Hardware Design with Generalized Arrows

Adam Megacz
megacz@cs.berkeley.edu

Computer Science Division, UC Berkeley

Abstract. Instances of the `GArrow` type class (Figure 2) are called *generalized arrows*. The `GArrow` class generalizes `Control.Arrow` by allowing any type-level monoid to take the place of the cartesian product `(,)` and by replacing `arr` with the "structural" functions usually defined in terms of it.

Multi-level terms with environment classifier types [TN03] may be *flattened* into single-level terms parameterized by an instance of the `GArrow` class. Multi-level terms and environment classifier types play the same role for generalized arrows that *Paterson notation* [Pat01] and its typing rules [PPJ] play for `Control.Arrow`.

This paper presents the first nontrivial application of generalized arrows. Previously, GHC had been extended[1] with environment classifiers and an additional compiler pass which implements the flattening transformation [Meg11]. In the present work this facility has been augmented to allow for programs in which level-0 terms consist of unrestricted Haskell, while level-1 terms are limited to a small κ-calculus [Has95] based language. The flattened, `GArrow`-parameterized term is then instantiated with the instance `GArrowVerilog`, which renders the term as a Verilog program, which is then synthesized and run on an FPGA.

The sample application presented here is a bit-serial circuit which searches for SHA-256 hash collisions. The circuit has been synthesized on a Xilinx Spartan-6 FPGA and functions correctly.

1 Introduction

1.1 Related Work

Many researchers have investigated the use of functional programming languages to describe hardware circuits [ACS05] [GMJ05] [JO'95] [LLIV00] [MCL98] [PKI08] [SM01] [SR95] [BCSS98] [GMJ05] [MCL98] [SM01] [SR95]. The allure is strong: combinational circuits and pure functions have much in common.

These efforts generally fall into two categories:

[1] `http://www.cs.berkeley.edu/~megacz/garrows/`

– In one approach, the Haskell program *is the circuit*; this is also called a "shallow embedding". This was the approach used in the first version of the original Lava [BCSS98]. This approach is very pleasant for users, since they simply reuse the binding, application, and abstraction mechanisms they are accustomed to from Haskell. However, in order to extract a graph from the Haskell program some sort of mechanism for observing sharing [CS99,Gil09] is required. There are several approaches to observing sharing in a one-level language, but generally they require either restricting the algebra of valid program transformations or else accepting some degree of nondeterminism or lack of precise semantics for the observation process.
– In the other approach, the Haskell program *builds the circuit*; this is also called a "deep embedding". This was used in the second version of the original Lava, which required that programs be written in value-recursive monadic style [EL00]. This avoids the pitfalls of observable sharing but requires that circuits be constructed using a totally different notation – for example, `mdo` must be used instead of Haskell's recursive `let..in`.

The present paper experiments with a solution which does both of these at the same time: the program both *is the circuit* and *builds the circuit*, yet is not affected by the issues that arise from letting a program observe its own sharing structure. This is accomplished through the use of a *two-level language* which enforces stratification of the levels in the type system. The language is an extension of Haskell with code terms and code types with environment classifiers [TN03]. This extension supports *heterogeneous metaprogramming*, which is to say that it does not assume that the type systems of the two levels are the same, nor that one is a subset of the other. The level-1 language – which is meant to represent circuits – is based on κ-calculus, a first-order analogue of λ-calculus.

2 κ-calculus

λ-calculus allows functions of *higher type*; that is, terms of type $(\tau \to \tau) \to \tau$. When the need arises to restrict the use of such functions, the most straightforward approach is to enforce the separation syntactically in the grammar of the types:

$$\sigma ::= \textbf{bool} \mid \textbf{int} \mid \ldots \qquad \textit{(ground types)}$$

$$\tau ::= \sigma \mid \sigma \to \tau \qquad \textit{(first-order types)}$$

Although effective, this approach does not scale well. Consider adding polymorphism: two syntactical categories of type variables are required (one for ground types and one for first-order types). This in turn requires two syntactic quantifier forms, and subsequently two different kinds for polymorphic terms; the duplication of effort grows rapidly.

