Multi-Level Languages are Generalized Arrows *

Adam Megacz
UC Berkeley
megacz@berkeley.edu

Abstract

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 host language expressions into guest language expressions; because of this, Arrows may be used for metaprogramming only when the guest language is a superset of the host language. This restriction is also present in multi-level languages which offer unlimited cross-level persistence. The converse restriction, that the host language is a subset of the guest language, is imposed for multi-stage languages – those multi-level languages with a run construct.

This paper 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 4.5.4). Generalized arrows impose no containment relationship between the guest language and host language; they facilitate heterogeneous metaprogramming. The category having all generalized arrows as its morphisms and the category having all multi-level languages as its morphisms are isomorphic categories (Theorem 4.7.8). This is proven formally in Coq, and the proof is offered as justification for the assertion that multi-level languages are generalized arrows.

Combined with the existence of a particular kind of retraction (Definition 4.8.3) in the host language, 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 guest language providers write generalized arrow instances while the users of those guest 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.

Haskell is one example of a host language with the necessary kind of retraction. A modified version of GHC with multi-level terms is offered as a proof of concept; the Haskell extraction of the Coq proofs mentioned above has been compiled into this modified GHC, and is made available as a new flattening pass.

1. Introduction

The next section (Section 2) surveys related work. The following three sections cover the same topics three times, each time from a different perspective:

• Section 3.1 reviews Arrows, introduces generalized arrows, reviews multi-level languages, and previews the translation from the latter to the former. This is done by example, in a completely informal way, giving an idea of the programmer’s perspective and some sample code. Section 3 contains no theorems, proofs, or formal definitions and is not rigorous in any sense.

• Section 3.2 defines generalized arrows, multi-level languages, and their respective categories, and proves that these categories are isomorphic. All definitions and theorems in this section have been formalized in Coq.

• Section 3.3 describes a proof-of-concept implementation, instantiating the theorems of the previous section for an extension of System FC [SCJD07, post-publication Appendix C] and using the Haskell extraction of the resulting Coq proofs as part of a modified GHC. All of the examples from Section 3.1 are included as examples with the compiler.

Section 4 summarizes the paper’s results, highlights known deficiencies, and mentions a few directions for future work.

2. Related Work

One of the earliest examples of language support for metaprogramming is the `eval` primitive, present from the earliest days of LISP. However, the `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

\[\text{pow } n = \text{ if } n=0 \text{ then } \langle [ \mathbf{x} \mapsto 1 ] \rangle \text{ else } \langle [ \mathbf{x} \mapsto x \cdot \langle \text{pow } (n-1) \rangle x ] \rangle\]

\[\text{if } n=0 \text{ then } \langle \text{ga}_\text{drop} \rangle \cdot \langle \text{ga}_\text{constant} 1 \rangle \text{ else } \langle \text{ga}_\text{copy} \rangle \cdot \langle \text{id} \rangle \cdot \langle \text{ga}_\text{first} \rangle \cdot \langle \text{ga}_\text{second} \rangle \cdot \langle \text{pow } (n-1) \rangle \cdot \langle \text{ga}_\text{mult} \rangle\]

\[\text{pow' } n = \text{ if } n=0 \text{ then } \langle \text{ga}_\text{drop} \rangle \cdot \langle \text{ga}_\text{constant} 1 \rangle \text{ else } \langle \text{ga}_\text{copy} \rangle \cdot \langle \text{id} \rangle \cdot \langle \text{ga}_\text{first} \rangle \cdot \langle \text{ga}_\text{second} \rangle \cdot \langle \text{pow' } (n-1) \rangle \cdot \langle \text{ga}_\text{mult} \rangle\]

Figure 1. An example two-level program `pow`, which computes \(x^n\) (top), its flattened representation as a generalized arrow term (bottom), and a diagrammatic visualization of the terms produced by each of the two branches of the if clause (center). Identifiers prefixed with “\text{ga}_” come from type classes described in Section 4.3. The intuition behind the translation is explained in Figures 2 and 4 and Section 4.1 specifies the translation in detail.
explain reflection. Wand later [Wan86] provided a non-circular explanation of this phenomenon.

More recent research on metaprogramming, specifically multi-level programming in a typed setting, can be divided into intentional metaprogramming, which permits inspection of terms from a later level, and non-intentional metaprogramming, which does not try. Although the former offers additional capabilities for decomposing code which has already been assembled, it requires sophisticated mechanisms for maintaining hygiene and ensuring that Σ-equivalent terms are truly indistinguishable. Most work in this area has dealt with languages that have an intrinsic notion of freshness and name exchange based on nominal logics, as in FreshML [SPG03]. The present paper deals only with non-intentional metaprogramming. Although non-intentional metaprogramming lacks the ability to inspect code terms, it does have a counterbalancing advantage: the compiled implementation is free to perform “reduction under lambda” on later-stage terms, altering their syntactic structure. Because this syntactic structure cannot be examined by earlier stages, such reductions are safe. Research in this area has clustered mainly around two approaches: level annotations and modal types.

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 [CLG+01]. 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].

Modal types served as an early foundation for investigations into multi-level programming, beginning with the use of temporal logics by Davies and Pfenning [Dav96], who note that “Moggi’s computational λ-calculus only distinguishes values from computations and does not allow us to express stage separation.” Along with Wickline and Lee [WLP98], they achieved an implementation capable of runtime code generation. This work was later extended with Harper and Mitchell to yield the important role of enrichment in the definition and use of categories. Their work was particularly notable in that it directly resulted in a unified approach: level annotations and modal types.

