A New Notation for Arrows

Ross Paterson
Department of Computing, City University, London

ABSTRACT

The categorical notion of monad, used by Moggi to structure denotational descriptions, has proved to be a powerful tool for structuring combinator libraries. Moreover, the monadic programming style provides a convenient syntax for many kinds of computation, so that each library defines a new sublanguage.

Recently, several workers have proposed a generalization of monads, called variously "arrows" or Freyd-categories. The extra generality promises to increase the power, expressiveness and efficiency of the embedded approach, but does not mesh as well with the native abstraction and application. Definitions are typically given in a point-free style, which is useful for proving general properties, but can be awkward for programming specific instances.

In this paper we define a simple extension to the functional language Haskell that makes these new notions of computation more convenient to use. Our language is similar to the monadic style, and has similar reasoning properties. Moreover, it is extensible, in the sense that new combining forms can be defined as expressions in the host language.

1. INTRODUCTION

A useful method for implementing of domain-specific languages (DSLs) is to embed them in a general-purpose language. Functional languages are particularly suitable, as originally noted by Landin [19] and widely exploited since. Hudak [11] gives a recent account.

Many of these libraries or sublanguages have a common structure; they involve a monad, a categorical structure that Moggi applied to the structuring of denotational descriptions [22] and Wadler subsequently applied to functional programming [30]. Much useful code can be written to the monad abstraction, and is thus useful with each such library.

In the monad-based view of computation [22], we move from expressions yielding values of type A to computations of type $M\ A$, where M is a functor with certain operations. A simple example of the monadic style of programming is the following Haskell [24] function that adds the results of two computations:

$$\begin{aligned} &addM :: Monad\ m \Rightarrow m\ Int \to m\ Int \to m\ Int \\ &u\ `addM`\ v = u \gg\!= \lambda x \to \\ &\qquad\qquad v \gg\!= \lambda y \to \\ &\qquad\qquad return\ (x + y) \end{aligned}$$

Though the setting is more general, variables like x and y still denote values, and may be bound using Haskell's λ-abstraction.

A neat example of a monad-based library is provided by recursive-descent parsers [13]. One can write a set of mutually recursive computations that closely mirror the original grammar, subject to the usual limitations of top-down parsing. However such parsers have a major flaw: they are typically designed to backtrack on error, which is both inefficient and makes useful error reporting very difficult. The deterministic parsing library designed by Swierstra and Duponcheel [28] solves these problems, but steps beyond the world of monads. The reason is the same as the source of the efficiency of this technique: the parser has a static component that is independent of the input, and this would be lost in any definition of the $\gg\!=$ combinator. Moreover, as they noted, this optimization technique is applicable in many other contexts, but the resulting libraries would not be monadic.

Hughes [12] showed that monads could be generalized to "arrows" relating inputs and outputs. Workers in denotational semantics have proposed similar frameworks [4, 26, 27]. Such arrows may represent "procedures" that have a static component independent of the input, or other kinds of procedure that accept multiple inputs, as well as monadic computations. The added generality is useful, but comes at a cost: since procedures are no longer functions, they cannot be manipulated using the abstraction and application features of the underlying language. One can use a point-free style resembling category theory, which is very convenient for proving general properties, but can be awkward for programming specific instances.

The contribution of this paper is to define a convenient notation for computation corresponding to these semantic notions, designed as an extension to the functional language Haskell [24]. Although arrows cannot in general be factored as functions, we are able to define limited forms of application and abstraction, as well as a notion of *control operator* for combining arrow-based computations. We also extend a notion of feedback to arrows to support recursion. The new constructs are defined by translation to standard Haskell.

The rest of the paper is organized as follows. In the next

ICFP'01, September 3–5, 2001, Florence, Italy.

$$
\begin{aligned}
arr\ id \ggg f &= f \\
f \ggg arr\ id &= f \\
(f \ggg g) \ggg h &= f \ggg (g \ggg h) \\
arr\ (g \cdot f) &= arr\ f \ggg arr\ g \\
first\ (arr\ f) &= arr\ (f \times id) \\
first\ (f \ggg g) &= first\ f \ggg first\ g \\
first\ f \ggg arr\ (id \times g) &= arr\ (id \times g) \ggg first\ f \\
first\ f \ggg arr\ fst &= arr\ fst \ggg f \\
first\ (first\ f) \ggg arr\ assoc &= arr\ assoc \ggg first\ f
\end{aligned}
$$

Figure 1: Arrow equations

section we briefly review Hughes's arrows. Section 3 presents our proposed extension to Haskell, illustrated using an example from [12]. A larger example, an embedded language for regular data parallel algorithms, is described in Section 4. In Section 5 we consider how arrows may be extended to allow recursive definition of values, and similarly extend our syntax. This extension is applied in our final example, an embedded language for circuit description, in Section 6.

Although our focus is on programming, many of the concepts here are inspired by category theory. Short discussions of the connections are given in subsections entitled *Theoretical Aside*. These may be useful to readers with the appropriate theoretical background, but they are not essential to the main development.

The extension to Haskell described here has been implemented by a preprocessor that produces Haskell 98. The preprocessor is itself written in Haskell 98, as an extension of a Haskell parser and pretty printer written by Sven Panne, Simon Marlow and Keith Wansborough. This paper was formatted from a literate script which has also been fed to the preprocessor, and thence to Haskell implementations.

2. ARROWS

We briefly recall Hughes's definitions from [12].

Definition 1. An *arrow type* is a binary type constructor a with the following data:

```
class Arrow a where
    arr :: (b → c) → a b c
    (>>>) :: a b c → a c d → a b d
    first :: a b c → a (b, d) (c, d)
```

satisfying the equations of Figure 1. The functions $(\times)$ and *assoc* used there are defined as follows:

```
(×) :: (a → a') → (b → b') → (a, b) → (a', b')
(f × g) (a, b) = (f a, g b)

assoc :: ((a, b), c) → (a, (b, c))
assoc ((a, b), c) = (a, (b, c))
```

There is no need to require a *second* function, as it is defined in terms of *first*:

```
second :: Arrow a ⇒ a b c → a (d, b) (d, c)
second f = arr swap >>> first f >>> arr swap

swap :: (a, b) → (b, a)
swap~(x, y) = (y, x)
```

The following definitions will also be useful:

```
(***) :: Arrow a ⇒ a b c → a b' c' → a (b, b') (c, c')
f *** g = first f >>> second g

(&&&) :: Arrow a ⇒ a b c → a b c' → a b (c, c')
f &&& g = arr (λb → (b, b)) >>> f *** g
```

Note that *** does not in general preserve composition; for example, the order in which effects occur is significant.

Ordinary functions are a special case

```
instance Arrow (→) where
    arr f = f
    f >>> g = g · f
    first f = f × id
```

The Kleisli arrows of a monad may also be cast as an arrow type.

```
newtype Kleisli m a b = K (a → m b)

instance Monad m ⇒ Arrow (Kleisli m) where
    arr f = K (return · f)
    K f >>> K g = K (λb → f b >>= g)
    first (K f) = K (λ(b, d) → f b >>= λc →
                        return (c, d))
```

However, there are other important examples, as we shall see later.

Some arrow types have additional constants. Hughes gave a class specifying an application operator

```
class Arrow a ⇒ ArrowApply a where
    app :: a (a b c, b) c
```

which is required to satisfy certain conditions [12]. The trivial arrow type → and Kleisli arrow types satisfy these conditions, and indeed any such arrow type is equivalent to a Kleisli arrow type [12, 27].

