what goes wrong?

1.3 Duplicating and destroying information

The vocabulary above gives no way to duplicate or destroy information, but there is a standard interface for doing so:[10]

$$
\begin{array}{l}
\textbf{class}\ \mathit{Cartesian}\ k\ \textbf{where} \\
\quad \mathit{exl}\ \ :: (a \times b)\ `k`\ a \\
\quad \mathit{exr}\ :: (a \times b)\ `k`\ b \\
\quad \mathit{dup} :: a\ `k`\ (a \times a)
\end{array}
$$

Again, the required homomorphism properties are already in solved form, so we can immediately write them down a sufficient instance:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Cartesian}\ \mathit{StackFun}\ \textbf{where} \\
\quad \mathit{exl}\ \ = \mathit{stackFun}\ \mathit{exl} \\
\quad \mathit{exr}\ = \mathit{stackFun}\ \mathit{exr} \\
\quad \mathit{dup} = \mathit{stackFun}\ \mathit{dup}
\end{array}
$$

These three operations are used in the translation from λ-calculus (e.g., Haskell) to categorical form. The two projections (*exl* and *exr*) arise from translation of pattern-matching on pairs, while duplication is used for translation of pair formation and application expressions, in the guise of the "fork" operation [Elliott, 2017, Section 3]:

$$
\begin{array}{l}
(\vartriangle) :: (a\ `k`\ c) \rightarrow (a\ `k`\ d) \rightarrow (a\ `k`\ (c \times d)) \\
f \vartriangle g = (f \times g) \circ \mathit{dup}
\end{array}
$$

1.4 Conditional composition (coproducts)

Just as we have *MonoidalP* and *Cartesian* for products (defined above), there are also *dual* counterparts that work on coproducts (sums) instead of products:[11]

$$
\begin{array}{l}
\textbf{class}\ \mathit{MonoidalS}\ k\ \textbf{where} \\
\quad (+) :: (a\ `k`\ c) \rightarrow (b\ `k`\ d) \rightarrow ((a + b)\ `k`\ (c + d))
\end{array}
$$

There is also a dual interface to *Cartesian*:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Cocartesian}\ k\ \textbf{where} \\
\quad \mathit{inl}\ \ :: a\ `k`\ (a + b) \\
\quad \mathit{inr}\ \ :: b\ `k`\ (a + b) \\
\quad \mathit{jam} :: (a + a)\ `k`\ a
\end{array}
$$

The homomorphism properties are easily satisfied:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Cartesian}\ \mathit{StackFun}\ \textbf{where} \\
\quad \mathit{inl}\ \ = \mathit{stackFun}\ \mathit{inl} \\
\quad \mathit{inr}\ \ = \mathit{stackFun}\ \mathit{inr} \\
\quad \mathit{jam} = \mathit{stackFun}\ \mathit{jam}
\end{array}
$$

Just as the $(\vartriangle)$ ("fork") operation for producing products is defined via $(\times)$ and *dup*, so is the $(\triangledown)$ ("join") operation for consuming coproducts/sums defined via $(+)$ and *jam*:

[10] [I've been experimenting with having *Cartesian* be independent of *MonoidalP*. A more conventional choice is to have the former require the latter. I think the clean split enables a generalization later on.]

[11] There are two special forms dual to *first* and *second*:

$$
\begin{array}{l}
\mathit{left} :: \mathit{MonoidalS}\ k \Rightarrow (a\ `k`\ c) \rightarrow ((a + b)\ `k`\ (c + b)) \\
\mathit{left}\ f = f + \mathit{id}
\end{array}
$$

$$
\begin{array}{l}
\mathit{right} :: \mathit{MonoidalS}\ k \Rightarrow (b\ `k`\ d) \rightarrow ((a + b)\ `k`\ (a + d)) \\
\mathit{right}\ g = \mathit{id} + g
\end{array}
$$

$$(\triangledown) :: (a \mathbin{`k`} c) \to (b \mathbin{`k`} c) \to ((a + b) \mathbin{`k`} c)$$
$$f \triangledown g = jam \circ (f + g)$$

[Consider skipping $(\times)$ and $(+)$ in favor of $(\vartriangle)$ and $(\triangledown)$, which is consistent with the CtoC paper [Elliott, 2017].]

