Compiling gracefully

Conal Elliott

Haskell Love 2020

Example

Source code:

$$\lambda(x, y) \rightarrow 2 * x + 3 * y$$

Object code:

$$[\mathit{Dup}, \mathit{Push}, \mathit{Dup}, \mathit{Push}, \mathit{Const}\ 2, \mathit{Pop}, \mathit{Swap}, \mathit{Push}, \mathit{Exl}, \mathit{Pop}$$
$$, \mathit{Swap}, \mathit{Mul}, \mathit{Pop}, \mathit{Swap}, \mathit{Push}, \mathit{Dup}, \mathit{Push}, \mathit{Const}\ 3, \mathit{Pop}, \mathit{Swap}$$
$$, \mathit{Push}, \mathit{Exr}, \mathit{Pop}, \mathit{Swap}, \mathit{Mul}, \mathit{Pop}, \mathit{Swap}, \mathit{Add}]$$

Compiling gracefully

Compiling gracefully

1. Identify essence of stack computation.
2. Specify by precise analogy.
3. Solve for correct implementation.

The essence of stack computation

$\forall z.(a \times z \to b \times z)$ -- "accumulator" and "stack"

The essence of stack computation

$\forall z.(a \times z \to b \times z)$ -- "accumulator" and "stack"

$first :: (a \to b) \to \forall z.(a \times z \to b \times z)$

$first\ f\ (a, z) = (f\ a, z)$

The essence of stack computation

$\forall z.(a \times z \to b \times z)$ -- "accumulator" and "stack"

$first :: (a \to b) \to \forall z.(a \times z \to b \times z)$

$first\ f\ (a, z) = (f\ a, z)$

Encapsulate in a new type:

newtype $StackFun\ a\ b = SF\ (\forall z.a \times z \to b \times z)$

$stackFun :: (a \to b) \to StackFun\ a\ b$
$stackFun\ f = SF\ (first\ f)$

Analogy (homomorphism)

Specification: *stackFun* defines a *precise (non-leaky) analogy*

- for a useful algebraic abstraction/vocabulary $\mathbb{L}$, i.e.,
- *stackFun* is an $\mathbb{L}$-homomorphism/analogy.

Identify a useful vocabulary

For functions and other function-like things:

$$\begin{aligned}&\textbf{class } \mathit{Category}\ (\rightsquigarrow)\ \textbf{where}\\ &\quad \mathit{id}\ \ :: a \rightsquigarrow a\\ &\quad (\circ) :: (b \rightsquigarrow c) \to (a \rightsquigarrow b) \to (a \rightsquigarrow c)\end{aligned}$$

Bonus: automatically translate from Haskell! (See *Compiling to categories.*)

Analogy/homomorphism properties ("functor"):

$$\mathit{id} = \mathit{stackFun}\ \mathit{id}$$

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

Solve each equation for its one unknown (LHS operation).

Identify a useful vocabulary

For functions and other function-like things:

$$\begin{aligned}&\textbf{class } \mathit{Category}\ k \textbf{ where}\\ &\quad \mathit{id} \ :: a\ `k`\ a\\ &\quad (\circ) :: (b\ `k`\ c) \to (a\ `k`\ b) \to (a\ `k`\ c)\end{aligned}$$

Bonus: automatically translate from Haskell! (See *Compiling to categories*.)

Analogy/homomorphism properties ("functor"):

$$\mathit{id} = \mathit{stackFun}\ \mathit{id}$$

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

Solve each equation for its one unknown (LHS operation).

Identity

Homomorphism equation:

$$id = stackFun\ id$$

Already in solved form.

Sequential composition

$$stackFun\ g \circ stackFun\ f = stackFun\ (g \circ f)$$

Simplify each side:

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

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

Equivalent specification:

$$SF\ (first\ g) \circ SF\ (first\ f) = SF\ (first\ g \circ first\ f)$$

Sequential composition

$$SF\ (first\ g) \circ SF\ (first\ f) = SF\ (first\ g \circ first\ f)$$

Generalize/strengthen:

$$SF\ g' \circ SF\ f' = SF\ (g' \circ f') \quad \text{-- Now in solved form.}$$

Sufficient definitions:

instance *Category StackFun* **where**
  $id = SF\ id$
  $SF\ g \circ SF\ f = SF\ (g \circ f)$

More easy operations

class *AssociativeCat* $(\rightsquigarrow)$ **where**
$rassoc :: ((a \times b) \times c) \rightsquigarrow (a \times (b \times c))$
$lassoc :: (a \times (b \times c)) \rightsquigarrow ((a \times b) \times c)$

class *BraidedCat* $(\rightsquigarrow)$ **where**
$swap :: (a \times b) \rightsquigarrow (b \times a)$