Hughes also defined structures on sum types dualizing those on product types:

```
class Arrow a ⇒ ArrowChoice a where
    left :: a b c → a (Either b d) (Either c d)

right :: ArrowChoice a ⇒
    a b c → a (Either d b) (Either d c)
right f = arr mirror >>> left f >>> arr mirror
            where mirror (Left x) = Right x
                  mirror (Right y) = Left y

(+++) :: ArrowChoice a ⇒
    a b c → a b' c' → a (Either b b') (Either c c')
f +++ g = left f >>> right g

(|||) :: ArrowChoice a ⇒
    a b d → a c d → a (Either b c) d
f ||| g = f +++ g >>> arr untag
            where untag (Left x) = x
                  untag (Right y) = y
```

As an illustration of the programming style used with arrows, here is an arrow operation corresponding to *addM* from the previous section:

```
addA :: Arrow a ⇒ a b Int → a b Int → a b Int
addA f g = f &&& g >>> arr (λ(x, y) → x + y)
```

2.1 Theoretical Aside

Similar structures have been independently proposed by workers in denotational semantics. We give here a simplified (but equivalent) version of a definition of Power and Thielecke [27].

Definition 2. A *Freyd-category* consists of

- a category $\mathcal{V}$ with finite products (the value category),
- a category $\mathcal{C}$ with the same objects as $\mathcal{V}$ (the computation category),
- a functor $\mathsf{inc} : \mathcal{V} \to \mathcal{C}$ that is the identity on objects,
- a functor $\ltimes : \mathcal{C} \times \mathcal{V} \to \mathcal{C}$ such that

$$\mathsf{inc}\ x \ltimes y \quad = \quad \mathsf{inc}\ (x \times y)$$

and the following natural isomorphisms in $\mathcal{V}$

$$\begin{aligned} \mathsf{assoc}_\times &: (A \times B) \times C \cong A \times (B \times C) \\ \mathsf{unitr}_\times &: A \times 1 \cong A \end{aligned}$$

extend to natural isomorphisms in $\mathcal{C}$:

$$\begin{aligned} \mathsf{inc}\ \mathsf{assoc}_\times &: (A \ltimes B) \ltimes C \cong A \ltimes (B \times C) \\ \mathsf{inc}\ \mathsf{unitr}_\times &: A \ltimes 1 \cong A \end{aligned}$$

The first four of Hughes's axioms correspond to the requirements of a category $\mathcal{C}$ and an object-preserving functor inc (corresponding to *arr*). Hughes's *first* corresponds to the family of functors $- \ltimes C$ for each object C, with the last two of his axioms corresponding to the naturality requirements above.

In this form, the definition is easily generalized to any symmetric monoidal category. Nor is the assumption of symmetry required; one merely assumes two bifunctors and additional axioms, obtaining what Power and Robinson call a *notion of computation* [26]. Even more general structures have been explored by Blute, Cockett and Seely [4].

Hughes [12] showed that the stream processors of the Fudgets library [5] comprised an arrow type, but were more often used as a dual arrow type. In Power and Robinson's terms, stream processors comprise a notion of computation where the underlying monoidal structure is that of sums rather than products.

A Freyd-category is said to be *closed* [27] if each functor $\mathsf{inc}\ \text{-} \ltimes A : \mathcal{V} \to \mathcal{C}$ has a right adjoint; this is equivalent to Hughes's *ArrowApply* class.

2.2 Deterministic Parsing

Hughes showed how Swierstra and Duponcheel's parser library may be recast using an arrow type, say *ParseArrow*, with composition corresponding to grammatical concatenation. For the empty language and union, we use the classes

```
class Arrow a ⇒ ArrowZero a where
    zeroArrow :: a b c

class ArrowZero a ⇒ ArrowPlus a where
    (<+>) :: a b c → a b c → a b c
```

which make sense for many kinds of arrow. An extra primitive is supplied for terminal symbols.

```
symbol :: Sym → ParseArrow () ()
```

```
data Expr = Plus Expr Expr | Minus Expr Expr | ···

expr :: ParseArrow () Expr
expr = term ⋙ exprTail

exprTail :: ParseArrow Expr Expr
exprTail = (
                arr (λe → (e, ())) ⋙
                second (symbol PLUS) ⋙
                second term ⋙
                arr (λ(e, t) → Plus e t) ⋙
                exprTail
        ) <+> (
                arr (λe → (e, ())) ⋙
                second (symbol MINUS) ⋙
                second term ⋙
                arr (λ(e, t) → Minus e t) ⋙
                exprTail
        ) <+> arr id

term :: ParseArrow () Expr
term = ...
```

Figure 2: Expression parser using arrows

Now we can write parsers using arrow combinators. For example, the parser in Figure 2 expresses the common example grammar

$$\begin{aligned} \mathit{expr} &::= \mathit{term}\ \mathit{exprTail} \\ \mathit{exprTail} &::= \mathit{PLUS}\ \mathit{term}\ \mathit{exprTail} \\ &\mid \mathit{MINUS}\ \mathit{term}\ \mathit{exprTail} \\ &\mid \epsilon \end{aligned}$$

In this program the underlying grammar is obscured by all the plumbing required to pass the results of earlier computations past later ones. (Indeed this is the reason for requiring *first* in the arrow definition.) This point-free style is typical of arrow-based programs. While convenient when defining general combinators and laws, it can be very cumbersome for specific definitions.

3. ARROW-BASED SUBLANGUAGES

We propose to address this problem by defining an extension to Haskell, with the meaning of new forms given by translation rules from the new expressions back into Haskell. This will be done in two stages. Firstly we define a syntax for arrow expressions, which will enable us to write programs resembling the raw monadic syntax (using $\gg\!=$ and $\gg$). Then on top of this we will define an analogue of Haskell's **do**-notation.

The new syntax for arrow expressions, with associated translation rules, is given in Figure 3. An arrow expression is defined by a new binding operator **proc**. The body of such an expression is a new form, which we call a *command*.

3.1 Arrow Application

The simplest kind of command is the arrow application $e_1 \prec e_2$, where e_2 is a Haskell expression to be input to the arrow described by the Haskell expression e_1. As noted above, there is in general no notion of application of arrows,

Syntax

$$
\begin{array}{rcl}
exp & ::= & \ldots \\
 & | & \textbf{proc}\ pat \to cmd \\
cmd & ::= & exp \prec exp \\
 & | & \textbf{form}\ exp\ cmd_1\ \ldots\ cmd_n \\
 & | & cmd_1\ op\ cmd_2 \\
 & | & \kappa\ pat \to cmd \\
 & | & (cmd)
\end{array}
$$

Translation rules

$$
\textbf{proc}\ p \to e_1 \prec e_2 \quad = \quad \begin{cases} arr\ (\lambda p \to e_2) \ggg e_1 \\ \quad \text{if } \mathsf{Vars}(p) \cap \mathsf{Vars}(e_1) = \varnothing \\ arr\ (\lambda p \to (e_1, e_2)) \ggg app \\ \quad \text{otherwise} \end{cases}
$$

$$
\textbf{proc}\ p \to \textbf{form}\ e\ c_1\ \ldots\ c_n = e\ (\textbf{proc}\ p \to c_1)\ \ldots\ (\textbf{proc}\ p \to c_n)
$$

$$
\textbf{proc}\ p \to c_1\ op\ c_2 \quad = \quad \textbf{proc}\ p \to \textbf{form}\ (op)\ c_1\ c_2
$$

$$
\textbf{proc}\ p \to \kappa\ p' \to c \quad = \quad \textbf{proc}\ (p, p') \to c
$$

Figure 3: Arrow expressions

but the rule allows two useful special cases. The first is

$$
\textbf{proc}\ p \to e_1 \prec e_2 \quad = \quad arr\ (\lambda p \to e_2) \ggg e_1
$$

Clearly this is meaningful only if e_1 contains no variables defined in p. A simple example of an expression for e_1 is the identity arrow

$$
\begin{array}{l}
returnA :: Arrow\ a \Rightarrow a\ b\ b \\
returnA = arr\ id
\end{array}
$$

Then we have

$$
\textbf{proc}\ p \to returnA \prec e \quad = \quad arr\ (\lambda p \to e)
$$

This arrow *returnA* will play a role analogous to *return* in monad notation.