Most work on giving category-theoretic semantics to multi-level languages has been based on indexed categories. Jacobs [Jac91] appears to have been the first to advocate their use to represent contexts, creating a separate category for each possible context to the left of the turnstile. Indexed categories were also used by Moggi in the form of presheaves to investigate two-level languages [Mog97]. This work was later extended with Harper and Mitchell to explain the phase distinction in the ML module system [HMM96].

Kameyama, Kiselyov, and Shan [KKc08] provide one translation from homogeneous multi-stage terms into single-stage terms. The resulting encoding uses cartesian tuples to represent contexts, which is workable so long as no substructural types are present in the language. Their work was particularly notable in that it directly addressed the influence of weak side effects such as nontermination and extrapolated to stronger side effects which might cause scope extrusion (for example, reference cells). More recently Choi, Aktemur, Yi, and Tatsuta have defined an “unstaging” translation [CAYT11] which can statically analyze homogeneous metaprograms.

The development of Arrows began with the premonoidal categories of Power and Robinson [PR97], which were introduced in order to represent contexts as objects in a category; the category Types(L) of this paper is based on the same idea. Power and Thielecke later examined the case where the center of such a category admits a strict monoidal identity-on-objects functor from a cartesian category [PT97], later termed a Freyd Category. It was later proved that premonoidal categories which admit such a functor are in bijective correspondence with strong κ-categories [LPT03]. This work was later extended with Harper and Mitchell to yield the important role of enrichment in the definition and use of categories.

3. Introduction By Example

3.1 Arrows in Haskell

From a programmer’s perspective, an Arrow is a type belonging to the type class below; the id and >>> declarations come from the Category superclass:

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

Instances of this class must obey the laws of [Pat01] Figure 1.

3.2 BiArrows in Haskell

BiArrows are meant to be Arrows with a notion of inversion. They were introduced in [ASoW-05] and further examined in [HJ09]. A BiArrow is an instance of the following class:

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

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 values 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. Moreover, nothing stops a program from

\(^{2}\) 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 the wrong result rather than fail via some
Some code examples are shown in Figure 2. The instance declaration showing that every Arrow is a generalized arrow. This may be combined with the instance declaration for Arrow (->) in Control.Arrow to provide an interpretation in Haskell for multi-level terms in languages which are a subset of Haskell.

class Category g => GArrow g (**) u where
  -- id = arr id
  -- (>>) = arr .
  ga_first :: g x y -> g (x ** y) (y ** z)
  ga_second :: g x y -> g (x ** z) (y ** y)
  ga_cancell :: g (u**x) x
  ga_swap :: g (x**y) (y**x)
  ga_copy :: g x (x**u)
  ga_drop :: g (x**u) x
  ga_assoc :: g ((x**y)**z) ((x*(y**z))
  ga_unassoc :: g (x**y z) ((x**y)**z)

instance Arrow a => GArrow a (,) () where
  instance Arrow a => GArrowCopy a (,) () where
  instance Arrow a => GArrowDrop a (,) () where
  instance Arrow a => GArrowSwap a (,) () where
  instance Arrow a => GArrowConstant a (,) () t t

Figure 2. The definition for the (GArrow) type class and its three most frequently implemented subclasses. The class Category comes from the standard Control.Category module.

Figure 3. Instance declaration showing that every Arrow is a generalized arrow. This may be combined with the instance declaration for Arrow (->) in Control.Arrow to provide an interpretation in Haskell for multi-level terms in languages which are a subset of Haskell.

3.3 Generalized Arrows in Haskell

Section 3.4 will define generalized arrows formally, in arbitrary languages. However, before presenting the abstract constructions, we will first survey one specific case: generalized arrows in Haskell, as embodied in the GArrow class shown in Figure 2. 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 any category. The GArrow class also has a first function, whose type and laws are identical to those of the Arrow class, although the accompanying “second” function (ga_second) does not have a default definition in terms of first.

Unlike the Arrow class, the GArrow class has a second type parameter ** of kind * -> * -> *, 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. Furthermore, allowing some type g to be an instance of GArrow via multiple different tensors will clarify the treatment of sums and products in Section 3.6.1.

The third type parameter u, of kind *, plays the role of () in Arrow; it is called the unit of the generalized arrow. 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, we find six functions: g a (un) cancel (1, r) and g a (un) assoc. These are used to re-arrange the inputs and outputs of a generalized arrow: the g a (un) cancel (1, r) functions add and remove superfluous inputs and outputs (which must be of the unit type) whereas the g a (un) assoc functions “re-parenthesize” the list of inputs. Readers familiar with category theory will notice that this definition closely parallels that of premonoidal categories [PR97].

The GArrow class has three subclasses which are used quite frequently. GArrowCopy allows one to duplicate inputs, GArrowSwap allows one to duplicate inputs, GArrowConstant allows one to duplicate inputs.
allows one to change the order of inputs, and \texttt{GArrowDrop} allows one to delete inputs of arbitrary type by using \texttt{ga\_drop} to turn them into inputs of the unit type which may then be discarded using \texttt{ga\_cancel[1,r]}. Note that the \texttt{Arrow} class does not permit this distinction: one can produce functions like \texttt{ga\_drop}, \texttt{ga\_copy}, and \texttt{ga\_swap} for any \texttt{Arrow} instance by using the mandatory \texttt{arr} function (the definitions appear in Figure \ref{fig:sample}). Readers familiar with type theory and proof theory will notice that these definitions closely parallel the structural operations of ordered linear logic \cite{Res09}.