Hasegawa's κ-calculus [Has95] provides a more manageable approach, motivated as a *syntax for morphisms in a contextually-closed category*; each expression of

κ-calculus inhabits a hom-set of the category. The following grammar is taken from [Has95, Section 3].

$$\tau ::= 1 \mid \tau \otimes \tau \mid \ldots \qquad \textit{(types)}$$

$$\Gamma ::= \cdot \mid x{:}\tau \rightsquigarrow \tau, \Gamma \qquad \textit{(contexts)}$$

$$e ::= x \mid \mathsf{lift}_A(e) \mid \kappa x{:}1 \rightsquigarrow A.e \mid \mathsf{id}_A \mid e \circ e \qquad \textit{(expressions)}$$

$$J ::= \Gamma \vdash e : \tau \rightsquigarrow \tau \qquad \textit{(judgments)}$$

Note that each variable in the context is assigned a *pair* of types, as is the expression in the succedent of a judgment. In the "first order λ-calculus" above, a function taking two arguments of types A and B and yielding a result of type C has the type $A \to (B \to C)$, whereas in κ-calculus it has the pair of types $A \otimes B \rightsquigarrow C$. The key difference to note here is the use of two distinct operators ($\otimes$ and $\rightsquigarrow$) rather than one ($\to$).

Also of note is the form of the type annotation on κ-abstraction: one may only abstract over terms whose type is of the form $1 \rightsquigarrow \tau$. This is how the "first order" nature of κ-calculus is enforced. If polymorphism is added, restrictions are enforced at the site of *use* rather than the site of *binding*.

In Hasegawa's presentation the underlying category's monoidal structure is assumed to be strict; as a consequence, the following type equalities hold in that presentation:

$$\begin{aligned} 1 \otimes \tau = {} & \tau = \tau \otimes 1 \\ \tau_1 \otimes (\tau_2 \otimes \tau_3) = {} & (\tau_1 \otimes \tau_2) \otimes \tau_3 \end{aligned}$$

In this presentation the equalities above are assumed only to be *isomorphisms* and will be invoked explicitly.

Note that there is no syntax for application, only for composition and "lift". The typing rules for these are given below:

$$\frac{}{\vdash \mathsf{id}_\tau : \tau \rightsquigarrow \tau}\,\mathrm{Id} \qquad \frac{}{x : \tau \rightsquigarrow \tau' \vdash x : \tau \rightsquigarrow \tau'}\,\mathrm{Var}$$

$$\frac{\Gamma \vdash e : 1 \rightsquigarrow \tau}{\Gamma \vdash \mathsf{lift}_{\tau'}(e) : \tau' \rightsquigarrow \tau \otimes \tau'}\,\mathrm{Lift}$$

$$\frac{\Gamma \vdash e_1{:}\tau_1 \rightsquigarrow \tau_2 \qquad \Gamma \vdash e_2{:}\tau_2 \rightsquigarrow \tau_3}{\Gamma \vdash e_2 \circ e_1 : \tau_1 \rightsquigarrow \tau_3}\,\mathrm{Comp}$$

$$\frac{\Gamma, x{:}1 \rightsquigarrow \tau_1 \vdash e{:}\tau_2 \rightsquigarrow \tau_3}{\Gamma \vdash \kappa x{:}1 \rightsquigarrow \tau_1\ .\ e\ : \tau_1 \otimes \tau_2 \rightsquigarrow \tau_3}\,\mathrm{Kappa}$$

Although this form most closely matches the category-theoretic foundations, it is more convenient to write programs using application, which may be defined as:

$$e_1 e_2 \overset{def}{=} e_1 \circ \mathsf{lift}(e_2)$$

The typing rule for this abbreviation is derivable:

$$\dfrac{\Gamma \vdash e_1 : \tau_1 \otimes \tau_2 \rightsquigarrow \tau_3 \qquad \dfrac{\dfrac{\Gamma \vdash e_2 : 1 \rightsquigarrow \tau_1}{\Gamma \vdash \mathsf{lift}_{\tau_2}(e_2) : 1 \otimes \tau_2 \rightsquigarrow \tau_1 \otimes \tau_2}\text{Lift}}{\Gamma \vdash \mathsf{lift}_{\tau_2}(e_2) : \tau_2 \rightsquigarrow \tau_1 \otimes \tau_2}\cong}{\Gamma \vdash e_1 \circ \mathsf{lift}_{\tau_2}(e_2) : \tau_2 \rightsquigarrow \tau_3}\text{Comp}$$

