Generalized Arrows

Adam Megacz Joseph

Generalized Arrows

by

Adam Megacz Joseph

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor John Wawrzynek, Chair
Professor George Necula
Professor George Bergman

Spring 2014

Generalized Arrows

Abstract

Generalized Arrows

by

Adam Megacz Joseph

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John Wawrzynek, Chair

Multi-level languages and arrows both facilitate metaprogramming, the act of writing a program which generates a program. The `arr` function required of all arrows turns arbitrary metalanguage expressions into object language expressions; because of this, arrows may be used for metaprogramming only when the object language is a superset of the metalanguage.

This thesis introduces *generalized arrows*, which are less restrictive than arrows in that they impose no containment relationship between the object language and metalanguage; this allows generalized arrows to be used for *heterogeneous* metaprogramming. This thesis also establishes a correspondence between two-level programs and one-level programs which take a generalized arrow instance as a distinguished parameter. A translation across this correspondence is possible, and is called a *flattening transformation*.

The flattening translation is not specific to any particular object language; this means that it needs to be implemented only once for a given metalanguage compiler. Support for various object languages can then be added by implementing instances of the generalized arrow type class; this does not require knowledge of compiler internals. Because of the flattening transformation the users of these object languages are able to program using convenient multi-level types and syntax; the conversion to one-level terms manipulating generalized arrow instances is handled by the flattening transformation.

A modified version of the Glasgow Haskell Compiler (GHC) with multi-level types and expressions has been produced as a proof of concept. The Haskell extraction of the Coq formalization in this thesis have been compiled into this modified GHC as a new flattening pass.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation, Overview, and Contribution

Robin Milner famously said "well typed programs don't go wrong" [Mil78]. To "go wrong" means to exhibit undefined behavior. For example, programs written in strongly statically typed programming languages cannot crash, write to random memory locations, or read past the end of a buffer. In the study of programming languages it is generally accepted that these are undesirable behaviors.

Higher-order functions are functions which take functions as arguments. Languages which facilitate the use of higher-order functions are called higher-order languages. Strong static typing and higher-order functions often occur together in the same language. This consonance can be explained by both practical engineering concerns as well as mathematical foundations in category theory. Languages with these features are called "HOT" (Higher Order Typed) languages; they include Haskell, ML, OCaml, Scala, and Coq.

A program which writes another program is called a *metaprogram*, and the program it writes is called the *object program*. Languages which facilitate the writing of metaprograms are called *metaprogramming languages*. Higher order functions are especially prevalent in metaprogramming languages, and in fact the use of higher-order functions has been construed as a kind of metaprogramming since the earliest days of LISP [Wan86]. When higher-order functions are used for metaprogramming both the metaprogram and the object program belong to the same language; this situation is called *homogeneous metaprogramming*. When the metalanguage and object language differ it is called *heterogeneous metaprogramming*. One recently-popularized application of heterogeneous metaprogramming is so-called embedded domain-specific languages (EDSLs).

When strongly statically typed languages are used for metaprogramming, something more than the Milner Property is desired: it should be the case that *strongly statically typed programs never create object programs which go wrong.* If the object language is strongly statically typed this property can be reduced to making sure that *strongly statically typed metaprograms never create ill-typed object programs*; this is sometimes called the Milner Property for Metaprograms. Metalanguage type systems which provide this guarantee are called *multi-level type systems*, and the languages that use them are called strongly typed multi-level languages.

At the time that this work was first published [Meg10; Meg11a; Meg11b] all known multi-level type systems were either for homogeneous metaprogramming or else custom-designed for a specific metalanguage and a specific object language. As Sheard and Linger [SL07] wrote "there is usually only a single object-language, and it must be built into the meta-language" – a general construction to produce a multi-level type system for arbitrary combinations of metalanguages and object languages did not exist. The principal contribution of this thesis is such a construction.

Viewed from the perspective of category theory this construction leads immediately to a *flattening transformation* which turns multi-level programs into ordinary one-level programs in a type-preserving and semantics-preserving (but certainly not readability-preserving) way. Viewed from the perspective of type theory the construction closely resembles *abstraction elimination* with a particular combinator basis. The flattened representation is based on a data structure (and category) that I choose to call a *generalized arrow*, since every arrow is a generalized arrow in a unique way, and every generalized arrow capable of *homogeneous* metaprogramming is an arrow. Generalized arrows *generalize arrows* to include heterogeneous metaprogramming.

Since the above-mentioned construction and flattening transformation are not specific to any particular object language the corresponding compiler pass can be implemented *once* per compiler and then reused for metaprogramming with any number of different object languages. Critically, adding a new object language does not require making modifications to the compiler or understanding compiler internals; it requires only implementing an instance of the generalized arrow type class. The flattening transformation encapsulates all the required knowledge of compiler internals, and the generalized arrow type class exposes the required information about the object language. To summarize, the flattening pass provides object-language-agnostic metaprogramming without sacrificing the Milner Property for Metaprograms. This flattening pass – and a procedure for adding such a pass to other compilers – is the principal practical benefit of this thesis.

1.2 Organization of this Thesis

The rest of this thesis is divided into four chapters. Each of the first three chapters explains generalized arrows from a different perspective:

1. *Programs* explains generalized arrows largely by example, from the programmer's perspective. This section does not include any proofs or formalizations; it is intended to be the most approachable and to give the best idea of the big picture.
2. *Categories* explains generalized arrows from a category theoretic perspective. This section is the most detailed and is formalized in Coq with machine-checked proofs.
3. *Compilers* explains generalized arrows from the compiler's perspective, focusing on the *flattening transformation* and this thesis' accompanying implementation of it as an additional pass for the GHC Haskell compiler. The implementation was extracted from the Coq formalization.

The fourth chapter covers an extended example application, a bit-serial SHA-256 circuit and its realization on an FPGA.

1.3 Background

Metaprogramming, the practice of writing programs which construct and manipulate other programs, has a long history in the computing literature, going back to the early days of LISP. However, the LISP `eval` primitive lacked access to the state of the current process; Smith [Smi83] remedied this, but did so via a definition which was self-referential in the sense that it used reflection to explain reflection. Wand later [Wan86] provided a non-circular explanation of this phenomenon.

Prior to [PL91] little of it dealt with metaprogramming in a statically typed setting where one wants to ensure not only that "well typed programs cannot go wrong," [Mil78] but also that well typed metaprograms *do not produce ill-typed object programs.*

One of the great successes of statically typed metaprogramming has been the use of the Kleisli category of a monad to provide a common abstraction for different *notions of computation* [Mog91]. This theory gave rise to a direct implementation in Haskell [Wad92] which rapidly gained popularity. Both the theory and implementation were later generalized: the Kleisli category of computations was generalized to *premonoidal* categories of computations by Power and Robinson [PR97], the use of monads in functional programming was generalized to arrows by Hughes [Hug00]. Because adding a new object language involves nothing more than implementing the functions required by the arrow type class, this approach to metaprogramming makes it quite easy to *provide* new object languages. Although all object languages share a common syntax [Pat01; LWY10], this syntax and its static semantics are profoundly different from those of the metalanguage, which can make it difficult to *use* new object languages.

By contrast, level annotations embed an object language in a manner which shares the binding, scoping, abstraction, and application mechanisms of the metalanguage. However, the type system of the metalanguage must reify the type system of the object language, so adding a new object language is quite difficult and generally requires making modifications to the compiler. Level annotations were originally motivated by the annotations used for manual binding-time analysis; these annotations and their use for explicit staging were first codified in MetaML [TS00] which was developed in quite a large number of subsequent papers, culminating in a full-featured implementation called MetaOCaml [LT06]. A major advance was the introduction of *environment classifiers* by Taha and Nielsen [TN03]. By careful use of these classifiers – type variables which are never instantiated except with other type variables – one can express, in the type system, the fact that a given term from a later stage has no free variables. This, in turn, can be used to ensure at compile time that the `run` operation is used safely. More recently, work has been done on inference for these classifiers [CMT04]. Languages offering these annotations are called *multi-level* languages [GJ91; GJ96; Dav96]; those which also offer the ability to to execute object-language expressions or persist metalanguage terms across level boundaries are called *multi-stage* languages [TS00].

It would seem that arrows and multi-level languages are used for very similar purposes. This leads us to wonder: how are they related? Is one a special case of the other, or do they derive from a common generalization? These are the questions which this thesis seeks to answer. They are particularly pressing in light of the fact that each paradigm has advantages which the other lacks: arrows are easier for object language *providers*, while level annotations offer a more integrated experience to the object language *users*.

Multi-level languages and arrows both facilitate metaprogramming, the act of writing a program which generates a program. The `arr` function required of all arrows turns arbitrary metalanguage expressions into object language expressions; because of this, arrows may be used for metaprogramming only when the object language is a superset of the metalanguage. This restriction is also present in multi-level languages which offer unlimited cross-level persistence.[1]

This thesis introduces *generalized arrows* and proves that they generalize arrows in the following sense: every arrow in a programming language arises from a generalized arrow with that language's term category as its codomain (Theorem 3.2.4.4). Generalized arrows impose no containment relationship between the object language and metalanguage; they facilitate *heterogeneous* metaprogramming. The category of generalized arrows and the category of reifications (i.e., multi-level languages) are isomorphic categories (Theorem 3.3.0.18). This is proven formally in Coq, and the proof is offered as justification for the slogan "*multi-level languages are generalized arrows.*"

This proof can be used to define an invertible translation from two-level terms to one-level terms parameterized by a generalized arrow instance. This is ergonomically significant: it lets object language *providers* write generalized arrow instances while the *users* of those object languages write multi-level terms. This is beneficial because implementing a generalized arrow instance is easier than modifying a compiler, whereas writing two-level terms is easier than manipulating generalized arrow terms.

A modified version of GHC with multi-level terms is offered as a proof of concept; the Haskell

[1]The converse restriction, that the metalanguage is a superset of the object language, is imposed for multi-*stage* languages – those multi-level languages with a `run` construct.

extraction of the Coq proofs mentioned above has been compiled into this modified GHC, and is made available as a new flattening pass.

Chapter 2

Programs

This chapter approaches generalized arrows from a programmer's perspective. It consists mostly of examples and practical perspective; it contains no formalizations, theorems or proofs.

The first section briefly reviews Haskell arrows as exposed in the `Arrow` type class from the Haskell standard library. The second section introduces generalized arrows as exposed in the proposed `GArrow` type class. The third section introduces multi-level types and expressions and a proposed syntax extension to allow multi-level types and expressions in Haskell. Each of these sections repeats the same three running examples:

- The `pow` function (exponentiation on the natural numbers) chosen as the simplest interesting example.
- Bidirectional programming, chosen as a simple example in which classic arrows cause problems and generalized arrows do not.
- Hardware design, the original motivation for this work and its best showcase due to the extreme contrast between the power of the metalanguage (arbitrary Haskell) and the object language (gate-level realizations of electronic circuits).

The fourth section discusses the *flattening transformation*, a type-preserving translation from multi-level expressions to expressions parameterized by an instance of the `GArrow` type class.

2.1 Arrows in Haskell

In Haskell, the `Arrow` class is used to represent abstract operations. If a type operator `a` is arrow, then for any input type `x` and output type `y`, there will be a type `a x y` of "abstract

```
class Category a where
  id    :: a x x
  (>>>) :: a x y -> a y z -> a x z

class Category a => Arrow a where
  arr   :: (x -> y) -> a x y
  first :: a x y -> a (x,z) (y,z)
```

Figure 2.1: The `Category` and `Arrow` classes from the Haskell standard library.

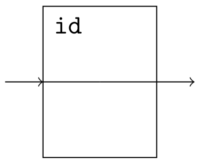
operations" and an implementation of the `Arrow` type class (shown in Figure 2.1) from the Haskell standard library.

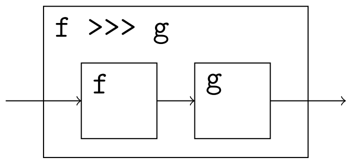
The `Arrow` class is a subclass of `Category`, so any instance of `Arrow` must implement four functions: `id` and `>>>` from `Category` as well as `arr` and `first` from `Arrow`. Here is a brief description of the two functions in the `Category` class:

- The `id` function of class `Category` is the *identity*. It can be thought of as an abstract operation that takes an input and returns it, unchanged, as its output. The type `a x x` indicates that for a given category `a` and any input type `x` the `id` function will produce an output of type `x`.
- The `>>>` function of class `Category` is the *composition* function. It can be thought of as taking two operations `f :: a x y` and `g :: a y z` and producing their composite operation `f>>>g :: a x z`. Notice that the "output" type of `f` must match the "input" type of `g`.

It is often useful to visualize these abstract operations using boxes-and-wires diagrams. There is a formal way to do this using *Penrose diagrams*, which were first introduced in [Pen71]; a recent summary of various extensions to the notation can be found in [Sel11].

Here are the Penrose diagrams for `id` and `>>>`:





Implementations of `Category` are required to satisfy the laws in Figure 2.2. The first two laws require that composing any operation `f` with `id` from either the left or right leaves `f` unchanged, and the third rule requires that when composing three operations `f`, `g`, and `h` it doesn't matter whether `f` and `g` or `g` and `h` are composed first – the result is the same in either case.

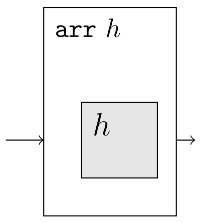
The `Arrow` class extends `Category` with two additional functions: `first` and `arr`.

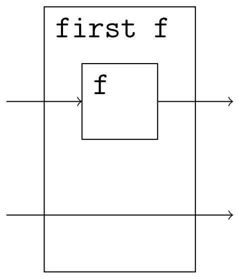
```
      id >>> f = f
               f = f >>> id
(f >>> g) >>> h = f >>> (g >>> h)
```

Figure 2.2: The category laws.

- The `first` function of class `Arrow` takes any abstract operation with input type `x` and output type `y` and produces a new operation that has an extra input and extra output, each of type `z`. The new abstract operation is just like the original operation, except that the extra input is passed unchanged to the extra output.
- The `arr` function of class `Arrow` lifts any Haskell function of type `x->y` into an abstract operation with input type `x` and output type `y`. It is important to note that this means that *any* Haskell function can be lifted into *every* arrow.

Below are the Penrose diagrams for the `arr` and `first` functions. The boxes labeled `f` and `g` in the previous diagram represented subdiagrams; the box labeled h below is fundamentally different: it is a placeholder not for an arrow expression, but for an arbitrary Haskell expression. It is shaded gray in order to emphasize this distinction.





The `arr` function hands quite a bit of power to the user of the arrow, who has access to the entire Haskell language “inside” the arrow. This places a great burden on the implementor of the arrow – the implementor must ensure that the arrow supports the entire Haskell language. Many arrows are derived from Haskell in some way and in those cases the requirement can often be satisfied without much trouble using higher order functions. However for other arrows this can be difficult or even impossible.

Instances of the `Arrow` class must obey the laws shown in Figure 2.3. The arrow laws formalize basic intuitions about how boxes-and-wires diagrams ought to work, particularly when arbitrary Haskell functions can be turned into boxes. For example, the first two laws require that the arrow’s `id` and `first` operations are the liftings of the “obvious” Haskell functions. The third law requires that lifting the composition of two Haskell functions gives the same result as composing their individual liftings. The fourth law requires that composing two operations and then adding an extra argument gives the same result as adding the extra argument to each operation individually before composing. The fifth law requires that the order of composition does not matter if neither operation uses the other’s output as its input

```
                      arr (λx -> x) = id
                      first (arr f) = arr (λ(x,y) -> (f x, y))
                arr (λx -> f (g x)) = arr f >>> arr g
                    first (f >>> g) = first f >>> first g
first f >>> arr (λ(x,y) -> (x,g y)) = arr (λ(x,y) -> (x,g y)) >>> first f
                first f >>> arr fst = arr fst >>> f
      first (first f) >>> arr assoc = arr assoc >>> first f
                                        where assoc (x,(y,z)) = ((x,y),z)
```

Figure 2.3: The arrow laws, from [Pat01].

and one of the operations is the result of lifting with `arr`. The last two laws ensure that inputs and outputs can be re-arranged without unexpected consequences.

2.1.1 The `pow` Function using Arrows

Consider the `pow` function, which computes the exponentiation $\mathtt{x}^{\mathtt{n}}$ where both `x` and `n` are natural numbers:

```
pow :: Int -> (Int -> Int)
pow n = if n==0
        then λx -> 1
        else λx -> x * (pow (n-1) x)
```

This function works in the usual way: if the exponent is zero, the `pow` function produces the result `λx -> 1`, a function which always returns `1` since any number raised to the power of `0` is `1`. If the exponent is nonzero, `pow` calls `pow (n-1)` to get the function that raises `x` to the power of `n-1`; it then uses that function on `x` and multiplies the result by `x` one additional time.

The `pow` function takes two natural-number arguments (`x` and `n`) and produces a natural-number result. It is also valid to think of `pow` as taking a *single* natural-number argument `n` and returning a *function* which is specialized to the task of raising natural numbers `x` to that particular power. For example, the `cube` function below is the result of applying `pow` to `3`:

```
cube :: Int -> Int
cube = pow 3
```

In Haskell both of these perspectives – a two-argument function returning numbers or a one-argument function returning functions – are equally valid and interchangeable. However sometimes the programmer wants to be specific about which case is intended. For example, a program that synthesizes circuits for computing exponentiation with a fixed exponent is very different from a single circuit that computes arbitrary exponentiations. An even simpler example comes from bitwise rotations: the circuit to left-rotate a word by 7 bits is nothing more than some wires, whereas the circuit to left-rotate a word by an arbitrary number of

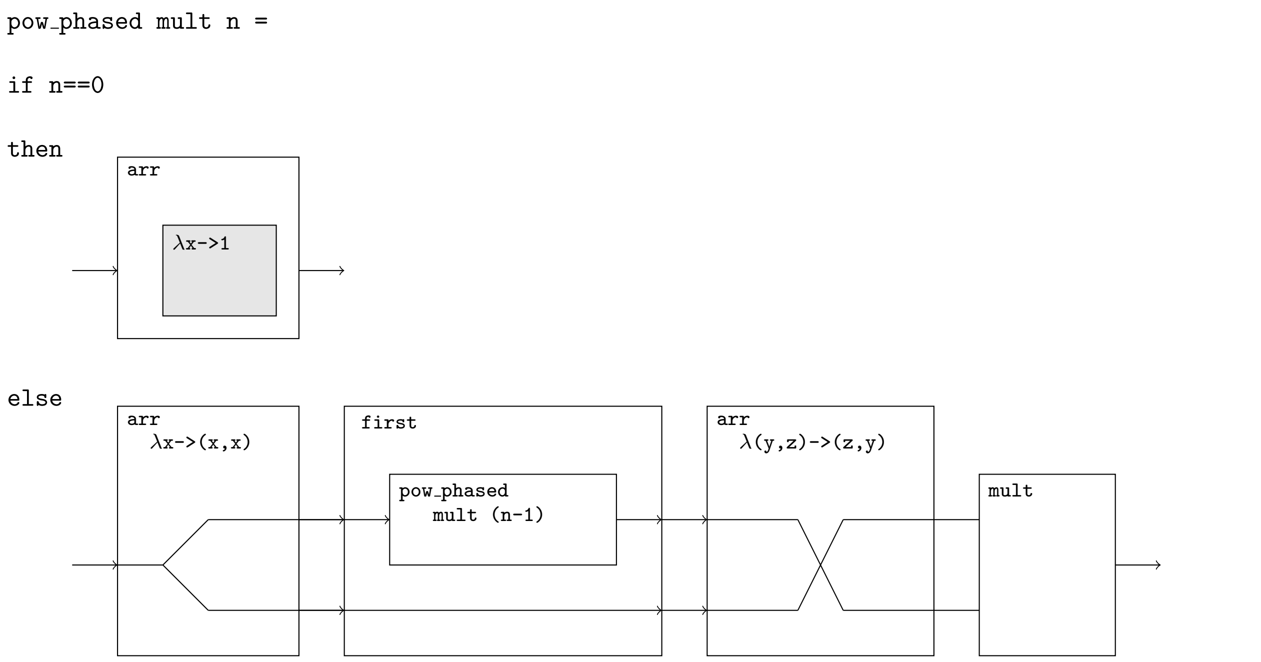


Figure 2.4: Penrose diagram for the `pow_phased` function.

bits (a "barrel shifter") is quite complex and subject to many subtle design tradeoffs. Arrows make it possible to express this difference, which is called a *phase distinction.*

Before going any further, it is necessary to make the `pow` function a bit more general. The `pow` function above uses the multiplication function from the Haskell standard library. Instead of being hardwired to use the standard library's integer multiplication function, the `pow` function can be rewritten for use with *any* multiplication function by adding an extra argument `mult` of type `(Int,Int)->Int`:

```
pow :: ((Int,Int) -> Int) -> Int -> (Int -> Int)
pow mult n = if n==0
             then λx -> 1
             else λx -> mult (x, (pow mult (n-1) x))
```

It is now possible to implement this more-general version of `pow` using arrows in two different ways; this will illustrate how arrows are able to capture phase distinctions. The first and most direct implementation simply lifts the `pow` function using `arr`:

```
pow_lifted :: Arrow a => ((Int,Int) -> Int) -> a (Int,Int) Int
pow_lifted mult = arr (uncurry (pow mult))
```

The other way to implement `pow` using arrows is to write a function that produces an arrow specialized to a particular value of `n`:

```
pow_phased :: Arrow a => a (Int,Int) Int -> Int -> a Int Int
pow_phased mult n =
                if n==0
                then arr (λx -> 1)
                else arr (λx -> (x,x))              >>>
                     first (pow_phased mult (n-1)) >>>
                     arr (λ(y,z) -> (z,y))         >>>
                     mult
```

The corresponding Penrose diagram is shown in Figure 2.4.

Writing arrow expressions by hand can be tedious and the resulting code is not easy to read, mostly because it includes expressions like `arr (λx -> (x,x))` and `arr (λ(y,z) -> (z,y))` which serve only to duplicate and re-arrange inputs. To avoid this unnecessary clutter, the Haskell language includes a special `proc`-syntax for arrows [Hug04, Section 3.4][Tea09, Section 7.10], derived from the `do`-syntax for monads. Using `proc` syntax, the `pow` program can be rewritten like this:

```
pow_phased_proc mult n =
    if n==0
    then proc x ->
         returnA -< 1
    else proc x ->
      do y       <- pow_phased_proc mult (n-1) -< x
         ret     <- mult                       -< (x,y)
         returnA -< ret
```

When written in this form, the programmer does not have to manually insert expressions to duplicate, rearrange, or discard inputs – the Haskell compiler reconstructs the necessary manipulations from the set of variables in scope at each subexpression.

Notice how `pow_lifted` and `pow_phased` have different types:

```
pow_lifted :: Arrow a =>  ((Int,Int) -> Int)        -> a (Int,Int) Int
pow_phased :: Arrow a => a (Int,Int)    Int  -> Int -> a      Int  Int
```

The phase distinction appears not only in the program, but also in its type. Two things are important to note. First, notice that `pow_phased` asks for the `mult` parameter to be supplied as a two-input, one-output arrow operation of type `a (Int,Int) Int` rather than a Haskell function of type `(Int,Int) -> Int`. Since values of the latter type can always be turned into values of the former type (using `arr`) this means that `pow_phased` is *less* demanding of its caller and therefore *more* general.

Second, notice that `pow_phased` takes the `n` argument as a parameter of type `Int` and returns an arrow. On the other hand, `pow_lifted` takes no arguments (other than `mult`), but the arrow it returns has more inputs (two instead of one).

To summarize, the `pow_phased` function expresses a phase distinction between the arrival of the `n` parameter and the execution of the exponentiation computation. This separation

```
class Arrow b => BiArrow b where
  biarr :: (x->y) -> (y->x) -> b x y
  inv   :: b x y -> b y x
```

Figure 2.5: The `BiArrow` class, from the Haskell standard library.

is exposed in its type: it takes `n` as an ordinary Haskell argument and returns a one-input, one-output arrow. On the other hand the `pow_lifted` function accepts both arguments "inside" the arrow, so its type ends with a two-input, one-output arrow.

2.1.2 Bidirectional Programming using Arrows

`BiArrows` are meant to represent abstract operations which are invertible. They were introduced in [Ali+05] and further examined in [JHI09]. A `BiArrow` is an instance of the `Arrow` subclass in Figure 2.5. The `BiArrow` class adds a new constructor `biarr`, which is intended to be used in place of `arr`. It takes a pair of functions which are assumed to be mutually inverse. The `inv` function attempts to invert a `BiArrow` value.

Types belonging to the class `BiArrow` consist of operations which *might be* invertible. Some `BiArrow` operations are actually not invertible, so the `inv` operation is partial and may fail at runtime. Since the type of `inv` does not include any way of reporting such failures (e.g., an `Either` type or `Error` monad), the behavior in these circumstances is undefined and must appeal to mechanisms outside the semantics of the language (e.g., `Prelude.error`). The type system is not capable of ensuring that "well-typed programs cannot go wrong" [Mil78] in this way[1]. Moreover, nothing stops a program from passing a `BiArrow` to a function whose type is `Arrow a=>...`. Such a function would have no reason to suspect that using `arr` rather than `biarr` in such a situation is dangerous. Not only can the use of `BiArrows` lead to undefined behavior, this behavior may be triggered by the composition of two programs which would each have been completely acceptable in isolation.