Categorical products and coproducts are related in *distributive* categories [Gibbons, 2002, Section 1.5.5]:[12]

$$\textbf{class}\ (Cartesian\ k, Cocartesian\ k) \Rightarrow Distributive\ k\ \textbf{where}$$
$$\quad distl :: (a \times (u + v)) \mathbin{`k`} ((a \times u) + (a \times v))$$

The $(\triangledown)$ and *distl* operations suffice to translate multi-constructor **case** expressions to categorical form [Elliott, 2017, Section 8]. The instance for stack functions is again trivial:

$$\textbf{instance}\ Distributive\ StackFun\ \textbf{where}$$
$$\quad distl = stackFun\ distl$$

With the *MonoidalS* and *Distributive* instances for $(\to)$, we can define a correct *MonoidalS* instance for *StackFun*:

Theorem 4 (Proved in Appendix A.4). Given the instance definition above, *stackFun* is a *MonoidalS* homomorphism.

$$\textbf{instance}\ MonoidalS\ StackFun\ \textbf{where}$$
$$\quad SF\ f + SF\ g = SF\ (undistr \circ (f + g) \circ distr)$$

1.5 Closed categories

[In progress. I don't know whether *StackFun* is closed. In any case, probably move to after Section 2.]

2 Stack programs

The definitions of *StackFun* and its type class instances above capture the essence of stack computation, while allowing evaluation as functions (via *evalStackFun*). For optimization and code generation, however, we will need to inspect the structure of a computation, which is impossible with *StackFun* due to its representation as a function. To remedy this situation, let's now make the notion of stack computation explicit as a data type having a precise relationship with the function representation.

As a first step, define a data type of reified primitive functions, along with an evaluator:

$$\textbf{data}\ Prim :: * \to * \to *\ \textbf{where}$$
$$\quad Exl\ :: Prim\ (a \times b)\ a$$
$$\quad Exr\ :: Prim\ (a \times b)\ b$$
$$\quad Dup :: Prim\ a\ (a \times a)$$
$$\quad \ldots$$
$$\quad Negate :: Num\ a \Rightarrow Prim\ a\ a$$
$$\quad Add, Sub, Mul :: Num\ a \Rightarrow Prim\ (a \times a)\ a$$
$$\quad \ldots$$

$$evalPrim :: Prim\ a\ b \to (a \to b)$$
$$evalPrim\ Exl \quad = exl$$

[12] There's also a right-distributing counterpart:

$$distr :: ((u + v) \times b) \mathbin{`k`} ((u \times b) + (v \times b))$$
$$distr = (swap + swap) \circ distl \circ swap$$

Inverses can be defined without *Distributive* [Gibbons, 2002, Section 1.5.5]:

$$undistl :: (MonoidalP\ k, MonoidalS\ k, Cocartesian\ k) \Rightarrow ((a \times u) + (a \times v)) \mathbin{`k`} (a \times (u + v))$$
$$undistl = second\ inl \triangledown second\ inr$$

$$undistr :: (MonoidalP\ k, MonoidalS\ k, Cocartesian\ k) \Rightarrow ((u \times b) + (v \times b)) \mathbin{`k`} ((u + v) \times b)$$
$$undistr = first\ inl \triangledown first\ inr$$

$$
\begin{aligned}
&\mathit{evalPrim}\ \mathit{Exr} &&= \mathit{exr} \\
&\mathit{evalPrim}\ \mathit{Dup} &&= \mathit{dup} \\
&\quad \ldots \\
&\mathit{evalPrim}\ \mathit{Negate} &&= \mathit{negateC} \\
&\mathit{evalPrim}\ \mathit{Add} &&= \mathit{addC} \\
&\mathit{evalPrim}\ \mathit{Sub} &&= \mathit{subC} \\
&\mathit{evalPrim}\ \mathit{Mul} &&= \mathit{mulC} \\
&\quad \ldots
\end{aligned}
$$

A stack program is a sequence of instructions, most of which correspond to primitive functions that replace the top of the stack without using the rest, and the others that re-associate:[13]

$$
\begin{aligned}
&\mathbf{data}\ \mathit{StackOp} :: * \to * \to *\ \mathbf{where} \\
&\quad \mathit{Prim} :: \mathit{Prim}\ a\ b \to \mathit{StackOp}\ (a \times z)\ (b \times z) \\
&\quad \mathit{Push} :: \mathit{StackOp}\ ((a \times b) \times z)\ (a \times (b \times z)) \\
&\quad \mathit{Pop}\ \ :: \mathit{StackOp}\ (a \times (b \times z))\ ((a \times b) \times z)
\end{aligned}
$$