So we have:

$$\dfrac{\Gamma \vdash e_1 : \tau_1 \otimes \tau_2 \rightsquigarrow \tau_3 \qquad \Gamma \vdash e_2 : 1 \rightsquigarrow \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2 \rightsquigarrow \tau_3}\text{KappaApp}$$

In practical use, the part of a type to the left of the $\rightsquigarrow$ can be thought of as a list of arguments, represented as a "right imbalanced" tree terminated by 1. For example, a function taking exactly three arguments of types A, B, and C and returning a result of type D would have the type

$$A \otimes (B \otimes (C \otimes 1)) \rightsquigarrow D$$

We will adopt the convention that $\otimes$ is right associative, and elide the parentheses:

$$A \otimes B \otimes C \otimes 1 \rightsquigarrow D$$

The 1 is still necessary as an indication that C is the *third and final argument* rather than *a list of all but the first two arguments*; this is similar to how the Haskell patterns `a:b:c:[]` and `a:b:c` give the identifier `c` different types. This distinction is important, since it shows how κ-calculus types $A \otimes B \otimes C \rightsquigarrow D$ differ from Haskell's "uncurried" function types `(A,B,C)->D`. In κ-calculus a function may be applied without knowing its arity – just as with curried functions in λ-calculus *but without the use of higher types*. For example, the following judgment is derivable for any α, and therefore for f a function of any arity.

$$f : A \otimes \alpha \rightsquigarrow B, e : 1 \rightsquigarrow A \vdash fe : \alpha \rightsquigarrow B$$

We extend the κ-calculus with an additional expression form **letrec** not found in Hasegawa's work:

$$\frac{\begin{array}{c}\Gamma, x{:}1 \rightsquigarrow A \vdash e_x : 1 \rightsquigarrow A \\ \Gamma, x{:}1 \rightsquigarrow A \vdash e : B \rightsquigarrow C\end{array}}{\Gamma \vdash \textbf{letrec}\ x = e_x\ \textbf{in}\ e : B \rightsquigarrow C}\text{LetRec}$$

3 A Two-Level $\boldsymbol{\lambda}$-$\boldsymbol{\kappa}$-calculus

We now proceed to embed κ-calculus within Haskell. First, the expressions of Haskell are extended with the usual "bracket" and "escape" operators of multi-level languages [NN92]:

e_0 ::= $\ldots \mid \langle\!\!\{ e_1 \rangle\!\!\}$ *(level-0 expressions)*

e_1 ::= $\kappa x{:}1 \rightsquigarrow \tau\ .\ e_1 \mid x \mid e_1 e_1 \mid$ ~~e_0 *(level-1 expressions)*

The grammar for types is extended with a code type, indexed by an environment classifier [TN03] (which is a type variable) and a pair of types:

τ ::= $\ldots \mid \langle\!\!\{ \tau \rightsquigarrow \tau \rangle\!\!\}@\alpha$ *(types)*

Although λ-application and κ-application are completely distinct nodes in the parsed abstract syntax tree, it is cumbersome to have to use an actual operator – rather than simple juxtaposition – for the latter. Therefore the parser measures the number of code brackets enclosing a subexpression, subtracts from it the number of escapes, and if the result is nonzero it treats occurrences of juxtaposition as κ-application. The syntax for λ-abstraction is overloaded similarly for κ-abstraction.

Here is a simple example program showing κ-application inside brackets:

```
applyBrak :: <[ (a,b) ~~> c ]>@d ->
             <[    () ~~> a ]>@d ->
             <[    b  ~~> c ]>@d
applyBrak x y = <[ ~~x ~~y ]>
```

To illustrate κ-abstraction, here is an example functional which reverses the order of the first two arguments of a function. Notice that this works for functions of any arity greater than one:

```
swap :: <[ (a,(b,c)) ~~> d ]>@e ->
        <[ (b,(a,c)) ~~> d ]>@e
swap f = <[ \x y -> ~~f y x ]>
```

The feature which distinguishes κ-abstraction from λ-abstraction is the inability to abstract over functions. For example, consider an attempt to write the Haskell `apply` function inside the brackets:

```
bad = <[ \f x -> f x ]>
```

This program is rejected by the typechecker:

```
Demo.hs:12:22:
    Couldn't match expected type `(t0, t1)'
       with actual type `()'
    Expected type: (t0, t1)~~>t3
      Actual type: ()~~>t2
    In the expression: f x
    In the expression: \ x -> f x
```

Since `f` is brought into scope by a κ-abstraction, the typechecker concludes from rule [Kappa] that `f` has type `()~~>t` for some type `t`. When it encounters the application `f x` it uses rule [KappaApp], attempting to unify `()~~>t` with `(a,b)~~>c`; this unification fails.