Unfortunately there is no way to fix this within the framework of arrows, because the arrow type class requires that `arr` be defined for arbitrary functions – even those which cannot possibly have an inverse. For example, consider the arrow program for `pow` given in the previous section. It included the subexpression `arr` λ`x -> 1`; this is an arrow operation which, no matter what input is provided, will always produce `1` as its output. Such a function cannot have an inverse.

Not even the most powerful dependent type system can help with this problem: any restriction

[1]Note that aside from failures of `inv` there is a second, separate, issue: the user of a `BiArrow` might supply a pair of functions to `biarr` which are not in fact mutually inverse to each other. However, in this case the program will simply compute an incorrect result rather than "going wrong" via an extra-semantical failure mechanism. Preventing this sort of problem is possible only with a type system rich enough to express facts about equivalence of functions, which typically requires fairly powerful dependent types (such as Coq's). Generalized arrows do not address this second issue.

```
class Arrow a => ArrowLoop a
  where
    loop :: a (x,y) (x,z) -> a y z
```

Figure 2.6: The `ArrowLoop` class, from the Haskell standard library

```
class ArrowLoop a => ArrowCircuit a
  where
    delay :: x -> a x x
```

Figure 2.7: The `ArrowCircuit` class, from [Pat01]

on the use of `arr` would result in an implementation which was no longer an `Arrow`. For example, suppose we want to call a function which demands a `BiArrow` argument:

```
someFunctionWeWillCall :: BiArrow b => b Int Int -> WorldPeace
```

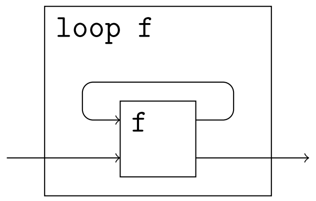
Nothing prevents calling the function like this:

```
bogus :: BiArrow b => WorldPeace
bogus = someFunctionWeWillCall (arr (λx -> 1))
```

Clearly the value `arr (λx -> 1)` passed to `someFunctionWeWillCall` isn't invertible, yet the type system didn't prevent the call. The problem is `arr` – it creates a "loophole" that can be used to sneak non-invertible functions into the `BiArrow`.

2.1.3 Circuits using Arrows

One of the very first applications of arrows was synchronous circuits – that is, circuits with a single global clock. In order to build interesting circuits it is necessary to be able to express feedback, and there is an `Arrow` subclass called `ArrowLoop` in the Haskell standard library that does this; it is shown in Figure 2.6. The `loop` function's type says that it takes an operation with two inputs and two outputs where one of the input types matches one of the output types and returns an arrow without them. Here is the Penrose diagram for `loop`:



Paterson's paper [Pat01] defines arrow circuits as a further subclass: arrows with feedback (`ArrowLoop`) and a *delay* mechanism analogous to a register. The class `ArrowCircuit` for these circuits is shown in Figure 2.7. The `delay` primitive can be interpreted as a register with input type `x`, output type `x`, and initial value given by its sole argument. The following describes a register which holds an `Int` and is initialized with the value `3`:

```
threeReg :: ArrowCircuit a => a Int Int
threeReg = delay 3
```

Here is the counter circuit from Paterson's paper:

```
counter :: ArrowCircuit a => a Bool Int
counter =
   proc reset -> do rec
                 output <- idA -< if reset
                                  then 0
                                  else next <- delay0 -< output+1
```

On each clock tick the circuit will output a zero if the `reset` input is true; otherwise it increments its internal state and outputs that value. The type `ArrowCircuit a => a Bool Int` therefore represents circuits with a one-bit input and an integer-valued output.

Unfortunately this representation of circuits as arrows works *too* well, since the following function has the same type:

```
higherOrderCircuit :: ArrowCircuit a => a Bool Int
higherOrderCircuit =
   arr (λb -> head (drop 100
         (iterate collatz (if b then 3 else 4))))

collatz :: Int -> Int
collatz 1 = 1
collatz n = if n `mod` 2 == 0
            then n `div` 2
            else 3*n + 1
```

Clearly this definition cannot be realized as a circuit: it involves a function which is higher-order and recursive.[2] At issue once again is the fact that *every* arrow must allow *all* Haskell functions to be absorbed via the `arr` function – even *higher-order* Haskell functions. Just as the previous section used `arr` to sneak non-invertible functions into a `BiArrow`, this section used `arr` to sneak higher-order functions into the description of a hardware circuit.

An even more extreme example is the perfectly well-typed program below, whose type purports to take "higher-order circuit" arguments:

[2]One proposed solution is to exploit *parametricity* by keeping the `Int` and `Bool` types abstract. In this approach the type of a circuit with integer and boolean inputs/outputs becomes
`forall i . forall b . ArrowCircuit i b a => a x y`
The quantifier-bound type variables `i` and `b` stand in for `Int` and `Bool`. Unfortunately this approach causes the types of circuit transformers to become higher-rank types of the form
`(forall i . forall b . ArrowCircuit i b a => a x y) -> ...`
Since higher-rank type inference is undecidable even the Haskell compilers which support this extension require that the programmer write manual type ascriptions by hand. Even more problematic is the fact that the parametricity-based approach requires explicitly enumerating every single in-arrow datatype as part of the definition of `ArrowCircuit`, meaning that circuit designers can no longer use abstract types or modules to enforce information hiding.

```
applyArrowCircuit :: ArrowCircuit a =>
                          a Int (Bool->Bool) ->
                          a      ()    Bool ->
                          a Int        Bool

applyArrowCircuit higherOrderCircuit arg =
  arr (λz -> ((),z))       >>>
  first arg                >>>
  arr (λ(y,z) -> (z,y))    >>>
  first higherOrderCircuit >>>
  arr (λ(f, x) -> f x)
```

2.2 Generalized Arrows in Haskell

A first attempt at solving the problem of the previous sections might be to try to do away with `arr`. Unfortunately that is impossible when using arrows: the `arr` function is tightly woven into the laws which prescribe the behavior of arrows, so solving the problem is not as simple as removing `arr`. Additionally the desugaring algorithm for `proc`-syntax depends on unrestricted use of `arr`; eliminating or restricting `arr` would mean going back to writing arrow expressions by hand.

Perhaps we might admit only certain special cases of `arr`. For example, we might require some operation which behaves like `arr λx->x`, but not allow arbitrary functions like `λx->1` to be lifted. But what criteria should be used for deciding which special cases to require? It would seem unwise to make a choice based on any one particular application such invertible programming or circuit design. One way of explaining generalized arrows is as the result of searching for a well-motivated collection of definitions to fill the void left by removing `arr`.

The `GArrow` class definition is shown in Figure 2.8, and the Penrose diagrams for its functions appear in Figure 2.9. Like the `Arrow` class, the `GArrow` class is a subclass of `Category`: every `GArrow` supports the `id` and `>>>` operations, and obeys the laws (associativity and neutrality) required of a category. The `GArrow` class also has a `ga_first` function, whose type and laws are identical to those of the `Arrow` class[3], although `ga_second` is not (always) definable in terms of `ga_first`.

Unlike the `Arrow` class, the `GArrow` class has a second type parameter `**` of kind $\star$`->`$\star$`->`$\star$, called the *tensor* of the generalized arrow. This plays a role for generalized arrows analogous to the role of pairing `(,)` in the `Arrow` class: using it, one may form binary trees whose leaves are types. Conceptually, one such tree is the type of the "input" and the other is the type of the "output." In contrast to the `Arrow` class, generalized arrows do not assume that this type operator is necessarily a cartesian product. The third type parameter `u`, of kind $\star$, plays the role of `()` in arrows; it is called the *unit* of the generalized arrow and is drawn as a

[3]One could argue that `first` belongs in a class `BinoidalCategory`, a superclass of `Arrow`.

```
class Category g => GArrow g (**) u where
  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 )
```

Figure 2.8: The definition for the GArrow type class

dotted line in the Penrose diagrams. In the "trees of types" analogy above the unit type is the empty leaf.

Note the conspicuous absence of the arr function for lifting arbitrary Haskell functions into the generalized arrow. In its place are six functions: ga_{un}cancel{l,r} and ga_{un}assoc. These are used to re-arrange the inputs and outputs of a generalized arrow: the ga_{un}cancel{l,r} functions add and remove superfluous inputs and outputs (which must be of the unit type) and the ga_{un}assoc functions "re-parenthesize" the list of inputs[4]

The GArrow class has three subclasses which are used very frequently, shown in Figure 2.10. GArrowCopy allows duplication of inputs, GArrowSwap allows reordering of inputs, and GArrowDrop allows deletion of inputs of *arbitrary* type (by using ga_drop to turn them into outputs of the unit type which may then be discarded using ga_cancel{l,r}). Note that the Arrow class does not permit this distinction: one can produce functions like ga_drop, ga_copy, and ga_swap for *any* Arrow instance using the mandatory arr function.

Conceptually, the nine functions ga_{un}cancel{l,r}, ga_{un}assoc, ga_drop, ga_copy, and ga_swap exist to "put back" what generalized arrows have lost – relative to the Arrow class – as a result of eschewing a mandatory arr. This can be made slightly more concrete by taking a look at the instance declaration in Figure 4.6 which gives every Arrow instance a canonical GArrow instance.

2.2.1 The pow Function using Generalized Arrows

One last subclass of GArrow is necessary before moving on to the implementation of pow. If the exponent is 0, The pow function needs to return an abstract operation whose output is always a constant 1. There is a GArrow subclass called GArrowConstant with an instance for each GArrow and type t such that the generalized arrow is capable of lifting *constant functions* with result type t:

[4]Why use binary trees instead of lists? One motivation is the desire to avoid having to prove facts about the associativity of list-concatenation at the level of types, something which cannot be done in the type systems of many languages.

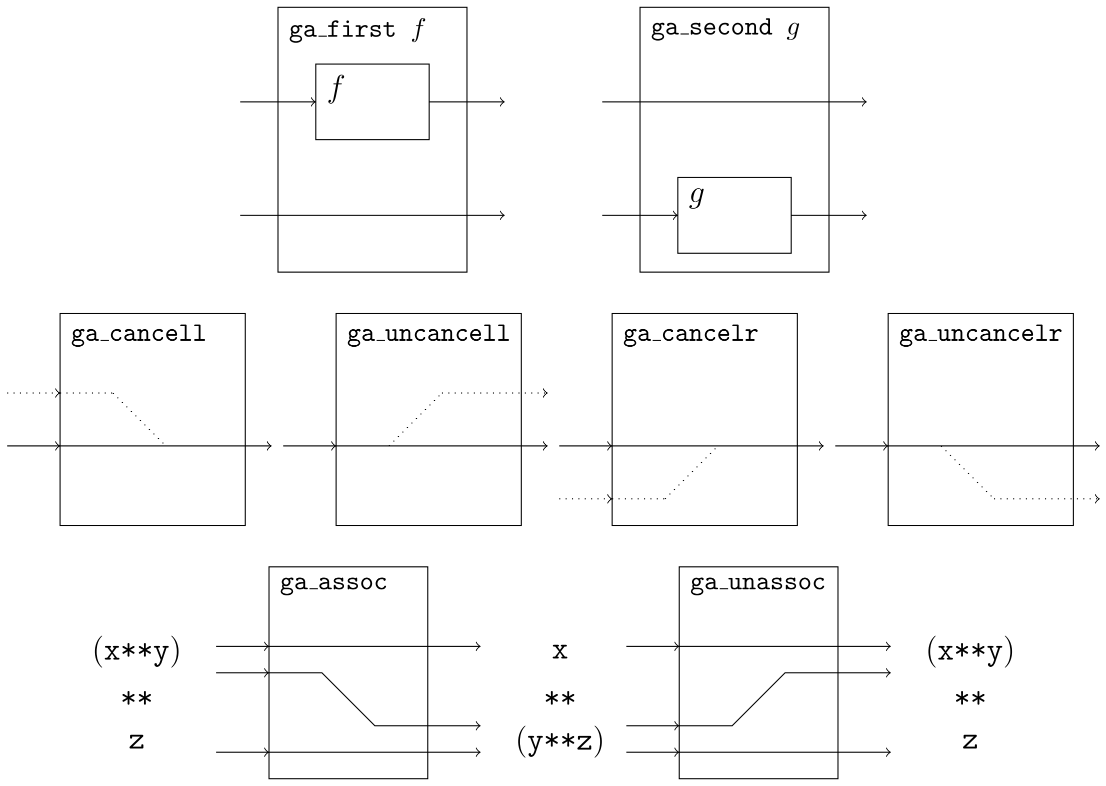


Figure 2.9: Penrose diagrams for the functions in the `GArrow` class

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

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

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

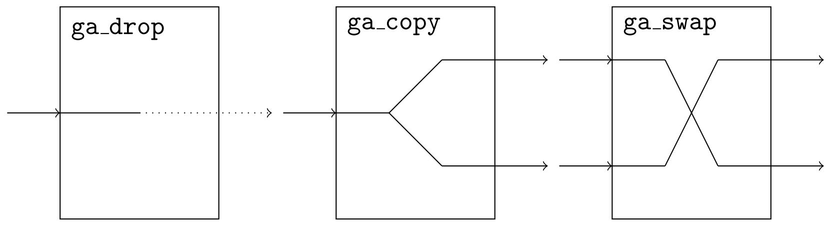


Figure 2.10: Three frequently-implemented subclasses of `GArrow` and their Penrose diagrams

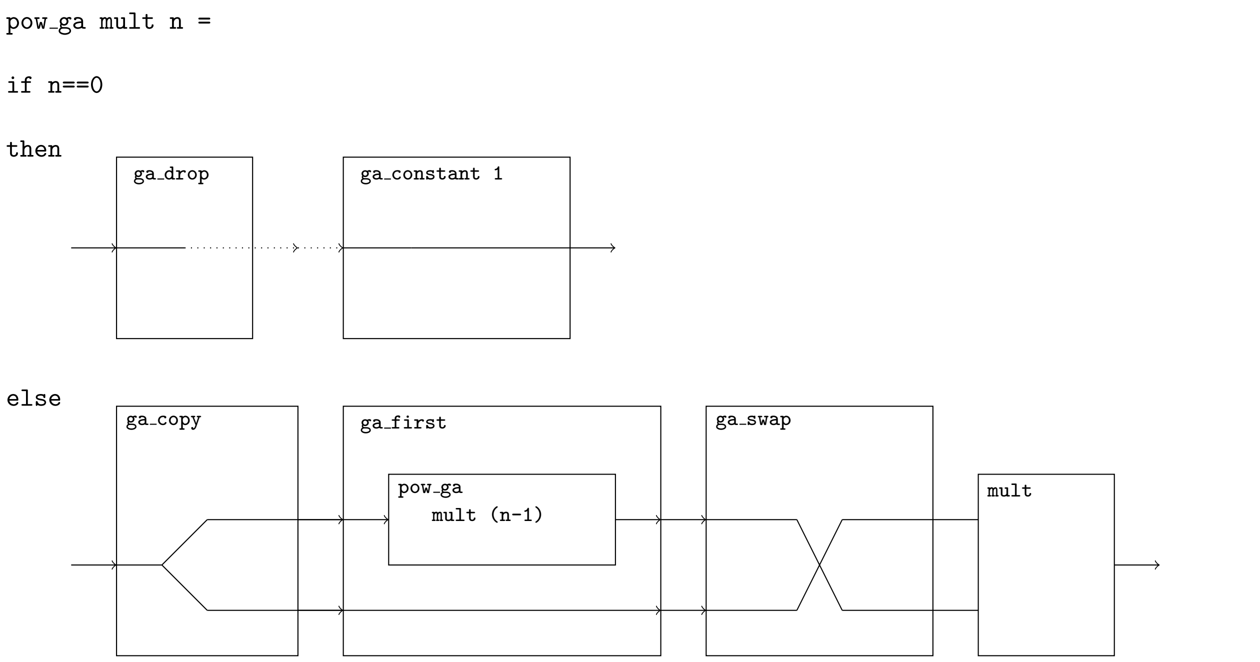


Figure 2.11: Penrose diagram for the `pow` function using generalized arrows.

```
class GArrow g (**) u => GArrowConstant g (**) u t where
  ga_constant :: t -> g u t
```

Now, here is the `pow` example again, but using generalized arrows. Unsurprisingly, it is very similar to the version using ordinary arrows.

```
pow_ga n mult = if n==0
                then   ga_drop                >>>
                       ga_constant 1
                else   ga_copy                      >>>
                       ga_first (pow_ga mult (n-1)) >>>
                       ga_swap                      >>>
                       mult
```

The corresponding Penrose diagram is shown in Figure 2.11

2.2.2 Bidirectional Programming using Generalized Arrows

It is now possible to define the class `BiGArrow`, which avoids the problems of the `BiArrow` class by using generalized arrows in place of arrows:

```
class GArrow g (**) u => BiGArrow g (**) u where
  biga_arr :: (x -> y) -> (y -> x) -> g x y
  biga_inv :: g x y -> g y x
```

Notice that the problem of non-invertible functions "leaking" in no longer exists: `biga_arr` is the only way to get Haskell functions into the `BiGArrow` (there is no `arr`) and it requires that functions be supplied in matched pairs.

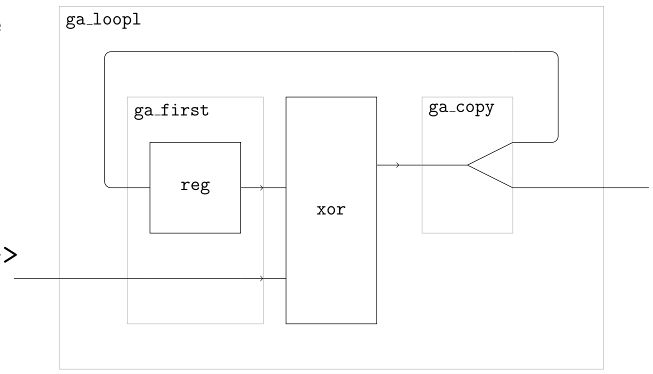
An implementation of `BiGArrow` should not also implement the class `GArrowConstant`, since `ga_constant` produces non-invertible operations (except at trivial types such as `()`). Separating `GArrowConstant` from the primary `GArrow` class enables this distinction.

2.2.3 Circuits using Generalized Arrows

Circuits can be described with generalized arrows in much the same way as with arrows. Shown below is an example oscillator circuit which outputs the exclusive or of its input and its previous output:

```
class GArrowLoop g => Hardware g where
  -- ...
  xor :: g Bit Bit
  reg :: g Bit Bit

oscillator :: Hardware g => g Bit Bit
oscillator = ga_loopl (ga_first reg >>>
                          xor >>>
                          ga_copy)
```



2.3 Two-level Languages

The `pow` example using generalized arrows in Section 2.2.1 was difficult to read and unnecessarily tedious to write, much like the `pow` example using ordinary arrows without the `proc`-syntax. Is there something similar to `proc`-syntax for generalized arrows?

In the case of ordinary arrows the language internal to the arrow is a superset of Haskell, so the `proc`-syntax is fairly lightweight; it adds a few extra constructs (`proc`, `-<`, and `<-`) but is able to recycle the Haskell expression grammar for most of its needs. In the case of generalized arrows it is no longer safe to assume that the internal language of the generalized arrow is a superset of Haskell, so the solution is a bit more difficult, and will require a detour into multi-level languages.

Multi-level languages involve two additional expression forms: brackets and escape. Brackets, written `<{...}>`, serve to demarcate expressions of the object language. For example, the definition below is the metalanguage representation of an object language's identity function:

```
  object_id = <{ λx -> x }>
```

The escape form, written `~~e`, serves to insert one object language expression into another.

For example, the definition below is the metalanguage function which composes two object language functions:

```
  object_comp f g = <{ λx -> ~~f (~~g x) }>
```

Multi-level languages also introduce an additional type form: code types, written `<{t}>@`α for some type `t` and environment classifier α [TN03]. Expressions wrapped in code brackets are given a type of this form, and expressions subject to the escape operator must bear a type of this form. This means that `object_comp` above has the following type:

```
  object_comp :: forall c. <{y->z}>@c -> <{x->y}>@c -> <{x->z}>@c
```

To be slightly more formal, adding multi-level types and expressions involves extending the Haskell grammar by adding expression forms `<{`e`}>` for homogeneous code-bracketing and `~~`e for escape, as well as code types `<{`τ`}>@`α, as shown below:

e ::= ... | `<{`e`}>` | `~~`e | ... *(expressions)*

τ ::= ... | `<{`τ`}>@`α | ... *(types)*

2.3.1 The `pow` Function using a Two-Level Language

Here is the `pow` example once again, this time using homogeneous code brackets:

```
pow_leveled :: <{ (Int,Int) -> Int }>@a -> Int -> <{ Int -> Int }>@a
pow_leveled mult n =
  if n==0
  then <{ λx -> 1 }>
  else <{ λx -> ~~mult (x, (~~(pow_leveled mult (n-1)) x)) }>
```

Notice the similarity between the type of `pow_leveled` and the type of `pow_phased`:

```
pow_phased  :: Arrow a => a (Int,Int)    Int      -> Int ->  a Int    Int
pow_leveled ::           <{ (Int,Int) -> Int }>@a -> Int -> <{ Int -> Int }>@a
```

2.3.2 Bidirectional Programming using a Two-Level Language

It reasonably straightforward to eliminate irreversible functions from the object language of a two-level language. In the Haskell type system the rule of *structural weakening* allows typing of expressions which discard or ignore inputs:

$$\frac{\Gamma \vdash e{:}\tau}{x : \tau', \Gamma \vdash e{:}\tau}\ \textsf{[Weak]}$$

In prose, this rule says that assuming e has type τ in context Γ, one can conclude that e has that same type τ in a context enlarged by an extra assumption $x : \tau'$. This allows typing of expressions that ignore some of the variables which are in scope; for example:

```
constantOne :: Int -> Int
constantOne = λx -> 1
```

This program is well-typed one-level Haskell according to the following derivation:

$$\dfrac{\dfrac{\dfrac{}{\vdash \texttt{1:Int}}\,[\textsf{Const}]}{\texttt{x : Int} \vdash \texttt{1:Int}}\,[\textsf{Weak}]}{\vdash \lambda\texttt{x->1:Int->Int}}\,[\textsf{Abs}]$$

In order to eliminate irreversible object language programs, all that is required is to forbid typing proofs for expressions of the form <{e}> from using the rule [Weak]; there are many ways to accomplish this. So a compiler for metaprograms which produce reversible object programs should reject a program like this:

```
constantOneObjectProgram = <{ λx -> 1 }>
```

Of course, the compiler will still accept the program `constantOne` since its body is an expression in the metalanguage, and the goal is to allow *unrestricted* metaprograms to produce only *reversible* object programs. Importantly, however, the phase distinction between the object language and metalanguage prevents functions like `constantOne` from leaking into the object language. For example, the program below is rejected by the compiler:

```
constantOneSneaky = <{ constantOne }>
```

The compiler rejects this program because `constantOne` is an identifier bound at the level of the metalanguage; it is not in scope[5] within the code brackets.

2.3.3 Circuits using a Two-Level Language

Unfortunately, the homogeneous code brackets are too powerful: they allow arbitrary λ-calculus expressions in the object language. This means that it is possible to write two-level expressions corresponding to "higher-order circuits":

```
  applyCircuit :: <{ (Bool -> Bool) -> Bool }>@a ->
                  <{  Bool -> Bool          }>@a ->
                  <{  Bool -> Bool          }>@a
  applyCircuit =
    <{ λhigherOrderCircuit -> λarg -> higherOrderCircuit arg }>
```

The expression above is syntactically correct and well-typed, yet cannot be translated into meaningful hardware. The problem is that λ-calculus allows functions of *higher type*; i.e., expressions of type (τ -> τ) -> τ. Recall that the grammar for homogeneous code brackets allows arbitrary Haskell expressions e inside the homogeneous code brackets:

[5]Many homogeneous metaprogramming languages include an operators for *cross-stage persistence*[TS00] which allow importing variables bound at the level of the metalanguage into expressions at the level of the object language; this feature is clearly inappropriate for the goal of writing unrestricted metaprograms which produce reversible object programs.

e ::= ... | `<{`e`}>` | `~~`e | ... *(expressions)*

We will instead create a separate grammar production k for object-language expressions, and use a new *heterogeneous* code bracket form `<[`k`]>` to distinguish it:

e ::= ... | `<[`k`]>` | ... *(metalanguage expressions)*

k ::= x | kk | `~~`e | λx `->` k *(object language expressions)*

Not much has changed so far: object language expressions k may include variables, application, abstraction, and escaped metalanguage expressions, just as before. The first major change comes with the grammar for heterogeneous two-level types:

τ ::= `()` | τ`*`τ | `<[`τ `~~>` τ`]>@`α | ... *(types)*

The first two productions are analogous to the usual unit and product types from ordinary Haskell. The third production is the type construct for object language expressions. Notice that instead of a single type τ it now involves *two* types; these are called the *input* and *output* types. Separating the argument (input) and return (output) types in the grammar makes it possible to avoid the use of the general `(->)` type constructor for function spaces. More importantly, however, it gives the type system a way to ban higher-order functions by using a modified version of the abstraction typing rule, taken from Hasegawa's κ-calculus [Has95].

$$\frac{x{:}\texttt{()}\texttt{~~>}\tau_1, \Gamma \vdash k{:}\tau_2\texttt{~~>}\tau_3}{\Gamma \vdash \lambda x{:}\tau_1 \texttt{ -> } k \ : \tau_1\texttt{*}\tau_2\texttt{~~>}\tau_3}\ [\kappa\text{-Abs}]$$

The key feature is the type of the variable `x` in the hypothesis: it must be in the form `()~~>`τ_1; any input to the function being typed must itself have a trivial input type. Note also the type of the expression k in the conclusion: its input type has been augmented but its output type is unchanged.