Stack operations have a simple interpretation as functions:[14]

$$
\begin{aligned}
&\mathit{evalStackOp} :: \mathit{StackOp}\ u\ v \to (u \to v) \\
&\mathit{evalStackOp}\ (\mathit{Prim}\ f) &&= \mathit{first}\ (\mathit{evalPrim}\ f) \\
&\mathit{evalStackOp}\ \mathit{Push} &&= \mathit{rassoc} \\
&\mathit{evalStackOp}\ \mathit{Pop} &&= \mathit{lassoc}
\end{aligned}
$$

We will form chains (linear sequences) of stack operations, each feeding its result to the next:[15]

$$
\begin{aligned}
&\mathbf{infixr}\ 5 \triangleleft \\
&\mathbf{data}\ \mathit{StackOps} :: * \to * \to *\ \mathbf{where} \\
&\quad \mathit{Nil} :: \mathit{StackOps}\ a\ a \\
&\quad (\triangleleft) :: \mathit{StackOp}\ a\ b \to \mathit{StackOps}\ b\ c \to \mathit{StackOps}\ a\ c
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{evalStackOps} :: \mathit{StackOps}\ u\ v \to (u \to v) \\
&\mathit{evalStackOps}\ \mathit{Nil} &&= \mathit{id} \\
&\mathit{evalStackOps}\ (\mathit{op} \triangleleft \mathit{rest}) &&= \mathit{evalStackOps}\ \mathit{rest} \circ \mathit{evalStackOp}\ \mathit{op}
\end{aligned}
$$

We'll want to compose these chains sequentially:

$$
\begin{aligned}
&\mathbf{infixr}\ 5 +\!\!+ \\
&(+\!\!+) :: \mathit{StackOps}\ a\ b \to \mathit{StackOps}\ b\ c \to \mathit{StackOps}\ a\ c \\
&\mathit{Nil} &&+\!\!+\ \mathit{ops}' = \mathit{ops}' \\
&(\mathit{op} \triangleleft \mathit{ops}) &&+\!\!+\ \mathit{ops}' = \mathit{op} \triangleleft (\mathit{ops} +\!\!+ \mathit{ops}')
\end{aligned}
$$

Lemma 5. Nil and $(+\!\!+)$ implement identity and composition on functions in the following sense:

$$\mathit{id} = \mathit{evalStackOps}\ \mathit{Nil}$$

$$\mathit{evalStackOps}\ g \circ \mathit{evalStackOps}\ f = \mathit{evalStackOps}\ (f +\!\!+ g)$$

Proof. The first property is immediate from the definition of $\mathit{evalStackOps}$. The second follows by structural induction on g. □

A complete stack program is a chain of stack operations that can change only the top of the stack:

$$\mathbf{data}\ \mathit{StackProg}\ a\ b = \mathit{SP}\ \{\mathit{unSP} :: \forall z. \mathit{StackOps}\ (a \times z)\ (b \times z)\}$$

[13] [Maybe rename the constructors to something like *FirstSO*, *RassocSO*, and *LassocSO*. Look for prettier alternatives.]

[14] The operations $\mathit{negateC}$, addC, etc are the categorical versions of negate, $(+)$, etc, uncurried where needed. We use the categorical versions here for easier generalization later.

[15] [Maybe I should change *StackOps* to preserve the composition structure. The calculations would be simpler, and the implementation more efficient.]