4 Generalized Arrows

```
sample2 =
  ga_uncancelr                >>>
  ga_first ga_copy            >>>
  ga_swap                     >>>
  ga_second (ga_first ga_drop >>>
             ga_cancell)      >>>
  ga_cancell
```

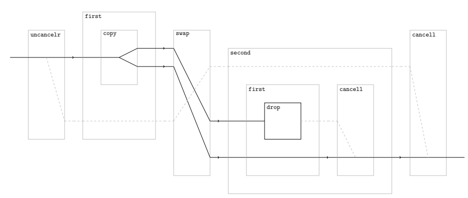


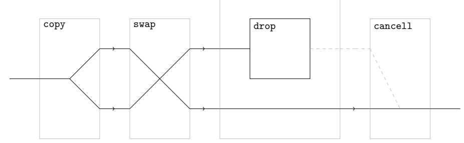
Fig. 1. A sample `GArrow` term and its visualization as a Penrose diagram.

Once a level-1 κ-calculus term has been produced, it is necessary to compile it. The multi-level terms introduced in the previous section are not executed directly; the back end of the compiler has not been extended to produce code for them. Instead, these terms are *flattened* into ordinary Haskell terms in which terms parameterized by an instance of the `GArrow` class take the place of level-1 terms. The definition of the `GArrow` class is shown in Figure 2.

As partial justification for the name *generalized arrow*, Figure 3 shows an instance declaration making any `Control.Arrow` (including `(->)`) a `GArrow`.

It will be convenient to visualize `GArrow` terms as Penrose diagrams [Sel09]. In fact, the `GArrow` instance `GArrowTikZ` does exactly this, emitting TikZ code to produce the diagram. For example, the following term:

```
sample1 =
  ga_copy          >>>
  ga_swap          >>>
  ga_first ga_drop >>>
  ga_cancell
```



A larger example can be found in Figure 1.

```
class Category g => GArrow g (**) u where
--id          :: g x x
--(>>>)       :: g x y -> g y z -> g x z
  ga_first     :: g x y -> g (x ** z) (y ** z)
  ga_second    :: g x y -> g (z ** x) (z ** y)
  ga_cancell   :: g (u**x)          x
  ga_cancelr   :: g    (x**u)       x
  ga_uncancell :: g     x       (u**x)
  ga_uncancelr :: g     x          (x**u)
  ga_assoc     :: g ((x** y)**z ) ( x**(y **z))
  ga_unassoc   :: g ( x**(y **z)) ((x** y)**z )

class GArrow g (**) u => GArrowCopy g (**) u where
  ga_copy      :: g x (x**x)

class GArrow g (**) u => GArrowDrop g (**) u where
  ga_drop      :: g x u

class GArrow g (**) u => GArrowSwap g (**) u where
  ga_swap      :: g (x**y) (y**x)

class GArrow g (**) u => GArrowLoop g (**) u where
  ga_loopr    :: g (x**z) (y**z) -> g x y
  ga_loopl    :: g (z**x) (z**y) -> g x y
```

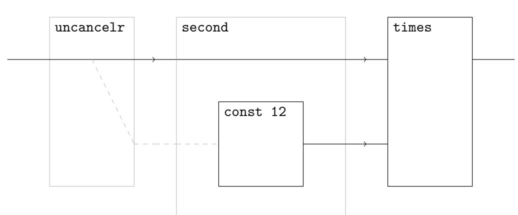
Fig. 2. The definition for the `GArrow` type class and its four most frequently implemented subclasses. The class `Category` comes from the standard `Control.Category` module.

5 Flattening

Having described the modifications to the syntax and type system, we now show a few examples of the flattening procedure (see [Meg11] for complete details on the algorithm).