The κ-application rule is similar:

$$\frac{\Gamma \vdash k : \tau_1\texttt{*}\tau_2\texttt{~~>}\tau_3 \qquad \Gamma \vdash k_1 : \texttt{()}\texttt{~~>}\tau_1}{\Gamma \vdash k\ k_1 : \tau_2\texttt{~~>}\tau_3}\ [\kappa\text{-App}]$$

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 ]>
```

The following functional illustrates κ-abstraction; it reverses the order of the first two arguments of a function.

```
swap :: <[ a*b*c ~~> d ]>@e ->
        <[ b*a*c ~~> d ]>@e
swap f = <[ λx y -> ~~f y x ]>
```

Here is the `pow` function written using the heterogeneous code brackets:

```
pow :: <[ Int*Int*()~~>Int ]>@a -> Int -> <[ Int*()~~>Int ]>@a
pow mult n =
  if n==0
  then <[ λx -> 1 ]>
  else <[ λx -> ~~mult x (~~(pow mult (n-1)) x) ]>
```

The following attempt to write the `applyCircuit` example using `<[..]>` brackets will fail:

```
  applyCircuit =
    <[ λhigherOrderCircuit -> λarg -> higherOrderCircuit arg ]>
```

The program above is rejected by the modified GHC typechecker accompanying this thesis, which gives the following error message:

```
  Fail.hs:2:37:
      Couldn't match expected type ‘t0*t1’ with actual type ‘()’
      Expected type: t0*t1~~>t3
        Actual type: ()~~>t2
      In the expression: higherOrderCircuit arg
      In the expression:  arg -> higherOrderCircuit arg
```

Since `higherOrderCircuit` is brought into scope by a κ-abstraction, the type inference algorithm uses rule [κ-Abs] and proceeds under the assumption that `higherOrderCircuit` has type `()~~>t2` for some type `t2`. When it encounters the application `higherOrderCircuit arg` it uses the rule [κ-App], attempting to unify `()~~>t2` with `t0*t1~~>t3`; this unification fails since there is no choice of `t0` and `t1` such that `t0*t1` unifies with `()`.

With the extended grammar and type system, it is now possible to express the `oscillator` example as a two-level program:

```
  class Hardware a where
    -- ...
    xor :: <[ Bit ~~> Bit ]>@a
    reg :: <[ Bit ~~> Bit ]>@a

  oscillator :: Hardware a => <[ Bit*() ~~> Bit ]>@a
  oscillator =
    <[ λinput ->
          let output  = ~~xor input delayed
              delayed = ~~reg output
          in output ]>
```

The full definition of the `Hardware` type class can be found in Figure 5.1.

2.4 The Flattening Transformation

It is very tedious to write expressions for both arrows and generalized arrows by hand. Haskell has a special `proc` syntax for writing `Arrow` expressions; this syntax is more comfortable for the user because it recycles Haskell's variable binding and scoping syntax for use in object language expressions. In fact it does more than that: the `proc` syntax allows the entire unrestricted Haskell grammar to be used in object language expressions. This is not a problem for homogeneous metaprogramming, but it is unworkable for the more general case of heterogeneous metaprogramming. One contribution of this thesis is the realization that a preexisting technology (multi-level types and expressions, with κ-calculus as the minimal object language) is ideally suited to playing the role of `proc`-syntax for a generalization of another preexisting technology (arrows).

This section will try to illustrate the similarities between generalized arrow expressions and two-level expressions, mostly by example. It does not attempt to prove any sort of formal correspondence (that is handled in Chapter 3) or detail the algorithm for translating one kind of expression into the other (that is the subject of Chapter 4). What follows in the remainder of this chapter is just an overview.

So far this chapter has shown three example programs using both generalized arrows and multi-level expressions. Let's return to the simplest example, `pow`, and put the programs side by side, with the inter-token whitespace deliberately contorted to emphasize the similarities.

```
pow_ga mult n =                           pow mult n =
 if n==0                                   if n==0
 then ga_drop          >>>                 then <[ λx ->
      ga_constant 1                                     1 ]>
 else ga_copy                     >>>      else <[ λx ->
      ga_first (pow_ga mult (n-1)) >>>              ~~mult x
      ga_swap                     >>>              (~~(pow mult (n-1)) x) ]>
      mult
```

2.4.1 From Generalized Arrows to Multi-Level Expressions

The translation from generalized arrow expressions to multi-level expressions is the easier of the two directions. In fact, it can be accomplished without any compiler modifications, simply by implementing the `GArrow` type class. The starting point is to notice that it is possible to implement a `GArrow` instance where abstract operations with input type `x` and output type `y` are realized as two-level expressions of type `<[x~~>y]>`. In this case, the required functions implementing the methods of the `Category` class would be something like:[6]

```
 id    :: <[ x ~~> x ]>
 (>>>) :: <[ x ~~> y ]> -> <[ y ~~> z ]> -> <[ x ~~> z ]>
```

[6]the code in the remainder of this section is abbreviated and omits a lot of syntactic noise, particularly the type variables serving as environment classifiers; the full implementation, which typechecks, can be found in Figure 4.7

It isn't hard to figure out how to write two-level expressions with these types; in fact these expressions have already appeared in earlier examples:

```
  id           = <[ λx -> x ]>
  f >>> g      = <[ λx -> ~~g (~~f x) ]>
```

For the `ga_{un}cancel{l,r}` and `ga_{un}assoc` functions of the `GArrow` class, it is no more difficult:

```
  ga_first  f  = <[ λ(x,y)     -> (~~f x,y) ]>
  ga_second f  = <[ λ(x,y)     -> (x,~~f y) ]>
  ga_cancell   = <[ λ(_,x)     -> x         ]>
  ga_cancelr   = <[ λ(x,_)     -> x         ]>
  ga_uncancell = <[ λx         -> ((),x)    ]>
  ga_uncancelr = <[ λx         -> (x,())    ]>
  ga_assoc     = <[ λ((x,y),z) -> (x,(y,z)) ]>
  ga_unassoc   = <[ λ(x,(y,z)) -> ((x,y),z) ]>
```

Even the `ga_drop`, `ga_copy`, and `ga_swap` functions have direct implementations:

```
  ga_drop       = <[ λ_     -> ()    ]>
  ga_copy       = <[ λx     -> (x,x) ]>
  ga_swap       = <[ λ(x,y) -> (y,x) ]>
  ga_constant i = <[ λ()    -> %i    ]>
```

At this point we have complete implementations of `GArrow`, `GArrowDrop`, `GArrowSwap`, and `GArrowCopy`. If we then simply *evaluate* the `pow` function using the type class dictionaries for these implementations, we wind up with this:

```
pow mult n =
 if n==0
 then   <[ λ_ -> () ]>                >>>
        <[ λ() -> 1 ]>
 else   <[ λx -> (x,x) ]>                          >>>
        <[ λ(x,y) -> (x,~~(pow mult (n-1)) y)]>  >>>
        <[ λ(x,y) -> (y,x) ]>                      >>>
        mult
```

The `else` branch contains reducible expressions; simplifying them a bit yields:

```
pow mult n =
 if n==0
 then   <[ λ_ -> 1 ]>
 else   <[ λx -> (x,~~(pow mult (n-1)) x)]>  >>>
        mult
```

And then completing the reduction produces the following expression, which is the multi-level expression originally written by hand back in Section 2.3.1.

```
pow mult n =
 if n==0
 then   <[ λ_ -> 1 ]>
 else   <[ λx -> ~~mult x (~~(pow mult (n-1)) x) ]>
```

2.4.2 From Multi-Level Expressions to Generalized Arrows

Translating generalized arrow expressions into multi-level expressions is easy, and in fact requires no compiler assistance. The opposite direction – translating multi-level expressions into generalized arrow expressions – is quite a bit more difficult, but much more useful.

When translating from generalized arrow expressions to multi-level expressions, the translation followed the structure of the syntax. For the reverse translation it will be necessary to instead follow the structure of the *proof of the typing judgment*, since it contains important information about variable use and reordering. These proofs can very quickly become extremely large (see Figure 4.4 for an example), so at this point we will limit our focus on only the `else` clause of the `pow` program, which was:

`<[ λx -> ~~mult x (~~(pow mult (n-1)) x) ]>`

Since this is a well-typed two-level program it has a typing proof. An abbreviated version of the proof is shown below, although at such a small scale it is nearly impossible to read:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathtt{mult} : \mathtt{<[Int*Int*()}\sim\sim\mathtt{>Int]>}}{\Gamma \vdash \sim\sim\mathtt{mult} : \mathtt{Int*Int*()}\sim\sim\mathtt{>Int}}\,\text{[Esc]} \quad \dfrac{}{\mathtt{x} : \mathtt{()}\sim\sim\mathtt{>Int} \vdash \mathtt{x} : \mathtt{()}\sim\sim\mathtt{>Int}}\,\text{[Var]}}{\Gamma, \mathtt{x} : \mathtt{()}\sim\sim\mathtt{>Int} \vdash \sim\sim\mathtt{mult\ x} : \mathtt{Int*()}\sim\sim\mathtt{>Int}}\,[\kappa\text{-App}] \quad \dfrac{\dfrac{\Gamma \vdash \mathtt{pow\ mult\ (n-1)} : \mathtt{<[Int*()}\sim\sim\mathtt{>Int]>}}{\Gamma \vdash \sim\sim\mathtt{(pow\ mult\ (n-1))} : \mathtt{Int*()}\sim\sim\mathtt{>Int}}\,\text{[Esc]} \quad \dfrac{}{\mathtt{x} : \mathtt{()}\sim\sim\mathtt{>Int} \vdash \mathtt{x} : \mathtt{()}\sim\sim\mathtt{>Int}}\,\text{[Var]}}{\mathtt{x} : \mathtt{()}\sim\sim\mathtt{>Int}, \Gamma \vdash \sim\sim\mathtt{(pow\ mult\ (n-1))\ x} : \mathtt{()}\sim\sim\mathtt{>Int}}\,[\kappa\text{-App}]}{\mathtt{x} : \mathtt{()}\sim\sim\mathtt{>Int}, \mathtt{x} : \mathtt{()}\sim\sim\mathtt{>Int}, \Gamma \vdash \sim\sim\mathtt{mult\ x\ (}\sim\sim\mathtt{(pow\ mult\ (n-1))\ x)} : \mathtt{()}\sim\sim\mathtt{>Int}}\,[\kappa\text{-App}]}{\mathtt{x} : \mathtt{()}\sim\sim\mathtt{>Int}, \Gamma \vdash \sim\sim\mathtt{mult\ x\ (}\sim\sim\mathtt{(pow\ mult\ (n-1))\ x)} : \mathtt{()}\sim\sim\mathtt{>Int}}\,\text{[Contr]}}{\Gamma \vdash \lambda\,\mathtt{x\ ->}\ \sim\sim\mathtt{mult\ x\ (}\sim\sim\mathtt{(pow\ mult\ (n-1))\ x)} : \mathtt{Int*()}\sim\sim\mathtt{>Int}}\,[\kappa\text{-Abs}]}{\Gamma \vdash \mathtt{<[}\lambda\,\mathtt{x\ ->}\ \sim\sim\mathtt{mult\ x\ (}\sim\sim\mathtt{(pow\ mult\ (n-1))\ x)]>} : \mathtt{<[Int*()}\sim\sim\mathtt{>Int]>}}\,\text{[Brak]}$$

Once the proof is written out in long-form like this, the actual expressions and variable names are superfluous; they can be recovered (modulo choice of variables names) from the structure of the proof itself. By omitting the unnecessary expressions and variable names, and abbreviating the conclusion of the Var rule from $\mathtt{x} : \mathtt{()}\sim\sim\mathtt{>Int} \vdash \mathtt{x} : \mathtt{()}\sim\sim\mathtt{>Int}$ to $\mathtt{()}\sim\sim\mathtt{>Int}$ the proof becomes more manageable. We will also drop the Brak and Esc rule invocations to focus on the part of the proof that deals with object language expressions:

$$\dfrac{\dfrac{\dfrac{[\kappa\text{-App}]\quad\dfrac{\Gamma \vdash \mathtt{Int*Int*()}\sim\sim\mathtt{>Int} \quad \dfrac{}{\mathtt{()}\sim\sim\mathtt{>Int}}\,\text{[Var]}}{\Gamma, \mathtt{()}\sim\sim\mathtt{>Int} \vdash \mathtt{Int*()}\sim\sim\mathtt{>Int}} \quad \dfrac{\Gamma \vdash \mathtt{Int*()}\sim\sim\mathtt{>Int} \quad \dfrac{}{\mathtt{()}\sim\sim\mathtt{>Int}}\,\text{[Var]}}{\mathtt{()}\sim\sim\mathtt{>Int}, \Gamma \vdash \mathtt{()}\sim\sim\mathtt{>Int}}\,[\kappa\text{-App}]}{\mathtt{()}\sim\sim\mathtt{>Int}, \mathtt{()}\sim\sim\mathtt{>Int}, \Gamma \vdash \mathtt{()}\sim\sim\mathtt{>Int}}\,[\kappa\text{-App}]}{\mathtt{()}\sim\sim\mathtt{>Int}, \Gamma \vdash \mathtt{()}\sim\sim\mathtt{>Int}}\,\text{[Contr]}}{\Gamma \vdash \mathtt{Int*()}\sim\sim\mathtt{>Int}}\,[\kappa\text{-Abs}]$$

The proof above contains the [κ-App] and [κ-Abs] rules described earlier, as well as the [Var] rule common to all type systems which expresses the vacuous fact that if you assume x has type τ, you can conclude that x has type τ. The only other rule in the proof is [Contr], the rule of contraction. This is one of the three standard substructural rules [Gen35]; the other two are exchange and weakening:

$$\frac{\Gamma_1, \Gamma_2 \vdash e : \tau}{\Gamma_2, \Gamma_1 \vdash e : \tau}\,\textsf{[Exch]} \qquad \frac{\Gamma_1, \Gamma_1 \vdash e : \tau}{\Gamma_1 \vdash e : \tau}\,\textsf{[Contr]} \qquad \frac{\Gamma_2 \vdash e : \tau}{\Gamma_1, \Gamma_2 \vdash e : \tau}\,\textsf{[Weak]}$$

The rule of exchange makes it possible to type programs which use variables in a different order than they are abstracted (for example $\lambda x.\lambda y.yx$). The rule of contraction makes it possible to type programs which use variables more than once (for example $\lambda x.xx$). The rule of weakening makes it possible to type programs which do not use all the variables in scope (for example $\lambda x.\lambda y.x$).

The typing proof for the `else` branch of the `pow` program needs the rule of contraction because the variable `x` appears more than once in the body of the function which binds it.

The flattening transformation proceeds by walking the structure of the typing proof, replacing each typing rule invocation with a generalized arrow expression. For example, the rule of contraction is replaced by `ga_copy`. Using $[\![-]\!]$ for the translation function and letting Δ stand for the rest of the proof above the contraction rule, we can write:

$$\left[\!\!\left[\frac{\Delta}{\Gamma_1 \vdash e : \tau}\textsf{[Contr]}\right]\!\!\right] = \texttt{ga\_copy} \ \texttt{>>>} \ [\![\Delta]\!]$$

Similarly, the $\textsf{[Var]}$ rule becomes `id`:

$$\left[\!\!\left[\frac{}{\texttt{x} : \tau \vdash \texttt{x} : \tau}\textsf{[Var]}\right]\!\!\right] = \texttt{id}$$

The $\textsf{[}\kappa\textsf{-App]}$ rule is a bit tricky:

$$\left[\!\!\left[\frac{\Delta_1 \qquad \Delta_2}{k_1 k_2 : \tau_1 \mathtt{\sim\sim>} \tau_2}\textsf{[}\kappa\textsf{-App]}\right]\!\!\right] = \texttt{ga\_second} \ [\![\Delta_2]\!] \ \texttt{>>>} \ [\![\Delta_1]\!]$$

Revisiting the typing proof for the `else` branch of the `pow` example again, with the judgments omitted, we have this:

$$\left[\!\!\left[\ \frac{\dfrac{\dfrac{\texttt{\sim\sim mult} \qquad \dfrac{}{\vdash}\textsf{[Var]}}{\vdash}\qquad \dfrac{\texttt{\sim\sim(pow mult (n-1))} \qquad \dfrac{}{\vdash}\textsf{[Var]}}{\vdash}\textsf{[}\kappa\textsf{-App]}}{\vdash}\textsf{[}\kappa\textsf{-App]}}{\vdash}\textsf{[Contr]}\ \right]\!\!\right]$$

with the left inference labelled $\textsf{[}\kappa\textsf{-App]}$.

Since the bottom-most rule invocation is $\textsf{[Contr]}$, this reduces to:

$$\texttt{ga\_copy >>>} \left[\!\!\left[\dfrac{ [\kappa\text{-App}]\,\dfrac{\texttt{\textasciitilde\textasciitilde mult} \quad \dfrac{}{\vdash}[\text{Var}]}{\vdash} \qquad \dfrac{\texttt{\textasciitilde\textasciitilde(pow mult (n-1))} \quad \dfrac{}{\vdash}[\text{Var}]}{\vdash}[\kappa\text{-App}] }{\vdash}[\kappa\text{-App}] \right]\!\!\right]$$

Now the bottom-most rule is [κ-App], so this reduces to:

$$\begin{array}{l}
\texttt{ga\_copy >>>} \\
\texttt{ga\_second} \left[\!\!\left[\dfrac{\texttt{\textasciitilde\textasciitilde(pow mult (n-1))} \quad \dfrac{}{\vdash}[\text{Var}]}{\vdash}[\kappa\text{-App}] \right]\!\!\right] \texttt{>>>} \\
\left[\!\!\left[[\kappa\text{-App}]\,\dfrac{\texttt{\textasciitilde\textasciitilde mult} \quad \dfrac{}{\vdash}[\text{Var}]}{\vdash} \right]\!\!\right]
\end{array}$$

Reducing the second term:

$$\begin{array}{l}
\texttt{ga\_copy >>>} \\
\texttt{ga\_second} \left[\!\!\left[\dfrac{\texttt{\textasciitilde\textasciitilde(pow mult (n-1))} \quad \dfrac{}{\vdash}[\text{Var}]}{\vdash}[\kappa\text{-App}] \right]\!\!\right] \texttt{>>>} \\
\texttt{ga\_second} \left[\!\left[\dfrac{}{\vdash}[\text{Var}] \right]\!\right] \texttt{>>>mult}
\end{array}$$

And then the first term:

$$\begin{array}{l}
\texttt{ga\_copy >>>} \\
\texttt{ga\_second (ga\_second} \left[\!\left[\dfrac{}{\vdash}[\text{Var}] \right]\!\right] \texttt{>>> pow mult (n-1)) >>>} \\
\texttt{ga\_second} \left[\!\left[\dfrac{}{\vdash}[\text{Var}] \right]\!\right] \texttt{>>>mult}
\end{array}$$

Finally reducing the [Var] invocations leaves:

```
ga_copy >>>
ga_second (ga_second id     >>>
           pow mult (n-1)) >>>
ga_second id >>>
mult
```

Since `(ga_second id)`=`id`, this is equivalent to:

```
        ga_copy >>>
        ga_second (id >>> pow mult (n-1)) >>>
        id >>>
        mult
```

And `(id >>> f)=f`, so:

```
        ga_copy                    >>>
        ga_second (pow mult (n-1)) >>>
        mult
```

In special cases like this we can rewrite `ga_second` into an expression using `ga_first` and `ga_swap` instead, giving the presentation as a generalized arrow term that we started with:

```
        ga_copy                   >>>
        ga_first (pow mult (n-1)) >>>
        ga_swap                   >>>
        mult
```

Obviously this is not a process that is pleasant to carry out by hand, even for a one-line program.

Chapter 3

Categories

This chapter formalizes the material demonstrated in the previous chapter, providing formal definitions, proofs, and theorems. The end goal of this chapter is the proof of its final theorem stating that the category of generalized arrows is isomorphic to the category of reifications (i.e. the category of multi-level languages).

The first section of this chapter covers basic category theory, formalizing it in the type theory of a particular dependently typed programming language, Coq. The goal here is to formalize enough category theory in order to state the definition of *enrichment*. Enrichment is one of the many ways one category can appear "inside of" another category.

The second section reviews how programming languages are defined using category theory, reconstructing the usual category of types and expressions[1].

The third section examines arrows in programming languages from a categorical perspective, and posits a generalization beyond the case where the arrow's internal language is a superset of the category in which it occurs. This will require introducing an additional kind of category for studying programming languages, beyond the usual category of types and expressions. The additional category is formed by the judgments and deductions of the programming language's type system; it has appeared before in the literature but only infrequently. A generalized arrow can then be defined as a functor from one language's category of *judgments* to another language's category of *types*. This is the formal statement behind the slogan that *generalized arrows represent one language's proof structure in another language's term structure*. It is shown that every arrow (i.e. enriched Freyd category) in a programming language gives rise to such a judgments-to-types functor, and therefore every arrow is a generalized arrow. The third section ends by proving that generalized arrows form a category.

The fourth section examines multi-level languages using categories, and shows that for each multi-level language there is a functor from its its object language's category of types to its

[1]which seems to be given at least a half-dozen different names in the literature

metalanguage's category of types, as well as a functor from its object language's category of judgments to its metalanguage's category of judgments. Moreover, these two functors form a commuting square with the inclusion functors from each language's category of types to its category of judgments; commuting squares of this kind will be called *reifications* and it will be shown that reifications themselves form a category.

The final section proves (Theorem 3.3.0.18) that the category of reifications is isomorphic to the category of generalized arrows; this is the formal statement behind the slogan *multi-level languages are generalized arrows* and the computational content of this theorem's proof is the basis for the *flattening transformation* which turns well-typed multi-level expressions into well-typed one-level expressions parameterized over a generalized arrow.

The material in this chapter is formalized as machine-checked Coq proofs in the accompanying archive. Specifically, theorems labeled **Formalized Theorem** have corresponding proofs in the Coq scripts; in most case the Coq proposition (but not the proof) appears in `typewriter font` immediately after the prose statement of the theorem. To improve readability, the following elements of Coq syntax have been elided from the printed version of this thesis: semicolons, curly braces, `Notation` clauses, `Implicit Argument` clauses, explicit instantiation of implicit arguments, and polymorphic type quantifiers (specifically, `forall` occurring immediately after a colon). Also, to reduce clutter, underscores (_) have been used in positions where an identifier is required but its name is unimportant (for example, the names of proof obligations).

3.1 Basic Category Theory

This section briefly reviews the basic definitions of category theory. Its main purpose is to show how these constructs are represented in Coq proofs.

Formalized Definition 3.1.0.1 ([Awo06, Definition 1.1]) A *category* is a collection `Ob` of *objects*, and for each pair of objects `a` and `b` a collection `a~>b` of *morphisms* along with an equivalence relation $\approx$ on that collection. For each object `a` there is a designated *identity* morphism `id a`, and for each pair of morphisms `f:a~>b` and `g:b~>c` there is a *composite* `f>>>g`. The `>>>` operator respects the equivalence relation, has `id a` as its left and right neutral element, and is associative.

```
Class Cat :=
 Ob    : Type
 (~>)  : Ob -> Ob -> Type
 id    : a~>a
 (>>>) : a~>b -> b~>c ->a~>c

 (≈)   : Equivalence (a~>b)
 _     : Proper ((≈) ==> (≈) ==> (≈)) comp

 _     :     id a >>> f  ≈ f
 _     :     f  >>> id b ≈ f
 _     : (f >>> g) >>> h ≈ f >>> (g >>> h)
```

Formalized Definition 3.1.0.2 ([Awo06, Definition 1.2]) A *functor* from category `c1` to category `c2` is a mapping which assigns an object of `c2` to each object of `c1` and a morphism of `c2` to each morphism of `c1` in a manner which respects object assignment, equivalence, identities, and composition.