$$
\begin{aligned}
&\textbf{instance } \mathit{Category}\ \mathit{StackProg}\ \textbf{where} \\
&\quad \mathit{id} = \mathit{SP}\ \mathit{Nil} \\
&\quad \mathit{SP}\ g \circ \mathit{SP}\ f = \mathit{SP}\ (f +\!\!+\ g) \\[6pt]
&\textbf{instance } \mathit{MonoidalP}\ \mathit{StackFun}\ \textbf{where} \\
&\quad \mathit{first}\ (\mathit{SP}\ \mathit{ops}) = \mathit{SP}\ (\mathit{Push} \triangleleft \mathit{ops} +\!\!+\ \mathit{Pop} \triangleleft \mathit{Nil}) \\
&\quad \mathit{second}\ g = \mathit{swap} \circ \mathit{first}\ g \circ \mathit{swap} \\
&\quad f \times g = \mathit{first}\ f \circ \mathit{second}\ g \\[6pt]
&\mathit{primProg} :: \mathit{Prim}\ a\ b \to \mathit{StackProg}\ a\ b \\
&\mathit{primProg}\ p = \mathit{SP}\ (\mathit{Prim}\ p \triangleleft \mathit{Nil}) \\[6pt]
&\textbf{instance } \mathit{Cartesian}\ \mathit{StackProg}\ \textbf{where} \\
&\quad \mathit{exl}\ \ = \mathit{primProg}\ \mathit{Exl} \\
&\quad \mathit{exr}\ = \mathit{primProg}\ \mathit{Exr} \\
&\quad \mathit{dup} = \mathit{primProg}\ \mathit{Dup} \\[6pt]
&\textbf{instance } \mathit{Num}\ a \Rightarrow \mathit{NumCat}\ \mathit{StackProg}\ a\ \textbf{where} \\
&\quad \mathit{negateC} = \mathit{primProg}\ \mathit{Negate} \\
&\quad \mathit{addC}\ = \mathit{primProg}\ \mathit{Add} \\
&\quad \mathit{subC}\ \ = \mathit{primProg}\ \mathit{Sub} \\
&\quad \mathit{mulC}\ = \mathit{primProg}\ \mathit{Mul} \\
&\qquad\quad \ldots
\end{aligned}
$$

Figure 1: Stack programs (specified by *progFun* as homomorphism and calculated in Appendix A.5)

To compile a stack program, convert it to a stack function:

$$
\begin{aligned}
&\mathit{progFun} :: \mathit{StackProg}\ a\ b \to \mathit{StackFun}\ a\ b \\
&\mathit{progFun}\ (\mathit{SP}\ \mathit{ops}) = \mathit{SF}\ (\mathit{evalStackOps}\ \mathit{ops})
\end{aligned}
$$

We can also convert all the way to a regular function:

$$
\begin{aligned}
&\mathit{evalProg} :: \mathit{StackProg}\ a\ b \to (a \to b) \\
&\mathit{evalProg} = \mathit{evalStackFun} \circ \mathit{progFun}
\end{aligned}
$$

This *evalProg* definition constitutes an interpreter for stack programs. Our quest, however, is the reverse. Given a function f, we want to construct a purely sequential, stack-manipulating program p such that $\mathit{evalProg}\ p = f$. As stated, this goal is impossible, since functions are not inspectable. Moreover, for a given function f there may be no program p that satisfy this requirement, or there may be many such programs. Although we cannot invert *evalProg* as written, we can transform this specification into a correct and effective implementation. As in Section 1, we can calculate instances of *Category* etc for *StackProg* resulting in Figure 1.

Theorem 6 (Proved in Appendix A.5)*.* Given the definitions in Figure 1, *progFun* is a homomorphism with respect to each instantiated class.

Corollary 6.1. Given the definitions in Figure 1, *evalProg* is also a homomorphism with respect to each instantiated class.

Proof. The composition of homomorphisms (here *evalStackFun* and *progFun*) is a homomorphism (*evalProg*). □

3 What's next?

[Working here.]

- Examples
- Optimization
- More with *Cocartesian*, including multi-constructor **case** expressions. Maybe start with conditionals. Hm! I don't think I can define $(+)$ on *StackProg*, because the representation is a linear sequence of stack operations.

4 Related work

- [Meijer, 1992]
- [Meijer, 1991]
- [Bahr and Hutton, 2015]
- [Vazou et al., 2018]
- [McKinna and Wright, 2006]

A Proofs

A.1 Lemma 1

We need to show that *evalStackFun* is a left inverse for *stackFun*, i.e., for all f, $\mathit{evalStackFun}\ (\mathit{stackFun}\ f) = f$. Reasoning equationally,

$$
\begin{array}{ll}
& \mathit{evalStackFun}\ (\mathit{stackFun}\ f) \\
= & \{\ \text{definition of } \mathit{stackFun}\ \} \\
& \mathit{evalStackFun}\ (\mathit{SF}\ (\mathit{first}\ f)) \\
= & \{\ \text{second definition of } \mathit{evalStackFun}\ \} \\
& \mathit{rcounit} \circ \mathit{first}\ f \circ \mathit{runit} \\
= & \{\ \text{definition of } (\circ) \text{ on functions}\ \} \\
& \lambda a \rightarrow \mathit{rcounit}\ (\mathit{first}\ f\ (\mathit{runit}\ a)) \\
= & \{\ \text{definition of } \mathit{rcounit} \text{ on functions}\ \} \\
& \lambda a \rightarrow \mathit{rcounit}\ (\mathit{first}\ f\ (a, ())) \\
= & \{\ \text{definition of } \mathit{first} \text{ on functions}\ \} \\
& \lambda a \rightarrow \mathit{rcounit}\ (f\ a, ()) \\
= & \{\ \text{definition of } \mathit{rcounit} \text{ on functions}\ \} \\
& \lambda a \rightarrow f\ a \\
= & \{\ \eta\text{-reduction}\ \} \\
& f
\end{array}
$$