Here is a simple example of a flattened term; the arguments `const` and `times` are "black boxes":

```
demo const times =
   <[ \y -> ~~times y ~~(const 12) ]>
```

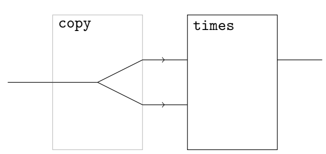


```
instance Arrow a => GArrow a (,) () where
  ga_first     =  first
  ga_second    =  second
  ga_cancell   =  arr (\((),x) -> x)
  ga_cancelr   =  arr (\(x,()) -> x)
  ga_uncancell =  arr (\x -> ((),x))
  ga_uncancelr =  arr (\x -> (x,()))
  ga_assoc     =  arr (\((x,y),z) -> (x,(y,z)))
  ga_unassoc   =  arr (\(x,(y,z)) -> ((x,y),z))

instance Arrow a => GArrowDrop a (,) () where
  ga_drop      =  arr (\x -> ())

instance Arrow a => GArrowCopy a (,) () where
  ga_copy      =  arr (\x -> (x,x))

instance Arrow a => GArrowSwap a (,) () where
  ga_swap      =  arr (\(x,y) -> (y,x))

instance ArrowLoop a => GArrowLoop a (,) () where
  ga_loopr     =  loop
  ga_loopl  f  =  loop (ga_swap >>> f >>> ga_swap)
```

Fig. 3. Instance declaration showing that every `Control.Arrow` is a generalized arrow.

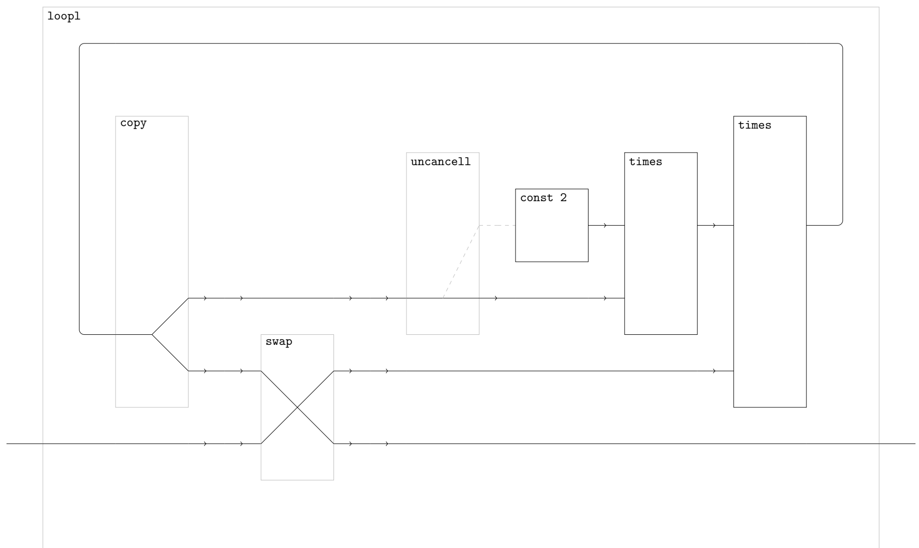
When an identifier appears more than once, the structural rule of *contraction* will appear in the term's proof tree. This is realized by the `ga_copy` method as shown in the following example:

```
demo const times =
  <[ \y -> ~~times y y ]>
```



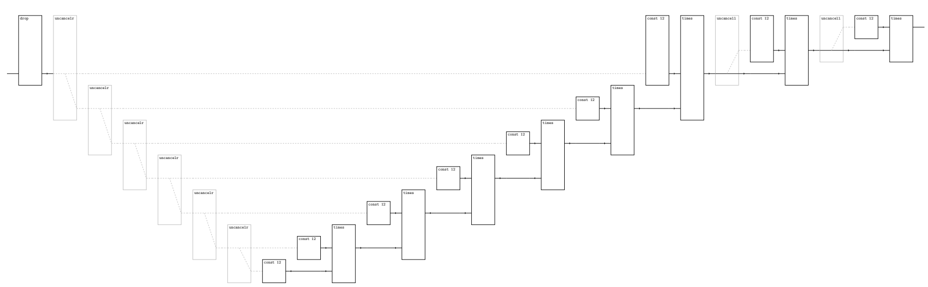
Finally, it is important to note that recursion *outside the code brackets* represents repetitive structures, whereas recursion *inside the brackets* represents feedback loops. This is illustrated by the following two examples; the first shows recursion *inside the brackets*, which produces feedback:

```
demo const times =
   <[ \x ->
      let out   = ~~times (~~times ~~(const 2) out) x
       in out
    ]>
```