The second translation is

$$
\textbf{proc}\ p \to e_1 \prec e_2 \quad = \quad arr\ (\lambda p \to (e_1, e_2)) \ggg app
$$

This version has no such syntactic restriction, but it does require that the arrow in use belong to the class *ArrowApply*, and thus be equivalent to a monad. Thus both rules are needed. The rules overlap, but from the axioms of *app* it is possible to show that in that case they produce equivalent translations.

Hence we must distinguish two kinds of variables:

local variables defined in the current arrow expression.

external variables defined outside.

In this paper we shall focus on arrows that are not equivalent to monads, so we shall use only the first form of arrow application. Nevertheless, the notation may be used with a variety of arrows, some of which are equivalent to monads.

3.2 Control Operators

Next we need a means to combine commands to make new ones. In the monad setting, we have operators like

$$
mplus :: MonadPlus\ m \Rightarrow m\ a \to m\ a \to m\ a
$$

This works well, because in an expression like

$$
e_1\ `mplus`\ e_2
$$

the two expressions may take inputs from environment variables bound in ordinary ways. However, we cannot in general factor an arrow type as a function from inputs, so an arrow combinator must route the inputs of the composite expression to each of the arguments. Hence the corresponding arrow operator has the signature

$$
(\lhd\!\rhd) :: ArrowPlus\ a \Rightarrow a\ b\ c \to a\ b\ c \to a\ b\ c
$$

In the arrow notation, a command describes an arrow from the local environment. We can use operators to combine commands by combining the resulting arrows, so for example we have

$$
\textbf{proc}\ p \to c_1 \lhd\!\rhd c_2 \quad = \quad (\textbf{proc}\ p \to c_1) \lhd\!\rhd (\textbf{proc}\ p \to c_2)
$$

In general an operator may be an arbitrarily complex Haskell expression meeting certain conditions (to be given below). The syntax requires a keyword **form** to distinguish these from commands. However in the special case of infix operators we can use an abbreviated syntax as above.

Parameter Passing. Some operators pass data to their arguments. For example, the monadic operator for exception handling has the form

$$
\begin{array}{l}
handle :: MonadHandle\ ex\ m \Rightarrow \\
\qquad m\ a \to (ex \to m\ a) \to m\ a
\end{array}
$$

If the second argument (the handler) is called, it is passed the exception raised. The arrow form will also have two arguments. Each will be passed the input, with the second also being given the exception:

$$
\begin{array}{l}
handleA :: ArrowHandle\ ex\ a \Rightarrow \\
\qquad a\ b\ c \to a\ (b, ex)\ c \to a\ b\ c
\end{array}
$$

We shall adopt the convention of adding argument data by pairing in this way. In general the input of an arrow will have the form

$$
((\ldots(v, v_1), \ldots), v_n)
$$

where v is the original input, named by the **proc** pattern p, and the v_i are additional arguments, as yet unnamed. The next form, the κ quantifier, applies another pattern to the innermost argument v_1 within a sub-command. A similar quantifier occurs in the abstract machine framework of Douence and Fradet [7], transferring a value from the argument stack to the environment.

Thus we can write a command like

$$
c_1\ `handleA`\ \kappa\ ex \to c_2
$$

This may be read just like the corresponding monadic form: the body c_1 is executed, and if it raises an exception then the handler c_2 is called, with ex bound to the exception raised. However, the arrow version of the operator passes the original environment to both commands, as we can see from the translation:

$$
\begin{array}{l}
\textbf{proc}\ p \to c_1\ `handleA`\ \kappa\ ex \to c_2 \\
= (\textbf{proc}\ p \to c_1)\ `handleA`\ (\textbf{proc}\ p \to \kappa\ ex \to c_2) \\
= (\textbf{proc}\ p \to c_1)\ `handleA`\ (\textbf{proc}\ (p, ex) \to c_2)
\end{array}
$$

An operator may also accept an argument from its caller in a similar way, as in the following operator to encapsulate state-transforming arrows:

$$
runWithState :: \ldots \Rightarrow a\ b\ c \to a'\ (b, s)\ c
$$

Naturality. We stated above that an operator delivers inputs of the composite arrow to its components. We can formalize this with a naturality condition for each operator. For example, the *handleA* operator will be required to satisfy

$$arr\ k \ggg (f\ `handleA`\ g) = (arr\ k \ggg f)\ `handleA`\ (arr\ (k \times id) \ggg g)$$

This ensures that inputs delivered by the operator to f or g were inputs to the whole expression. In general, an input to the whole expression need not be delivered to each argument; in the above example g is called only if an exception occurs in f. But any input that is delivered must have been an input to the whole arrow.

In the special case of a Kleisli arrow of a monad m, this naturality condition ensures that the operator is equivalent to a monadic operator. In this case, the type of *handleA* is equivalent to

$$(b \to m\ c) \to ((b, ex) \to m\ c) \to b \to m\ c$$

Currying the second argument gives the type

$$(b \to m\ c) \to (b \to ex \to m\ c) \to b \to m\ c$$

Since this is natural in b (and the Kleisli arrows factor as functions) it is equivalent to the type of the corresponding monad operator

$$m\ c \to (ex \to m\ c) \to m\ c$$

Many monadic operators have similar generalizations in the arrow setting.

Formal Definition of a Control Operator. In order to specify which Haskell expressions may serve as control operators, we need a preliminary definition:

Definition 3. Let τ stand for a Haskell value type. We introduce a new sort of types

$$\text{Command types} \quad \theta \ ::= \ a\backslash\tau \ \mid \ \tau \rightharpoonup \theta$$

For each Haskell type τ and command type θ, we define a Haskell type $\tau \rightsquigarrow \theta$ as follows

$$\begin{aligned} \tau \rightsquigarrow a\backslash\tau' &= a\ \tau\ \tau' \\ \tau \rightsquigarrow (\tau' \rightharpoonup \theta) &= (\tau, \tau') \rightsquigarrow \theta \end{aligned}$$

If $k :: \tau_1 \to \tau_2$, we can define a function $k \rightsquigarrow \theta :: (\tau_2 \rightsquigarrow \theta) \to (\tau_1 \rightsquigarrow \theta)$ by

$$\begin{aligned} k \rightsquigarrow a\backslash\tau &= (arr\ k \ggg) \\ k \rightsquigarrow (\tau \rightharpoonup \theta) &= (k \times id) \rightsquigarrow \theta \end{aligned}$$

Definition 4. A Haskell expression e is a *control operator* of signature $\theta_1 \to \cdots \theta_n \to \theta$ if

1. No local variables occur free in e,
2. e has type
$$\forall b.\ (b \rightsquigarrow \theta_1) \to \cdots (b \rightsquigarrow \theta_n) \to (b \rightsquigarrow \theta)$$
where b does not occur free in any of the θs, and
3. e satisfies a corresponding naturality property
$$e\ ((k \rightsquigarrow \theta_1)\ x_1)\ \ldots\ ((k \rightsquigarrow \theta_n)\ x_n) = (k \rightsquigarrow \theta)\ (e\ x_1\ \ldots\ x_n)$$

```
expr :: ParseArrow () Expr
expr = proc () →
            (term ↢ ()) `bind` κ t →
            exprTail ↢ t

exprTail :: ParseArrow Expr Expr
exprTail = proc e → (
                (symbol PLUS ↢ ())`bind_`
                (term ↢ ()) `bind` κ t →
                exprTail ↢ Plus e t
      ) <+> (
                (symbol MINUS ↢ ())`bind_`
                (term ↢ ()) `bind` κ t →
                exprTail ↢ Minus e t
      ) <+> (returnA ↢ e)
```

Figure 4: Expression parser in arrow notation

The first two conditions would be checked by the implementation. It may be that the naturality property can be obtained automatically from parametricity results.