Homomorphisms already in solved form. Satisfy by definition:

instance *AssociativeCat StackFun* **where**
$rassoc = stackFun\ rassoc$
$lassoc = stackFun\ lassoc$

instance *BraidedCat StackFun* **where**
$swap = stackFun\ swap$

Parallel composition

$$\textbf{class } \mathit{Monoidal}_\times\ (\rightsquigarrow) \textbf{ where}$$
$$(\times) :: (a \rightsquigarrow c) \to (b \rightsquigarrow d) \to ((a \times b) \rightsquigarrow (c \times d))$$

Convenient specializations:

$$\mathit{first} :: \mathit{Monoidal}_\times\ (\rightsquigarrow) \Rightarrow (a \rightsquigarrow c) \to ((a \times b) \rightsquigarrow (c \times b))$$
$$\mathit{first}\ f = f \times \mathit{id}$$

$$\mathit{second} :: \mathit{Monoidal}_\times\ (\rightsquigarrow) \Rightarrow (b \rightsquigarrow d) \to ((a \times b) \rightsquigarrow (a \times d))$$
$$\mathit{second}\ g = \mathit{id} \times g$$

Focus on *first*, since

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

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

Homomorphism property for *first*

$$first\ (stackFun\ f) = stackFun\ (first\ f)$$

Inlining,

$$first\ (SF\ (first\ f)) = SF\ (first\ (first\ f))$$

Can we eliminate a *first*?

Homomorphism property for *first*

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

Inlining,

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

Can we eliminate a *first*?

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

Homomorphism property for *first*

$$first\ (first\ f) = lassoc \circ first\ f \circ rassoc$$

Accumulator/stack evolution:

$$\begin{array}{lll}
 & ((a, b) \quad , z) & \\
rassoc \longmapsto & (a \qquad , (b, z)) & \text{-- } push \\
first\ f \longmapsto^* & (f\ a \qquad , (b, z)) & \text{-- steps for } f \\
lassoc \longmapsto & ((f\ a, b), z) & \text{-- } pop
\end{array}$$

The stack's purpose revealed!

Homomorphism property for *first*

Now

$$first\ (SF\ (first\ f)) = SF\ (lassoc \circ first\ f \circ rassoc)$$

Strengthen/generalize:

$$first\ (SF\ f') = SF\ (lassoc \circ f' \circ rassoc) \quad \text{-- Now in solved form.}$$

Sufficient definition:

instance $Monoidal_\times\ StackFun$ **where**
$first\ (SF\ f) = SF\ (lassoc \circ f \circ rassoc)$
$second\ g = swap \circ first\ g \circ swap$
$f \times g = first\ f \circ second\ g$

Note right-to-left argument evaluation. For left-to-right, define $f \times g = second\ g \circ first\ f$.

Parallel composition

Inlining,

$$\mathit{stackFun}\ f \times \mathit{stackFun}\ g = \mathit{SF}\ (\mathit{lassoc} \circ \mathit{first}\ f \circ \mathit{rassoc} \circ \mathit{first}\ \mathit{swap} \circ \mathit{lassoc} \circ \mathit{first}\ g \circ \mathit{rassoc} \circ \mathit{first}\ \mathit{swap})$$

Accumulator/stack evolution:

$$\begin{array}{lll}
 & ((a,b) & ,z) \\
\mathit{first}\ \mathit{swap} \longmapsto & ((b,a) & ,z) \\
\mathit{rassoc} \longmapsto & (b & ,(a,z)) \\
\mathit{first}\ g \longmapsto^* & (g\ b & ,(a,z)) \quad \text{-- steps for } g \\
\mathit{lassoc} \longmapsto & ((g\ b,a) & ,z) \\
\mathit{first}\ \mathit{swap} \longmapsto & ((a,g\ b) & ,z) \\
\mathit{rassoc} \longmapsto & (a & ,(g\ b,z)) \\
\mathit{first}\ f \longmapsto^* & (f\ a & ,(g\ b,z)) \quad \text{-- steps for } f \\
\mathit{lassoc} \longmapsto & ((f\ a,g\ b),z) & \text{-- } = \mathit{first}\ (f \times g)\ ((a,b),z)
\end{array}$$

Parallel composition

$$stackFun\ f \times stackFun\ g = SF\ (lassoc \circ first\ f \circ rassoc \circ first\ swap \circ lassoc \circ first\ g \circ rassoc \circ first\ swap)$$

We've recursively flattened to *purely sequential* compositions of:

- *first* p for a few primitive functions p,
- *rassoc* and *lassoc* in balanced pairs ("push" and "pop").