The following example demonstrates recursion *outside the brackets*, which produces repetitive structures:

```
-- ‘‘pow n x’’ computes x^n
pow 0 x = const (1::Int)
pow 1 x = const x
pow n x = <[ ~~times
                 ~~(pow 1 x)
                 ~~(pow (n-1) x) ]>

demo const times =
  <[ \y -> ~~(pow 9 12) ]>
```



This distinction between recursion inside the brackets and recursion outside the brackets is closely related to monadic *value recursion* [EL00]. In fact, a term which uses recursion (LetRec) inside the brackets will be flattened to a `GArrow` term which relies on `ga_loop`. This term may then be instantiated for any `MonadFix`, since `Control.Arrow.Kleisli` provides a `ArrowLoop` instance for any `MonadFix`, and Figure 3 provides a `GArrowLoop` instance for any `ArrowLoop`;

when instantiated in this manner, `ga_loop` will be realized as `mfix`. By contrast, recursion outside the brackets will not be altered by the flattener aside from an adjustment to its type.

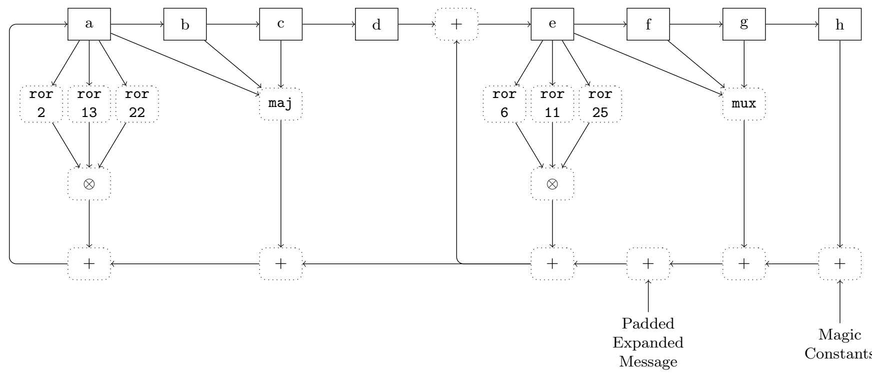


Fig. 4. The SHA-256 Algorithm. Each solid rectangle is a 32-bit state variable; the path into each rectangle computes its value in the next round based on the values of the state variables in the previous round. The standard specifies initialization values for the state variables prior to the first message block. The $\otimes$ symbol is bitwise xor, the $+$ symbol is addition modulo 2^{32}, `ror` is bitwise right rotation, `maj` is bitwise majority, and `mux` is bitwise mux (`e[i]?f[i]:g[i]`). For each block of the message the algorithm above is iterated for 64 rounds; the values in the eight state registers afterwards are added to the values they held before the block before starting the next block. The hash of a message consists of the concatenation of the values in the eight state variables after the last block has been processed.

6 Specifying Hardware

6.1 Primitives

The SHA-256 engine is defined in terms of the primitives shown in Figure 5, which appear as opaque elements in Haskell. Each of the primitives was manually implemented in Verilog; Haskell is essentially used as a language for connecting them.

The first two primitives provide a constant logic zero and one. The next six primitives are basic combinational logic elements, and the seventh element is a simple register (the design assumes only a single global clock).

The `loop` element outputs a repeating sequence of bits (which is fixed at design time). The `fifo` element is a simple one-bit first-in-first-out queue.