The three subclasses above frequently occur together; to reduce verbosity, the class \texttt{GArrowSTKC} collects them:

\begin{verbatim}
class (GArrowDrop g (**)) u,
   GArrowCopy g (**)) u,
   GArrowSwap g (**)) u =>
   GArrowSTKC g (**)) u
\end{verbatim}

Instances of \texttt{GArrowSTKC} provide implementations of guest languages which are supersets of the Simply Typed Kappa Calculus \cite{Has95}.

Conceptually, the nine functions \texttt{ga\_cancel[1,r]}, \texttt{ga\_assoc}, \texttt{ga\_drop}, \texttt{ga\_copy}, and \texttt{ga\_swap} exist to "put back" what generalized arrows have lost -- relative to the \texttt{Arrow} class -- as a result of eschewing a mandatory \texttt{arr}. This can be made slightly more concrete by taking a look at the instance declaration in Figure \ref{fig:sample} which gives every \texttt{Arrow} instance a canonical \texttt{GArrow} instance. This instance, in conjunction with the standard instance \texttt{Arrow} \texttt{(&->)}, may be used as an evaluation mechanism for guest languages which happen to be a subset of Haskell. The insertion of (sometimes quite a large number of) these re-arrangement function invocations is typically done by the compiler (see Remark \ref{remark:implementation}). Programmers are responsible only for supplying the implementation for these re-arrangement functions.

### 3.4 Example: BiGArrows

We can now define the class \texttt{BiGArrow}, which avoids the problems of the \texttt{BiArrow} class by using generalized arrows in place of \texttt{Arrows}:

\begin{verbatim}
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
\end{verbatim}

The tutorial which accompanies the modified compiler includes an example \texttt{BiGArrow} instance which also implements class \texttt{GArrowDrop} using a \texttt{logging translation} \cite{MHT04}.

### 3.5 Multi-Level Terms and Types

Haskell Arrows come with a very convenient syntax \cite{Pat01} Figure 7., modeled on the monadic do-notation. Further work has been done on a calculus which types terms of the Arrow's internal language directly \cite{WY10}. Is there an equivalent for generalized arrows?

There is. However, rather than being a custom syntax designed for a particular type class or category, it turns out than an existing syntax - which is in fact a typed calculus - is not only usable, but ideal. This calculus is the term syntax and type system of multi-level languages \cite{NN92} augmented with environment classifiers \cite{TN03}. This connection will be spelled out in formal detail in Section \ref{sec:connection} where we give only the most cursory preview of multi-level terms, their types, and the connection.

Multi-level terms involve two additional expression forms: brackets and escape. Brackets, written \texttt{<}[,\ldots](, serve to demarcate an expression of the guest language. For example, the definition below

\begin{verbatim}
newtype Code x y =
   Code { unC :: forall a. <[ x -> y ]> @a }
\end{verbatim}

is the host language representation of a guest language's identity function:

\begin{verbatim}
guest_id = <[ \x -> x ]>
\end{verbatim}

The escape form, written \texttt{~e.} serves to insert one guest language expression into another. For example, the definition below is the host language function which composes two guest language functions:

\begin{verbatim}
guest_comp f g = <[ \x -> \texttt{~f (~g x) } ]>
\end{verbatim}

Multi-level languages also introduce an additional type form: code types, written \texttt{<}[t@c)] where \texttt{t} is a type variable and \texttt{c} is an \texttt{environment classifier} \cite{TN03}. In homogeneous multi-level programming, environment classifiers are used to ensure that certain terms of code type contain no free variables, and are therefore safe to execute. In heterogeneous metaprogramming an execution mechanism may or may not be available; even when it is unavailable these classifiers still serve an important purpose: a single host program might manipulate terms in multiple different guest languages; unification of environment classifier variables ensures that if one guest language term is pasted into another term (via the escape operator), the same guest language implementation (i.e., generalized arrow) is used to realize them.

### 3.6 Multi-Level Terms to Generalized Arrows and Back

As a preview of the connection between generalized arrows and multi-level languages, consider Figure \ref{fig:sample} which shows that level-1 terms may be used to implement the \texttt{GArrow} class (at level-0). This class instance is not typically used in practice -- it is not involved in
the flattening process, nor do users typically rely on it. However, it is valid code (the modified compiler accepts it) and will help illustrate the flattening process. It provides one possible “translation back” from generalized arrows to multi-level terms.

The formal definition of the flattening process is explained in complete detail in Section 4, what follows is simply a sketch. Consider the pow program from Figure 1:

\[
\text{pow n} = \begin{cases} 
0 & \text{if } n=0 \\
1 & \text{if } n=1 \\
\text{else} & \begin{cases} 
\text{if } x \rightarrow x \rightarrow (x,x) \rightarrow (x,x) & \text{else} \\
\text{firstc} & \text{secondc} & \text{mult} & \\
\text{firstc} & \text{secondc} & \text{mult} & \\
\text{if } n=0 & \text{else} & \text{if } n=0 & \text{else} \\
\text{then} & \text{then} & \text{then} & \text{else} \\
\text{ga_drop} & \text{ga_constant} & \text{ga_second} & \text{ga_second} \\
\end{cases} 
& \end{cases} 
\]

(\text{pow \(n-1\)) x

One might imagine rewriting the program above in point-free form as shown in the middle column of Figure 5. The rewritten program has a very important property: all expressions inside code brackets happen to occur in the \text{GArrow Code} instance of Figure 4. Because of this, we can replace these expressions with functions defined in the \text{GArrow} class; when instantiated with \text{GArrow Code}, the resulting program will be identical (modulo the \text{newtype} required as a hint to the compiler’s type inference engine) from the \text{GArrow} Code instance of Figure 4, maintaining left-to-right order of expressions. Because no actual code terms appear in the third program, it is polymorphic in the \text{GArrow} instance and can be used with instances other than \text{instance Code}.