A.2 Lemma 2

[Adapt Joachim Breitner's proof in haskell-cafe email 2018-07-23, giving him credit.]

A.3 Lemma 3

We need to show that *stackFun* is injective, i.e., $\mathit{stackFun}\ f = \mathit{stackFun}\ f' \Longrightarrow f = f'$. Since $\mathit{stackFun} = \mathit{SF} \circ \mathit{first}$, and *SF* is *injective*, we only need show that *first* as injective:

$$
\begin{array}{ll}
& \mathit{first}\ f = \mathit{first}\ f' \\
\Longleftrightarrow & \{\ \text{equality on functions (extensionality)}\ \} \\
& \forall x\ z.\mathit{first}\ f\ (x, z) = \mathit{first}\ f'\ (x, z)
\end{array}
$$

$$
\begin{aligned}
\iff & \ \{ \text{ definition of } \mathit{first} \ \} \\
& \forall x\ z. (f\ x, z) = (f'\ x, z) \\
\iff & \ \{ \text{ equality on pairs } \} \\
& \forall x\ z. f\ x = f'\ x \wedge z = z \\
\iff & \ \{ \text{ trivial conjunct } \} \\
& \forall x. f\ x = f'\ x \\
\iff & \ \{ \text{ equality on functions } \} \\
& f = f'
\end{aligned}
$$

A.4 Theorem 4

The *MonoidalS* homomorphism property:

$$\mathit{stackFun}\ f + \mathit{stackFun}\ g = \mathit{stackFun}\ (f + g)$$

Using the definition of *stackFun*,

$$\mathit{SF}\ (\mathit{first}\ f) + \mathit{SF}\ (\mathit{first}\ g) = \mathit{SF}\ (\mathit{first}\ (f + g))$$

Simplify the RHS:

$$
\begin{aligned}
& \mathit{first}\ (f + g) \\
= & \ \{ \ \mathit{undistr} \circ \mathit{distr} = \mathit{id} \ \} \\
& (\mathit{undistr} \circ \mathit{distr}) \circ \mathit{first}\ (f + g) \circ (\mathit{undistr} \circ \mathit{distr}) \\
= & \ \{ \text{ associativity of } (\circ) \ \} \\
& \mathit{undistr} \circ (\mathit{distr} \circ \mathit{first}\ (f + g) \circ \mathit{undistr}) \circ \mathit{distr} \\
= & \ \{ \text{ Lemma 7 } \} \\
& \mathit{undistr} \circ (\mathit{first}\ f + \mathit{first}\ g) \circ \mathit{distr}
\end{aligned}
$$

The required *MonoidalS* homomorphism is thus equivalent to

$$\mathit{SF}\ (\mathit{first}\ f) + \mathit{SF}\ (\mathit{first}\ g) = \mathit{SF}\ (\mathit{undistr} \circ (\mathit{first}\ f + \mathit{first}\ g) \circ \mathit{distr})$$

Strengthen by generalizing from $\mathit{first}\ f$ and $\mathit{first}\ g$, resulting in a sufficient definition:

```
instance MonoidalS StackFun where
  SF f + SF g = SF (undistr ∘ (f + g) ∘ distr)
```

The needed lemma:

Lemma 7. $\mathit{distr} \circ \mathit{first}\ (f + g) \circ \mathit{undistr} = \mathit{first}\ f + \mathit{first}\ g$

Proof. It will be convenient to prove an equivalent, slightly different form, eliminating *distr*:

$$\mathit{first}\ (f + g) \circ \mathit{undistr} = \mathit{undistr} \circ (\mathit{first}\ f + \mathit{first}\ g)$$