For example, the control operator *handleA* has signature $a\backslash c \to (ex \rightharpoonup a\backslash c) \to a\backslash c$.

Examples. Some functions we have already seen are also examples of control operators:

```
(&&&) :: Arrow a ⇒ a b c → a b d → a b (c, d)
(<+>) :: ArrowPlus a ⇒ a b c → a b c → a b c
zeroArrow :: ArrowZero a ⇒ a b c
```

Others may be defined using the features of Haskell. For example, the arrow counterpart of the monadic binding operator $\gg=$ may be defined as

```
bind :: Arrow a ⇒ a b c → a (b, c) d → a b d
u `bind` f = arr id &&& u >>> f
```

Using this operator, we can redefine *addA* (which is also an operator):

```
addA :: Arrow a ⇒ a b Int → a b Int → a b Int
addA f g = proc z →
                (f ↢ z) `bind` κ x →
                (g ↢ z) `bind` κ y →
                returnA ↢ x + y
```

Another useful operator is the special case of *bind* where the result of the first computation is ignored, corresponding to the monadic $\gg$ combinator:

```
bind_ :: Arrow a ⇒ a b c → a b d → a b d
u `bind_` v = u `bind` (arr fst >>> v)
```

Now we can rewrite the arrow parser of Figure 2, obtaining the version of Figure 4. As promised, the form of this program is very similar to what we would write with monadic parser combinators [13]. The point is that this program works not merely for monadic parsers but also for any parser that can be cast in the more general arrow form, including the optimized ones of Swierstra and Duponcheel [28]. All the plumbing of the previous version is hidden.

3.3 Theoretical Aside

Power and Thielecke [27] showed that each Freyd-category is equivalent to a kind of indexed category called a *κ-category*. Each category H_A models computations in a context A, and has the same objects as $\mathcal{C}$, with morphism sets

$$H_A(B, C) = \mathcal{C}(A \times B, C)$$

Our command sublanguage could be viewed as a language for such indexed categories, with A corresponding to the input context and B to additional arguments. The κ quantifier then corresponds to the obvious isomorphism

$$H_A(B \times C, D) \cong H_{A \times B}(C, D)$$

as in Hasegawa's κ-calculus [9]. A control operator defines a natural family of functions, one for each category H_A. These generalize the controls of elementary control structures [25], which are used to model concurrency. Our definition suggests a higher-order generalization, although such operators appear to be less useful.

3.4 Type-checking of Commands

One could use the equations of Figure 3 to transform any arrow expression into ordinary Haskell, where it will be type-checked, but it would obviously be easier for users to deal with a type system for the command sublanguage. There is not room here for a formal treatment, not least because there is no complete definition of Haskell's type system to refer to, but the basic ideas are simple. Each command is assigned a command type as follows:

- If $e_1 :: a\ \tau\ \tau'$ and $e_2 :: \tau$, then $e_1 \prec e_2$ has type $a \backslash \tau'$.
- If c has type θ assuming $p :: \tau$, then $\kappa\, p \to c$ has type $\tau \rightharpoonup \theta$.
- If each c_i has type θ_i and e is a control operator of signature $\theta_1 \to \cdots \theta_n \to \theta$, then **form** $e\ c_1 \ldots c_n$ has type θ.

It follows (by induction on c) that if c has type θ assuming $p :: \tau$, then **proc** $p \to c :: \tau \rightsquigarrow \theta$, and

$$((\lambda p' \to e) \rightsquigarrow \theta)\ (\mathbf{proc}\ p \to c) \quad = \quad \mathbf{proc}\ p' \to [e/p]c$$

This equation expresses the naturality of commands with respect to the environment, allowing us to change the representation of the environment, for example to improve efficiency. We shall return to this point in Section 3.7.

3.5 Equivalences

It is also useful to reason directly with commands.

Definition 5. We write $c_1 \equiv_p c_2$ for

$$\mathbf{proc}\ p \to c_1 = \mathbf{proc}\ p \to c_2$$

and $c_1 \equiv c_2$ to mean $c_1 \equiv_p c_2$ for all legal p.

Then we have the following equivalences for *returnA* and *bind*, corresponding to the familiar monad laws:

$$\begin{aligned}
(\mathit{returnA} \prec e)\ \text{`}\mathit{bind}\text{`}\ \kappa\, x \to c &\equiv [e/x]c \\
c\ \text{`}\mathit{bind}\text{`}\ \kappa\, x \to \mathit{returnA} \prec x &\equiv c \\
c_1\ \text{`}\mathit{bind}\text{`}\ \kappa\, x_1 \to (c_2\ \text{`}\mathit{bind}\text{`}\ \kappa\, x_2 \to c_3) &\equiv \\
(c_1\ \text{`}\mathit{bind}\text{`}\ \kappa\, x_1 \to c_2)\ \text{`}\mathit{bind}\text{`}\ \kappa\, x_2 \to c_3 &
\end{aligned}$$

An arrow library would typically supply a collection of arrows and operators with associated laws, ideally expressed as equivalences between commands.

Syntax

$$\begin{aligned}
\mathit{cmd} &::= \ldots \\
&\mid \mathbf{do}\ \{\mathit{stmt}_1; \ldots \mathit{stmt}_n; \mathit{cmd}\} \\
\mathit{stmt} &::= \mathit{cmd} \\
&\mid \mathit{pat} \leftarrow \mathit{cmd} \\
&\mid \mathbf{rec}\ \{\mathit{stmt}_1; \ldots; \mathit{stmt}_n\}
\end{aligned}$$

Translation rules

$$\begin{aligned}
\mathbf{do}\ \{c\} &\equiv c \\
\mathbf{do}\ \{p \leftarrow c; A\} &\equiv c\ \text{`}\mathit{bind}\text{`}\ \kappa\, p \to \mathbf{do}\ \{A\} \\
\mathbf{do}\ \{c; A\} &\equiv c\ \text{`}\mathit{bind}\_\text{`}\ \mathbf{do}\ \{A\} \\
\mathbf{do}\ \{\mathbf{rec}\ A; B\} &\quad \text{(see Section 5.3)}
\end{aligned}$$

Figure 5: do-notation for arrows

```
expr :: ParseArrow () Expr
expr = proc () → do
           t ← term ≺ ()
           exprTail ≺ t

exprTail :: ParseArrow Expr Expr
exprTail = proc e → do
               symbol PLUS ≺ ()
               t ← term ≺ ()
               exprTail ≺ Plus e t
     ⋘ do
               symbol MINUS ≺ ()
               t ← term ≺ ()
               exprTail ≺ Minus e t
     ⋘ do
               returnA ≺ e
```

Figure 6: Expression parser using do-notation

3.6 do-notation for Arrows

We can take the correspondence further, by defining a **do**-notation for commands in a similar fashion to Haskell's monadic **do**-notation. The syntax and translation rules are given in Figure 5. The **rec** construct, which allows recursive bindings, will be discussed in Section 5.3.

Then the above operator *addA* could be rewritten as

```
addA :: Arrow a ⇒ a b Int → a b Int → a b Int
addA f g = proc z → do
    x ← f ≺ z
    y ← g ≺ z
    returnA ≺ x + y
```

Similarly, the parser example of Figures 2 and 4 may be rewritten as in Figure 6, which is similar to the monadic version, though it works for a wider variety of parsers.