The benefits of separating Haskell’s (,) from the generalized arrow’s tensor can be seen in the \text{GArrowSum} class, which is roughly analogous to \text{ArrowChoice}[Hug00, 5.1]:

\[
\begin{align*}
\text{class } & (\text{GArrow g (**) u,} \\
& \text{GArrow g \langle\rangle v) } \\
& \text{GArrowSum g (**) u v \langle\rangle where} \\
& \text{ga_sum : : g (x\langle\rangle x) x} \\
& \text{ga_never : : g v v x} \\
\end{align*}
\]

Note that g is required to be a generalized arrow via two separate tensors: the ** tensor is used for representing contexts, and the \langle\rangle tensor used to represent the \text{sum} of two types. The \text{ga_first} and \text{ga_second} functions on the \langle\rangle tensor give us the left and right of the \text{ArrowChoice} class, and the other functions of that class may be defined in terms of them. The \text{ArrowChoice} class fixes \text{Either} as the type operator used to represent sums, whereas GArrowSum leaves it abstract. For this reason we need \text{ga_merge}, which is akin to the obvious function of type \text{Either a a} \rightarrow \text{a and ga_never}, which produces a disjunct that “never” occurs. The left and right injections of the sum are given by:

\[
\begin{align*}
\text{ga_inl} & = \text{ga_uncancell} \gg \text{ga_first} \gg \text{ga_never} \\
\text{ga_inr} & = \text{ga_uncancell} \gg \text{ga_second} \gg \text{ga_never} \\
\end{align*}
\]

Note how this example does not assume that the units of the two generalized arrow instances are the same. If this were the case, it would mean that the generalized arrow had a function similar to the Haskell function of type \text{forall } \alpha. \alpha \rightarrow \alpha. In Haskell the only function of this type is \text{⊥}[Wad89]; total functional languages [Tur04] such as Coq have no functions of this type at all. Clearly the ability to separate the units of the two instances is important here.

The class for products is defined dually:

\[
\begin{align*}
\text{class } & (\text{GArrow g (**) u,} \\
& \text{GArrow g \langle\rangle v) } \\
& \text{GArrowProd g (**) u \langle\rangle v where} \\
& \text{ga_prod_copy : : g x (x\langle\rangle x) x} \\
& \text{ga_prod_drop : : g x v} \\
\end{align*}
\]

There is an important symmetry here: the type of \text{ga_merge} is dual to \text{ga_copy} and the type of \text{ga_never} is dual to that of \text{ga_drop}. There are category-theoretic reasons for this: the unit of a coproduct is the initial object, while the unit of a product is the terminal object. Products have diagonal morphisms while coproducts have codiagonal morphisms.
In most cases, instances of \texttt{GArrowProd} will use the same type for both units \(\mathbf{u}\) and \(\mathbf{v}\); this is because cartesian products let you “throw away” either coordinate, and languages without substructural types similarly let you “throw away” any element of the context (weakening). One can add \texttt{GArrow} subclasses for type operators simulating the other connectives of linear logic \cite{PFL92} – additive conjunction, additive disjunction, multiplicative conjunction, multiplicative disjunction – in a similar fashion; in these cases the units must be different. Another important example occurs in asynchronous datalow, where a pair of streams is quite different from a stream of pairs – the latter has more stringent synchronization behavior. The inability to make this distinction forced the Fudgets library \cite{CH93} to co-opt the \textit{coproduct} structure of the underlying type system to represent pairs of streams, causing an anomaly that Paterson notes \cite{Pa01} Section 5.1 in the type of the Fudgets \texttt{loop} function.

4. Foundations

Having introduced generalized arrows in a cursory informal manner, these ideas will now be made rigorous. This section will make frequent use of finite binary trees whose leaves are either a distinguished “empty” value or else of some particular sort \(S\):

\[
\text{Tree}(S) = \langle \emptyset \rangle \mid \langle S \rangle \mid (\text{Tree}(S), \text{Tree}(S))
\]

This section will state results in terms of programming languages in general, though a few very basic assumptions are necessary. When speaking of a programming language \(L\) we will assume that it has types, terms, and variables. We will assume that a context is a finite binary tree of types. We will assume some equivalence relation \(\equiv\) on terms of the same type in the same context; typically this is a denotational semantics or contextual equivalence under some operational semantics, but those are not the only possibilities. We will assume some sort of syntactical substitution mechanism (let-binding, lambda-abstraction and application, explicit substitution, or some other mechanism) on terms which is associative up to \(\equiv\) and for which the identity terms are left and right neutral. We will assume that the language comes with some collection of inference rules such that for every term \(\mathbf{e}\) which has type \(\tau\) in context \(\Gamma\) there is a natural deduction proof with conclusion \(\Gamma \vdash \langle \tau \rangle\) and no open hypotheses and, and furthermore that \(\mathbf{e}\) can be recovered from this proof.