```
Class Functor (c1:Cat(Ob:=Ob1))(c2:Cat(Ob:=Ob2)) :=
 Fobj  : Ob1 -> Ob2
 Fmor  : a~>b -> (Fobj a)~>(Fobj b)
 _     : Proper ((≈) ==> (≈)) Fmor
 _     : Fmor (id a) ≈ id (Fobj a)
 _     : Fmor f >>> Fmor g ≈ Fmor (f >>> g)
```

Theorem 3.1.0.3 A collection of functors forms a category if it is closed under composition and includes the identity functor on the domain and codomain of every functor in the category.

3.1.1 Monoidal Categories

Formalized Definition 3.1.1.1 ([PR97, Definition 3.2]) A *binoidal category* is a category $\mathbb{C}$ given with a family of pairs of functors indexed by the objects of $\mathbb{C}$. For an object A, the first functor will be called the *left product* $(- \ltimes A)$ and the second functor will be called the *right product* $(A \rtimes -)$. Additionally, it is required that the action on B of the first functor at A be the same as the action on A of the second functor at B; this common action on objects is written $A \otimes B$.

```
Class BinoidalCat :=
  _   :> Cat
 (⊗) :  Obj  -> Obj -> Obj
 (⋊) :  ∀a:Obj, Functor c c (Fobj:=(λx.a⊗x))
 (⋉) :  ∀a:Obj, Functor c c (Fobj:=(λx.x⊗a))
```

Remark 3.1.1.2 Binoidal categories will be used to model computations in which *evaluation order* is significant. The fact that the two functors agree on objects reflects the fact that type systems do not track which coordinate of a tuple was computed first. The fact that the functors may disagree on morphisms reflects the fact that evaluating the left coordinate first may yield a different result than evaluating the right coordinate first.

Formalized Definition 3.1.1.3 ([PR97, Definition 3.3]) A morphism in a binoidal category is *central* if its left product at every object commutes with the right product of every other morphism, and its right product at every object commutes with the left product of every other morphism.

```
Class CentralMor (f:A~>B) : Prop :=
 _ : ∀g, f ⋉ _ >>> _ ⋊ g ≈ _ ⋊ g >>> f ⋉ _
 _ : ∀g, g ⋉ _ >>> _ ⋊ f ≈ _ ⋊ f >>> g ⋉ _
```

Remark 3.1.1.4 Central morphisms model computations which are *pure* and therefore commute (in time) with all others. Note that for morphisms f and g the expression $f \otimes g$ is not well-defined unless at least one of f or g is central.

Formalized Definition 3.1.1.5 (`Center`) For $\mathbb{C}$ a binoidal category, the *center of* $\mathbb{C}$, written $Z(\mathbb{C})$, is the collection of all central morphisms of $\mathbb{C}$.

Formalized Theorem 3.1.1.6 For $\mathbb{C}$ a binoidal category, $Z(\mathbb{C})$ is a subcategory of $\mathbb{C}$.

Formalized Definition 3.1.1.7 ([LCH09]) A binoidal category is *commutative* if every morphism is central.

```
Class CommutativeCat :=
 _ :> BinoidalCat
 _ :  ∀f, CentralMor f
```

Formalized Definition 3.1.1.8 ([PR97, Section 3.5]) A *premonoidal category* is a binoidal category with an object I and central natural isomorphisms $A \otimes (B \otimes C) \cong (A \otimes B) \otimes C$ and $X \otimes I \cong X \cong I \otimes X$ such that all possible naturality squares commute and in addition Mac Lane's *pentagon* and *triangle* coherence conditions of Figure 3.1 hold.

```
Class PreMonoidalCat :=
 _        :> BinoidalCat _
 I        :  Obj
 assoc    :          (a⊗b)⊗x ≅ a⊗(b⊗x)
 cancell  :              I⊗a ≅ a
 cancelr  :              a⊗I ≅ a
 triangle : cancelr ⋉ b  ≈ assoc >>> a ⋊ cancell
 pentagon : assoc >>> assoc ≈ assoc ⋉ d >>> assoc >>> a ⋊ assoc
```

Premonoidal categories provide enough machinery to deal with finite lists of objects by representing them syntactically as the nonempty leaves of binary trees with $\otimes$ at all interior nodes. The `assoc`, `cancell`, and `cancelr` naturalities of a premonoidal category let us rotate these binary trees (or "re-parenthesize" them), and the commutative diagrams ensure that any two trees are interchangeable so long as they have the same leaves in the same order (i.e. the "parenthesization" does not matter).

$$\begin{tikzcd}
 & (A\otimes B)\otimes(C\otimes D) \arrow[dr, "\mathtt{assoc}"] & \\
((A\otimes B)\otimes C)\otimes D \arrow[ur, "\mathtt{assoc}"] \arrow[d, "\mathtt{assoc}\ltimes D"'] & & A\otimes(B\otimes(C\otimes D)) \\
(A\otimes(B\otimes C))\otimes D \arrow[rr, "\mathtt{assoc}"'] & & A\otimes((B\otimes C)\otimes D) \arrow[u, "A\rtimes\mathtt{assoc}"']
\end{tikzcd}$$

$$\begin{tikzcd}
(A\otimes I)\otimes B \arrow[rr, "\mathtt{assoc}"] \arrow[dr, "\mathtt{cancelr}\ltimes B"'] & & A\otimes(I\otimes B) \arrow[dl, "A\rtimes\mathtt{cancell}"] \\
 & A\otimes B &
\end{tikzcd}$$

Figure 3.1: The pentagon and triangle coherence conditions of monoidal categories from [Mac71, p162].

The $\ltimes$ and $\rtimes$ functors of a premonoidal category ensure that any two morphisms can be combined; the domain and codomain of the combined morphism will be the binary tree having the original two morphisms' domain and codomain as its two child nodes. For example, if $f : A\texttt{->}B$ and $g : C\texttt{->}D$ then

$$f \ltimes C\texttt{>>>}B \rtimes g : A \otimes C\texttt{->}B \otimes D$$

$$A \rtimes g\texttt{>>>}f \ltimes D : A \otimes C\texttt{->}B \otimes D$$

Although these two compositions have the same domain and codomain, they are not necessarily equal in a premonoidal category. This is how premonoidal categories model the concept of *evaluation order* in a programming language. Similarly, the central morphisms of a premonoidal category model the *pure functions* of a programming language. Just as there are some programming languages (e.g. Coq, Haskell) in which all functions are pure, there are categories in which all morphisms are central:

Formalized Definition 3.1.1.9 A *monoidal category* is a commutative premonoidal category.

```
Class MonoidalCat :=
 mon_pm           := pm
 mon_commutative  :> CommutativeCat pm
```

Many of the concepts in this chapter come in two flavors: one for binoidal categories in general and another which is specific to commutative binoidal categories. In order to save space and reduce repetition, this chapter will follow established convention: a term which is prefixed with "pre-" indicates the version which is not necessarily commutative. For example, a monoidal category is necessarily commutative, while a *premonoidal* category is not necessarily. The Coq development is based on definitions for the "pre-" versions; the commutative versions add the additional hypothesis `CommutativeCat C`.

Formalized Definition 3.1.1.10 ([PR97, Definition 3.8]) A *monoidal functor* is a functor between premonoidal categories which preserves their premonoidal structure *including centrality* of morphisms.

```
Class MonoidalFunctor
   (c1:PreMonoidalCat(I:=I1))
   (c2:PreMonoidalCat(I:=I2)) :=
  _     :> Functor c1 c2
  _     :  ∀A ∀B, Fobj (A⊗B) =  (Fobj A)⊗(Fobj B)
  _     :  ∀f ∀A, Fmor (f⋉A) ≈ (Fmor f)⋉(Fobj A)
  _     :  ∀f ∀A, Fmor (A⋊f) ≈ (Fobj A)⋊(Fmor f)
  _     :  I2 ≅ Fobj I1
  _     :  ∀f, CentralMor f -> CentralMor (Fmor f)
  _     :  cancell ≈ Fmor cancell
  _     :  cancelr ≈ Fmor cancelr
  _     :  assoc   ≈ Fmor assoc
```

Three common operations on contexts in programming languages are weakening, contraction, and exchange; these correspond to free variables which are ignored, reused, or reordered. The categories which allow the corresponding operations are called terminal, symmetric, and diagonal categories:

Formalized Definition 3.1.1.11 ([Awo06, Definition 2.7]) An object 1 of a category $\mathbb{C}$ is a *terminal object* if there is exactly one morphism into 1 from every object. This morphism will be written $\texttt{drop}_A : A$->1.

```
Class TerminalCat :=
  _      :> Cat
  1      :  Obj
  drop   :  ∀a, a ~> 1
  unique :  ∀a ∀f:a~>1, f ≈ drop a
```

A *braided premonoidal category* is a category with a natural isomorphism $A \otimes B \cong B \otimes A$ in which the *hexagon* and *braided triangle* coherence conditions of Figure 3.2 are met.

```
Class BraidedCat :=
 _          :> PreMonoidalCat
 swap       :  a⊗b ≅ b⊗a
 hexagon1   :  assoc  >>> swap >>> assoc ≈ swap ⋉ c >>> assoc >>> b ⋊ swap
 hexagon2   :  assoc⁻¹ >>> swap >>> assoc⁻¹ ≈ a ⋊ swap >>> assoc⁻¹ >>> swap ⋉ b
 triangleb  :  cancelr ≈ swap >>> cancell
```

$$\begin{tikzcd}
 & A \otimes (B \otimes C) \arrow[r, "\texttt{swap}"] & (B \otimes C) \otimes A \arrow[dr, "\texttt{assoc}"] & \\
(A \otimes B) \otimes C \arrow[ur, "\texttt{assoc}"] \arrow[dr, "\texttt{swap} \rtimes C"'] & & & B \otimes (C \otimes A) \\
 & (B \otimes A) \otimes C \arrow[r, "\texttt{assoc}"'] & B \otimes (A \otimes C) \arrow[ur, "B \ltimes \texttt{swap}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
 & (A \otimes B) \otimes C \arrow[r, "\texttt{swap}"] & C \otimes (A \otimes B) \arrow[dr, "\texttt{assoc}^{\texttt{-1}}"] & \\
A \otimes (B \otimes C) \arrow[ur, "\texttt{assoc}^{\texttt{-1}}"] \arrow[dr, "A \ltimes \texttt{swap}"'] & & & (C \otimes A) \otimes B \\
 & A \otimes (C \otimes B) \arrow[r, "\texttt{assoc}^{\texttt{-1}}"'] & (A \otimes C) \otimes B \arrow[ur, "\texttt{swap} \rtimes B"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
A \otimes I \arrow[rr, "\texttt{swap}"] \arrow[dr, "\texttt{cancelr}"'] & & I \otimes A \arrow[dl, "\texttt{cancell}"] \\
 & A &
\end{tikzcd}$$

Figure 3.2: The hexagon and braided triangle coherence conditions for braided categories.

Formalized Definition 3.1.1.12 A *symmetric premonoidal category* is a braided premonoidal category in which each pair of mediating isomorphisms are mutually inverse:

```
Class SymmetricCat :=
  _ :> BraidedCat
  _ :  swap ≈ swap^-1
```

Formalized Definition 3.1.1.13 A category with diagonal morphisms or *diagonal category* is a premonoidal category with with a morphism $\texttt{copy}_X : X \texttt{->} X \otimes X$ for each object X such that $\texttt{copy}_I \approx \texttt{cancelr}_I{}^{\texttt{-1}} \approx \texttt{cancell}_I{}^{\texttt{-1}}$.

Formalized Definition 3.1.1.14 A *braided diagonal category* is a braided premonoidal category with diagonal morphisms in which the diagram below commutes.

```
Class DiagonalCat :=
  _      :> BinoidalCat
  copy   :  a ~> a⊗a
  _      :  copy >>> swap ≈ copy
```

$$\begin{tikzcd}
I \arrow[d, "\texttt{copy}"'] \arrow[dr, "\texttt{copy}"] & \\
I \otimes I \arrow[r, "\texttt{swap}"'] & I \otimes I
\end{tikzcd}$$

Formalized Definition 3.1.1.15 A *symmetric monoidal functor* is a functor between symmetric premonoidal categories which preserves their braiding structure.

Formalized Definition 3.1.1.16 A *cartesian category* is a diagonal monoidal category in which I is a terminal object and the diagram below commutes.

```
Class CartesianCat :=
 _ :> TerminalCat
 _ :> MonoidalCat(I:=1)
 _ :> DiagonalCat
 _ :  id a ≈ copy >>> drop a ⋊      a >>> cancell
 _ :  id a ≈ copy >>>      a ⋉ drop a >>> cancelr
```

$$\begin{tikzcd}
1 \otimes X \arrow[d, "\texttt{cancell}"'] & X \otimes X \arrow[l, "\texttt{drop}_X \ltimes X"'] \arrow[d, "X \rtimes \texttt{drop}_X"] \\
X \arrow[ur, "\texttt{copy}"] \arrow[loop below, "\texttt{id}_X"] & X \otimes 1 \arrow[l, "\texttt{cancelr}"]
\end{tikzcd}$$

3.1.2 Enriched Categories

There are many ways that one category can exist "inside" another category; the one most useful in the study of programming languages is *enrichment*:

Formalized Definition 3.1.2.1 ([Kel82, p. 1.2]) A $\mathbb{V}$-*enriched category* or *category enriched over* $\mathbb{V}$ or *category enriched in* $\mathbb{V}$ consists of:

1. A monoidal category $\langle \mathbb{V}, \otimes, I \rangle$
2. A collection $\mathbb{C}$ of objects
3. For each pair $A, B \in \mathbb{C}$ a $\mathbb{V}$-object $\mathbb{C}(A, B)$.
4. For each $A \in \mathbb{C}$ a $\mathbb{V}$-morphism $\texttt{eid}_A : I \texttt{->} \mathbb{C}(A, A)$ called *enriched identity*
5. For every triple $A, B, C \in \mathbb{C}$ a $\mathbb{V}$-morphism $\texttt{ecomp}_{ABC} : \mathbb{C}(A, B) \otimes \mathbb{C}(B, C) \texttt{->} \mathbb{C}(A, C)$ called *enriched composition*

Such that the $\mathbb{V}$-diagrams of Figure 3.3 commute for any $f : I \texttt{->} \mathbb{C}(A, B)$.

The extraneous collection of $\mathbb{C}$-objects in the definition above can be eliminated by identifying each $\mathbb{C}$-object A with $\mathbb{C}(I, A)$ which is a $\mathbb{V}$-object. The usual definition of enrichment requires that $\mathbb{V}$ is monoidal; when $\mathbb{V}$ is merely premonoidal "a $\mathbb{V}$-enriched category" will be understood to mean a $Z(\mathbb{V})$-enriched category.

$$\begin{tikzcd}
\mathbb{C}(A,A)\otimes\mathbb{C}(A,B) \arrow[dr, "\mathtt{ecomp}"] & & \mathbb{C}(A,B)\otimes\mathbb{C}(B,B) \arrow[dl, "\mathtt{ecomp}"'] \\
\mathbb{C}(A,A)\otimes I \arrow[u, "\mathbb{C}(A,A)\rtimes f"] & \mathbb{C}(A,B) & I\otimes\mathbb{C}(B,B) \arrow[u, "f\ltimes\mathbb{C}(B,B)"'] \\
I\otimes I \arrow[u, "\mathtt{eid}_A\ltimes I"] & I \arrow[l, "\mathtt{cancell}^{\mathtt{-1}}"] \arrow[r, "\mathtt{cancelr}^{\mathtt{-1}}"'] \arrow[u, "f"'] & I\otimes I \arrow[u, "I\rtimes\mathtt{eid}_B"']
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbb{C}(A,C)\otimes\mathbb{C}(C,D) \arrow[ddr, bend left, "\mathtt{ecomp}"] & \\
(\mathbb{C}(A,B)\otimes\mathbb{C}(B,C))\otimes\mathbb{C}(C,D) \arrow[u, "\mathtt{ecomp}\ltimes\mathbb{C}(C,D)"] \arrow[dd, bend left, "\mathtt{assoc}"] & \\
 & \mathbb{C}(A,D) \\
\mathbb{C}(A,B)\otimes(\mathbb{C}(B,C)\otimes\mathbb{C}(C,D)) \arrow[uu, bend left, "\mathtt{assoc}^{\mathtt{-1}}"] \arrow[d, "\mathbb{C}(A,B)\rtimes\mathtt{ecomp}"'] & \\
\mathbb{C}(A,B)\otimes\mathbb{C}(B,D) \arrow[uur, bend right, "\mathtt{ecomp}"'] &
\end{tikzcd}$$

Figure 3.3: Diagrams in $\mathbb{V}$ which must commute in order for $\mathbb{C}$ to be an enriched category.

Technically a "$\mathbb{V}$-enriched category" is not really a category; it is a collection of objects and morphisms in another category $\mathbb{V}$. However, given an enriched category, we can build a real category:

Formalized Definition 3.1.2.2 ([Kel82, p. 1.3]) If $\mathbb{C}$ is a $\mathbb{V}$-enriched category, then the *underlying category* of $\mathbb{C}$, written $\mathbb{C}_0$, has $\mathbb{C}$-objects as its objects and a morphism $f_0 : A\texttt{->}B$ for each $\mathbb{V}$-morphism $f : I\texttt{->}\mathbb{C}(A,B)$. The composition of two morphisms f_0 and g_0 is defined as

$$f_0\texttt{>>>}g_0 \overset{\text{def}}{\equiv} \mathtt{cancell}^{\mathtt{-1}} \texttt{ >>> } (f\otimes g) \texttt{ >>> } \mathtt{ecomp}$$

Associativity of composition and left/right neutrality of identity follow from the commutative diagrams in Figure 3.3.

Intuitively, an enrichment of $\mathbb{C}$ in $\mathbb{V}$ expresses the fact that a smaller category $\mathbb{C}$ is "inside" a larger category $\mathbb{V}$ and moreover that the morphism-combining operation of composition `>>>` can be expressed in terms of the morphisms of $\mathbb{V}$. This "containment with expressible composition" is one of three essential features of how arrows encapsulate one category (or language) within another. The second feature requires an additional condition:

Formalized Definition 3.1.2.3 We will say that $\mathbb{C}$ is *binoidally* enriched in $\mathbb{V}$ if $\mathbb{C}$ is enriched over $\mathbb{V}$, $\mathbb{C}_0$ is binoidal, and if for each triple of objects A, B, C in $\mathbb{C}$ there are a pair of central $\mathbb{V}$-morphisms

$$\begin{aligned}\texttt{efirst}_{ABC} &: \mathbb{C}(A,B)\texttt{->}\mathbb{C}(A\otimes C, B\otimes C)\\ \texttt{esecond}_{ABC} &: \mathbb{C}(A,B)\texttt{->}\mathbb{C}(C\otimes A, C\otimes B)\end{aligned}$$

Such that $(f\ltimes A)_0 = f_0\texttt{>>>efirst}$ and $(A\rtimes f)_0 = f_0\texttt{>>>esecond}$ in $\mathbb{V}$, and such that composition with `efirst` and `esecond` in $\mathbb{V}$ preserves enriched identity morphisms and enriched composition[2].

Formalized Definition 3.1.2.4 (`PremonoidalEnrichment`) A *premonoidal enrichment*, written $\mathbb{C}\in\mathbb{V}$, is a monoidal category $\langle\mathbb{V},\oplus,I_v\rangle$ and a premonoidal category $\langle\mathbb{C},\ltimes,\rtimes,I\rangle$ such that $\mathbb{C}$ is the underlying category of some binoidally $\mathbb{V}$-enriched category.

Intuitively, a premonoidal enrichment $\mathbb{C}\in\mathbb{V}$ requires that the morphism-combining operations $\ltimes$ and $\rtimes$ of $\mathbb{C}$ can be expressed in terms of the morphisms of $\mathbb{V}$. This "containment with expressible combinators" is the second essential feature of how arrows encapsulate one category (or language) within another. The next section will complete the picture by adding a third feature.

3.1.3 Arrows in Category Theory

Shortly after arrows in the programming language Haskell were introduced by Hughes [Hug00] papers appeared attempting to provide a category-theoretic explanation of the phenomenon. Early attempts [JHI09; JH06; HJ06; Atk08; Asa10; AH10] identified arrows with *Freyd categories*, a mathematical concept previously introduced by Power, Robinson, and Thielecke:

Formalized Definition 3.1.3.1 (`FreydCategory`) A *Freyd Category* [PT97; PT99] is an identity-on-objects symmetric monoidal functor $\mathcal{J}{:}\mathbb{V}\texttt{->}Z(\mathbb{C})$ where $\mathbb{V}$ is a cartesian category and $\mathbb{C}$ (called the *Kleisli category*) is symmetric premonoidal.