Still more easy operations

$$\begin{aligned}&\textbf{class } \mathit{Monoidal}_{\times}\ (\rightsquigarrow) \Rightarrow \mathit{Cartesian}\ (\rightsquigarrow) \textbf{ where}\\ &\quad \mathit{exl} :: (a \times b) \rightsquigarrow a\\ &\quad \mathit{exr} :: (a \times b) \rightsquigarrow b\\ &\quad \mathit{dup} :: a \rightsquigarrow (a \times a)\end{aligned}$$

Homomorphisms are again in solved form.

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

Reifying stack computation

- Code generation needs inspection.
- Introduce *data* representation denoting stack functions.
- Specify by homomorphism; solve for implementation.

Primitive operations

```
data Prim :: * → * → * where    -- Notation
  Exl  :: Prim (a × b) a
  Exr  :: Prim (a × b) b
  Dup  :: Prim a (a × a)
  ...
  Negate :: Num a ⇒ Prim a a
  Add, Sub, Mul :: Num a ⇒ Prim (a × a) a
  ...

evalPrim :: Prim a b → (a → b)   -- Denotation
evalPrim Exl    = exl        -- fst
evalPrim Exr    = exr        -- snd
evalPrim Dup    = dup        -- dup
                 ...
evalPrim Negate = negateC    -- negate
evalPrim Add    = addC       -- uncurry (+)
                 ...
```

Stack operations

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

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

Stack programs

$$\textbf{data}\ \mathit{StackOps} :: * \to * \to * \ \textbf{where}$$
$$\quad \mathit{Nil} :: \mathit{StackOps}\ a\ a$$
$$\quad (\prec) :: \mathit{StackOp}\ a\ b \to \mathit{StackOps}\ b\ c \to \mathit{StackOps}\ a\ c$$

$$\mathit{evalStackOps} :: \mathit{StackOps}\ u\ v \to (u \to v)$$
$$\mathit{evalStackOps}\ \mathit{Nil} = \mathit{id}$$
$$\mathit{evalStackOps}\ (\mathit{op} \prec \mathit{ops}) = \mathit{evalStackOps}\ \mathit{ops} \circ \mathit{evalStackOp}\ \mathit{op}$$

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

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

Compiler specification: *progFun* is homomorphic.

Compiler solution

```
instance Category StackProg where
  id = SP Nil
  SP g ∘ SP f = SP (f ++ g)

instance Monoidal× StackProg where
  first (SP ops) = SP (Push :< ops ++ Pop :< Nil)
  second g = swap ∘ first g ∘ swap
  f × g = first f ∘ second g

primProg :: Prim a b → StackProg a b
primProg p = SP (Prim p :< Nil)

instance Cartesian StackProg where
  exl  = primProg Exl
  exr  = primProg Exr
  dup = primProg Dup
```

Example

Haskell:

$$\lambda(x, y) \rightarrow 2 * x + 3 * y$$

Standard algebraic translation:

$$\begin{aligned} & addC \\ \circ\ & (mulC \circ (const\ 2 \times exl) \circ dup \times mulC \circ (const\ 3 \times exr) \circ dup) \\ \circ\ & dup \end{aligned}$$

Stack program:

$$\begin{aligned} & [Dup, Push, Dup, Push, Const\ 2, Pop, Swap, Push, Exl, Pop \\ & , Swap, Mul, Pop, Swap, Push, Dup, Push, Const\ 3, Pop, Swap \\ & , Push, Exr, Pop, Swap, Mul, Pop, Swap, Add] \end{aligned}$$

Sum types and higher-order functions

- Sums ("coproducts") are dual to products: $(+), \mathit{inl}, \mathit{inr}, \mathit{jam}$.
- Higher-order functions ("exponentials"): $\mathit{curry}, \mathit{uncurry}, \mathit{apply}$.

Additional "primitives":

$$\begin{aligned}
&\textbf{data } \mathit{Prim} :: * \to * \to * \textbf{ where} \quad \text{-- Notation} \\
&\quad \ldots \\
&\quad (:+) :: \mathit{StackOps}\ a\ c \to \mathit{StackOps}\ b\ d \to \mathit{Prim}\ (a + b)\ (c + d) \\
&\quad \mathit{Apply} :: \mathit{Prim}\ ((a \to b) \times a)\ b \\
&\quad \mathit{Curry} :: \mathit{StackProg}\ (a \times b)\ c \to \mathit{Prim}\ a\ (b \to c)
\end{aligned}$$

Or relax the representation.

Compiling gracefully

- Identify simple essence of stack computation.
- Relate to regular functions.
- Identify common algebraic vocabulary.
- Reify stack computation.
- Solve standard homomorphism equations ⇒ compiler.
- Standard translation from Haskell to algebraic form.

To do

- Better generic optimization
- More ambitious machine models