4.1 The Category \(\text{Types}(L)\)

For a particular programming language \(L\) (e.g., Haskell), there is a well-established way to produce a category. Following the unfortunate convention that categories are named after their objects (Cat, Grp, Sets) rather than their morphisms, we have:

\textbf{Formalized Definition 4.1.1 (\textsf{TypesL})} Let \(L\) be a programming language. \(\text{Types}(L)\) is the following category. \textbf{Objects:} finite binary trees with a type of \(L\) at each nonempty leaf. \textbf{Morphisms} from \(A\) to \(B\): a binary tree \(T\) of the same shape as \(B\), where each leaf of \(T\) is a proof having all the leaves of \(A\) as its hypotheses and the corresponding leaf of \(B\) as its conclusion. \textbf{Equality} of morphisms: a pair of morphisms \(f, g : A \rightarrow B\) are equal if the expressions recovered from the conclusions of their constituent proofs are \((\equiv)\)-equivalent.

\textbf{Formalized Theorem 4.1.2 (\textsf{TypesL_PreMonoidal})} \(\text{Types}(L)\) is a premonoidal category, where the functor \(- \times B\) sends an object \(A\) to \((A, B)\) and a morphism \(f : A \rightarrow C\) to \((f, \text{id}_B)\) and the functor \(A \times -\) sends an object \(B\) to \((A, B)\) and a morphism \(f : B \rightarrow C\) to \(\langle \text{id}_A, f \rangle\).

This category is not \textit{strict} premonoidal. Its strictification \cite[Theorem XI.3.1]{Lan71} – a category in which the objects are \textit{lists} of types rather than binary trees – is more common in the literature, but removes some structure we will need later on. Perhaps the earliest example comes from Lawvere’s thesis \cite{Law63}, later revisited as \cite{Law96}, explaining quantifiers as adjunctions, which led to the general concept being called a Lawvere Theory. More verbosely, \(\text{Types}(L)\) could be called “the non-strict multi-sorted Lawvere Theory of the term algebra of \(L\)”. In the context of logic, Awodey refers to this as the formula category \cite{Aw06} 9.5). Crole refers to it as the classifying category of a theory \cite[4.8.4]{Cro04}. Other names include “clone category” \cite[Definition 8.9.4]{Ber98} and “category of derived operations of an algebra.” The full subcategory of \(\text{Types}(L)\) consisting of only those objects which have exactly one leaf is often called the \textit{Lindenbaum category} of an algebra (in this case, the term algebra), and in the case of the language Haskell this is the category most frequently referred to as “\texttt{Hask}.” \(\text{Types}(L)\) occurs in \textit{non}-strictified form as a special case of far more general work by Blute, Cockett, and Seely \cite{BCS97}.

4.2 The Category \(\text{Judgments}(L)\)

Having defined the category of types of a language, we now move to a second category: its category of \textit{judgments}. Let a context be a binary tree of types (i.e. an object of \(\text{Types}(L)\)) and let a judgment be a pair of contexts, written \(\Gamma_1 \vdash_1 \Gamma_2\).

\textbf{Formalized Definition 4.2.1 (\textsf{JudgmentsL})} Let \(\text{Judgments}(L)\) be the following category. \textbf{Objects:} finite binary trees with a \textit{judgment} at each non-empty leaf. \textbf{Morphisms} from \(A\) to \(B\): a binary tree \(T\) of the same shape as \(B\), where each leaf of \(T\) is a proof having all the leaves of \(A\) as its hypotheses and the corresponding leaf of \(B\) as its conclusion. \textbf{Equality} of morphisms: a pair of morphisms \(f, g : A \rightarrow B\) are equal if the expressions recovered from the conclusions of their constituent proofs are \((\equiv)\)-equivalent.

Observe that we allow a context (tree of types) rather than merely a single type as the succeedant of a judgment. In contrast to natural deduction for classical logic, these trees are conjunctive rather than disjunctive: a proof of \(\Gamma \vdash \langle \tau_1, \tau_2 \rangle\) establishes the well-typedness of both \(\tau_1\) and \(\tau_2\).

Like \(\text{Types}(L)\), this category can be found elsewhere in the literature, under many names: its use in the study of programming languages and type systems originated with Cartmell \cite[Section 13]{Car85}, who calls this the “category of contexts and relations” \(R(U)\) for a term category \(U\) and explains how to construct it for cases where the base category has all finite limits (though Types (Haskell) does not) and certain projection morphisms exist – roughly equivalent to assuming that the unit object is terminal. Lambeek \cite{Lam09} p79 calls this the \textit{deductive system on a category} (in this case, the category \(\text{Types}(L)\)). Szabo \cite[1.1.26.2.5.3]{Sza78} calls it a \textit{sequential category generated by an unlabeled deductive system} (i.e., category of sequents). Szabo’s work also demonstrates that \(\text{Types}(L)\) (which he calls \(FM(X)\)) is isomorphic to a subcategory of this category via the covariant hom-functor embedding. This last result is the closest thing to Theorem \ref{thm:non-mono} we have been able to find in the literature.

\textbf{Formalized Definition 4.2.2 (\textsf{Judgments-cartesian})} \(\text{Judgments}(L)\) is a cartesian category.

4.3 Relating \(\text{Judgments}(L)\) and \(\text{Types}(L)\)

Of great importance to the rest of the development is the relationship between \(\text{Types}(L)\) and \(\text{Judgments}(L)\):

\textbf{Formalized Theorem 4.3.1 (\textsf{TypesEnrichedInJudgments})} \(\text{Types}(L)\) is enriched \cite[1.3]{Kel82} in \(\text{Judgments}(L)\).

\textbf{Formalized Definition 4.3.2 (\textsf{Types_first, Types_second})} The functors \(- \times A\) and \(A \times -\) which give \(\text{Types}(L)\) its
premonoidal structure are \textit{Judgments(\mathcal{L})}-enriched functors \cite{kelley:1982} 1.10.