The `oracle` is much like `loop`, except that the value being repeated can be modified remotely from outside the FPGA using the device's JTAG connection. This same JTAG connection can be used to query the value of any `probe`. Each takes an `Int` argument which is used as an "address" to identify the `probe` or `oracle` within the running design.

```
class BitSerialHardwarePrimitives g where
  type Wire

  high    :: <[                  ()    ~~> Wire ]>@g
  low     :: <[                  ()    ~~> Wire ]>@g

  not     :: <[             Wire,()    ~~> Wire ]>@g
  xor     :: <[       Wire,(Wire,())   ~~> Wire ]>@g
  or      :: <[       Wire,(Wire,())   ~~> Wire ]>@g
  and     :: <[       Wire,(Wire,())   ~~> Wire ]>@g
  mux2    :: <[ Wire,(Wire,(Wire,()))  ~~> Wire ]>@g
  maj3    :: <[ Wire,(Wire,(Wire,()))  ~~> Wire ]>@g
  reg     :: <[             Wire,()    ~~> Wire ]>@g

  loop    ::       [Bool] -> <[    ()  ~~> Wire ]>@g
  fifo    ::       Int -> <[  Wire,()  ~~> Wire ]>@g

  probe   ::       Int -> <[ Wire,()   ~~> Wire ]>@g
  oracle  ::       Int -> <[      ()   ~~> Wire ]>@g
```

Fig. 5. Type class containing the primitives needed for the SHA-256 circuit

There are a few basic subcircuits to build before assembling the SHA-256 hashing engine. First, we define a three-input `xor` gate in the obvious manner:

```
xor3 = <[ \x y z -> xor (xor x y) z ]>
```

Using this, we are now able to write code for a bit-serial adder. The `firstBit` produces a repeating pattern of 32 bits, the first of which is a one; this signal is used to clear the internal carry-bit state (`carry_out`).

```
adder =
  <[ \in1 in2 ->
     let firstBit  = ~~(loop [ i/=0 | i<-[0..31] ])
         carry_out = reg (mux2 firstBit
                               zero
                               carry_in)
         carry_in  = maj3 carry_out in1 in2
     in  xor3 carry_out in1 in2
   ]>
```

Finally, the circuit below performs a bitwise right-rotation. Since the circuit is bit-serial, it has a latency of 32 bits.

```
rotRight n =
  <[ \input ->
     let sel   = ~~(loop [ i >= 32-n | i<-[0..31] ])
         fifo1 = ~~(fifo (32-n)) input
         fifo2 = ~~(fifo  32   ) fifo1
     in  mux2 sel fifo1 fifo2
   ]>
```

```
sha256round =
  <[ \load input k_plus_w ->
     let a    = ~~(fifo 32) (mux2 load a_in input)
         b    = ~~(fifo 32) a
         c    = ~~(fifo 32) b
         d    = ~~(fifo 32) c
         e    = ~~(fifo 32) (mux2 load e_in d)
         f    = ~~(fifo 32) e
         g    = ~~(fifo 32) f
         h    = ~~(fifo 32) g
         s0   = xor3
                    (~~(rotRight  2) a_in)
                    (~~(rotRight 13) a_in)
                    (~~(rotRight 22) a_in)
         s1   = xor3
                    (~~(rotRight  6) e_in)
                    (~~(rotRight 11) e_in)
                    (~~(rotRight 25) e_in)
         a_in = adder t1 t2
         e_in = adder t1 d
         t1   = adder
                   (adder h s1)
                   (adder (mux2 e g f)
                          k_plus_w)
         t2   = adder s0 (maj3 a b c)
     in h
   ]>
```

Fig. 6. Core algorithm for one pass of SHA-256. The input `k_plus_w` is a wire input carrying the sum of the SHA-256 constant table entry and (K) and the message being hashed (W). The `load` input switches the circuit between computation mode and loading mode; when in loading mode the eight state registers form one long shift register; a new state can be shifted in via `input` and the old state shifted out via the circuit's (sole) output.

Using these subcircuits, it is now possible to express the SHA-256 algorithm, which can be found in Figure 4.

Figure 6 shows the implementation of the core of the SHA-256 algorithm. The circuit is initialized by holding `load` high for 8×32 cycles while shifting in the initial hash state on the `input` wire. The 64 rounds of the SHA-256 algorithm are then performed by holding `load` low and waiting for 64×32 clocks. Finally the result is read out by holding `load` high and monitoring the circuit's output for the following 8×32 clocks.