3.7 Improving the Translation

The rules of Figures 3 and 5 define the meaning of the new constructs in clear way, but may produce less efficient programs than one might have written by hand. For example, the arrow *addA* above would be translated to

```
addA f g = arr id &&& f ⋙
           arr id &&& (arr (λ(z, x) → z) ⋙ g) ⋙
           arr (λ((z, x), y) → x + y)
```

Note that both the original input z and the first result x are held during the computation of g, even though z is not required. We can project out z when it is no longer needed, obtaining the improved version

$$\begin{aligned} addA\ f\ g = {} & arr\ id \mathbin{\&\&\&} f \ggg \\ & arr\ (\lambda(z,x) \to (x,z)) \ggg second\ g \ggg \\ & arr\ (\lambda(x,y) \to x+y) \end{aligned}$$

which is essentially equivalent to what we would write by hand. These projections may also be moved through operators, thanks to their naturality property. The prototype implementation incorporates many such improvements.

4. EXAMPLE: DATA PARALLELISM

For each set S, the type $a^S \to b^S$ defines an arrow. Such arrows may be used to model data parallel computation; here S represents the set of processors, and the *arr* operation executes the same function on each processor. Additional combinators will be required for the various operations supported by a particular model.

Here we shall focus on a special case: algorithms operating on 2^n elements, whose behaviour is defined by induction on n. These arise in circuit design (cf. Ruby [15]), and descriptions of parallel algorithms (cf. Misra's powerlists [21]).

The objects of interest then consist of infinite sequences of functions on arrays of increasing size

$$\prod_{n=0}^{\infty} a^{2^n} \to b^{2^n}$$

We can model a^{2^n} as $Pair^n\ a$, where

$$\textbf{type}\ Pair\ a = (a,a)$$

Thus the elements are organized as a perfectly balanced binary tree of depth n, and we are interested in functions that preserve this depth. We call them "homogeneous functions" and model them with the following Haskell datatype:

$$\textbf{data}\ Hom\ a\ b = (a \to b) :\&: Hom\ (Pair\ a)\ (Pair\ b)$$

Elements of this type have the form

$$f_0 :\&: f_1 :\&: f_2 :\&: \ldots$$

where $f_n :: Pair^n\ a \to Pair^n\ b$.

Before writing programs with this datatype, we need a framework for executing them. We will define a type for perfectly balanced binary trees:

$$\begin{aligned} &\textbf{data}\ BalTree\ a = Zero\ a \mid Succ\ (BalTree\ (Pair\ a)) \\ &\quad \textbf{deriving}\ Show \end{aligned}$$

Here are some example elements:

$$\begin{aligned} tree0 &= Zero\ 1 \\ tree1 &= Succ\ (Zero\ (1,2)) \\ tree2 &= Succ\ (Succ\ (Zero\ ((1,2),(3,4)))) \\ tree3 &= Succ\ (Succ\ (Succ\ (Zero\ (((1,2),(3,4)), \\ &\qquad ((5,6),(7,8)))))) \end{aligned}$$

The elements of this type have a string of constructors expressing a depth n as a Peano numeral, enclosing a nested pair tree of 2^n elements.

The following function applies a homogeneous function to a perfectly balanced tree, yielding another perfectly balanced tree of the same depth:

$$\begin{aligned} &apply :: Hom\ a\ b \to BalTree\ a \to BalTree\ b \\ &apply\ (f :\&: fs)\ (Zero\ x) = Zero\ (f\ x) \\ &apply\ (f :\&: fs)\ (Succ\ t) = Succ\ (apply\ fs\ t) \end{aligned}$$

Few other operations can be expressed in terms of the balanced tree type. Typically one wants to split a tree into two subtrees, do some processing on the subtrees and combine the results. But the type system cannot discover that the two results are of the same depth (and thus combinable). Of course, this is exactly what homogeneous functions can do, so we shall focus on them; the balanced tree type is used only for test runs of our algorithms.

Firstly, *Hom* is an arrow:

$$\begin{aligned} &\textbf{instance}\ Arrow\ Hom\ \textbf{where} \\ &\quad arr\ f = f :\&: arr\ (f \times f) \\ &\quad (f :\&: fs) \ggg (g :\&: gs) = (g \cdot f) :\&: (fs \ggg gs) \\ &\quad first\ (f :\&: fs) = first\ f :\&: \\ &\qquad (arr\ transpose \ggg first\ fs \ggg arr\ transpose) \\ \\ &transpose :: ((a,b),(c,d)) \to ((a,c),(b,d)) \\ &transpose\ ((a,b),(c,d)) = ((a,c),(b,d)) \end{aligned}$$

The function *arr* maps a function over the leaves of the tree. The composition $\ggg$ composes sequences of functions pairwise. The $\mathbin{\&\&\&}$ operator unriffles a tree of pairs (a,b) into a tree of as and a tree of bs, applies the appropriate function to each tree and riffles the results.

When describing algorithms, one often provides a pure function for the base case (trees of one element) and a expression for trees of pairs, usually invoking the same algorithm on smaller trees.

Parallel Prefix. This operation (also called *scan*) maps the sequence

$$x_0, x_1, x_2, \ldots, x_{2^n-1}$$

to the sequence

$$x_0, x_0 \oplus x_1, x_0 \oplus x_1 \oplus x_2, \ldots, x_0 \oplus x_1 \oplus \cdots \oplus x_{2^n-1}$$

for some associative operation $\oplus$.

If there is only one element (i.e. the tree has zero depth) then obviously the scan should be the identity function. Otherwise, we need to deal with a tree of pairs, so the general *scan* operation will have the form

$$\begin{aligned} &scan :: (a \to a \to a) \to a \to Hom\ a\ a \\ &scan\ (\oplus)\ b = id :\&: \textbf{proc}\ (x,y) \to \ldots \end{aligned}$$

where b is the identity of the $\oplus$ operation[1]. The missing part will be defined using recursive calls of *scan*, but operating on smaller trees.

An efficient scheme, devised by Ladner and Fischer [18], is first to sum the elements pairwise:

$$x_0 \oplus x_1, x_2 \oplus x_3, x_4 \oplus x_5, \ldots$$

and then to compute the scan of this list (which is half the length of the original), yielding

$$x_0 \oplus x_1, x_0 \oplus x_1 \oplus x_2 \oplus x_3, x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5, \ldots$$

This list is half of the desired answer; the other elements are

$$x_0, x_0 \oplus x_1 \oplus x_2, x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4, \ldots$$

[1]It is possible to do without the identity, at the cost of slightly complicating the code.

which can be obtained by shifting our partial answer one place to the right and adding $x_0, x_2, x_4, \ldots$. We can express this idea directly in our notation:

$$\begin{aligned}
&\mathit{scan} :: (a \to a \to a) \to a \to \mathit{Hom}\ a\ a\\
&\mathit{scan}\ (\oplus)\ b = \mathit{id} :\&: \mathbf{proc}\ (x, y) \to \mathbf{do}\\
&\qquad\qquad\qquad\qquad y' \leftarrow \mathit{scan}\ (\oplus)\ b \prec x \oplus y\\
&\qquad\qquad\qquad\qquad \mathit{yl} \leftarrow \mathit{rsh}\ b \prec y'\\
&\qquad\qquad\qquad\qquad \mathit{returnA} \prec (\mathit{yl} \oplus x, y')
\end{aligned}$$

The auxiliary arrow $\mathit{rsh}\ b$ shifts each element in the tree one place to the right, placing b in the now-vacant leftmost position, and discarding the old rightmost element. This could be supplied as a primitive, but it is also possible to code it directly:

$$\begin{aligned}
&\mathit{rsh} :: a \to \mathit{Hom}\ a\ a\\
&\mathit{rsh}\ b = \mathit{const}\ b :\&: \mathbf{proc}\ (x, y) \to \mathbf{do}\\
&\qquad\qquad\qquad\qquad \mathit{yl} \leftarrow \mathit{rsh}\ b \prec y\\
&\qquad\qquad\qquad\qquad \mathit{returnA} \prec (\mathit{yl}, x)
\end{aligned}$$