The following simple definition will be helpful:

\textbf{Formalized Definition 4.3.3 (Enrichment)} An enrichment, written \( C \in \mathcal{V} \), is a monoidal category \( \langle \mathcal{V}, \otimes, I_\mathcal{V} \rangle \) and a premonoidal category \( \langle C, \otimes, \times, I \rangle \) such that \( C \) is the underlying category of some \( \mathcal{V} \)-enriched category, and all of the functors \( A \otimes - \) and \( - \times A \) are \( \mathcal{V} \)-enriched functors.

Enriched categories are defined in only those categories which are monoidal, so the monoidal structure of \( \mathcal{V} \) is a necessary part of this definition. Asking that \( C \) be premonoidal and including the “extra structure” which makes it so as part of the enrichment will simplify later definitions. Lastly, recall the definition of the \textit{center} of a premonoidal category and the fact that it is a monoidal category:

\textbf{Formalized Definition 4.3.4 (Center)} For \( C \) a premonoidal category, the \textit{center} of \( C \), written \( Z(\mathcal{C}) \), is the collection of all central morphisms of \( C \).

\textbf{Formalized Lemma 4.3.5 (CenterMonoidal)} For \( C \) a premonoidal category, \( Z(\mathcal{C}) \) is a monoidal subcategory of \( C \).

\subsection{4.4 An Arrow in \( \mathcal{L} \)}

Within this framework of languages-as-categories, what is an instance of the \textit{Arrow} class?

\textbf{Formalized Definition 4.4.1 (FreydCategory)} A \textit{Freyd Category} \cite{freyd:1972} \cite{pf:1999} is an identity-on-objects strict symmetric monoidal functor \( \mathcal{F} : C \rightarrow Z(\mathcal{C}) \) where \( C \) is a cartesian category and \( \mathcal{C} \) (called the \textit{Kleisli category}) is symmetric premonoidal.

\textbf{Formalized Definition 4.4.2 (ArrowInProgrammingLanguage)} An \textit{Arrow} in a programming language \( \mathcal{L} \) with \( \text{Types}(\mathcal{L}) \) cartesian is an enrichment \( \mathcal{K} \in \mathcal{V}_\mathcal{K} \) with \( \mathcal{V}_\mathcal{K} \subseteq \text{Types}(\mathcal{L}) \) and a Freyd Category \( \mathcal{F} : \text{Types}(\mathcal{L}) \rightarrow Z(\mathcal{C}) \) \cite{ake:2008} Theorem 3.7.

For example, in Haskell any particular instance \textit{Arrow} \( T \) determines a full subcategory of \( \text{Types}(\text{Haskell}) \) consisting of types of the form \( T \times Y \) for any \( X \) and \( Y \). The Kleisli category \( \mathcal{K} \) of the \textit{Arrow} is enriched in this subcategory of \( \text{Types}(\text{Haskell}) \).

\subsection{4.5 Generalized Arrows}

We are now ready to define a generalized arrow in a programming language:

\textbf{Formalized Definition 4.5.1 (GeneralizedArrowInLanguage)} For programming languages \( G \) and \( H \), a generalized arrow from language \( G \) to language \( H \) is a monoidal functor from \textit{Judgments}(\( G \)) to \( Z(\text{Types}(H)) \).

\textbf{Remark 4.5.2} Note that the domain of the functor is a judgment category whereas its codomain is \textit{the center of} a term category; in this sense, generalized arrows represent the \textit{proof structure of one language within the term structure of another language}.

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

\textbf{Formalized Definition 4.5.3 (GeneralizedArrow)} For enrichments \( \mathcal{K} \in \mathcal{V}_\mathcal{K} \) and \( \mathcal{C} \in \mathcal{V} \), a generalized arrow from \( \mathcal{K} \in \mathcal{V}_\mathcal{K} \) to \( \mathcal{C} \in \mathcal{V} \) is a monoidal functor from \( \mathcal{V}_\mathcal{K} \) to \( Z(\mathcal{C}) \).

\textbf{Formalized Theorem 4.5.4 (ArrowsAreGeneralizedArrows)} If \( \mathcal{J} : \text{Types}(\mathcal{L}) \rightarrow Z(\mathcal{K}) \) is an \textit{Arrow} in programming language \( \mathcal{L} \), the inclusion functor from the enriching category of \( \mathcal{K} \) to \text{Types}(\( \mathcal{L} \)) is a generalized arrow from \( Z(\mathcal{K}) \in \mathcal{V}_\mathcal{K} \) to \text{Types}(\( \mathcal{L} \)) \subseteq \mathcal{V}_\mathcal{L} \).

\textbf{Remark 4.5.5} Often, given an \textit{Arrow} \( \mathcal{J} \) in a programming language as above, one can cook up a language \( \mathcal{L}^A \) such that \( Z(\mathcal{K}) \cong \text{Types}(\mathcal{L}^A) \) and \( \mathcal{V}_\mathcal{K} \cong \text{Judgments}(\mathcal{L}^A) \), making the result of Theorem 4.5.4 not just a generalized arrow between enrichments (Definition 4.5.3) but also a generalized arrow between languages \( \mathcal{L}^A \) and \( \mathcal{L} \) (Definition 4.5.1).

\textbf{Remark 4.5.6} A great deal of work has been devoted to isolating the right definition of an \textit{Arrow} in an arbitrary category \cite{hase:2009} \cite{jones:2006} \cite{ake:2008} \cite{asa:2010} \cite{ah:2011}, such that \textit{Arrows} in programming languages occur as a special case. These \textit{Arrows} in arbitrary categories are outside the scope of the present paper; focus here is limited to those \textit{Arrows} which occur in a programming languages. For example, \textit{Arrows} on the category of sets or on the category of vector spaces are not considered.