```
Class FreydCategory (V C:Cat) :=
  _    : CartesianCat V
  _    : SymmetricCat C
  _    : StrictMonoidalFunctor(Fobj:=(λx.x)) V C
  _    : ∀f, CentralMor (arr f)
```

Remark 3.1.3.2 The naming convention here is counterintuitive. A Freyd *category* is actually a *functor*.

[2]This is equivalent to requiring that the left product and right product functors of $\mathbb{C}$ be $\mathbb{V}$-*enriched functors* [Kel82, (1,5),(1.6)]

This is only a partial characterization of arrows; the correct and complete definition was provided by Atkey [Atk08, Definition 3.2]:

Formalized Definition 3.1.3.3 An *arrow between categories* is a Freyd category $\mathcal{J}$:$\mathbb{V}$`->`$Z(\mathbb{C})$ such that $\mathbb{C}$ is the underlying category of a premonoidal enrichment $\mathbb{C} \Subset \mathbb{V}$.

In less formal terms, an arrow between categories is defined to be the situation where one category $\mathbb{V}$ "contains" another category $\mathbb{C}$ and:

1. The morphism-composing operation `>>>` of $\mathbb{C}$ can be expressed in terms of the morphisms of $\mathbb{V}$
2. The morphism-combining operations $\ltimes$ and $\rtimes$ of $\mathbb{C}$ can be expressed in terms of the morphisms of $\mathbb{V}$
3. Any morphism f of the "outer" category can be transported into a morphism $\mathcal{J}(f)$ of the "inner" category $\mathbb{C}$.

Hopefully at this point it is clear that the `Arrow` class in the Haskell programming language works in a very similar way to the category-theoretic definitions introduced in this section. For example, the argument and return types of the `first` method of the Haskell `Arrow` class bear strong similarity to the domain and codomain of the `efirst` morphism of a premonoidally enriched category, the `>>>` operation of the Haskell `Category` class works in a very similar way to the `ecomp` morphism, and the $\mathcal{J}$ functor plays a role similar to `arr`. The next section will introduce the framework needed to formalize the precise nature of this similarity.

3.2 Categories for Programming Languages

The previous section built up the category theory needed to provide the definition of an *arrow between categories*. This is not quite the same thing as the arrows that programmers are accustomed to using (e.g. the Haskell `Arrow` type class), but they are very closely related concepts. This section will formalize some basic tools for representing programming languages in category theory in order to state the precise relationship.

3.2.1 Admissible Languages

Formalized Definition 3.2.1.1 A programming language specification $\mathcal{L}$ is: a grammar specifying its expressions e including variables x, a grammar specifying its types τ, a ternary substitution operation $e_1[x \mapsto e_2]$ given as a recursive function on the syntax of expressions, a denotation relation $\approx$ which is an equivalence relation (not necessarily recursive) on expressions, and a typing relation $\vdash$ between finite lists of variable-type pairs and expression-type pairs.[3]

The following definition spells out some fairly obvious properties that a programming language's specification ought to have. These are not conditions on the language but rather conditions on its presentation in terms of grammar and typing rules. Like the definition of an enriched category, the following definition has been deliberately engineered to ensure that the result forms a category.

Formalized Definition 3.2.1.2 A programming language specification $\mathcal{L}$ is *admissible* if it meets the following conditions:

1. **Denotation Ignores Variable Names**: The denotation of an expression is not affected by permutation of its variable names.
2. **Reflexive Typing Relation**: For every type τ the relation $x : \tau \vdash x : \tau$ holds.
3. **Substitution Has Left and Right Identities**: $e \approx x[x \mapsto e] \approx e[x \mapsto x]$.
4. **Substitution Preserves Types**:[4]
   if $\Gamma_2 \vdash f : \tau_2$ and $\Gamma_1, x : \tau_2, \Gamma_3 \vdash g : \tau$ then $\Gamma_1, \Gamma_2, \Gamma_3 \vdash g\ [x \mapsto f] : \tau$
5. **Substitution Respects Denotation**:
   If $f \approx f'$ and $g \approx g'$ then $g[x \mapsto f] \approx g'[x \mapsto f']$.
6. **Substitution is Associative up to Denotation**:
   $h[x \mapsto g[y \mapsto f]] \approx h[x \mapsto g][y \mapsto x]$ if y is not free in h.

Since denotation ignores variable names we will, without loss of generality, assume that in any typing relation $\Gamma \vdash e : \tau$ the variables of the variable-type list Γ are distinguished only by their ordinal position in the list (i.e. always in the form $x_0, x_1, x_2 \ldots$). When normalized this way it is no longer necessary to include the variable names in the context, so the typing relation on variable-name-normalized expressions can be reduced to a relation between lists of types and an expression-type pair. Finally, we will add an extra degree of freedom by allowing the list of types to the left of the turnstile to be a *binary tree*, each of whose leaves is either a type τ or else empty ($\top$):

$$\Gamma ::= \top \mid \langle \Gamma, \Gamma \rangle \mid \tau$$

It is now possible to extend typing to relate type-trees to expression-type pairs by saying that the extended relation $\Gamma \vdash e : \tau$ holds if and only if the unextended relation holds for the erasure $\varepsilon(\Gamma) \vdash e : \tau$ where erasure is defined as:

[3] Following the convention of [Sco93, p427] the relation $\approx$ on closed expressions can be extended to a relation on all expressions by considering two expressions equivalent if they are equivalent under every possible well-typed substitution for their free variables.

[4] also called the "substitution lemma" [Pie02, p. 9.3.8])

$$\begin{aligned}
\varepsilon(\top) &= \\
\varepsilon(\tau) &= \tau \\
\varepsilon(\langle \Gamma_1, \Gamma_2 \rangle) &= \varepsilon(\Gamma_1), \varepsilon(\Gamma_2)
\end{aligned}$$

There is a similar grammar Σ for binary trees of expression-type pairs:

$$\Sigma ::= \top \mid \langle \Sigma, \Sigma \rangle \mid e : \tau$$

And it has a similar erasure:

$$\begin{aligned}
\varepsilon(\top) &= \\
\varepsilon(e : \tau) &= \tau \\
\varepsilon(\langle \Sigma_1, \Sigma_2 \rangle) &= \varepsilon(\Sigma_1), \varepsilon(\Sigma_2)
\end{aligned}$$

Following work by Blute, Cockett, and Seely [BCS97] (which is far more general than the particular case here) we will call these expression-type trees *co-contexts*. Any language requiring only one-element co-contexts can simply encode them as one-element trees, and rules for vertical and horizontal *context expansion* can be adjoined.

Formalized Lemma 3.2.1.3 Given a language specification $\mathcal{L}$ there are admissible strengthenings of *reflexive sequents*, *left and right identities exist*, and *substitution preserves types* in which the context and co-context may be arbitrary (but matching) contexts and co-contexts rather than being limited to single-leaf contexts and co-contexts.

Proof. `strong_substitution` □

3.2.2 Categories of Types and Expressions

Formalized Definition 3.2.2.1 The *category of types and closed expressions* $\text{Types}^0(\mathcal{L})$ of an admissible programming language $\mathcal{L}$ is the smallest category which has:

1. A designated object I
2. An object $[\![\tau]\!]$ for each type τ of $\mathcal{L}$
3. A morphism $[\![e]\!] : I$`->`$[\![\tau]\!]$ assigned to each *closed* expression e such that $\top \vdash e : \tau$.

This is not a very interesting category because its morphisms don't say anything about how expressions interact. The object I is the domain of *every* non-identity morphism.

$$\begin{tikzcd}
& & I \arrow[dll, "\llbracket e_1\rrbracket"'] \arrow[dl, "\llbracket e_2\rrbracket"'] \arrow[d, "\llbracket e_3\rrbracket"'] \arrow[dr, "\llbracket e_4\rrbracket"] & & \\
\llbracket\tau_1\rrbracket & \llbracket\tau_2\rrbracket & \llbracket\tau_3\rrbracket & \llbracket\tau_4\rrbracket & \cdots
\end{tikzcd}$$

Formalized Definition 3.2.2.2 The *category of types and terms with one free variable* $\text{Types}^1(\mathcal{L})$ of an admissible programming language $\mathcal{L}$ is the smallest category which has:

1. An object $\llbracket\tau\rrbracket$ for each type τ of $\mathcal{L}$
2. A morphism $\llbracket e\rrbracket : \llbracket\tau_1\rrbracket \texttt{->} \llbracket\tau_2\rrbracket$ assigned to each term e with exactly one free variable x such that $x : \tau_1 \vdash e : \tau_2$.

Two terms are assigned the same morphism *if and only if* they have the same denotation for every choice of substitution for their free variable. In other words, $\llbracket e_1\rrbracket = \llbracket e_2\rrbracket$ iff $e_1 \approx e_2$.

Since equality of morphisms is based on the semantic relation $\approx$ different choices of semantics will yield non-isomorphic categories, even for the same grammar and type system. This category is known by different names in the literature: formula category [Awo06, p. 9.5], Lindenbaum category of an algebra.

There's really no reason to stop at one free variable, and in fact moving to arbitrarily (but finitely) many free variables gives a construction that is quite standard in categorical logic (see $\text{Form}(\bar{x})$ of [Awo06, p. 9.5]), having originally been used by Lawvere to explain quantifiers as adjunctions [Law96].

Formalized Definition 3.2.2.3 ([Cro94, p. 4.8.4]) The *category of types and terms* $\text{Types}^\omega(\mathcal{L})$ of an admissible programming language $\mathcal{L}$ is the smallest category which has:

1. An object $\llbracket\tau\rrbracket$ for each type τ of $\mathcal{L}$
2. An assignment of an object to each *type tree* T:

$$\begin{aligned} \llbracket\top\rrbracket &= I \\ \llbracket\tau\rrbracket &= \llbracket\tau\rrbracket \\ \llbracket\langle T_1, T_2\rangle\rrbracket &= \llbracket T_1\rrbracket \otimes \llbracket T_2\rrbracket \end{aligned}$$

3. For each e such that $\Gamma \vdash e : \tau$ a morphism $\llbracket e\rrbracket : \llbracket\varepsilon(\Gamma)\rrbracket \texttt{->} \llbracket\tau\rrbracket$.

Two terms are assigned the same morphism *if and only if* they have the same denotation for every substitution for their free variables.

This category is known by many other names in different contexts: Lawvere theory, classifying category of a theory [Cro94, p. 3.8], category of derived operations of an algebra, clone category, algebra of types [Sco93].

Formalized Lemma 3.2.2.4 The category $\text{Types}^0(\mathcal{L})$ is a subcategory of the category $\text{Types}^\omega(\mathcal{L})$.

Formalized Theorem 3.2.2.5 (`TypesL_PreMonoidal`) Types$^\omega(\mathcal{L})$ is a premonoidal category, where the functor $- \ltimes B$ sends an object A to $\langle A, B\rangle$ and a morphism $f : A\texttt{->}C$ to $\langle f, \mathsf{id}_B\rangle$ and the functor $A \rtimes -$ sends an object B to $\langle A, B\rangle$ and a morphism $f : B\texttt{->}C$ to $\langle \mathsf{id}_A, f\rangle$.

This category is not *strict* premonoidal. Its strictification [Mac71, Theorem XI.3.1] – a category in which the objects are *lists* of types rather than binary trees – removes some structure we will need later on. Perhaps the earliest example comes from Lawvere's thesis [Law63], later revisited as [Law96], explaining quantifiers as adjunctions, which led to the general concept being called a Lawvere theory.

Formalized Definition 3.2.2.6 (`ArrowInProgrammingLanguage`) An *Arrow in a programming language* $\mathcal{L}$ with Types$^\omega(\mathcal{L})$ cartesian is an enrichment $\mathbb{C} \Subset \mathbb{V}$ with $\mathbb{V} \subseteq$ Types$^\omega(\mathcal{L})$ and a Freyd Category $\mathcal{J}$:Types$^\omega(\mathcal{L})\texttt{->}Z(\mathbb{C})$ [Atk08, Theorem 3.7].

For example, in Haskell any particular `instance Arrow a` determines a full subcategory of Types$^\omega$(Haskell) consisting of types of the form `a X Y` for any `X` and `Y`. The Kleisli category $\mathbb{C}$ of the arrow is enriched in this subcategory of Types(Haskell).

3.2.3 Categories of Judgments and Deductions

Arrows in a programming language $\mathcal{L}$ can be explained using only the category Types$^\omega(\mathcal{L})$ provided that it is self-enriched. In this case an entire copy of the metalanguage appears "inside" the arrow (i.e. within the object language), ensuring that they are essentially equal in expressive power. This agrees with the fact that arrows are suitable for homogeneous metaprogramming: in heterogeneous metaprogramming the metalanguage and object language have the same expressive power, and with an arrow between languages the two categories are joined (by the Freyd category functor $\mathcal{J}$ in one direction and by enrichment in the other).

Generalized arrows lift the requirement that the object language is a superset of the metalanguage. This means that the categories corresponding to these two languages could be considerably different, so a categorical model for generalized arrows is not going to involve a single category self-enrichment. There needs to be a second category. That category is the *category of judgments and deductions*, which this section will define.

Before defining the category of judgments we will first need to introduce some admissibility conditions on the proofs of the language's type system, similar to how we introduced admissibility conditions on the expressions of the language in order to define Types$^\omega(\mathcal{L})$.

Formalized Definition 3.2.3.1 A deductive system for the type system of a language $\mathcal{L}$ is a collection of inference rules R according to the following grammar:

$J ::= \Gamma \vdash \Sigma$ *(judgments)*

$J^\star ::= J J^\star \mid \cdot$ *(judgment lists)*

$$R ::= \frac{J^\star}{J} \qquad \textit{(inference rules)}$$

Formalized Definition 3.2.3.2 A deductive system for the type system of a language $\mathcal{L}$ is admissible if:

1. **Term Irrelevance**: The hypotheses of inference rules must impose no structure on the term to the right of the turnstile. For example, the inference rule below is not admissible:
$$\frac{\ldots \vdash (\lambda x.e) : \tau}{\ldots}$$
This requirement ensures that in order to know what inference rules can be applied to continue a proof, all we need to look at is the context to the left of the turnstile and the type to the right of the colon – the term might affect the result of applying the rule, but it cannot influence whether or not the rule is applicable. This will ensure that composition is a total function in future constructions. Term irrelevance mirrors *proof irrelevance* under the Curry-Howard correspondence [How80].

2. **Vertical Context Expansion**: if there is a proof
$$\frac{\dfrac{\Gamma \vdash \Sigma}{\vdots}}{\Gamma' \vdash \Sigma'}$$
then for every Γ_0 whose variables are disjoint from those of Γ, Γ', Σ, and Σ', and Σ_0 whose expressions consist only of lone variables there are proofs
$$\frac{\dfrac{\Gamma_0, \Gamma \vdash \Sigma_0, \Sigma}{\vdots}}{\Gamma_0, \Gamma' \vdash \Sigma_0, \Sigma'} \qquad \frac{\dfrac{\Gamma, \Gamma_0 \vdash \Sigma, \Sigma_0}{\vdots}}{\Gamma', \Gamma_0 \vdash \Sigma', \Sigma_0}$$
Moreover, the denotation of Σ' in the latter proofs is the same as in the original proof.

3. **Context Associativity**: If Γ_1 and Γ_2 have exactly the same leaves in exactly the same order, and if Σ_1 and Σ_2 have exactly the same leaves in exactly the same order, then the following is admissible:
$$\frac{\Gamma_1 \vdash \Sigma_1}{\Gamma_2 \vdash \Sigma_2}$$
This will ensure that the category of judgments is monoidal.

4. **Horizontal Context Expansion**: For every type τ and every Γ, the following rules must be admissible for some choice of variable name x not occurring in Γ or Σ:
$$\frac{\Gamma \vdash \Sigma}{x : \tau, \Gamma \vdash x : \tau, \Sigma} \qquad \frac{\Gamma \vdash \Sigma}{\Gamma, x : \tau \vdash \Sigma, x : \tau}$$
This will ensure that the enrichment is binoidal.

While $\text{Types}^\omega(\mathcal{L})$ accounts for expressions as morphisms between tuples of types, the category of judgments accounts for *proofs* as morphisms between tuples of *judgments*:

Formalized Definition 3.2.3.3 The *category of judgments* $\text{Judgments}(\mathcal{L})$ of an admissible programming language $\mathcal{L}$ is a category which assigns to each tuple of judgments $J^\star$ an object and to each proof

$$\cfrac{\cfrac{J_0 \ldots J_{n-1}}{\vdots}}{J}$$

a morphism $[\![J_0]\!] \times \ldots \times [\![J_{n-1}]\!]\texttt{->}[\![J]\!]$. Two morphisms with the same domain and codomain are equal if and only if for every choice of expression in the co-contexts of $J_0 \ldots J_{n-1}$ and every substitution for the variables in the contexts of J and $J_0 \ldots J_{n-1}$, all of the expressions in the co-context of J have the same denotation.

Formalized Lemma 3.2.3.4 $\text{Judgments}(\mathcal{L})$ is a cartesian category via $\times$

Proof. `Judgments_cartesian` □

Lambek [Lam69, p79] calls this the *deductive system on a category* (in this case, the category $\text{Types}^\omega(\mathcal{L})$). Szabo [Sza78, pp. 1.1.26,2.5.3] calls it a *sequential category generated by an unlabeled deductive system* (i.e., category of sequents). Szabo's work also demonstrates that $\text{Types}^\omega(\mathcal{L})$ (which he calls $Fm(X)$) is isomorphic to a subcategory of this category via the covariant hom-functor embedding.

Remark 3.2.3.5 Note that there are two distinct monoidal structures here: $\otimes$ and $\times$. The $\otimes$ operation on contexts (which can be extended to an operation on judgments) represents lists of types in a context; the $\times$ operation on judgments represents lists of judgments in an inference rule. So, for example,

$$\frac{\begin{array}{r} x : \tau_1, y : \tau_2 \vdash e_1 : \tau_3 \\ z : \tau_z \vdash e_2 : \tau_4 \end{array}}{q : \tau_q \vdash e' : \tau_5}$$

would be interpreted by a morphism from $((\tau_1 \otimes \tau_2) \vdash \tau_3) \times (\tau_z \vdash \tau_4)$ to $(\tau_q \vdash \tau_5)$ which would be identified for purposes of equality with the possible denotations of e' under all possible choices of e_1, e_2, z, and q.

Remark 3.2.3.6 The choice of denotational equality in all contexts as the equivalence relation for morphisms is important. Consider the following two inference rules as an example:

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : \texttt{Int} \\ \Gamma \vdash e_2 : \texttt{Int} \end{array}}{\Gamma \vdash e_1\texttt{+}e_2 : \texttt{Int}} \qquad \frac{\begin{array}{c} \Gamma \vdash e_1 : \texttt{Int} \\ \Gamma \vdash e_2 : \texttt{Int} \end{array}}{\Gamma \vdash e_1\texttt{*}e_2 : \texttt{Int}}$$

For every choice of Γ, each of these rules entails a morphism from $(\Gamma \vdash \texttt{Int}) \times (\Gamma \vdash \texttt{Int})$ to $(\Gamma \vdash \texttt{Int})$. However these morphisms are not equal because they have different denotation after substituting $e_1 := \texttt{0}$ and $e_2 := \texttt{1}$.

Having defined the category $\text{Judgments}(\mathcal{L})$, we can now prove its relationship to $\text{Types}^\omega(\mathcal{L})$:

Formalized Theorem 3.2.3.7 Types$^\omega(\mathcal{L})$ is premonoidally enriched in Judgments($\mathcal{L}$).

Proof. `TypesEnrichedInJudgments`

Proof Outline

1. The category of judgments is actually binoidal in two different ways: $\otimes$ and $\times$. For purposes of defining the enrichment, we choose $\times$ (Lemma 3.2.3.4).
2. Each object of Types$^\omega(\mathcal{L})$ corresponds to a type tree, so for every pair of objects $[[T_1]]$ and $[[T_2]]$ there is an object $T_1 \vdash T_2$ of Judgments($\mathcal{L}$).
3. For every enriched identity morphism $\texttt{eid}_\tau : [[T]]\texttt{->}[[T]]$ we have the following by (strong) reflexivity of sequents (Definition 3.2.1.2):

$$\frac{\top \vdash \top}{\Sigma \vdash \Sigma}$$

4. For every three objects $[[T_1]]$, $[[T_2]]$, and $[[T_3]]$ in Types$^\omega(\mathcal{L})$ there is a composition morphism $(T_1 \vdash T_2) \otimes (T_2 \vdash T_3)\texttt{->}(T_1 \vdash T_3)$ in Judgments($\mathcal{L}$) arising from the proofs required by (strong) preservation of types under substitution (Definition 3.2.1.2)
5. The requisite diagrams commute. Commutativity of the upper diagram of Definition 3.1.2.1 relies on the presence of left and right identity terms (Definition 3.2.1.2) and the lower diagram relies on associativity of substitution (Definition 3.2.1.2).

□

Formalized Lemma 3.2.3.8 The enrichment of Types$^\omega(\mathcal{L})$ in Judgments($\mathcal{L}$) has a hom functor $\mathsf{hom}_\mathcal{L}$.

Proof. Choose $X = I$, so let $\mathsf{hom}_\mathcal{L}([[T]]) = (\top \vdash T)$. A morphism $f_0 : [[T_1]]\texttt{->}[[T_2]]$ in Types$^\omega(\mathcal{L})$ underlies a morphism $(\top \vdash \top)\texttt{->}(T_1 \vdash T_2)$ in Judgments($\mathcal{L}$), so there must be a proof tree:

$$\cfrac{\cfrac{\top \vdash T_1}{\vdots}}{\top \vdash T_2}$$

Let $\mathsf{hom}_\mathcal{L}(f)$ be this proof. □

3.2.4 Generalized Arrows

Formalized Definition 3.2.4.1 (`GeneralizedArrowInLanguage`) For programming languages $\mathcal{O}$ and $\mathcal{M}$, a generalized arrow $\mathcal{G}$ from language $\mathcal{O}$ to language $\mathcal{M}$ is a monoidal functor from Judgments($\mathcal{O}$) to Z(Types($\mathcal{M}$)).

$$\begin{tikzcd}
\text{Judgments}(\mathcal{O}) \arrow[d, dashed, no head, "\psi"'] \arrow[dr, "\mathcal{G}"] & \text{Judgments}(\mathcal{M}) \arrow[d, dashed, no head, "\psi"'] \\
\text{Types}(\mathcal{O}) & Z(\text{Types}(\mathcal{M}))
\end{tikzcd}$$

Remark 3.2.4.2 Note that the domain of the functor is a judgment category whereas its codomain is (the center of) a *term* category; in this sense, generalized arrows represent the *proof structure of one language within the term structure of another language.*

We can expand the scope of this definition to arbitrary categories:

Formalized Definition 3.2.4.3 (`GeneralizedArrow`) For enrichments $\mathbb{K} \in \mathbb{V}_{\mathbb{K}}$ and $\mathbb{C} \in \mathbb{V}$, a generalized arrow from $\mathbb{K} \in \mathbb{V}_{\mathbb{K}}$ to $\mathbb{C} \in \mathbb{V}$ is a monoidal functor from $\mathbb{V}_{\mathbb{K}}$ to $Z(\mathbb{C})$.

$$\begin{tikzcd}
\mathbb{V}_{\mathbb{K}} \arrow[d, dashed, no head, "\psi"'] \arrow[dr, "\mathcal{G}"] & \mathbb{V} \arrow[d, dashed, no head, "\psi"'] \\
\mathbb{K} & Z(\mathbb{C})
\end{tikzcd}$$

Formalized Theorem 3.2.4.4 (`ArrowsAreGeneralizedArrows`) If $\mathcal{J}$:Types$^\omega(\mathcal{L})$->$Z(\mathbb{K})$ is an arrow in programming language $\mathcal{L}$, the inclusion functor from the enriching category of $\mathbb{K}$ to Types$^\omega(\mathcal{L})$ is a generalized arrow from $Z(\mathbb{K}) \in \mathbb{V}_{\mathbb{K}}$ to Types$^\omega(\mathcal{L}) \in$ Judgments$(\mathcal{L})$:

$$\begin{tikzcd}