Butterfly Circuits. In many divide-and-conquer schemes, one recursive call processes the odd-numbered elements and the other processes the even ones [14]:

$$\begin{aligned}
&\mathit{butterfly} :: (\mathit{Pair}\ a \to \mathit{Pair}\ a) \to \mathit{Hom}\ a\ a\\
&\mathit{butterfly}\ f = \mathit{id} :\&: \mathbf{proc}\ (x, y) \to \mathbf{do}\\
&\qquad\qquad\qquad\qquad x' \leftarrow \mathit{butterfly}\ f \prec x\\
&\qquad\qquad\qquad\qquad y' \leftarrow \mathit{butterfly}\ f \prec y\\
&\qquad\qquad\qquad\qquad \mathit{returnA} \prec f\ (x', y')
\end{aligned}$$

The recursive calls operate on halves of the original tree, so the recursion is well-defined. (The Fast Fourier Transform has a similar structure.) Some examples of butterflies:

$$\begin{aligned}
&\mathit{rev} :: \mathit{Hom}\ a\ a\\
&\mathit{rev} = \mathit{butterfly}\ \mathit{swap}
\end{aligned}$$

$$\begin{aligned}
&\mathit{unriffle} :: \mathit{Hom}\ (\mathit{Pair}\ a)\ (\mathit{Pair}\ a)\\
&\mathit{unriffle} = \mathit{butterfly}\ \mathit{transpose}
\end{aligned}$$

Batcher's ingenious sorter for bitonic sequences [1] is another example of a butterfly circuit:

$$\begin{aligned}
&\mathit{bisort} :: \mathit{Ord}\ a \Rightarrow \mathit{Hom}\ a\ a\\
&\mathit{bisort} = \mathit{butterfly}\ \mathit{cmp}\\
&\qquad \mathbf{where}\ \mathit{cmp}\ (x, y) = (\mathit{min}\ x\ y, \mathit{max}\ x\ y)
\end{aligned}$$

This can be used (with *rev*) as the merge phase of a sorting function.

5. RECURSION

Since arrows are Haskell values, they may be recursively defined in the usual way, as we have seen. A different kind of recursion involves recursive definition of values within a computation, where an output is used as an input. To express this, we will define a feedback operator on arrows, though not all arrows will have such an operator. We expect that it would generalize the fixed point operator on monads, which has signature

$$\begin{aligned}
&\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadFix}\ m\ \mathbf{where}\\
&\qquad \mathit{mfix} :: (a \to m\ a) \to m\ a
\end{aligned}$$

An axiomatization of this operator is given by Erkök and Launchbury [8]. Not all monads have such an operator, but several important ones do, including state transformers, readers, writers and Haskell's built-in monads ST and IO.

The straightforward generalization of *mfix* would be the class

$$\begin{aligned}
&\mathbf{class}\ \mathit{Arrow}\ a \Rightarrow \mathit{ArrowFix}\ a\ \mathbf{where}\\
&\qquad \mathit{fixA} :: a\ (b, c)\ c \to a\ b\ c
\end{aligned}$$

This could work, but it is neater to separate the output from the feedback data, giving the more symmetrical definition

$$\begin{aligned}
&\mathbf{class}\ \mathit{Arrow}\ a \Rightarrow \mathit{ArrowLoop}\ a\ \mathbf{where}\\
&\qquad \mathit{loop} :: a\ (b, d)\ (c, d) \to a\ b\ c
\end{aligned}$$

The trivial arrow type has such an operator:

$$\begin{aligned}
&\mathbf{instance}\ \mathit{ArrowLoop}\ (\to)\ \mathbf{where}\\
&\qquad \mathit{loop} = \mathit{simple_loop}\\
&\\
&\mathit{simple_loop} :: ((b, d) \to (c, d)) \to b \to c\\
&\mathit{simple_loop}\ f\ b = c\ \mathbf{where}\ (c, d) = f\ (b, d)
\end{aligned}$$

Monads with *mfix* give rise to Kleisli arrows with a *loop* operator:

$$\begin{aligned}
&\mathbf{instance}\ \mathit{MonadFix}\ m \Rightarrow\\
&\qquad\qquad \mathit{ArrowLoop}\ (\mathit{Kleisli}\ m)\ \mathbf{where}\\
&\qquad \mathit{loop}\ (K\ f) = K\ (\mathit{liftM}\ \mathit{fst} \cdot \mathit{mfix} \cdot f')\\
&\qquad\qquad \mathbf{where}\ f'\ x\ y = f\ (x, \mathit{snd}\ y)
\end{aligned}$$

We shall require that the *loop* operator satisfy the equations of Figure 7. These axioms are also presented in a graphical form in Figure 8. Here the ovals represent *loop* operators, which feed part of the output of the arrow inside back to its input.

Our intent, as with *mfix*, is that a value is recursively defined, but the computation is executed only once. Thus computations at the start or end that are independent of the recursively defined value can by moved out of *loop*, using the tightening rules. On the other hand, the sliding rule can move only pure computations on the recursively defined value from the end of the *loop* to the start; moving general computations would change the order of computational effects. The vanishing rule states that nested recursive definitions are equivalent to simultaneous recursive definitions. Superposing adds unrelated data to the recursion. Finally, we require that *loop* should extend *simple_loop*.

Instances of *loop* for specific arrows may well satisfy additional axioms. For example, effect-free synchronous circuits would satisfy a stronger version of the sliding axiom, in which arbitrary circuits could be moved around the loop.

5.1 Theoretical Aside

The *simple_loop* operator is an example of a *trace operator*, as defined by Joyal, Street and Verity [16, 10]. Their definition assumed a braided monoidal category (a relaxation of a symmetric monoidal category). The equations of Figure 7 generalize their axioms to Freyd-categories, and the names of all but the last are taken from the corresponding trace axioms.

In this setting, *loop* defines a family of functions

$$\mathcal{C}(B \ltimes D, C \ltimes D) \to \mathcal{C}(B, C)$$

Then the tightening rules amount to naturality in B and C, while sliding specifies dinaturality in D.

Left tightening	$loop\ (first\ h \ggg f)$	$=$	$h \ggg loop\ f$
Right tightening	$loop\ (f \ggg first\ h)$	$=$	$loop\ f \ggg h$
Sliding	$loop\ (f \ggg arr\ (id \times k))$	$=$	$loop\ (arr\ (id \times k) \ggg f)$
Vanishing	$loop\ (loop\ f)$	$=$	$loop\ (arr\ assoc^{-1} \ggg f \ggg arr\ assoc)$
Superposing	$second\ (loop\ f)$	$=$	$loop\ (arr\ assoc \ggg second\ f \ggg arr\ assoc^{-1})$
Extension	$loop\ (arr\ f)$	$=$	$arr\ (simple\_loop\ f)$

Figure 7: Loop equations

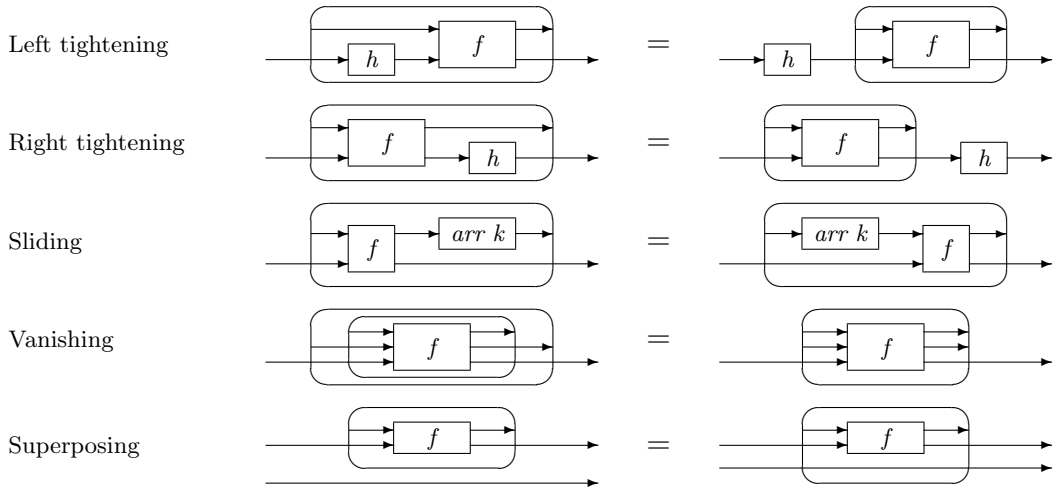


Figure 8: Loop equations in graphical form

It is straightforward to further generalize the signature of *loop* and these axioms to any symmetric notion of computation. Indeed the Fudgets stream processor library [5] already includes a version of *loop* based on sums rather than products.