\subsection{4.6 Multi-Level Languages}

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

\textbf{Formalized Definition 4.6.1 (Reification)} Given two enrichments \( \mathcal{K} \in \mathcal{V}_\mathcal{K} \) and \( \mathcal{C} \in \mathcal{V} \), a \textit{reification} from \( \mathcal{K} \in \mathcal{V}_\mathcal{K} \) to \( \mathcal{C} \in \mathcal{V} \) is a monoidal functor \( \mathcal{R} : \mathcal{V}_\mathcal{K} \rightarrow \mathcal{V} \) such that for every \( K \in \text{Ob}(\mathcal{K}) \) there exists a functor \( \exists_K : \mathcal{K} \rightarrow Z(\mathcal{C}) \) making the following diagram commute up to natural isomorphism:

\textbf{Remark 4.6.2} With this definition in place, the definition of a two-level language is straightforward.
The motivation for this definition can be found in literature all the host language does not function types – may be used to define multi-level languages in which the 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 host language does not have such function types (for example, languages based on λ-calculus).

Formalized Definition 4.6.2 (TwoLevelLanguage) A two-level language \((G, H)\) consists of a pair of languages \(G\) and \(H\) and a reification from \(\text{Types}(G) \in \text{Judgments}(G)\) to \(\text{Types}(H) \in \text{Judgments}(H)\).

\[
\text{Judgments}(G) \xrightarrow{\text{R}} \text{Judgments}(H)
\]

Remark 4.6.3 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 4.5.2.

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-formed rule or axiom \(R\) for types in \(L\), and each binding time \(b \in B\), we add the rule or axiom \(R_b^{b}\); their specification for \(R_b^{b}\) – translated into the terminology of category theory – forms the functor \(R\) above when the host language has function types \(\Rightarrow\) for which \(\text{Hom}(X, Y) \cong \text{Hom}(I, X \Rightarrow Y)\).

Remark 4.6.4 Note that the functors \(R\) and \(!_K\) are not necessarily full or faithful. This will be revisited in Section 4.5.

By iterating and taking limits, we arrive at \(n\)-level languages and \(\omega\)-level languages:

Formalized Definition 4.6.5 (NLevelLanguage) Every language is a one-level language. An \((n + 1)\)-level language is a two-level language \((L_n, L)\) where \(L_n\) is an \(n\)-level language.

Formalized Definition 4.6.6 (OmegaLevelLanguage) An \(\omega\)-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 \(n \in \mathbb{N}\).

4.7 Multi-Level Languages are Generalized Arrows

Although \(\text{Types}(L) \in \text{Judgments}(L)\) gives an enrichment for any programming language, not all enrichments arise this way. The next three properties hold for all enrichments which do arise this way (Lemma 4.7.5) from languages with cartesian product types. They capture the formal facts needed to drive the remainder of the proof.

Formalized Definition 4.7.1 (MonoidalEnrichment) An enrichment \(C \in \mathcal{V}\) is monic if there is an object \(X\) such that \(\text{Hom}(X, -) : C \to \mathcal{V}\) is faithful.

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

Formalized Definition 4.7.2 (SurjectiveEnrichment) An enrichment \(C \in \mathcal{V}\) is surjective if every object of \(\mathcal{V}\) is the tensor of finitely many objects each of which is either the unit object \(I\), of \(\mathcal{V}\) or a hom-object of \(C\).

Intuitively, surjective enrichments are those in which the enriching category has no extraneous objects unnecessary to the enrichment (note that this says nothing about morphisms). Intuitively, morphology is not a serious limitation; one can simply restrict the focus to the least full subcategory in which \(C\) is enriched.

Formalized Definition 4.7.3 (MonoidalEnrichment) An enrichment \(C \in \mathcal{V}\) is monoidal if \(\text{Hom}(I, -) : C \to \mathcal{V}\) 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 is:

Formalized Theorem 4.7.4 (CartesianMonoidalEnrichment) Any enrichment \(C \in \mathcal{V}\) with both \(C\) and \(\mathcal{V}\) cartesian is a monoidal enrichment.

Formalized Lemma 4.7.5 (LanguagesWithProductsAreSMME) If a programming language \(L\) has cartesian contexts (i.e., weakening, exchange, and contraction) (i.e. pairs), then \(\text{Types}(L) \in \text{Judgments}(L)\) is a surjective monoidal monoidal enrichment.

In a vague sense, surjective monoidal monoidal enrichments “look enough like programming languages” for the proof to go through.

Formalized Definition 4.7.6 (CategoryOfReifications) The category of reifications is the category with Objects: surjective monoidal monoidal enrichments, Morphisms from \(E_1\) to \(E_2\): reifications from \(E_1\) to \(E_2\), Equality of morphisms: reifications whose \(R\) functors are naturally isomorphic.

Formalized Definition 4.7.7 (CategoryOfGeneralizedArrows) The category of generalized arrows is the category with Objects: surjective monoidal monoidal enrichments, Morphisms from \(E_1\) to \(E_2\): generalized arrows from \(E_1\) to \(E_2\), Equality of morphisms: generalized arrows which are naturally isomorphic as functors. Composition of two generalized arrows \(A : E_2 \to E_1\) and \(A' : E_1 \to E_2\) is given by \(A \circ \text{Hom}_{E_2}(I, -) \circ A'\). This leads to the main technical result of the paper:

Formalized Theorem 4.7.8 (ReificationsAreGeneralizedArrows) The category of reifications is isomorphic to the category of generalized arrows.