\mathbb{V}_{\mathbb{K}} \arrow[d, dashed, no head, "\psi"'] \arrow[drr, "\subseteq"] & & \text{Judgments}(\mathcal{L}) \arrow[d, dashed, no head, "\psi"'] \\
Z(\mathbb{K}) & \text{Types}^\omega(\mathcal{L}) \arrow[l, "\mathcal{J}"'] & = Z(\text{Types}^\omega(\mathcal{L}))
\end{tikzcd}$$

Note that Types$^\omega(\mathcal{L})$ = Z(Types$^\omega(\mathcal{L})$) because the domain of a Freyd Category must be cartesian (all morphisms are central).

Formalized Definition 3.2.4.5 (`CategoryOfGeneralizedArrows`) The category of generalized arrows is the category whose objects are surjective monic monoidal enrichments, with a morphism from E_1 to E_2 for each generalized arrow from E_1 to E_2, and in which two morphisms are equal if the corresponding generalized arrows are naturally isomorphic as functors. Composition of two generalized arrows $\mathcal{G} : E_2$->E_3 and $\mathcal{G}' : E_1$->E_2 is given by $\mathcal{G}'$>>>$\mathsf{hom}_{E_2}(I, -)$>>>$\mathcal{G}$.

3.3 Categories for Multi-Level Languages

Having defined generalized arrows, we now turn to multi-level languages. First, however, a definition in terms of arbitrary enrichments:

Formalized Definition 3.3.0.6 (`Reification`) Given two enrichments $\mathbb{K} \in \mathbb{V}_{\mathbb{K}}$ and $\mathbb{C} \in \mathbb{V}$, a reification from $\mathbb{K} \in \mathbb{V}_{\mathbb{K}}$ to $\mathbb{C} \in \mathbb{V}$ is a monoidal functor $\mathcal{R} : \mathbb{V}_{\mathbb{K}}$`->`$\mathbb{V}$ such that for every $K {\in} Ob(\mathbb{K})$ there exists a functor $\exists_K : \mathbb{K}$`->`$Z(\mathbb{C})$ making the following diagram commute up to natural isomorphism:

$$\begin{tikzcd}
\mathbb{V}_{\mathbb{K}} \arrow[rr, "\mathcal{R}", "\oplus" marking] & & \mathbb{V} \\
\mathbb{K} \arrow[u, "\mathsf{hom}(K,-)"] \arrow[rr, "\exists_K"'] & & Z(\mathbb{C}) \arrow[u, tail, "\mathsf{hom}(I,-)"']
\end{tikzcd}$$

With this definition in place, the definition of a two-level language is straightforward:

Formalized Definition 3.3.0.7 (`TwoLevelLanguage`) A two-level language $(\mathcal{O}, \mathcal{M})$ consists of a pair of languages $\mathcal{O}$ and $\mathcal{M}$ and a reification from $\text{Types}(\mathcal{O}) \in \text{Judgments}(\mathcal{O})$ to $\text{Types}(\mathcal{M}) \in \text{Judgments}(\mathcal{M})$.

$$\begin{tikzcd}
\text{Judgments}(\mathcal{O}) \arrow[r, "\mathcal{R}", "\oplus" marking] & \text{Judgments}(\mathcal{M}) \\
\text{Types}(\mathcal{O}) \arrow[u, "\mathsf{hom}(K,-)"] \arrow[r, "\exists_K"'] & Z(\text{Types}(\mathcal{M})) \arrow[u, tail, "\mathsf{hom}(I,-)"']
\end{tikzcd}$$

Remark 3.3.0.8 The essence of a multi-level language is a functor from one Judgments category to another; in this sense, multi-level languages represent the *proof structure of one language within the proof structure of another language.* Compare this with Remark 3.2.4.2.

The motivation for this definition can be found in literature all the way back to the earliest work on typed multi-level languages. In [NN92, p39] the authors "sketch the form of the general construction [of a two-level language]. For each well-formedness rule or axiom R for types in L, and each binding time $b{\in}B$, we add the rule or axiom R^b;" their specification for R^b – translated into the terminology of category theory – forms the functor $\mathcal{R}$ above when the metalanguage language has function types $\Rightarrow$ for which $\mathsf{Hom}(X, Y) \cong \mathsf{Hom}(I, X{\Rightarrow}Y)$[5].

Remark 3.3.0.9 Note that the functors $\mathcal{R}$ and $\exists_K$ are not necessarily full or faithful.

By iterating and taking limits, we arrive at n-level languages and ω-level languages:

[5]This was the case for all of the systems considered in [NN92], and is also for all typed λ-calculi. The definition given here – without reference to function types – may be used to define multi-level languages in which the metalanguage does *not* have such function types (for example, languages based on κ-calculus).

Formalized Definition 3.3.0.10 (`NLevelLanguage`) Every language is a one-level language. An $(n+1)$-level language is a two-level language $(\mathcal{L}_n, \mathcal{L})$ where $\mathcal{L}_n$ is an n-level language.

Formalized Definition 3.3.0.11 (`OmegaLevelLanguage`) An ω-level language is a function f whose domain is the natural numbers such that $(f(n), f(n{+}1))$ is a two-level language for every $n \in \mathbb{N}$.

Formalized Definition 3.3.0.12 (`MonicEnrichment`) An enrichment $\mathbb{C} \Subset \mathbb{V}$ is *monic* if there is an object X such that $\mathsf{hom}(X, -) : \mathbb{C}\texttt{->}\mathbb{V}$ is faithful.

The enrichment of any concrete category in **Sets** is monic. Intuitively, to say that an enrichment is monic is to say that its base category $\mathbb{C}$ is "concrete with respect to" $\mathbb{V}$ (rather than **Sets**). Often X is called a *generator* for $\mathbb{C}$. The enriched category of a monic enrichment is called *well-pointed* if X is a terminal object.

Formalized Definition 3.3.0.13 (`SurjectiveEnrichment`) An enrichment $\mathbb{C} \Subset \mathbb{V}$ is *surjective* if every object of $\mathbb{V}$ is the tensor of finitely many objects each of which is either the unit object I_v of $\mathbb{V}$ or a hom-object $\mathbb{C}(-,-)$.

Intuitively, surjective enrichments are those in which the enriching category $\mathbb{V}$ has no extraneous *objects* unnecessary to the enrichment (note that this says nothing about *morphisms*). This is not a serious limitation; one can always get a suitable enrichment by simply restricting focus to the least full subcategory in which $\mathbb{C}$ is enriched.

Formalized Definition 3.3.0.14 (`MonoidalEnrichment`) An enrichment $\mathbb{C} \Subset \mathbb{V}$ is *monoidal* if $\mathsf{hom}(I, -) : Z(\mathbb{C})\texttt{->}\mathbb{V}$ is a monoidal functor.

Intuitively, a monoidal enrichment is one in which the base category's tensor is "rich enough" to represent the structure of its own judgments. The most common case arises as a result of $\mathbb{C}$ being cartesian:

Formalized Theorem 3.3.0.15 (`CartesianEnrMonoidal`) Any enrichment $\mathbb{C} \Subset \mathbb{V}$ with both $\mathbb{C}$ and $\mathbb{V}$ cartesian is a monoidal enrichment.

Formalized Lemma 3.3.0.16 (`LanguagesWithProductsAreSMME`) If a programming language $\mathcal{L}$ has cartesian contexts (i.e., weakening, exchange, and contraction), then $\text{Types}^{\omega}(\mathcal{L}) \Subset \text{Judgments}(\mathcal{L})$ is a surjective monic monoidal enrichment.

Formalized Definition 3.3.0.17 (`CategoryOfReifications`) The category of reifications is the category whose objects are surjective monic monoidal enrichments, whose morphisms from E_1 to E_2 are reifications from E_1 to E_2, and where two morphisms are equal if the corresponding reifications' $\mathcal{R}$-functors are naturally isomorphic.

This leads to the main technical result of this chapter:

Formalized Theorem 3.3.0.18 The category of reifications is isomorphic to the category of generalized arrows.

Proof. `ReificationsAreGArrows`

Proof Outline

In intuitive terms, the proof of isomorphism of categories requires a (identity-preserving, composition-respecting) mapping M_1 from generalized arrows to reifications, a (identity-preserving, composition-respecting) mapping M_2 from reifications to generalized arrows, and proofs that $M_1(M_2(\mathcal{R}))$ is naturally isomorphic to $\mathcal{R}$ and $M_2(M_1(\mathcal{A}))$ is naturally isomorphic to $\mathcal{A}$. Coming up with M_1 and M_2 is, of course, the hard part. For M_1 we simply pre-compose the generalized arrow functor with the $\mathsf{hom}(I, -)$ functor on its codomain. Finding its companion, M_2, is more difficult: its construction relies on the fact that, because the domain of the reification functor $\mathcal{R}$ is part of a *surjective* enrichment, we can reason by induction on trees of hom-objects. The base case relies on the fact that the reification functor forms a commuting square with $\mathsf{hom}(I, -)$ (via $\exists_K$) for *every* $\mathsf{hom}(K, -)$, and the inductive step exploits the fact that the codomain enrichment is monoidal, allowing us to "represent" trees of objects. Together these provide an isomorphism between the range of $\mathcal{R}$ as a subcategory and the base category of its codomain; this isomorphism may be post-composed with the reification to produce a generalized arrow. $\square$

Chapter 4

Compilers

To demonstrate the feasibility of the ideas presented in the previous two chapters, a modified version of the Glasgow Haskell Compiler (GHC) [Tea09] has been created. This modified GHC offers a new flag, `-XModalTypes`, which adds new expression syntax for code brackets and escape, new type syntax for code types, inference of environment classifiers using GHC's existing type inference engine [Dim05], and an expansion of the type checker to include the additional typing rules described in this chapter.

4.1 System FC

GHC uses System FC [Sul+07, post-publication Appendix C] as its intermediate language. This language is rich enough to capture the type structure of Haskell including its many GHC-supported extensions. This chapter introduces System FC$^{\alpha}$, which is nothing more than the union of the grammars and typing rules of System FC and λ^{α} [TN03] with straightforward modifications to harmonize the two systems. System FC$^{\alpha}$ is of little direct interest as a type system on its own; it is defined only in order to facilitate the implementation and avoids deviating from the simple union of System FC and λ^{α} except where absolutely necessary. Those deviations are:

1. Inference rules for annotations (Note), recursive let bindings (LetRec), and literals (Lit) have been added because these constructs appear in GHC's `CoreSyn` but not in [Sul+07, post-publication Appendix C].

2. The Alt rule of System FC has been inlined into the Case rule in order to have only a single sort of judgment on expressions.

3. The environments for kinding of type variables (Γ), classification of coercion variables (Δ), and typing of value variables (Σ) are kept separate.

$$
\begin{aligned}
T &:= \text{type constructors}\\
K &:= \text{data constructors}\\
S_n &:= n\text{-ary type functions}\\
\alpha,\beta &:= \text{type variables}\\
\eta &:= \;\cdot\; \mid \alpha,\eta\\
c &:= \text{coercion variables}\\
n &:= \text{notes}\\
p &:= \text{patterns}\\
\kappa,\iota &:= \;\star\; \mid \kappa \Rightarrow \kappa\\
\tau,\sigma,\delta &:= \alpha \mid T \mid \texttt{(->)} \mid \texttt{(+>)}\\
&\quad \mid \tau\tau \mid \forall\alpha{:}\kappa.\tau\\
&\quad \mid S_n\tau_1\ldots\tau_n\\
&\quad \mid \texttt{<\{}\tau\texttt{\}>}^{\alpha}\\
&\quad \mid \texttt{<[}\tau\texttt{\~{}\~{}>}\tau\texttt{]>}^{\alpha}\\
\gamma &:= \text{coercions}\\
L_\tau &:= \text{literals of type } \tau\\
x &:= \text{value variables}\\
\Gamma &:= \;\cdot\; \mid \alpha{:}\kappa,\Gamma\\
\Delta &:= \;\cdot\; \mid c{:}\tau\sim\tau,\Delta\\
\Sigma &:= \;\langle\rangle\; \mid \;\langle\Sigma,\Sigma\rangle\; \mid \;\langle\tau@\eta\rangle
\end{aligned}
$$

$$
\begin{aligned}
e &:= x\\
&\quad \mid L_\tau\\
&\quad \mid \mathbf{note}\\
&\quad \mid n\ e\\
&\quad \mid \mathbf{let}\ x = e\ \mathbf{in}\ e\\
&\quad \mid \mathbf{letrec}\ \overline{x = e}\ \mathbf{in}\ e\\
&\quad \mid \mathbf{cast}\ \gamma\ e\\
&\quad \mid \mathbf{case}\ e\ \mathbf{of}\ \overline{p\texttt{->}e}\\
&\quad \mid e\ e \mid \lambda x{:}\tau.e\\
&\quad \mid e\ \tau \mid \Lambda\alpha{:}\kappa.e\\
&\quad \mid e\ \gamma \mid \Lambda^{c}c{:}\tau\sim\tau.e\\
&\quad \mid \texttt{<\{}e\texttt{\}>}\\
&\quad \mid \texttt{\~{}\~{}}e\\
&\quad \mid \texttt{<[}k\texttt{]>}\\
k &:= x\\
&\quad \mid \texttt{\~{}\~{}}e\\
&\quad \mid k\ k\\
&\quad \mid \lambda x.k\\
&\quad \mid \mathbf{let}\ x = e\ \mathbf{in}\ e\\
&\quad \mid \mathbf{letrec}\ \overline{x = e}\ \mathbf{in}\ e
\end{aligned}
$$

Figure 4.1: System FC$^{\alpha}$ grammar

$$\frac{}{\langle\langle\rangle, \Sigma\rangle \rightsquigarrow \Sigma}\textsf{CanL} \qquad \frac{}{\langle\Sigma, \langle\rangle\rangle \rightsquigarrow \Sigma}\textsf{CanR} \qquad \frac{}{\Sigma \rightsquigarrow \langle\langle\rangle, \Sigma\rangle}\textsf{uCanL} \qquad \frac{}{\Sigma \rightsquigarrow \langle\Sigma, \langle\rangle\rangle}\textsf{uCanR}$$

$$\frac{}{\langle\langle\Sigma_1, \Sigma_2\rangle, \Sigma_3\rangle \rightsquigarrow \langle\Sigma_1, \langle\Sigma_2, \Sigma_3\rangle\rangle}\textsf{Assoc} \qquad \frac{}{\langle\Sigma_1, \langle\Sigma_2, \Sigma_3\rangle\rangle \rightsquigarrow \langle\langle\Sigma_1, \Sigma_2\rangle, \Sigma_3\rangle}\textsf{uAssoc}$$

$$\frac{\Sigma_1 \rightsquigarrow \Sigma_2}{\langle\Sigma', \Sigma_1\rangle \rightsquigarrow \langle\Sigma', \Sigma_2\rangle}\textsf{Left} \qquad \frac{\Sigma_1 \rightsquigarrow \Sigma_2}{\langle\Sigma_1, \Sigma'\rangle \rightsquigarrow \langle\Sigma_2, \Sigma'\rangle}\textsf{Right}$$

$$\frac{}{\langle\Sigma_1, \Sigma_2\rangle \rightsquigarrow \langle\Sigma_2, \Sigma_1\rangle}\textsf{Exch} \qquad \frac{}{\langle\Sigma, \Sigma\rangle \rightsquigarrow \Sigma}\textsf{Cont} \qquad \frac{}{\langle\rangle \rightsquigarrow \Sigma}\textsf{Weak}$$

$$\frac{\Sigma_1 \rightsquigarrow \Sigma_2 \quad \Sigma_2 \rightsquigarrow \Sigma_3}{\Sigma_1 \rightsquigarrow \Sigma_3}\textsf{Comp} \qquad \frac{\Sigma_1 \rightsquigarrow \Sigma_2 \quad \Gamma; \Delta; \Sigma_1 \vdash \Sigma}{\Gamma; \Delta; \Sigma_2 \vdash \Sigma}\textsf{Arrange}$$

Figure 4.2: System FC$^\alpha$ structural rules.

4. The branches of a **case** statement, the bindings of a **letrec**, and the environment assigning types to value variables are represented using binary trees and explicit structural manipulations rather than lists and variable names. Because of this, expressions need not appear in judgments – the expression whose well-typedness is witnessed by a proof can be reconstructed, up to α-renaming, from the structure of the proof itself. The auxiliary judgment $\Sigma_1 \rightsquigarrow \Sigma_2$ asserts that the context Σ_2 may be produced from Σ_1 by invocation of structural operations.

5. Rather than represent contexts as type-variable pairs with an auxiliary variable-level map σ as in [TN03], contexts are represented as (trees of) type-level pairs with no auxiliary map.

6. Rather than decorate each judgment with a named level, the *succedent* of the judgment is decorated.

The grammar for System FC$^\alpha$ is shown in Figure 4.1. The grammar for coercions γ and patterns p and the rules for the type-well-formedness judgment $\vdash_{\text{TY}}$ and coercion-well-formedness judgment $\vdash_{\text{CO}}$ are identical to those of System FC and are not shown; they may be found in [Sul+07, post-publication Appendix C].

The structural rules for System FC$^\alpha$ are shown in Figure 4.2. These are the usual Gentzen substructural rules [Gen35] formulated to operate on trees rather than lists. Contexts are represented as a binary tree in which each leaf can be either empty or a type; this binary tree is used to represent a list of types as the nonempty leaves of the tree. The auxiliary judgment $\rightsquigarrow$ defines a binary reachability relation on contexts. The first six rules make any two trees with the same non-empty leaves in the same order mutually reachable. The Left

$$\dfrac{\Gamma;\Delta;\Sigma \vdash \langle \tau @ \alpha, \eta \rangle}{\Gamma;\Delta;\Sigma \vdash \langle \texttt{<\{}\tau\texttt{\}>}^{\alpha} @ \eta \rangle}\,\textsf{Brak} \qquad \dfrac{\Gamma;\Delta;\Sigma \vdash \langle \texttt{<\{}\tau\texttt{\}>}^{\alpha} @ \eta \rangle}{\Gamma;\Delta;\Sigma \vdash \langle \tau @ \alpha, \eta \rangle}\,\textsf{Esc}$$

$$\dfrac{\Gamma;\Delta;\Sigma \vdash \langle \tau \rangle}{\Gamma;\Delta;\Sigma \vdash \langle \tau \rangle}\,\textsf{Note}(n) \qquad \dfrac{}{\Gamma;\Delta;\langle\rangle \vdash \langle \tau @ \cdot \rangle}\,\textsf{Lit}(L_\tau) \qquad \dfrac{}{\Gamma;\Delta;\langle \tau @ \eta \rangle \vdash \langle \tau @ \eta \rangle}\,\textsf{Var}$$

$$\dfrac{\begin{array}{c}\Gamma;\Delta;\Sigma_1 \vdash \langle \tau_1 @ \eta \rangle \\ \Gamma;\Delta;\langle \Sigma_2, \langle \tau_1 @ \eta \rangle \rangle \vdash \langle \tau_2 @ \eta \rangle\end{array}}{\Gamma;\Delta;\langle \Sigma_1, \Sigma_2 \rangle \vdash \langle \tau_2 @ \eta \rangle}\,\textsf{Let} \qquad \dfrac{\begin{array}{c}\Gamma;\Delta;\Sigma \vdash \left\langle T\overline{\delta} @ \eta \right\rangle \qquad \theta = [\overline{\alpha \mapsto \delta}] \\ (\forall i)\ K_i : \forall \overline{\alpha{:}\kappa} \forall \overline{\beta{:}\iota} \overline{c{:}\sigma{\sim}\sigma'}.\sigma.\overline{\tau'} \to T\overline{\alpha} \\ (\forall i)\ \Gamma, \overline{\beta{:}\theta(\iota)}; \overline{\sigma_i{\sim}\sigma'_i}, \Delta; \left\langle \Sigma, \overline{\theta(\sigma_i)} \right\rangle \vdash \langle \tau @ \eta \rangle\end{array}}{\Gamma;\Delta;\left\langle \textsf{mkTree}(\overline{\Sigma_i}), \Sigma \right\rangle \vdash \langle \tau @ \eta \rangle}\,\textsf{Case}$$

$$\dfrac{}{\Gamma;\Delta;\langle\rangle \vdash \langle\rangle}\,\textsf{Void} \qquad \dfrac{\begin{array}{c}\Gamma;\Delta;\Sigma_1 \vdash \Sigma'_1 \\ \Gamma;\Delta;\Sigma_2 \vdash \Sigma'_2\end{array}}{\Gamma;\Delta;\langle \Sigma_1, \Sigma_2 \rangle \vdash \langle \Sigma'_1, \Sigma'_2 \rangle}\,\textsf{Join} \qquad \dfrac{\begin{array}{c}\langle \tau @ \eta' \rangle \in \Sigma_2 \ \Rightarrow\ \eta' = \eta \\ \Gamma;\Delta;\langle \Sigma_1, \Sigma_2 \rangle \vdash \langle \tau_1 @ \eta, \Sigma_2 \rangle\end{array}}{\Gamma;\Delta;\Sigma_1 \vdash \langle \tau_1 @ \eta \rangle}\,\textsf{LetRec}$$

$$\dfrac{\Gamma;\Delta;\langle \Sigma, \langle \tau_x @ \eta \rangle \rangle \vdash \langle \tau_e @ \eta \rangle}{\Gamma;\Delta;\Sigma \vdash \langle \texttt{(->)}\,\tau_x \tau_e @ \eta \rangle}\,\textsf{Abs} \qquad \dfrac{\begin{array}{c}\Gamma;\Delta;\Sigma_2 \vdash \langle \texttt{(->)}\,\tau_1 \tau_2 @ \eta \rangle \\ \Gamma;\Delta;\Sigma_1 \vdash \langle \tau_1 @ \eta \rangle\end{array}}{\Gamma;\Delta;\langle \Sigma_1, \Sigma_2 \rangle \vdash \langle \tau_2 @ \eta \rangle}\,\textsf{App}$$

$$\dfrac{\begin{array}{c}\alpha \notin \Gamma \cup \Delta \cup \Sigma \\ \alpha{:}\kappa, \Gamma;\Delta;\Sigma \vdash \langle \sigma @ \cdot \rangle\end{array}}{\Gamma;\Delta;\Sigma \vdash \langle \forall \alpha{:}\kappa.\sigma @ \cdot \rangle}\,\textsf{AbsT} \qquad \dfrac{\begin{array}{c}\Gamma \vdash_{\mathrm{TY}} \tau : \kappa \\ \Gamma;\Delta;\Sigma \vdash \langle \forall \alpha{:}\kappa.\sigma @ \cdot \rangle\end{array}}{\Gamma;\Delta;\Sigma \vdash \langle \sigma[\alpha := \tau] @ \cdot \rangle}\,\textsf{AppT}$$

$$\dfrac{\begin{array}{c}c \notin \Delta \cup \Sigma \\ \Gamma; c{:}\sigma_1{\sim}\sigma_2, \Delta;\Sigma \vdash \langle \tau @ \cdot \rangle\end{array}}{\Gamma;\Delta;\Sigma \vdash \langle \texttt{(+>)}\,\sigma_1 \sigma_2 \tau @ \cdot \rangle}\,\textsf{AbsC} \qquad \dfrac{\begin{array}{c}\Gamma;\Delta \vdash_{\mathrm{CO}} \gamma : \sigma_1{\sim}\sigma_2 \\ \Gamma;\Delta;\Sigma \vdash \langle \texttt{(+>)}\,\sigma_1 \sigma_2 \tau @ \cdot \rangle\end{array}}{\Gamma;\Delta;\Sigma \vdash \langle \tau @ \cdot \rangle}\,\textsf{AppC} \qquad \dfrac{\begin{array}{c}\Gamma;\Delta \vdash_{\mathrm{CO}} \gamma : \sigma_1{\sim}\sigma_2 \\ \Gamma;\Delta;\Sigma \vdash \langle \sigma_1 @ \cdot \rangle\end{array}}{\Gamma;\Delta;\Sigma \vdash \langle \sigma_2 @ \cdot \rangle}\,\textsf{Cast}$$

Figure 4.3: System FC$^\alpha$, essentially the union of System FC and λ^α.

and Right rules indicate that reachability of subtrees may be extended to otherwise-identical parent trees. The Exch, Cont, and Weak rules are the usual substructural rules of exchange, contraction, and weakening. The Comp rules indicates that reachability is a transitive relation. The Arrange rule incorporates reachability of contexts into typing judgments. The remaining typing rules for System FC$^\alpha$ are shown in Figure 4.3.

Remark 4.1.0.19 The modified rules for Var and Lit require that the context contain no extraneous entries; this forces the proof-builder to invoke the Weak rule in order to get rid of these entries. Because the flattening functor sends the Weak rule to the `ga_drop` function, this requirement is essential to ensuring that the result of the flattening process is well-formed (and well-typed). Similarly, the App, Let, Case, and LetRec rules require that the conclusion context be "partitioned" amongst the hypotheses (Cont/`ga_copy`) in the correct order (Exch/`ga_swap`); reconstructing this partitioning from a `CoreSyn` expression accounts for a significant portion of the complexity in the Coq development, but this reconstruction is the "engine" of the flattening transformation and it is here that the compiler does the work of piecing together `ga_swap`, `ga_copy`, `ga_drop`, `ga_{un}cancel{l,r}`, and `ga_{un}assoc` – the work which makes generalized arrow instances so tedious to use without help from the compiler.

A few conservative restrictions have been imposed in places where there was uncertainty about the interaction between levels and features of System FC; these may be loosened in the future:

- The variables bound by a **letrec** must have the same named level.
- The following may occur only at level zero: the **cast** expression, coercion lambda, coercion application, and **case** expressions which bind coercion variables.

4.2 Support for GHC Passes written in Coq

In order to be able to use extracted proof content for the compiler pass, support for compiler passes written in Coq has been added to GHC. Since Coq already has the ability to extract proof-content as Haskell code this support consists mostly of exposing the GHC compiler's internal representation as Coq structures.

Since Coq is a much richer language than Haskell there is considerable latitude in how this is done; the ultimate solution consists of four representations, each more dependently typed than the one before it. These four representations are HaskCore, HaskWeak, HaskStrong, and HaskProof, and are often used as a pipeline:

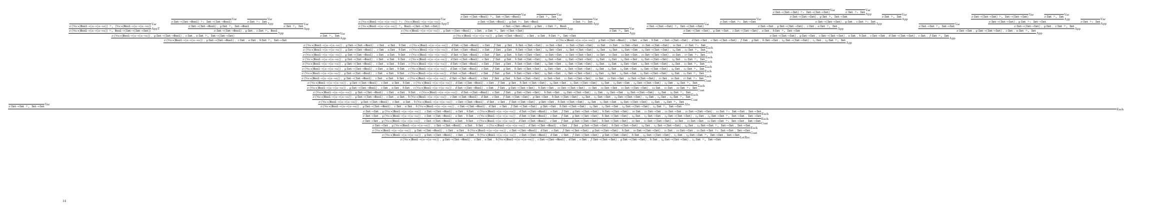
- **HaskCore** is a Coq representation which extracts exactly to `GHC.CoreSyn`, on the nose. It is a literal transcription of the GHC internal structures into the Coq language, with as few changes as possible. Variables are represented as the opaque-to-Coq type `CoreVar`, which admits no operations besides decidable quality and an allocation monad. This can be very cumbersome to manipulate in Coq due to the fact that Coq functions must be total and recursion must be structurally-decreasing, so most compiler passes will immediately convert this to HaskWeak.
- **HaskWeak** is a slightly more Coq-friendly version of HaskCore; it is still weakly (non-dependently) typed, but makes it easier to write total functions. Constructs which are treated differently by the typing rules (e.g., application of the function space tycon `(->)`