5.2 Comparison with *mfix*

Our axioms, restricted to the special case of Kleisli arrows, may be compared to the axiomatization of *mfix* given by Erkök and Launchbury [8]. The three axioms they postulate correspond to our extension, left tightening and vanishing axioms respectively. They reject a possible law corresponding to right tightening, because it fails for certain monads, the most important of which are those involving exceptions. Parametricity of *mfix* implies a weaker form of the sliding law, in which k must be strict, and this proves to be necessary for the exception monads. They suggest that a slightly stronger version of parametricity holds for all monads of interest; this would imply a counterpart of the superposing law.

It may be that a similar relaxation would be desirable in the arrow context. For example, a *loop* operator on parser arrows could be used to pass attributes between parsers in either direction. (The parsers themselves are values of arrow type, and would be recursively defined using the ordinary recursion of Haskell.) However, such a loop operator would not satisfy the right tightening axiom, because the computation h might cause the parse to fail, which would make the attributes undefined if h were inside the *loop*. Similarly the sliding axiom would fail for non-strict k, if the parse inside the loop were to fail.

5.3 Extending the do-notation

We could use the *loop* operator directly, but is is more convenient to add recursive bindings to our **do**-notation, as foreshadowed in Figure 5. We use a form modelled on the recursive **let** (O'Haskell [23] has a similar notation for monadic **do**), rather than the recursive **do** of Erkök and Launchbury. This form is more flexible, and has a simple correspondence to *loop*, given by the following translation rule:

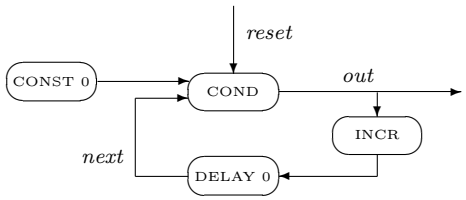
$$\begin{aligned}\mathbf{do}\,\{\mathbf{rec}\ A; B\} \equiv \mathbf{do}\ pB \leftarrow \mathbf{form}\ loop\ (\kappa\, pA \rightarrow \mathbf{do} \\ A \\ returnA \prec (pA, pB)) \\ B\end{aligned}$$

where pA is a pattern containing those variables defined in A that are required in A, and pB is a pattern containing those variables defined in A that are required in B.

6. EXAMPLE: SYNCHRONOUS CIRCUITS

A synchronous circuit receives an input and produces an output on each tick of some global clock. The output for a given tick may depend on the input for that tick, as well as previous inputs. Such circuits fit well with the data-flow model of computation, and several languages of that type have been used to model them [2, 6, 29].

Consider the following simple circuit (taken from [20]):



This circuit represents a resettable counter, taking a Boolean input and producing an integer output, which will be the number of clock ticks since the input was last *True*. To achieve this, the output is incremented and fed back, delayed by one clock cycle. The first output of the DELAY component is its argument, here 0; its second output is its first input, and so on.

Hardware description languages embedded in Haskell can achieve considerable flexibility by parameterizing descriptions over type classes. The microarchitecture design language Hawk [20] abstracts over the type of values that may pass through wires. Low-level descriptions deal with bits (*Bool*), but any Haskell type may be used, allowing Hawk to scale to much more abstract descriptions, and also allowing the same circuit description to be simulated or symbolically executed. Further interpretations are possible with the hardware description language Lava [3], where circuits have the form

$$Value \to Monad\ Value'$$

where both value and monad types are parameters described by Haskell classes. By selecting appropriate instances, a single description may be simulated, symbolically executed or presented in a variety of styles[2].

6.1 A Circuit Class

We propose to generalize, treating circuits as arrows, so that a wider range of interpretations will be possible. It suffices to consider circuits with a single input and output, because multiple inputs may be treated as input of a tuple, and similarly for output.

- The *arr* operation defines circuits where each output is a pure function of the corresponding input (e.g. COND and INCR in the above circuit).
- Composition connects the output of the first circuit to the input of the second.
- The *first* operation channels part of the input to a subcircuit, with the rest copied directly to the output.

As usual, we shall require additional operations. We define circuits as arrows that support cycles and a delay arrow:

```
class ArrowLoop a ⇒ ArrowCircuit a where
    delay :: b → a b b
```

The argument supplies the initial output; subsequent outputs are copied from the input of the previous tick. A circuit built with *loop* must include a *delay* somewhere on its second input before using it, as in the example above. One could enforce this by combining the two in a single construct, but the present formulation is better suited to algebraic manipulation.

[2]The most recent release of Lava has however removed monads from the language, partly by pushing impure features into their variant of Haskell.

```
newtype SeqMap b c = SM (Seq b → Seq c)

instance Arrow SeqMap where
    arr f = SM (mapSeq f)
    SM f ⋙ SM g = SM (g · f)
    first (SM f) =
        SM (zipSeq · (f × id) · zipSeq⁻¹)

instance ArrowLoop SeqMap where
    loop (SM f) =
        SM (simple_loop (zipSeq⁻¹ · f · zipSeq))

instance ArrowCircuit SeqMap where
    delay x = SM (Cons x)
```

Figure 9: A circuit arrow type

Here is the resettable counter circuit in arrow notation:

```
counter :: ArrowCircuit a ⇒ a Bool Int
counter = proc reset → do
            rec next ← delay 0 ≺ out + 1
                out ← returnA ≺
                        if reset then 0 else next
            returnA ≺ out
```

This corresponds rather directly to the graphical presentation given earlier. The variables denote the values passing through wires on a particular clock tick.

6.2 Interpretations

One implementation uses an idea introduced by Kahn [17]: components define functions from infinite sequences of inputs to infinite sequences of outputs. This idea is the basis for several data-flow languages [2, 6, 29], for which hardware simulation is just one application, as well as the microarchitecture design language Hawk mentioned above.

Infinite sequences may be modelled in Haskell by defining

```
data Seq a = Cons a (Seq a)
```

A circuit description in Hawk consists of a simultaneous recursive definition of several such sequences (there called *signals*), each representing the entire sequence of values that pass through a particular wire over time. We can use this idea to define a circuit arrow type as in Figure 9. The definitions use the obvious functions

```
mapSeq :: (a → b) → Seq a → Seq b
zipSeq :: (Seq a, Seq b) → Seq (a, b)
zipSeq⁻¹ :: Seq (a, b) → (Seq a, Seq b)
```

The underlying model is the same as in Hawk, but programming with arrows has a different feel. In Hawk one works with circuits and wires, with variables denoting the entire history of wires. Each primitive operation on values must be lifted to an operation on sequences, and one often has to convert back and forth between tuples of sequences and sequences of tuples. In the arrow formulation, one works with circuits and values. The conversions are still happening, but

they are built into the arrow combinators, and thus hidden by the arrow notation.