Here is the type inferred by GHC for `sha256round`:

```
$ inplace/bin/ghc-stage2 SHA256.hs

TYPE SIGNATURES
 sha256round ::
    forall (t :: * -> * -> *) a.
       (Num a, BitSerialHardwarePrimitives t) =>
       (a -> <[(Wire, ())~~>Wire]>@t)
       -> <[(Wire, (Wire, (Wire, ())))~~>Wire]>@t
```

7 Implementation

One caveat should be noted: the Haskell code in the current implementation emits not Verilog, but a graph in text form whose nodes are the primitives above. This graph is then read in by a separate Java program, which emits the actual Verilog and supervises the execution of the synthesis tools. In principle there is no reason why the Java code could not be rewritten in Haskell (it was inherited from an earlier project and works quite well).

The syntax leaves a bit to be desired. Although the mathematical notation assumes right-associativity for $\otimes$, the Haskell parser interprets `(x,y,z)` as a triple (distinct from a pair whose second coordinate is a pair). The low-precedence application operator `($)` is unavailable inside code brackets because it is an ordinary Haskell function (not a language primitive) with a higher-order type. However, it is so effective at eliminating parentheses that it might be worth including it in the grammar.

8 Conclusion and Future Work

Generalized arrows make the structural laws (weakening, exchange, contraction, and associativity) of a typing proof explicit in the resulting generalized arrow term. Consequently, syntactical properties like variable order are retained and may be exploited. In the context of hardware design, this may lead to a strategy for conveniently specifying *relative location* (RLOC) constraints [Sin11].

References

[ACS05] E Axelsson, K Claessen, and M Sheeran. Wired: Wire-aware circuit design. *LECTURE NOTES IN COMPUTER SCIENCE*, Jan 2005.

[BCSS98] P Bjesse, K Claessen, M Sheeran, and S Singh. Lava: hardware design in haskell. *Proceedings of the third ACM SIGPLAN international ...*, Jan 1998.

[CS99] Claessen and Sands. Observable sharing for functional circuit description. 1999.

[EL00] Levent Erkök and John Launchbury. Recursive monadic bindings. pages 174–185, 2000.

[Gil09] Andy Gill. Type-safe observable sharing in haskell. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 117–128. ACM, 2009.

[GMJ05] J Grundy, T Melham, and JO'leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, Jan 2005.

[Has95] M Hasegawa. Decomposing typed lambda calculus into a couple of categorical programming languages. *Lecture Notes in Computer Science*, 953, 1995.

[JO'95] JO'Donnell. From transistors to computer architecture: Teaching functional circuit specification in hydra. *Functional Programming Languages in Education: First ...*, Jan 1995.

[LLIV00] Y Li, M Leeser, S Inc, and M View. Hml, a novel hardware description language and its translation tovhdl. *IEEE Transactions on Very Large Scale Integration (VLSI) ...*, Jan 2000.

[MCL98] J Matthews, B Cook, and J Launchbury. Microprocessor specification in hawk. *Computer Languages*, Jan 1998.

[Meg11] Adam Megacz. Multi-level languages are generalized arrows. *submitted, preprint available on arxiv.org*, http://arxiv.org/abs/1007.2885, 2011.

[NN92] F. Nielson and H. R. Neilson. *Two-Level Functional Languages.* Cambridge University Press, Cambridge, Mass., 1992.

[Pat01] Ross Paterson. A new notation for arrows. In *ICFP*, pages 229–240, 2001.

[PKI08] S Park, J Kim, and H Im. Functional netlists. *portal.acm.org*, Jan 2008.

[PPJ] Ross Paterson and Simon Peyton Jones. Type and translation rules for arrow notation in ghc.

[Sel09] Peter Selinger. A survey of graphical languages for monoidal categories, August 23 2009.

[Sin11] Satnam Singh. The RLOC is dead - long live the RLOC. In John Wawrzynek and Katherine Compton, editors, *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*, pages 185–188. ACM, 2011.

[SM01] R Sharp and A Mycroft. A higher-level language for hardware synthesis. *LECTURE NOTES IN COMPUTER SCIENCE*, Jan 2001.

[SR95] R Sharp and O Rasmussen. Using a language of functions and relations for vlsi specification. *Proceedings of the seventh international conference on . . .*, Jan 1995.

[TN03] Walid Taha and Michael Florentin Nielsen. Environment classifiers. pages 26–37, 2003.