```
putStrLn :: String -> IO ()
main = putStrLn "Hello, World"
```

$$\dfrac{\dfrac{\dfrac{}{\vdash_e\ e{:}(\forall\alpha{:}\star)(\texttt{GHC.Types.IO}\alpha)\to(\texttt{GHC.Types.IO}\alpha)}\text{Global}}{\vdash_e\ e{:}(\texttt{GHC.Types.IO()})\to(\texttt{GHC.Types.IO()})}\text{AppT} \qquad \dfrac{}{\vdash_e\ e{:}\texttt{GHC.Types.IO()}}\text{Global}}{\vdash_e\ e{:}\texttt{GHC.Types.IO()}}\text{App}$$

The `HelloWorld.hs` program, along with the result of running `inplace/bin/ghc-stage2 -dcoqpass` on it and passing the result through `pdflatex`.

```
fib n =
  ifThen (n==zero)
         one
         (ifThen (n==one)
                 one
                 ((fib (n - one)) + (fib (n - two))))
```



A program to compute the $\mathtt{n}^{\text{th}}$ number in the Fibonacci sequence, and its typing proof as reconstructed by running `inplace/bin/ghc-stage2 -dcoqpass` on it and passing the result through `pdflatex`. Even small and simple programs produce enormous typing proofs.

Figure 4.4: The well-typedness proof for a 26-line program in the `garrow` package tutorial, included strictly for comic value.

vs. application other tycons) and constructs which have their own introduction forms (e.g., type-lambda and coercion-lambda) have separate inductive type constructors in HaskWeak. HaskWeak variables are represented as a type which wraps a CoreVar and is indexed by the variable's sort (expression variable, type variable, coercion variable).

- **HaskStrong** is a very-dependently-typed version of HaskWeak; any value of type HaskStrong is a valid, well-Haskell-typed Haskell term. Type variables and coercion variables are represented using PHOAS [Chl08] since the rules for Haskell well-typedness involve type-level reduction. Value variables are represented in first-order style since Haskell well-typedness does not involve *value* reduction. Any type with decidable equality and an infinite supply of fresh values may be used to represent HaskStrong variables.
- **HaskProof** is an explicit natural-deduction proof tree, in System FC$^{\alpha}$, of the well-typedness of a given Haskell term. Types and coercions are represented in the same way as in HaskStrong, but value variables are represented Gentzen style with explicit structural operations.

Conversions in both directions between all adjacent representation pairs in the pipeline are provided; in the "forward" direction these conversions are partial functions (i.e., they may fail) and in the "reverse" direction the conversions are total. Support is also included for turning a HaskProof into a LaTeX document displaying the actual proof tree, as shown in Figure 4.4. These proofs are *unlabeled deductions* [Bar84]; variable names and expressions do not appear in the proof tree because they can be recovered from its structure.

Invoking GHC with the new flag `-fcoqpass` will run all the top-level bindings through the function `coqPassCoreToCore` defined in `Extraction.v` immediately after GHC's "desugarer" pass; the bindings returned by that function are then sent through the rest of the compiler. Invoking with the new flag `-ddump-proofs` will pass those bindings instead to `coqPassCoreToString` and dump the string it returns. Finally, `-ddump-coqpass` will dump the core expression returned by `coqPassCoreToCore`.

4.3 Kappa Calculus as the Minimal Object Language

As explained in the circuits examples of Chapter 2, one of the major goals of this work is to allow object languages which lack first-class functions. The straightforward approach to restricting first-class functions is a "syntactic separation" in the grammar for types:

$\sigma ::= \texttt{Bool} \mid \texttt{Int} \mid \ldots$ *(ground types)*

$\tau ::= \sigma \mid \sigma \texttt{ -> } \tau$ *(first-order types)*

This approach is effective, but 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 expressions; the duplication of effort grows rapidly: most language constructs must occur in two different "flavors."

Hasegawa's κ-calculus [Has95] provides a more manageable approach, originally motivated as a *syntax for morphisms in a contextually-closed category.* κ-calculus can also be seen[1] as a type theory for Milner's algebraic theory of action calculi [Mil96]. The following grammar is taken from Hasgawa's paper [Has95, Section 3], substituting[2] symbols which are more familiar to Haskell users.

τ ::= `()` | τ`*`τ | ... *(types)*

k ::= x | `id` | `first` k | $\lambda x{:}\tau$ `->` k | k `>>>` k *(expressions)*

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

$$\frac{}{\vdash \texttt{id} : \tau\texttt{\char`~\char`~>}\tau}\,[\mathsf{Id}] \qquad \frac{}{x : \tau\texttt{\char`~\char`~>}\tau' \vdash x : \tau\texttt{\char`~\char`~>}\tau'}\,[\mathsf{Var}]$$

$$\frac{\Gamma \vdash k : \tau_1\texttt{\char`~\char`~>}\tau_2}{\Gamma \vdash \texttt{first}\ k : \tau\texttt{*}\tau_1\texttt{\char`~\char`~>}\tau\texttt{*}\tau_2}\,[\mathsf{First}]$$

$$\frac{\Gamma \vdash k_1{:}\tau_1\texttt{\char`~\char`~>}\tau_2 \qquad \Gamma \vdash k_2{:}\tau_2\texttt{\char`~\char`~>}\tau_3}{\Gamma \vdash k_1\texttt{>>>}k_2 : \tau_1\texttt{\char`~\char`~>}\tau_3}\,[\mathsf{Comp}]$$

$$\frac{\Gamma, x{:}\texttt{()}\texttt{\char`~\char`~>}\tau_1 \vdash k{:}\tau_2\texttt{\char`~\char`~>}\tau_3}{\Gamma \vdash \lambda x{:}\tau_1\ \texttt{->}\ k\ : \tau_1\texttt{*}\tau_2\texttt{\char`~\char`~>}\tau_3}\,[\kappa\text{-}\mathsf{Abs}]$$

A typing judgment $\Gamma \vdash k : \tau_1$`~~>`τ_2 assigns an expression k a *pair* of types: τ_1 is the *source type* and τ_2 is the *target type.* Similarly, contexts Γ associate a pair of types to each variable. The source type of an expression can be thought of as a list of arguments, represented as a "right imbalanced" `*`-tree terminated by `()`. For example, a function taking exactly three arguments of types `A`, `B`, and `C` and returning a result of type `D` would have the types `A*(B*(C*())) ~~> D`. [3]

The following rule is also admissible; the derivation proceeds by straightforward induction on the proof of $\tau_1 \rightsquigarrow \tau_2$; the important cases are shown in Figure 4.5:

[1]Specifically, κ-abstraction corresponds to the "non-functorial" version of the abstraction operator $\texttt{ab}_x$ described in [Mil96, Section 4.9]. The κ-calculus expression `first f` is similar to the action structure `f` $\otimes$ `id`, although the algebraic laws of action calculi [Mil96, Section 2.2] require that for all `f` and `g` the equalities (`f` $\otimes$ `id`)`>>>`(`id` $\otimes$ `g`) = (`f>>>id`) $\otimes$ (`id>>>g`) = (`id>>>f`) $\otimes$ (`g>>>id`) = (`id` $\otimes$ `g`)`>>>`(`f` $\otimes$ `id`) hold. Since these equalities do not necessarily hold for all arrows, the theory of action calculi is not quite general enough for use as syntax for arrows or generalized arrows.

[2]We write `()` for 1, `first` for `lift`, `*` for $\otimes$, `f>>>g` for `g` $\circ$ `f`, and $\lambda x{:}\tau$`->`k for $\kappa x.k$.

[3]In both Hasegawa's and Milner's presentations the underlying category's monoidal structure is assumed to be strict; as a consequence, the type *equalities* `()*`τ = τ = τ`*()` and τ_1`*`(τ_2`*`τ_3) = (τ_1`*`τ_2)`*`τ_3 hold. In this chapter these are assumed only to be *isomorphisms*; they will be invoked explicitly via a family of typing rules [$\cong$].

$$\frac{\tau_1 \rightsquigarrow \tau_2 \qquad \Gamma \vdash k : \tau_2\texttt{\~{}\~{}>}\tau_3}{\Gamma \vdash k : \tau_1\texttt{\~{}\~{}>}\tau_3}\,[\cong]$$

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

$$\frac{\begin{array}{c}\Gamma, x{:}\texttt{()}\texttt{\~{}\~{}>}A \vdash k_x : \texttt{()}\texttt{\~{}\~{}>}A \\ \Gamma, x{:}\texttt{()}\texttt{\~{}\~{}>}A \vdash k : B\texttt{\~{}\~{}>}C\end{array}}{\Gamma \vdash \textbf{letrec}\ x = k_x\ \textbf{in}\ k : B\texttt{\~{}\~{}>}C}\,[\text{LetRec}]$$

The most significant difference between κ-calculus and λ-calculus is the typing rule for abstractions; as with λ-abstraction, the typing rule for κ-abstraction is applicable when the body of the abstraction is typeable under an additional assumption about the type of the abstracted variable. However, in the case of κ-calculus, that assumption must have the unit type `()` as the source type of the free variable. This is how the first order nature of κ-calculus is enforced. If polymorphism is added there is no need for two "flavors" of each language construct since the first order restriction is enforced at the site of *use* rather than the site of *binding*.

In the syntactically-separated first-order λ-calculus a function taking two arguments of types `A` and `B` and yielding a result of type `C` has the type `A -> (B -> C)`. In κ-calculus such a function has the pair of types `(A*(B*()))~~>C`. The `()` is necessary as an indication that `B` is the *second and final argument* rather than *a list of all but the first argument*; this is similar to how the principal typings of the Haskell expressions `a:b:[]` and `a:b` differ in the type of `b`.

Juxtaposition $k_1 k_2$ is an abbreviation for `first` k_2 `>>>` k_1; its typing rule [κ-App] is derivable:

$$[\text{Comp}]\ \frac{[\cong]\ \dfrac{[\text{First}]\ \dfrac{\Gamma \vdash k_2 : \texttt{()}\texttt{\~{}\~{}>}\tau_1}{\Gamma \vdash \texttt{first}\ k_2 : \texttt{()*}\tau_2\texttt{\~{}\~{}>}\tau_1\texttt{*}\tau_2}}{\Gamma \vdash \texttt{first}\ k_2 : \tau_2\texttt{\~{}\~{}>}\tau_1\texttt{*}\tau_2} \qquad \Gamma \vdash k_1 : \tau_1\texttt{*}\tau_2\texttt{\~{}\~{}>}\tau_3}{\Gamma \vdash \texttt{first}\ k_2\ \texttt{>>>}\ k_1 : \tau_2\texttt{\~{}\~{}>}\tau_3}$$

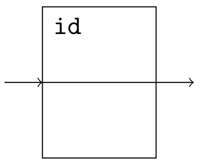
Unlike uncurried function types `(A,B,C)->D`, κ-calculus functions can be partially applied without knowledge of their arity – just as with curried functions in λ-calculus *but without the use of higher types.* For example, the following judgment is derivable for any α, and therefore for functions `f` of any arity greater than zero:

$$k_1 : \texttt{A*}\alpha\texttt{\~{}\~{}>B}\ ,\ \ k_2{:}\texttt{()\~{}\~{}>A} \quad \vdash \quad k_1\, k_2 : \alpha\texttt{\~{}\~{}>B}$$

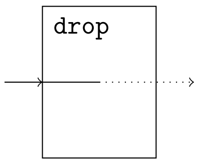
The ability to type a partial application without knowing the function's arity is crucial to ensuring that standard Damas/Hindley/Milner type inference algorithms can be used.

To improve readability, the parser and pretty-printer treat `*` as right-associative and omit parentheses where possible. Also, a source type enclosed in an outermost layer of parentheses is another way of writing a trailing `()`. This means that the type `A*(B*(C*()))~~>D` can be written as `(A*B*C)~~>D`; however, this is different from `A*B*C~~>D`, which abbreviates `A*(B*C)~~>D`.

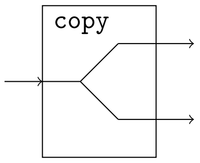
$$\begin{aligned}
\texttt{id}_\tau &= \lambda x.x && : \tau\texttt{->}\tau \\
\texttt{drop} &= \lambda x.\texttt{id} && : \tau\texttt{->()} \\
\texttt{copy} &= \lambda x.x \texttt{ >>> first } x && : \tau \texttt{ -> } \tau \times \tau \\
\texttt{swap} &= \lambda x.\lambda y.x \texttt{ >>> first } y && : \tau_1 \times \tau_2 \texttt{ -> } \tau_2 \times \tau_1
\end{aligned}$$



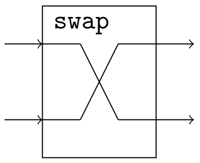
$$\dfrac{\dfrac{\dfrac{}{x{:}\texttt{()->}\tau \vdash x \;:\; \texttt{()->}\tau}\;\text{[Var]}}{\vdash \lambda x.x \;:\; \tau \times \texttt{()} \texttt{ -> } \tau}\;\text{[Abs]}}{\vdash \lambda x.x \;:\; \tau \texttt{ -> } \tau}\;\text{[CancelR]}$$



$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\vdash \texttt{id} \;:\; \texttt{()->()}}}{x{:}\texttt{()->}\tau \vdash \texttt{id} \;:\; \texttt{()->()}}\;\text{[Weak]}}{\vdash \lambda x.\texttt{id} \;:\; \tau \times \texttt{()} \texttt{ -> ()}}\;\text{[Abs]}}{\vdash \lambda x.\texttt{id} \;:\; \tau \texttt{ -> ()}}\;\text{[CancelL]}$$



$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}\texttt{()->}\tau \vdash x \;:\; \texttt{()->}\tau}\;\text{[Var]}}{x{:}\texttt{()->}\tau \vdash \texttt{first } x \;:\; \texttt{()} \times \tau\texttt{->}\tau \times \tau}\;\text{[First]}}{x{:}\texttt{()->}\tau \vdash \texttt{first } x \;:\; \tau\texttt{->}\tau \times \tau}\;\text{[CancelL]} \qquad \dfrac{}{x{:}\texttt{()->}\tau \vdash x \;:\; \texttt{() -> } \tau}\;\text{[Var]}}{x{:}\texttt{()->}\tau, x{:}\texttt{()->}\tau \vdash x \texttt{ >>> first } x \;:\; \texttt{() -> } \tau \times \tau}\;\text{[Comp]}}{x{:}\texttt{()->}\tau \vdash x \texttt{ >>> first } x \;:\; \texttt{() -> } \tau \times \tau}\;\text{[Contr]}}{\vdash \lambda x.x \texttt{ >>> first } x \;:\; \tau \times \texttt{() -> } \tau \times \tau}\;\text{[Abs]}}{\vdash \lambda x.x \texttt{ >>> first } x \;:\; \tau \texttt{ -> } \tau \times \tau}\;\text{[CancelR]}$$



$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y{:}\texttt{()->}\tau_2 \vdash y \;:\; \texttt{()->}\tau_2}\;\text{[Var]}}{y{:}\texttt{()->}\tau_2 \vdash \texttt{first } y \;:\; \texttt{()} \times \tau_1\texttt{->}\tau_2 \times \tau_1}\;\text{[First]}}{y{:}\texttt{()->}\tau_2 \vdash \texttt{first } y \;:\; \tau_1\texttt{->}\tau_2 \times \tau_1}\;\text{[CancelL]} \qquad \dfrac{}{x{:}\texttt{()->}\tau_1 \vdash x{:}\texttt{()->}\tau_1}\;\text{[Var]}}{y{:}1\texttt{->}\tau_2, x{:}\texttt{()->}\tau_1 \vdash x \texttt{ >>> first } y \;:\; \texttt{() -> } \tau_2 \times \tau_1}\;\text{[Comp]}}{x{:}\texttt{()->}\tau_1, y{:}1\texttt{->}\tau_2 \vdash x \texttt{ >>> first } y \;:\; \texttt{() -> } \tau_2 \times \tau_1}\;\text{[Exch]}}{x{:}\texttt{()->}\tau_1 \vdash \lambda y.x \texttt{ >>> first } y \;:\; \tau_2 \times \texttt{() -> } \tau_2 \times \tau_1}\;\text{[Abs]}}{x{:}\texttt{()->}\tau_1 \vdash \lambda y.x \texttt{ >>> first } y \;:\; \tau_2 \texttt{ -> } \tau_2 \times \tau_1}\;\text{[CancelR]}}{\vdash \lambda x.\lambda y.x \texttt{ >>> first } y \;:\; \tau_1 \times \tau_2\texttt{->}\tau_2 \times \tau_1}\;\text{[Abs]}$$

Figure 4.5: The Penrose diagrams and κ-calculus typing derivations for `id`, `drop`, `copy`, and `swap`

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

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

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

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

instance Arrow a => GArrowReify a (,) () x y x y
 where
  ga_reify     =  arr

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

instance ArrowApply a => GArrowApply a (,) () a
 where
  ga_applyl    = ga_swap >>> app
  ga_applyr    = app
```

Figure 4.6: A `GArrow` instance for ordinary `Arrows`

4.4 Flattening with various `GArrow` Instances

4.4.1 A `GArrow` instance for Arrows

One of the most simple and direct `GArrow` instances is one that turns an ordinary arrow (instance of `Arrow`) into a generalized arrow. This instance is shown in Figure 4.6.

```
newtype Code x y =
 Code  unC :: forall a. <[ x -> y ]>@a

instance Category Code where
 id     = Code <[ λx -> x ]>
 f . g  = Code <[ λx -> ~~(unC f) (~~(unC g) x) ]>

instance GArrow Code (,) () where
 ga_first  f  = Code <[ λ(x,y) -> (~~(unC f) x,y)]>
 ga_second f  = Code <[ λ(x,y) -> (x,~~(unC f) y)]>
 ga_cancell   = Code <[ λ(_,x) -> x ]>
 ga_cancelr   = Code <[ λ(x,_) -> x ]>
 ga_uncancell = Code <[ λx -> ((),x) ]>
 ga_uncancelr = Code <[ λx -> (x,()) ]>
 ga_assoc     = Code <[ λ((x,y),z) -> (x,(y,z)) ]>
 ga_unassoc   = Code <[ λ(x,(y,z)) -> ((x,y),z) ]>

instance GArrowDrop Code (,) () where
 ga_drop        = Code <[ λ_ -> u ]>

instance GArrowCopy Code (,) () where
 ga_copy        = Code <[ λx -> (x,x) ]>

instance GArrowSwap Code (,) () where
 ga_swap        = Code <[ λ(x,y) -> (y,x) ]>
```

Figure 4.7: A `GArrow` instance for code types

4.4.2 A `GArrow` instance for Unflattening

Another simple `GArrow` instance provides a generalized arrow implementation for code values (i.e. values of type `<[t]>`) in a two-level language. This instance is shown in Figure 4.7; the `newtype Code` declaration is required as a hint to Haskell's instance inference engine. This instance can be considered an "unflattening instance" since what it does (turn `GArrow` expressions into equivalent two-level expressions) the opposite of what the flattening pass does (turn two-level expressions into equivalent `GArrow` expressions).

4.4.3 A `GArrow` Instance for Pretty-Printing

A slightly more complicated `GArrow` instance is `GArrowPretty`, which uses pretty-printing combinators [Hug95] to turn a `GArrow` expression into equivalent Haskell source code; it is analogous to the `Show` class. Here is a partial definition:

```
  import Text.PrettyPrint.HughesPJ
  type Precedence = Int
  data SourceCode a b = SC Precedence Doc

  instance GArrow SourceCode (,) () where
    ga_first (SC 0 f) = SC 1 $ text "ga_first" f
    ga_first (SC _ f) = SC 1 $ text "ga_first" $ parens f
    -- ...
    ga_cancell        = SC 0 $ text "ga_cancell"
    -- ..
```

It is also possible to give a `Hardware` instance for `SourceCode`:

```
  instance Hardware SourceCode where
    high        = SC 0 $ text "high"
    -- ...
    fifo   len  = SC 0 $ text "fifo"   <+> (text . show) len
    probe  id   = SC 0 $ text "probe"  <+> (text . show) id
    loop   vals = SC 0 $ text "loop"   <+> brackets $ hcat $ vals'
      where
       vals' = punctuate comma $ map (text . show) vals
```

Using these instances, it is possible to write:

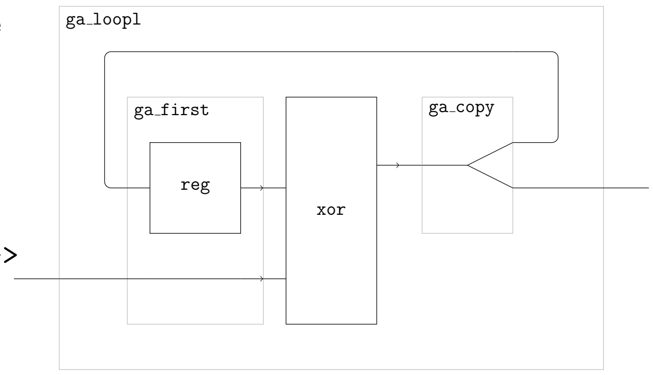
```
  showHardware ::
     (forall g . (Hardware g, GArrow g) => g x y)
     -> String
```

Notice the rank-2 type; unlike circuit-transformers the `showHardware` operation must be able to supply its own type `g` and `GArrow` instance to its argument; on the other hand, the type of the input `x` and output `y` are chosen by the caller. The `GArrow` representation of object language expressions is an example of *induction principle representation* of datatypes in System F_ω. This representation was as described by Pfenning and Paulin-Mohring [PP90, Definitions 13 and 14]; it was first used by Pfenning and Lee [PL91] in LEAP to represent polymorphic λ-calculus.

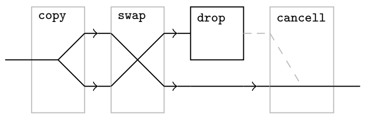
Recall the `oscillator` program from Chapter 2:

```
class GArrowLoop g => Hardware g where
  -- ...
  xor :: g Bit Bit
  reg :: g Bit Bit

oscillator :: Hardware g => g Bit Bit
oscillator = ga_loopl (ga_first reg >>>
                       xor >>>
                       ga_copy)
```



```
tikzExample1 =
  ga_copy          >>>
  ga_swap          >>>
  ga_first ga_drop >>>
  ga_cancell
```



```
tikzExample2 =
  ga_uncancelr        >>>
  ga_first ga_copy    >>>
  ga_swap             >>>
  ga_second
    (ga_first ga_drop >>>
     ga_cancell)      >>>
  ga_cancell
```

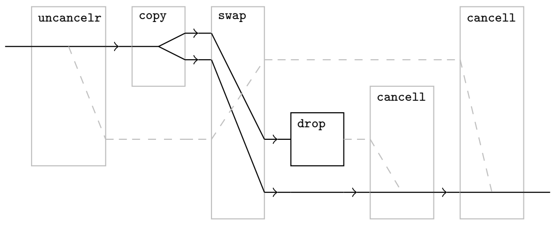


Figure 4.8: Two sample `GArrow` expressions and their visualization using `GArrowTikZ`.

Instantiating `oscillator` with `GArrowPretty` and rendering the resulting `Doc` is just what one would expect:

```
  ga_loopl (ga_first reg >>>
            xor >>>
            ga_copy)
```

The `SourceCode` instance demonstrates another reason for using `GArrows` instead of `Arrows`. The `Arrow` type class allows `arr` to be applied to any Haskell function `f`. Unfortunately, arbitrary Haskell functions cannot be inspected – there is no `Show` instance for function types – so there can never be a complete `Show` implementation for ordinary arrows.

4.4.4 A `GArrow` Instance for Drawing Penrose Diagrams

As explained in Chapter 2, reading and writing generalized arrow expressions in text form can be quite difficult for larger examples; it is often more convenient to visualize `GArrow` terms as Penrose diagrams. The instance `GArrowTikZ` renders these diagrams as TikZ code, using the `lp_solve` linear solver to produce a pleasing visual layout. Figure 4.8 shows the result of using the `GArrowPretty` instance on two larger examples.

The `pow` program from Chapter 2 was a very crude implementation of the exponentiation function; in particular it required time $O(\mathtt{n})$ to compute $\mathtt{x}^{\mathtt{n}}$. A slightly more intelligent implementation would divide the exponent by two on each recursive call instead of decrementing it; an implementation in this style, along with its Penrose diagram for the $\mathtt{9}^{12}$ case, is shown in Figure 4.9.

```
pow_smart const times =
  <[ λy -> ~~(pow 9 12) ]>
      where
        pow 0 x                  = const (1::Int)
        pow 1 x                  = const x
        pow n x | n `mod` 2 == 0 =
                    <[ let x_to_n_over_2 = ~~(pow (n `div` 2) x)
                          in  ~~times x_to_n_over_2 x_to_n_over_2 ]>
                | otherwise    =
                    <[ ~~times ~~(pow (n-1) x) ~~(const x) ]>
```

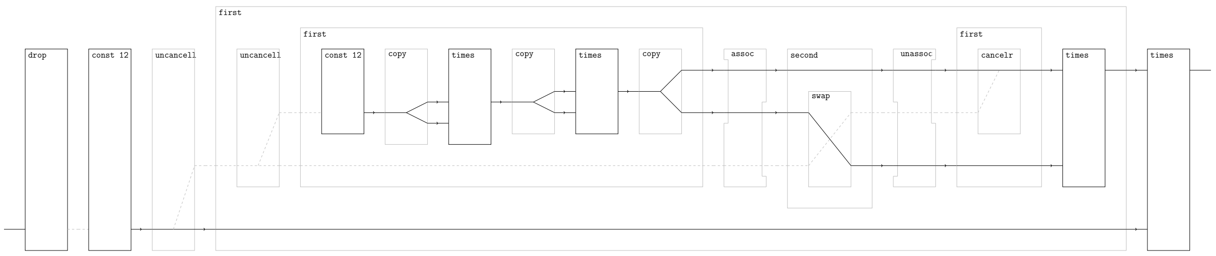


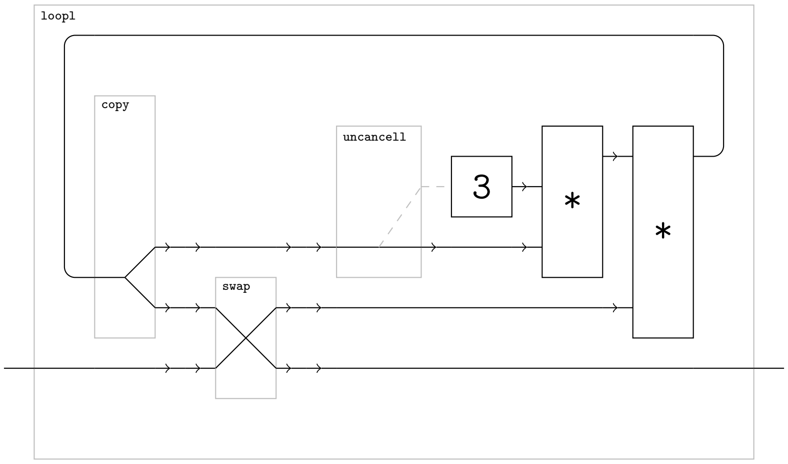
Figure 4.9: A slightly better implementation of `pow`, and its Penrose diagram.

4.5 Repetition vs Feedback

Before concluding this chapter, a few words are in order regarding the *meaning* of the brackets, and why they can't simply be erased. One of the most important reasons is that recursion inside the brackets has a meaning completely different from recursion outside the brackets.