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 \(M_1(M_2(R))\) is naturally isomorphic to \(R\) and \(M_2(M_1(A))\) is naturally isomorphic to \(A\). 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 \(\text{Hom}(I, -)\) 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 \(R\) 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 \(\text{Hom}(I, -)\) (via \(!_K\)) for every \(\text{Hom}(K, I, -)\), 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 \(R\) as a subcategory and the base category of its codomain; this isomorphism may be post-composed with the reification to produce a generalized arrow. The Coq proof is normative; this is only a sketch.
5. Proof of Concept

To demonstrate the feasibility of these ideas, a modified version of the Glasgow Haskell Compiler (GHC) [Lear9] has been created. This modified GHC offers a new flag, -Moda1Types, which adds new expression syntax for code brackets and escape, new type syntax for code types (with environment classifiers), inference of environment classifiers using GHC’s existing arbitrary-rank-polymorphism type inference engine [Dim05], and an expansion of the type checker to include the additional typing rules described in this section.

This paper’s accompanying Coq script includes a complete formalization of System FCα (Section 5.1), a reification from a simple PCF-like language to System FCα, and a flattening functor for this reification. The Coq code is extracted to Haskell code, which is then compiled into the compiler, providing a new flag -fcooppas which invokes the flattening transformation. A library GHC RetMet.GArrow is also provided; it includes known-to-the-compiler type classes for generalized arrows.

5.1 System FCα

GHC uses System FC [SCJ10] post-publication Appendix C] as its intermediate language (as of this writing GHC has not yet been updated to use System FC2 [WVJZ11]). This language is rich enough to capture the type structure of Haskell as well as the large variety of extensions supported by GHC. System FCα is little more than the union of the grammars and typing rules of System FC and λα [TN03]. The remainder of this section explains the modifications made to harmonize the two systems and ensure that its proofs form a category Judgments(System FCα). System FCα 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 naive union of System FC and λα 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 [SCJ10] 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.
4. The branches of a case statement, letrec bindings, 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 Σ1 ⊢ A asserts that the context Σ1 may be produced from Σ1 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.

Remark 5.1.1 The 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
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, literals may appear only at level zero (the compiler comes with special functions to transport literals from level zero to positive levels) and the following expressions may occur only at level zero: the cast expression, coercion lambda, coercion application, and case expressions which bind coercion variables.

Formalized Definition 5.1.2 (SystemFCa) System FC^∗ is the
programming language of Figure 6.

Formalized Definition 5.1.3 (PCF) PCF is the sublanguage of System FC^∗ consisting of proofs which do not use the following rules: Case, Cast, AbsT, AppT, AbsC, AppC. This is roughly equivalent to the Programming Language for Computable Functions [Plo77] without integers or booleans.

Formalized Theorem 5.1.4 (PCF_SystemFCa_two_level) (PCF, System FC^∗) is a two-level language for which the the retraction of Definition 4.3.3 exists.
Although the current proof scripts establish that there is a reification from PCF to System FC*, the actual functor used in the -fcoqpass extraction is constructed directly, rather than by invoking Theorem 4.7.8 and post-composing the resulting functor with a retraction.

### 6.2 Future Work

A number of interesting classes of Arrows arise by requiring that the hom-sets of the Kleisli category of the Arrow each carry some sort of additional structure. For example, the ArrowPlus class assumes that for each pair of types \( x \) and \( y \), if \( a \) is an instance of ArrowPlus, then the values of \( a \times x \) form a monoid, with the zeroArrow and \( \langle \rangle \) methods of ArrowPlus as the identity element and multiplication operator. Arrows with semiring, ring, lattice, boolean algebra, relation algebra, and Kleene algebra structures on their hom-sets are also possible; the last is particularly useful for concurrency. Even the structure of a category on the hom-set is possible, making the Arrow’s Kleisli category \( \mathbb{K} \) a 2-category. All of these possibilities carry over directly to generalized arrows; the question of what the corresponding multi-level language features are has yet to be answered.

The fact that the proofs above work even for reifications whose functors are not full or not faithful (Remark 4.6.3) suggests other interesting directions. A non-faithful reification functor would suggest a guest language with a finer equivalence relation on terms than its host language; for example, a guest language with a syntactical notion of equality and a host language in which equality is considered modulo \( \alpha \)-equivalence. A non-full reification functor suggests a host language which has functions between the images of guest types which, nonetheless, are not themselves the images of guest functions.

In the present implementation it is not possible to determine, solely from the type of a multi-level term, which of \( \text{GArrowDrop} \), \( \text{GArrowSwap} \), and \( \text{GArrowCopy} \) must be implemented in the generalized arrow supplied to the flattened version of the term. In order to do so, one must track the dereferencing, reordering, and duplication of variables in the types of the source language; GHC currently does not have support for this.

### REFERENCES

[AH10] Kazuyuki Asada and Ichiro Hasuo. Categorifying computations into components via arrows as profunctors. Electr. Notes Theor. Comput. Sci, 264(2):25–45, 2010.

[Asa10] K. Asada. Arrows are strong monads. In Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming, pages 33–42. ACM, 2010.

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

[Ber98] George M. Bergman. An invitation to general algebra and universal constructions. Henry Nelson, Berkeley, CA, 1998.

[Ble95] G Blelloch. Nesl: A nested data-parallel language (version 3.1). www-2.cs.cmu.edu, Jan 1995.