Simplify the LHS:

$$
\begin{aligned}
& \mathit{first}\ (f + g) \circ \mathit{undistr} \\
= & \ \{ \text{ definition of } \mathit{undistr} \text{ [Gibbons, 2002, Section 1.5.5 variation] } \} \\
& \mathit{first}\ (f + g) \circ (\mathit{first}\ \mathit{inl} \triangledown \mathit{first}\ \mathit{inr}) \\
= & \ \{ \ r \circ (p \triangledown q) = (r \circ p) \triangledown (r \circ q) \text{ [Gibbons, 2002, Section 1.5.2] } \} \\
& \mathit{first}\ (f + g) \circ \mathit{first}\ \mathit{inl} \triangledown \mathit{first}\ (f + g) \circ \mathit{first}\ \mathit{inr} \\
= & \ \{ \text{ property of } \mathit{first} \text{ and } (\circ) \ \} \\
& \mathit{first}\ ((f + g) \circ \mathit{inl}) \triangledown \mathit{first}\ ((f + g) \circ \mathit{inr}) \\
= & \ \{ \text{ [Gibbons, 2002, Section 1.5.2 variation] } \} \\
& \mathit{first}\ (\mathit{inl} \circ f) \triangledown \mathit{first}\ (\mathit{inr} \circ g)
\end{aligned}
$$

Then the RHS:

$$\begin{array}{ll} & \mathit{undistr} \circ (\mathit{first}\ f + \mathit{first}\ g) \\ = & \{\ \text{definition of } \mathit{undistr}\ \} \\ & (\mathit{first}\ \mathit{inl} \triangledown \mathit{first}\ \mathit{inr}) \circ (\mathit{first}\ f + \mathit{first}\ g) \\ = & \{\ (\triangledown)\ /\ (+) \text{ law [Gibbons, 2002, Section 1.5.2] }\} \\ & (\mathit{first}\ \mathit{inl} \circ \mathit{first}\ f) \triangledown (\mathit{first}\ \mathit{inr} \circ \mathit{first}\ g) \\ = & \{\ \text{Property of } \mathit{first} \text{ and } (\circ)\ \} \\ & \mathit{first}\ (\mathit{inl} \circ f) \triangledown \mathit{first}\ (\mathit{inr} \circ g) \end{array}$$

□

A.5 Theorem 6

Let's see how the definitions in Figure 1 follow from homomorphism properties.

A.5.1 *Category*

The homomorphic requirement for id:

$$\mathit{progFun}\ \mathit{id} = \mathit{id}$$

Simplify the LHS:

$$\begin{array}{ll} & \mathit{progFun}\ \mathit{id} \\ = & \{\ \mathit{SP} \text{ and } \mathit{unSP} \text{ are inverses }\} \\ & \mathit{progFun}\ (\mathit{SP}\ (\mathit{unSP}\ \mathit{id})) \\ = & \{\ \text{definition of } \mathit{progFun}\ \} \\ & \mathit{SF}\ (\mathit{evalStackOps}\ (\mathit{unSP}\ \mathit{id})) \end{array}$$

Then the RHS:

$$\begin{array}{ll} & \mathit{id} \\ = & \{\ \text{definition of } \mathit{id} \text{ on } \mathit{SF}\ \} \\ & \mathit{SF}\ \mathit{id} \\ = & \{\ \text{Lemma 5 }\} \\ & \mathit{SF}\ (\mathit{evalStackOps}\ \mathit{Nil}) \\ = & \{\ \mathit{unSP} \text{ and } \mathit{SP} \text{ are inverses }\} \\ & \mathit{SF}\ (\mathit{evalStackOps}\ (\mathit{unSP}\ (\mathit{SP}\ \mathit{Nil}))) \end{array}$$

The simplified id homomorphism requirement:

$$\begin{array}{ll} & \mathit{SF}\ (\mathit{evalStackOps}\ (\mathit{unSP}\ \mathit{id})) = \mathit{SF}\ (\mathit{evalStackOps}\ (\mathit{unSP}\ (\mathit{SP}\ \mathit{Nil}))) \\ \Longleftarrow & \{\ \mathit{SF} \circ \mathit{evalStackOps} \circ \mathit{unSP} \text{ is a function }\} \\ & \mathit{id} = \mathit{SP}\ \mathit{Nil} \end{array}$$

The homomorphic requirement for $(\circ)$:

$$\mathit{progFun}\ (\mathit{SP}\ g \circ \mathit{SP}\ f) = \mathit{progFun}\ (\mathit{SP}\ g) \circ \mathit{progFun}\ (\mathit{SP}\ f)$$