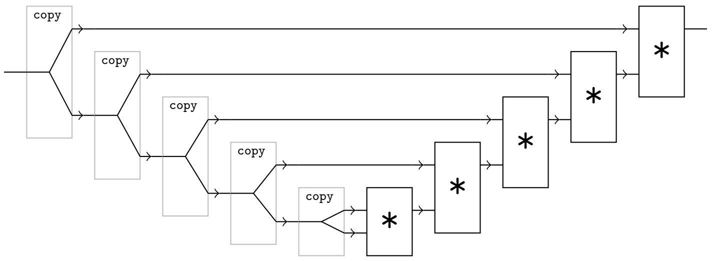
Recursion *outside the code brackets* represents repetitive structures, whereas recursion *inside the brackets* represents feedback loops. This is illustrated by the following two examples; the first shows recursion *inside the brackets*, which produces feedback:

```
  feedbackDemo =
     <[ λx -> let out   = (3 * out) * x
              in  out ]>
```



The following example demonstrates recursion *outside the brackets*, which produces repetitive structures, using the pow_smart function from Figure 4.9:

```
  repetitiveStructureDemo = pow_smart 6
```



This distinction between recursion inside the brackets and structure recursion outside the brackets is closely related to monadic *value recursion* [EL00]. In fact, an expression which uses recursive let inside the brackets will be flattened to a GArrow expression which relies on ga_loopl. This expression may then be instantiated for any MonadFix, since Control.Arrow.Kleisli provides a ArrowLoop instance for any MonadFix, and the appendix provides a GArrowLoop instance for any ArrowLoop. When instantiated in this manner, ga_loopl will be realized as mfix. By contrast, recursion outside the brackets is not altered by the flattener.

The contrast between repetitive structure and feedback is a straightforward example of how expression brackets convey essential information and can't simply be erased. The same is true of type brackets, but a different example comes to mind: in asynchronous dataflow, a *pair of streams* is very different from a *stream of pairs* – the latter has more stringent synchronization behavior. The inability to make this distinction forced the Fudgets library [CH93] to co-opt the *coproduct* structure of the underlying type system to represent pairs of streams, causing an anomaly that Paterson notes [Pat01, Section 5.1] in the type of the Fudgets loop function.

Chapter 5

Extended Example

This chapter serves as a small yet nontrivial example of a metaprogramming task carried out using a two-level language and generalized arrows. The task in question is one of hardware design: create a bit-serial circuit that implements the SHA-256 hash function.

5.1 Background

Hardware design in functional languages has a long history, starting with Backus' 1978 lecture [Bac78] introducing the combinator language FP, which served as the basis for Sheeran's μFP [She84]. Since then, many researchers have investigated the use of functional programming languages to describe hardware circuits. There are several challenges in this area; this chapter will focus on four in particular:

1. Enforcing a phase distinction
2. Disallowing higher-order functions in circuits
3. The need for application-specific compiler modifications
4. Distinguishing recursive structure from feedback

Not all of these challenges are necessarily regarded as major problems, but the solutions chosen are helpful in distinguishing different approaches.

Enforcing a phase distinction. Hardware design in functional languages is an example of heterogeneous metaprogramming: a software program generates a hardware program, and the resulting hardware program must be free of any dependencies on further software computation. This is independence is called a *phase distinction*. Monadic systems such as Chalmers Lava ([Bje+98] [Bje+98]) make a phase distinction in expressions by limiting the primitives which may be used to construct circuits. However, there is no phase distinction in the type system; for example, `CircuitMonad m => m Bit -> m Bit` is both the type of one-input/one-output circuits as well as the type of circuit-transformers that turn a one-output circuit into another one-output circuit. A few systems – notably reFL$^{\mathrm{ect}}$ [GMO05] and Sheard's Ωmega [SL07] – are based on multi-level languages [NN92] in which the type $\langle \tau_1$`->`$\tau_2 \rangle$ of object language functions is different from the type $\langle \tau_1 \rangle$`->`$\langle \tau_2 \rangle$ of meta language functions that transform object language expressions. In homogeneous multi-level languages these types are related by the "isomorphism for code types" [TS00, Section 8].

Disallowing higher-order functions in circuits. Functional languages allow higher-order functions: functions which take other functions as arguments. Unfortunately this does not correspond to anything meaningful in the world of circuits: one cannot have a circuit which takes another circuit as its input. Any approach to hardware design in a functional language must address this issue: higher-order functions must not be allowed *in circuits themselves*, but are desirable in the software programs which *generate the circuits*. Most existing work addresses this issue via a sort of implied well-foundedness lemma, whereby circuits with a type suitable for top-level operations (synthesis, execution, etc) can be proved to be free of irreducible higher-order subcircuits. For example, Chalmers Lava represents a circuit with input of type `A` and output of type `B` as a Haskell value of type `CircuitMonad m => A -> m B`. Since `Circuit` is a subclass of `Monad`, it is possible to create values of type `CircuitMonad m => (Bit -> m Bit) -> m B`. Values of such a type would appear to correspond to "higher-order circuits." Fortunately, since the primitives of the `CircuitMonad` class (such as `xor`, `nand`, etc) do not include higher-order operations, it can be shown that any paradoxical "higher-order circuits" occur within ordinary circuits only as part of a reducible subexpression.

The need for application-specific compiler modifications. Converting a functional program to Verilog or VHDL requires an understanding of the application domain (hardware). Ideally the compiler writer should not have to understand the application domains in which the compiler might be used. This is usually a disadvantage of systems based on multi-level languages; as Sheard and Linger [SL07] write "there is usually only a single object-language, and it must be built into the meta-language." Chalmers Lava avoids application-specific compiler modifications by using a *monadic embedding* of the circuit object language into unmodified Haskell. Producing Verilog is then a matter of selecting the symbolic interpretation [Bje+98, Section 3.2] by supplying the appropriate `instance CircuitMonad T` for some type `T` that represents Verilog program text. Kansas Lava [Gil+09] avoids hardware-specific compiler modifications by requiring only that compiler's `IO` monad implementation support the `StableName` extension [Gil09].

Distinguishing recursive structure from feedback. One of the most difficult problems with representing circuits in functional languages is dealing with sharing. In Chalmers Lava's monadic embedding this distinction is made by using different language constructs for recursion and feedback: Haskell's `let` is used for recursive structure and a special `mfix` operator (of the `MonadFix` class [EL00]) is used for feedback. In Hawk [MCL98] both repetitive structure and feedback are represented using Haskell's recursive `let` construct; in order to produce HDL it must be possible to distinguish these. Later versions of Hawk extracted this additional information from the program text using a mechanism called *observable sharing*

[CS99; Gil09]. Early versions of Kansas Lava took a similar approach, but replaced observable sharing with IO-based reification [Gil09]; later versions carry both a shallow embedding and a deep embedding at the same time [Gil+09].

5.2 Advantages of using Generalized Arrows

The example outlined this chapter demonstrates how an approach based on a two-level language and generalized arrows provides a unique combination of solutions to the challenges above:

1. The phase distinction is enforced using a two-level source language with environment classifiers [TN03]. The phase distinction extends to the types: the type of circuits with input `A` and output `B` is different from the type of circuit-transformers whose arguments has output type `A` and whose results have output type `B`.
2. Higher-order functions are kept out of circuits by restricting the object language to κ-calculus.
3. Application-specific compiler modifications are not required. The *flattening transformation*, which converts two-level programs into one-level programs polymorphic in a `GArrow` instance, is the only compiler modification involved and it is not specific to hardware design in any way.
4. Recursive structure is distinguished from feedback: recursive `let` bindings in the object language produce feedback while recursive `let` bindings in the meta language produce repetitive structure.

This chapter presents an example application of this approach: a bit-serial circuit which searches for SHA-256 hash collisions. A two-level circuit-building program is passed through the GHC flattening pass (which is not specific to hardware design in any way) and the resulting one-level program is combined with a `GArrow` instance that emits Verilog code; this instance is an ordinary Haskell library which can be written without knowledge of compiler internals.

5.3 A `GArrow` instance to emit Verilog

This section will describe the `GArrowVerilog` instance for generating HDL output. The Verilog output is produced in two passes. The first pass allocates `wire` declarations to components using a bidirectional propagation algorithm similar to unification. Once `wire` declarations have been allocated, producing the HDL text is straightforward. Here is the result of instantiating `oscillator` with `GArrowVerilog`:

```
data Bit = High | Low

class Hardware g where
  high   :: <[           () ~~> Bit ]>@g
  low    :: <[           () ~~> Bit ]>@g
  not    :: <[        (Bit) ~~> Bit ]>@g
  xor    :: <[    (Bit*Bit) ~~> Bit ]>@g
  or     :: <[    (Bit*Bit) ~~> Bit ]>@g
  and    :: <[    (Bit*Bit) ~~> Bit ]>@g
  mux2   :: <[ (Bit*Bit*Bit) ~~> Bit ]>@g
  maj3   :: <[ (Bit*Bit*Bit) ~~> Bit ]>@g
  reg    :: <[        (Bit) ~~> Bit ]>@g
  loop   :: [Bool] -> <[  () ~~> Bit ]>@g
  fifo   :: Int   -> <[(Bit) ~~> Bit ]>@g
  probe  :: Int   -> <[(Bit) ~~> Bit ]>@g
  oracle :: Int   -> <[   () ~~> Bit ]>@g
```

Figure 5.1: Primitives needed for the SHA-256 circuit

```
  module demo(inputWire)
    wire wire0;
    wire wire1;
    wire wire2;
    reg (wire1, wire0);
    xor (wire2, wire1, inputWire);
  endmodule
```

The SHA-256 engine is defined in terms of the primitives shown in Figure 5.1, 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.

5.4 Basic Building Blocks

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 ]>
```

The `xor3` can be used to code a bit-serial adder, shown below. 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 =
  <[ λin1 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 =
  <[ λinput ->
     let sel   = ~~(loop [ i >= 32-n | i<-[0..31] ])
         fifo1 = ~~(fifo (32-n)) input
         fifo2 = ~~(fifo  32   ) fifo1
     in  mux2 sel fifo1 fifo2
   ]>
```

5.5 SHA-256 Implementation

Using these subcircuits, it is now possible to express the SHA-256 algorithm, which can be found in Figure 5.2. 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.

Below is 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.

```
sha256round =
  <[ λload 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
   ]>
```

The modified GHC infers the following type for `sha256round`:

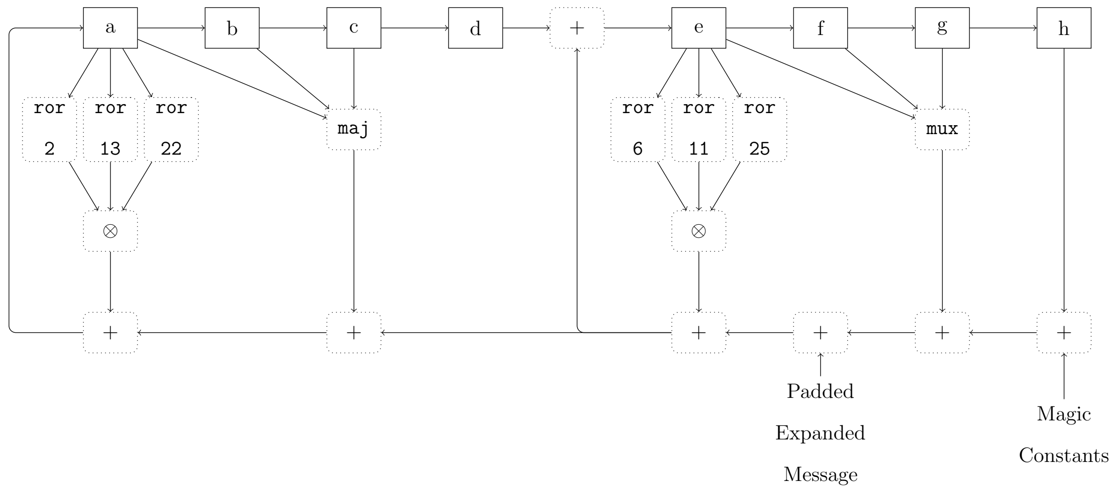


Figure 5.2: The SHA-256 Algorithm.

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

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

Bibliography

[AH10] Asada, Kazuyuki and Hasuo, Ichiro. Categorifying Computations into Components via Arrows as Profunctors. *Electr. Notes Theor. Comput. Sci* 264, 2 (2010), 25–45. doi: `10.1016/j.entcs.2010.07.012`.

[Ali+05] Alimarine, Artem et al. There and back again: arrows for invertible programming. In: 2005, 86–97. doi: `10.1145/1088348.1088357`.

[Asa10] Asada, K. Arrows are strong monads. In: *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*. ACM. 2010, 33–42. doi: `10.1145/1863597.1863607`.

[Atk08] Atkey, R. What is a categorical model of arrows? *Mathematically Structured Functional Programming* (2008). doi: `10.1016/j.entcs.2011.02.014`.

[Awo06] Awodey, Steve. *Category Theory*. Oxford Logic Guides, 2006.

[Bac78] Backus, John. Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM* 21, 8 (Jan. 1978), 613–641. doi: `10.1145/359576.359579`.

[Bar84] Barendregt, H. P. *The Lambda Calculus — Its Syntax and Semantics*. Revised. Vol. 103. Studies in Logic and the Foundations of Mathematics. North-Holland, 1984. isbn: 0-444-86748-1.

[BCS97] Blute, R F, Cockett, J R B, and Seely, R A G. Categories for Computation in Context and Unified Logic. *Journal of Pure and Applied Algebra* 116 (1997), 49–98. doi: `10.1016/S0022-4049(96)00162-4`.

[Bje+98] Bjesse, P et al. Lava: hardware design in Haskell. *ICFP '98* (Jan. 1998). doi: `10.1145/289423.289440`.

[CH93] Carlsson, Magnus and Hallgren, Thomas. FUDGETS: A Graphical User Interface in a Lazy Functional Language. In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. FPCA '93. ACM, Copenhagen, Denmark, 1993, 321–330. isbn: 0-89791-595-X. doi: `10.1145/165180.165228`.

[Chl08] Chlipala, Adam. Parametric higher-order abstract syntax for mechanized semantics. *ICFP '08* (Sept. 2008). doi: `10.1145/1411204.1411226`.

[CMT04] Calcagno, Cristiano, Moggi, Eugenio, and Taha, Walid. ML-Like Inference for Classifiers. In: *Programming Languages and Systems*. Ed. by Schmidt, David. Vol. 2986. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, 79–93. isbn: 978-3-540-21313-0. doi: `10.1007/978-3-540-24725-8_7`.

[Cro94] Crole, Roy. *Categories for Types.* Cambridge University Press, 1994. isbn: 0521450926.

[CS99] Claessen, Koen and Sands, David. Observable Sharing for Functional Circuit Description. In: *Advances in Computing Science — ASIAN'99.* Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1999, 78–78. doi: 10.1007/3-540-46674-6_7.

[Dav96] Davies. A Temporal-Logic Approach to Binding-Time Analysis. In: *LICS Symposium.* 1996. doi: 10.1109/LICS.1996.561317.

[Dim05] Dimitrios Vytiniotis, Stephanie Weirich, Simon Peyton Jones. Practical type inference for arbitrary-rank types (July 2005). doi: 10 . 1017 / S0956796806006034.

[EL00] Erkök, Levent and Launchbury, John. Recursive monadic bindings. In: *Proceedings of ICFP '00.* ACM, 2000, 174–185. doi: 10.1145/351240.351257.

[Gen35] Gentzen, Gerhard. Untersuchungen xfffdxfffdber das logische Schliexfffdxfffden. I. German. *Mathematische Zeitschrift* 39, 1 (1935), 176–210. issn: 0025-5874. doi: 10.1007/BF01201353.

[Gil+09] Gill, Andy et al. Introducing Kansas Lava. In: *21st International Symposium on Implementation and Application of Functional Languages.* LNCS 6041. LNCS 6041, Nov. 2009. doi: 10.1007/978-3-642-16478-1_2.

[Gil09] Gill, Andy. Type-safe observable sharing in Haskell. In: *Haskell Symposium.* ACM, 2009, 117–128. isbn: 978-1-60558-508-6. doi: 10.1145/1596638.1596653.

[GJ91] Gomard, C. K. and Jones, N. D. A Partial Evaluator for the Untyped Lambda-Calculus. *Journal of Functional Programming* 1, 1 (1991), 21–69. doi: 10.1017/S0956796800000058.

[GJ96] Glueck, R. and Joergensen, J. Fast Binding-Time Analysis for Multi-level Specialization. *Lecture Notes in Computer Science* 1181 (1996), 261. issn: 0302-9743. doi: 10.1007/3-540-62064-8_22.

[GMO05] Grundy, Jim, Melham, Tom, and O'leary, John. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming* 16, 2 (Jan. 2005), 157–196. doi: 10.1017/S0956796805005757.

[Has95] Hasegawa, Masahito. Decomposing typed lambda calculus into a couple of categorical programming languages. In: *Category Theory and Computer Science.* Ed. by Pitt, David, Rydeheard, David, and Johnstone, Peter. Vol. 953. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1995, 200–219. isbn: 978-3-540-60164-7. doi: 10.1007/3-540-60164-3_28.

[HJ06] Heunen, Chris and Jacobs, Bart. Arrows, like Monads, are Monoids. *ENTCS* 158 (2006), 219–236. doi: 10.1016/j.entcs.2006.04.012.

[How80] Howard, W. The formulae-as-types notion of construction. In: *To H.B.Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism.* 1980. isbn: 0123490502.

[Hug00] Hughes, J. Generalising monads to arrows. *Science of computer programming* (Jan. 2000). doi: 10.1016/S0167-6423(99)00023-4.

[Hug04] Hughes, John. Programming with Arrows. In: *LNCS 3622.* Vol. 3622. 2004, 73–129. doi: 10.1007/11546382_2.

[Hug95] Hughes, John. The design of a pretty-printing library. In: *Advanced Functional Programming*. Ed. by Jeuring, Johan and Meijer, Erik. Vol. 925. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1995, 53–96. isbn: 978-3-540-59451-2. doi: `10.1007/3-540-59451-5_3`.

[JH06] Jacobs, B. and Hasuo, I. Freyd is Kleisli, for Arrows. In: *Workshop on Mathematically Structured Functional Programming (MSFP 2006)*. eWiC. 2006. doi: `Non-DOI-Publication`.

[JHI09] Jacobs, Bart, Heunen, Chris, and Ichiro, Hasuo. Categorical semantics for arrows. *Journal of Functional Programming* 19, 3-4 (2009), 403–438. doi: `10.1017/S0956796809007308`.

[Kel82] Kelly, G M. *Basic Concepts of Enriched Category Theory*. 1982. isbn: 0521287022.

[Lam69] Lambek, Joachim. Deductive Systems and Categories: II: Standard Constructions and Closed Categories. In: *Category theory, homology theory and their applications*. Lecture Notes in Mathematics 86. Springer-Verlag, 1969, 76–122. doi: `10.1007/BFb0079385`.

[Law63] Lawvere, F. William. Functorial Semantics of Algebraic Theories. PhD thesis. Columbia University, 1963.

[Law96] Lawvere, Bill. Adjointness in foundations. *Pure and Applied Mathematics* (Jan. 1996). doi: `10.1111/j.1746-8361.1969.tb01194.x`.

[LCH09] Liu, Hai, Cheng, Eric, and Hudak, Paul. Causal commutative arrows and their optimization. In: *ICFP 2009*. ACM, 2009, 35–46. isbn: 978-1-60558-332-7. doi: `10.1145/1596550.1596559`.

[LT06] Lengauer, Christian and Taha, Walid. Special Issue on the First MetaOCaml Workshop 2004. *Science of Computer Programming* 62, 1 (2006), 1–2. issn: 0167-6423. doi: `10.1016/j.scico.2006.05.001`.

[LWY10] Lindley, S., Wadler, P., and Yallop, J. The arrow calculus. *JFP* 20 (2010), 51–69. doi: `10.1017/S095679680999027X`.

[Mac71] Mac Lane, Saunders. *Categories for the Working Mathematician*. Springer, 1971. isbn: 0387984038.

[MCL98] Matthews, J., Cook, B., and Launchbury, J. Microprocessor specification in Hawk. In: *Computer Languages, 1998. Proceedings. 1998 International Conference on*. May 1998, 90–101. doi: `10.1109/ICCL.1998.674160`.

[Meg10] Megacz, Adam. *Multi-Level Languages are Generalized Arrows*. arXiv, `http://arxiv.org/abs/1007.2885`. July 2010.

[Meg11a] Megacz, Adam. Hardware Design with Generalized Arrows. In: *Implementation of Functional Languages 2011, Draft Proceedings*. Ed. by Gill, Andy. Technical Report ITTC-FY2012-TR-29952012-01. University of Kansas. 2011.

[Meg11b] Megacz, Adam. Hardware Design with Generalized Arrows. In: *Implementation and Application of Functional Languages 2011, Selected Papers*. Ed. by Gill, Andy and Hage, Jurriaan. Vol. 7257. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, 164–180. isbn: 978-3-642-34406-0. doi: `10.1007/978-3-642-34407-7_11`.

[Mil78] Milner, R. A Theory of Type Polymorphism in Programming. *JCSS* 17 (1978), 348–375. doi: `10.1016/0022-0000(78)90014-4`.

[Mil96] MILNER, Robin. Calculi for Interaction. *Acta Inf* 33, 8 (1996), 707–737. DOI: 10.1007/BF03036472.

[Mog91] MOGGI, E. Notions of Computation and Monads. *Information and Computation* 93 (1991), 55–92. DOI: 10.1016/0890-5401(91)90052-4.

[NN92] NIELSON, F. and NEILSON, H. R. *Two-Level Functional Languages*. Cambridge University Press, Cambridge, Mass., 1992. ISBN: 0521403847.

[Pat01] PATERSON, Ross. A New Notation for Arrows. In: *ICFP*. 2001, 229–240. DOI: 10.1145/507546.507664.

[Pen71] PENROSE, Roger. Applications of negative dimensional tensors. *Combinatorial mathematics and its applications* (1971).

[Pie02] PIERCE, Benjamin. *Types and Programming Languages*. The MIT Press, 2002. ISBN: 0262162091.

[PL91] PFENNING, Frank and LEE, Peter. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science* 89, 1 (1991), 137–159. ISSN: 0304-3975. DOI: 10.1016/0304-3975(90)90109-U.

[PP90] PFENNING, Frank and PAULIN-MOHRING, Christine. Inductively defined types in the Calculus of Constructions. In: *Mathematical Foundations of Programming Semantics*. Vol. 442. Lecture Notes in Computer Science. 10.1007/BFb0040259. Springer Berlin / Heidelberg, 1990, 209–228. ISBN: 978-0-387-97375-3. DOI: 10.1007/BFb0040259.

[PR97] POWER, John and ROBINSON, Edmund. Premonoidal Categories and Notions of Computation. *Mathematical Structures in Computer Science* 7, 5 (1997), 453–468. DOI: 10.1017/S0960129597002375.

[PT97] POWER, J and THIELECKE, H. Environments, Continuation Semantics and Indexed Categories. *Lecture Notes in Computer Science* 1281 (1997). DOI: 10.1007/BFb0014560.

[PT99] POWER and THIELECKE. Closed Freyd- and κ-Categories. In: 1999. DOI: 10.1007/BFb0014560.

[Sco93] SCOTT, Dana S. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theor. Comput. Sci.* 121, 1-2 (1993), 411–440. ISSN: 0304-3975. DOI: 10.1016/0304-3975(93)90095-B.

[Sel11] SELINGER, P. A Survey of Graphical Languages for Monoidal Categories. English. In: *New Structures for Physics*. Ed. by COECKE, Bob. Vol. 813. Lecture Notes in Physics. Springer Berlin Heidelberg, 2011, 289–355. ISBN: 978-3-642-12820-2. DOI: 10.1007/978-3-642-12821-9_4.

[She84] SHEERAN, Mary. μFP, A Language for VLSI Design. In: *LISP and Functional Programming*. 1984, 104–112. DOI: 10.1145/800055.802026.

[SL07] SHEARD, Tim and LINGER, Nathan. Programming in Ωmega. In: 2007. DOI: 10.1007/978-3-540-88059-2_5.

[Smi83] SMITH, Brian C. *Reflection and Semantics in LISP*. Technical Report. ACM, 1983. DOI: 10.1145/800017.800513.

[Sul+07] SULZMANN, Martin et al. System F with Type Equality Coercions. TLDI '07 (2007), 53–66. DOI: 10.1145/1190315.1190324.

[Sza78] Szabo, M. E. (Manfred Egon). *Algebra of proofs*. Vol. vol.88. Studies in logic and the foundations of mathematics. North-Holland, 1978, xii, 297p. isbn: 978-0-7204-2286-3.

[Tea09] Team, The GHC. The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.12.1 (2009).

[TN03] Taha, Walid and Nielsen, Michael Florentin. Environment Classifiers. In: vol. 38. 1. ACM, New York, NY, USA, Jan. 2003, 26–37. doi: `10.1145/640128.604134`.

[TS00] Taha and Sheard. MetaML and Multi-stage Programming with Explicit Annotations. *TCS: Theoretical Computer Science* 248 (2000). doi: `10.1016/S0304-3975(00)00053-0`.

[Wad92] Wadler, Philip. The Essence of Functional Programming. In: *POPL 92*. 1992, 1–14. doi: `10.1145/143165.143169`.

[Wan86] Wand, Mitchell. The Mystery of the Tower Revealed: a Non-Reflective Description of the Reflective Tower. In: *Symposium on LISP and Functional Programming*. Aug. 1986, 298–307. doi: `10.1007/BF01806174`.