Other implementations of the *ArrowCircuit* class are possible. Instead of maps of sequences, we could use automata that map an input to an output and a new circuit, as follows:

```
newtype Auto b c = A (b → (c, Auto b c))
```

The external behaviour is the same, but this interpretation may have different operational characteristics.

We can define further implementations, and thus additional interpretations, by two strategies.

1. We can generalize the types *SeqMap* and *Auto*, replacing the function type with an arrow parameter, so that they become arrow transformers that may be applied to any arrow type that provides *loop*, such as state transformers.

2. Alternatively, we can apply other arrow transformers to an existing circuit arrow type to create a new one with additional features.

For example, to add debugging probes to circuits, we define a class

```
class ArrowCircuit a ⇒ ProbedCircuit a where
    probe :: Show b ⇒ String → a b ()
```

so we can extend the counter example

```
counter :: ProbedCircuit a ⇒ a Bool Int
counter = proc reset → do
              rec probe "Reset" ≺ reset
                  next ← delay 0 ≺ out + 1
                  out ← returnA ≺
                            if reset then 0 else next
                  probe "Output" ≺ out
              returnA ≺ out
```

The intention is that when this circuit is run, the sequence of values passing through the named wires will be recorded. Achieving this in Hawk seems to require non-declarative extensions to Haskell [20].

In the arrow setting, we can use the second technique above, defining a *Writer* arrow transformer that adds output to any arrow, and indeed preserves all the *ArrowCircuit* structure, as in Figure 10. (We have used the Haskell string output type *ShowS*, but this is easily generalized to any monoid). The simulator may then pick off the probe output from the circuit outputs.

```
newtype Writer a b c = W (a b (c, ShowS))

instance Arrow a ⇒ Arrow (Writer a) where
    arr f = W (arr (λx → (f x, id)))
    W f ⋙ W g = W (proc x → do
        (y, s1) ← f ≺ x
        (z, s2) ← g ≺ y
        returnA ≺ (z, s1 · s2))
    first (W f) = W (proc (x, y) → do
        (x', s) ← f ≺ x
        returnA ≺ ((x', y), s))

instance ArrowLoop a ⇒
    ArrowLoop (Writer a) where
    loop (W f) = W (proc b → do
        rec~(~(c, d), s) ← f ≺ (b, d)
        returnA ≺ (c, s))

instance ArrowCircuit a ⇒
    ArrowCircuit (Writer a) where
    delay x = W (delay x &&& arr (const id))

write :: Arrow a ⇒ Writer a ShowS ()
write = W (arr (λs → ((), s)))

instance ArrowCircuit a ⇒
    ProbedCircuit (Writer a) where
    probe label = proc x →
        write ≺ showString label·
            showString "␣=␣"·
            shows x · showChar '\n'
```

Figure 10: Adding output to an arrow

6.3 Conditionals

All the interpretations considered above are also instances of *ArrowChoice*. For example, here is an instance for the *SeqMap* type:

```
instance ArrowChoice SeqMap where
    left (SM f) =
        SM (λxs → replace xs (f (getLeft xs)))

getLeft :: Seq (Either a b) → Seq a
getLeft (Cons (Left x) xs) = Cons x (getLeft xs)
getLeft (Cons (Right _) xs) = getLeft xs

replace :: Seq (Either a b) → Seq c → Seq (Either c b)
replace (Cons (Left _) xs)~(Cons y ys) =
    Cons (Left y) (replace xs ys)
replace (Cons (Right x) xs) ys =
    Cons (Right x) (replace xs ys)
```

The subsequence of inputs tagged with *Left* is extracted by *getLeft* and fed to the subcircuit f. The outputs of f are then tagged with *Left* and merged with the original sequence by *replace*, replacing the corresponding inputs. The effect is to conditionally propagate the clock to subcircuits, as with the **when** construct of Lustre [6].

For arrows in *ArrowChoice*, we can define conditional commands as follows (**case** commands may be defined similarly):

```
proc p → if e then c1 else c2 =
    arr (λp → if e then Left p else Right p) ⋙
    (proc p → c1) ||| (proc p → c2)
```

A circuit inside an **if-then-else** command only takes an input and produces an output on clock ticks for which the condition is true. For example, the command

```
if b then counter ≺ reset
     else counter ≺ reset
```

is not equivalent to *counter* ≺ *reset*, because it maintains two counters, only one of which is reset or incremented on each clock tick (depending on the value of b on that tick).

7. CONCLUSION

Arrow types, as defined by Hughes, or the equivalent Freyd-categories defined by Power and Thielecke, provide useful expressiveness beyond that of monads. We have shown

how a simple and useful arrow-based sublanguage may be embedded in the functional language Haskell. As with the monadic style, we can use the full machinery of the host language in defining new operators, and thus defining new sublanguages. We have explored three examples here, but expect that many existing DSLs could be simplified and strengthened by being recast in this form.

8. ACKNOWLEDGEMENTS

I am indebted to John Hughes, Levent Erkök and John Launchbury for discussions on parts of this work.

Sven Panne, Simon Marlow and Keith Wansborough wrote the Haskell parser and pretty-printer on which I based the prototype preprocessor. The preparation of this paper was greatly eased by Ralf Hinze's *lhs2TEX* program.

9. REFERENCES

[1] K. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Conference*, pages 307–314, 1968.

[2] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*. ACM, 1998.

[4] R. Blute, J. Cockett, and R. Seely. Categories for computation in context and unified logic. *Journal of Pure and Applied Algebra*, 116:49–98, 1997.

[5] M. Carlsson and T. Hallgren. Fudgets – a graphical user interface in a lazy functional language. In *FPCA*, pages 321–330. ACM Press, 1993.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, pages 178–188, Munich, 1987.

[7] R. Douence and P. Fradet. Towards a taxonomy of functional language implementations. In *Programming Languages: Implementations, Logics and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 34–45, 1995.

[8] L. Erkök and J. Launchbury. Recursive monadic bindings. In *ICFP*, 2000.

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

[10] M. Hasegawa. Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In *Proceedings, Third International Conference on Typed Lambda Calculi and Applications (TLCA'97)*, volume 1210 of *Lecture Notes in Computer Science*. Springer, 1997.

[11] P. Hudak. Modular domain specific languages and tools. In *International Conference on Software Reuse*, 1998.

[12] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[13] G. Hutton and E. Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, 1998.

[14] G. Jones and M. Sheeran. Collecting butterflies. Technical Monograph PRG-91, Oxford University Computing Laboratory, Programming Research Group, Feb. 1991.

[15] G. Jones and M. Sheeran. Designing arithmetic circuits by refinement in Ruby. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 208–232. Springer, 1993.

[16] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996.

[17] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74*. North Holland, 1974.

[18] R. Ladner and M. Fischer. Parallel prefix computation. *J. ACM*, 27:831–838, 1980.

[19] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–164, 1966.

[20] J. Launchbury, J. Lewis, and B. Cook. On embedding a microarchitecture design language within Haskell. In *International Conference on Functional Programming*. ACM, 1999.

[21] J. Misra. Powerlist: A structure for parallel recursion. *ACM Trans. Prog. Lang. Syst.*, 16(6):1737–1767, Nov. 1994.

[22] E. Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science*. IEEE, 1989.

[23] J. Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Dept. of Computing Science, Chalmers University of Technology, 1999.

[24] S. Peyton Jones, J. Hughes, et al. Haskell 98: A non-strict, purely functional language, Feb. 1999.

[25] A. Power. Elementary control structures. In *CONCUR'96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 1996.

[26] J. Power and E. Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5):453–468, Oct. 1997.

[27] J. Power and H. Thielecke. Closed Freyd- and kappa-categories. In *ICALP*, volume 1644 of *LNCS*. Springer, 1999.

[28] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer, 1996.

[29] W. Wadge and E. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

[30] P. Wadler. The essence of functional programming. In *19th ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, NM, Jan. 1992.