Simplify the LHS:

$$\begin{array}{ll} & \mathit{progFun}\ (\mathit{SP}\ g \circ \mathit{SP}\ f) \\ = & \{\ \text{definition of } \mathit{progFun}\ \} \\ & \mathit{SF}\ (\mathit{evalStackOps}\ (\mathit{unSP}\ (\mathit{SP}\ g \circ \mathit{SP}\ f))) \end{array}$$

Then the RHS:

$$\begin{array}{ll} & \mathit{progFun}\ (\mathit{SP}\ g) \circ \mathit{progFun}\ (\mathit{SP}\ f) \\ = & \{\ \text{definition of } \mathit{progFun}\ \} \end{array}$$

$$
\begin{aligned}
& \mathit{SF}\ (\mathit{evalStackOps}\ g) \circ \mathit{SF}\ (\mathit{evalStackOps}\ f) \\
=\ & \{\ \text{definition of } (\circ) \text{ for } \mathit{StackFun}\ \} \\
& \mathit{SF}\ (\mathit{evalStackOps}\ g \circ \mathit{evalStackOps}\ f) \\
=\ & \{\ \text{Lemma 5}\ \} \\
& \mathit{SF}\ (\mathit{evalStackOps}\ (f +\!\!+\, g))
\end{aligned}
$$

These simplified $(\circ)$ homomorphism requirement:

$$
\begin{aligned}
& \mathit{SF}\ (\mathit{evalStackOps}\ (\mathit{unSP}\ (\mathit{SP}\ g \circ \mathit{SP}\ f))) = \mathit{SF}\ (\mathit{evalStackOps}\ (f +\!\!+\, g)) \\
\Longleftarrow\ & \{\ \mathit{SF} \circ \mathit{evalStackOps} \text{ is a function}\ \} \\
& \mathit{unSP}\ (\mathit{SP}\ g \circ \mathit{SP}\ f) = f +\!\!+\, g \\
\Longleftrightarrow\ & \{\ \mathit{SP} \text{ is bijective}\ \} \\
& \mathit{SP}\ (\mathit{unSP}\ (\mathit{SP}\ g \circ \mathit{SP}\ f)) = \mathit{SP}\ (f +\!\!+\, g) \\
\Longleftrightarrow\ & \{\ \mathit{SP} \text{ and } \mathit{unSP} \text{ are inverses}\ \} \\
& \mathit{SP}\ g \circ \mathit{SP}\ f = \mathit{SP}\ (f +\!\!+\, g)
\end{aligned}
$$

These simplified homomorphic specifications are in solved form and so suffice as a correct implementation.

A.5.2 Primitive functions

The *primProg* function (Figure 1) captures primitive functions in the following sense:

Lemma 8. $\mathit{progFun}\ (\mathit{primProg}\ \mathit{op}) = \mathit{stackFun}\ (\mathit{evalPrim}\ \mathit{op})$

Proof. Reason equationally:

$$
\begin{aligned}
& \mathit{progFun}\ (\mathit{primProg}\ \mathit{op}) \\
=\ & \{\ \text{definition of } \mathit{primProg}\ \} \\
& \mathit{progFun}\ (\mathit{SP}\ (\mathit{Prim}\ \mathit{op} \triangleleft \mathit{Nil})) \\
=\ & \{\ \text{definition of } \mathit{progFun}\ \} \\
& \mathit{SF}\ (\mathit{evalStackOps}\ (\mathit{Prim}\ \mathit{op} \triangleleft \mathit{Nil})) \\
=\ & \{\ \text{definition of } \mathit{evalStackOps}\ \} \\
& \mathit{SF}\ (\mathit{evalStackOps}\ \mathit{Nil} \circ \mathit{evalStackOp}\ (\mathit{Prim}\ \mathit{op})) \\
=\ & \{\ \text{definitions of } \mathit{evalStackOps} \text{ and } \mathit{evalStackOp}\ \} \\
& \mathit{SF}\ (\mathit{id} \circ \mathit{first}\ (\mathit{evalPrim}\ \mathit{op})) \\
=\ & \{\ \mathit{Category} \text{ law}\ \} \\
& \mathit{SF}\ (\mathit{first}\ (\mathit{evalPrim}\ \mathit{op})) \\
=\ & \{\ \text{definition of } \mathit{stackFun}\ \} \\
& \mathit{stackFun}\ (\mathit{evalPrim}\ \mathit{op})
\end{aligned}
$$

□

As a typical use of *evalPrim*, consider the homomorphism equation $\mathit{progFun}\ \mathit{exl} = \mathit{exl}$, beginning with the RHS:

$$
\begin{aligned}
& \mathit{exl} \\
=\ & \{\ \mathit{stackFun} \text{ is a } \mathit{Cartesian} \text{ homomorphism}\ \} \\
& \mathit{stackFun}\ \mathit{exl} \\
=\ & \{\ \text{definition of } \mathit{evalPrim}\ \} \\
& \mathit{stackFun}\ (\mathit{evalPrim}\ \mathit{Exl}) \\
=\ & \{\ \text{Lemma 8}\ \} \\
& \mathit{progFun}\ (\mathit{opP}\ \mathit{Exl})
\end{aligned}
$$

Our homomorphic specification is thus

$$
\begin{aligned}
& \mathit{progFun}\ \mathit{exl} = \mathit{progFun}\ (\mathit{opP}\ \mathit{Exl}) \\
\Longleftarrow\ & \{\ \mathit{progFun} \text{ is a function}\ \} \\
& \mathit{exl} = \mathit{opP}\ \mathit{Exl}
\end{aligned}
$$

A.5.3 *MonoidalP*

The required homomorphism:

$$progFun\ (first\ f) = first\ (progFun\ f)$$

In other words,

$$progFun\ (first\ (SP\ ops)) = first\ (progFun\ (SP\ ops))$$

Simplify the RHS:

$$\begin{aligned}
& first\ (progFun\ (SP\ ops)) \\
= {} & \{\ \text{definition of } progFun\ \} \\
& first\ (SF\ (evalStackOps\ ops)) \\
= {} & \{\ \text{definition of } first \text{ on } StackFun\ \} \\
& SF\ (lassoc \circ evalStackOps\ ops \circ rassoc) \\
= {} & \{\ \text{definition of } evalStackOps\text{; Lemma 5}\ \} \\
& SF\ (evalStackOps\ (Push \triangleleft ops + Pop \triangleleft Nil)) \\
= {} & \{\ \text{definition of } progFun\ \} \\
& progFun\ (SP\ (Push \triangleleft ops + Pop \triangleleft Nil))
\end{aligned}$$

The simplified homomorphism:

$$progFun\ (first\ (SP\ ops)) = progFun\ (SP\ (Push \triangleleft ops + Pop \triangleleft Nil))$$

A sufficient definition:

$$first\ (SP\ ops) = SP\ (Push \triangleleft ops + Pop \triangleleft Nil)$$

[*MonoidalS*. Doesn't seem possible with the current *StackProg* definition.]

References

Artem Alimarine, Sjaak Smetsers, Arjen Weelden, Marko Eekelen, and Rinus Plasmeijer. There and back again: arrows for invertible programming. In *In Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 86–97, 2006.

Steve Awodey. *Category theory*, volume 49 of *Oxford Logic Guides*. Oxford University Press, 2006.

Patrick Bahr and Graham Hutton. Calculating correct compilers. *Journal of Functional Programming*, 25, 2015.

Conal Elliott. Compiling to categories. In *Proceedings of the ACM on Programming Languages (ICFP)*, 2017.

Conal Elliott. The essence of automatic differentiation. In *Proceedings of the ACM on Programming Languages (ICFP)*, 2018.

Jeremy Gibbons. Calculating functional programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

F. William Lawvere and Stephen H. Schanuel. *Conceptual Mathematics: A First Introduction to Categories*. Cambridge University Press, 2nd edition, 2009.

Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer New York, 1998.

James McKinna and Joel Wright. Functional Pearl: A type-correct, stack-safe, provably correct, expression compiler in Epigram. 2006.

Erik Meijer. More Advice on Proving a Compiler Correct: Improve a Correct Compiler. In *Declarative Programming*, 1991.

Erik Meijer. *Calculating Compilers*. PhD thesis, Nijmegen University, feb 1992.

nLab. retract. wiki page, 2009–2018. URL `https://ncatlab.org/nlab/history/retract`.

Ross Paterson. Arrows and computation. In *The Fun of Programming*, pages 201–222, 2003.

Niki Vazou, Joachim Breitner, Will Kunkel, David Van Horn, and Graham Hutton. Theorem proving for all: Equational reasoning in Liquid Haskell. In *Haskell Symposium*, June 2018.