Bimonadic Semantics for Basic Pattern Matching Calculi

Wolfram Kahl[*] — kahl@cas.mcmaster.ca
Jacques Carette — carette@cas.mcmaster.ca
Xiaoheng Ji[*] — jixiaoheng@gmail.com

Software Quality Research Laboratory
Department of Computing and Software, McMaster University
Hamilton, Ontario, Canada L8S 4K1

June 2006

Abstract

The pattern matching calculi introduced by the first author are a refinement of the λ-calculus that integrates mechanisms appropriate for fine-grained modelling of non-strict pattern matching.

While related work in the literature only uses a single monad, typically Maybe, for matchings, we present an axiomatic approach to semantics of these pattern matching calculi using two monads, one for expressions and one for matchings.

Although these two monads only need to be relatively lightly coupled, this semantics implies soundness of all core PMC rules, and is a useful tool for exploration of the design space for pattern matching calculi.

Using lifting and Maybe monads, we obtain standard Haskell semantics, and by adding another level of Maybe to both, we obtain a denotational semantics of the "matching failure as exceptions" approach of Erwig and Peyton Jones. Using list-like monads opens up interesting extensions in the direction of functional-logic programming.

A short version of this report appears as [Kahl, Carette+ 2006].

Software Quality Research Laboratory

McMaster University

SQRL Report No. 33

[*] This research has been supported by the National Science and Engineering Research Council (NSERC), Canada

Contents

1 Introduction

Although (pure) functional programming in general is very accessible to equational reasoning, the addition of pattern-matching function definitions introduces non-equations looking like equations, for example the second line in

```
isEmptyList (x : xs) = False
isEmptyList ys       = True
```

The operational semantics of such definitions employs the *functional rewriting strategy* defined over several pages by Plasmeijer and van Eekelen [Plasmeijer, van Eekelen 1993] or, essentially equivalently, in the section on `case` expressions in the Haskell report [Peyton Jones+ 2003]. This implies that syntactic use of the definitions of a program in reasoning about that program has to take into account that complex strategy, and loses the simplicity of equational reasoning.

The pattern matching calculus (PMC) introduced by Kahl [Kahl 2004] remedies this situation. It separates pattern matching aspects into a separate syntactic category of "matchings", not unlike groups of "case branches `p -> e`" considered by Harrison *et al.* [Harrison, Sheard+ 2002], but with an additional "argument supply" constructor that rationalises and generalises the pattern guards proposed by Erwig and Peyton Jones [Erwig, Peyton Jones 2001]. PMC is equipped with a confluent (second-order) rewriting system, thereby enabling equational reasoning starting from the definitions of a program. The rewriting system directly gives a normalisation strategy [Kahl 2004].

PMC allows straightforward internalisation of pattern matching definitions without the ballast of having to introduce the new variables necessary as `case` arguments (the syntax will be explained in detail in Sect. 2):

$$\mathit{isEmptyList} = \{\!|\, (x : \mathit{xs}) \Mapsto \upharpoonleft \mathsf{False} \upharpoonright \ \|\ \mathit{ys} \Mapsto \upharpoonleft \mathsf{True} \upharpoonright \,|\!\}$$

Application to the empty list [] induces the following reduction sequence:

$$\begin{array}{lll}
\mathit{isEmptyList}\ [] & \longrightarrow & \{\!|\, (x : \mathit{xs}) \Mapsto \upharpoonleft \mathsf{False} \upharpoonright \ \|\ \mathit{ys} \Mapsto \upharpoonleft \mathsf{True} \upharpoonright \,|\!\}\ [] \\
& \longrightarrow & \{\!|\, [] \rhd ((x : \mathit{xs}) \Mapsto \upharpoonleft \mathsf{False} \upharpoonright \ \|\ \mathit{ys} \Mapsto \upharpoonleft \mathsf{True} \upharpoonright) \,|\!\} \\
& \longrightarrow & \{\!|\, [] \rhd (x : \mathit{xs}) \Mapsto \upharpoonleft \mathsf{False} \upharpoonright \ \|\ [] \rhd \mathit{ys} \Mapsto \upharpoonleft \mathsf{True} \upharpoonright \,|\!\} \\
& \longrightarrow & \{\!|\, \lightning \ \|\ [] \rhd \mathit{ys} \Mapsto \upharpoonleft \mathsf{True} \upharpoonright \,|\!\} \\
& \longrightarrow & \{\!|\, [] \rhd \mathit{ys} \Mapsto \upharpoonleft \mathsf{True} \upharpoonright \,|\!\} \\
& \longrightarrow & \{\!|\, \upharpoonleft \mathsf{True} \upharpoonright \,|\!\} \\
& \longrightarrow & \mathsf{True}
\end{array}$$

There is also a "conservative embedding" of the λ-calculus into PMC: Application is the same, and abstraction translates into a one-alternative variable-pattern matching:

$$\lambda\ v\ .\ e \quad := \quad \{\!|\, v \Mapsto \upharpoonleft e \upharpoonright \,|\!\}$$

With this definition, β-reduction can be emulated by a three-step reduction sequence in PMC (the reduction rules are listed in Fig. 5 and explained in Appendix B):

$$\begin{array}{rcl}
(\lambda\ v\ .\ e)\ a & = & \{\!|\, v \Mapsto \upharpoonleft e \upharpoonright \,|\!\}\ a \\
& \xrightarrow[(\{\!|\ |\!\}@)]{} & \{\!|\, a \rhd v \Mapsto \upharpoonleft e \upharpoonright \,|\!\} \\
& \xrightarrow[(\rhd v)]{} & \{\!|\, \upharpoonleft e \upharpoonright [v \backslash a] \,|\!\} \\
& = & \{\!|\, \upharpoonleft e[v \backslash a] \upharpoonright \,|\!\} \\
& \xrightarrow[(\{\!|\upharpoonleft\upharpoonright|\!\})]{} & e[v \backslash a]
\end{array}$$

β-normal forms translate into PMC normal forms, and PMC reduction sequences starting from translations of λ-terms essentially correspond to β-reduction sequences, so the embedding is faithful.

Pattern guards extend Boolean guards with the ability to bind additional variables; Peyton Jones' standard example is:

```
clunky env v1 v2 |  Just r1 <- lookup env v1
                 , Just r2 <- lookup env v2  = r1 + r2
                 | otherwise                 = v1 + v2
```

This directly translates into PMC, with a slightly different structure (with appropriate conventions, we could omit more parentheses):

$$\begin{array}{l}
\mathit{clunky} = \{\!|\, \mathit{env} \Mapsto v_1 \Mapsto v_2 \Mapsto ((\mathit{lookup}\ \mathit{env}\ v_1 \rhd \mathsf{Just}(r_1) \Mapsto \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{lookup}\ \mathit{env}\ v_2 \rhd \mathsf{Just}(r_2) \Mapsto \upharpoonleft r_1 + r_2 \upharpoonright) \\
\qquad\qquad\qquad\qquad\qquad\ \ \|\, \upharpoonleft v_1 + v_2 \upharpoonright) \\
\qquad\quad |\!\}
\end{array}$$

PMC is really a family of calculi based on a common core syntax: starting from the rewriting system corresponding to Haskell evaluation or the standard functional strategy and *exchanging a single rule*, we obtain a system that corresponds to Erwig and Peyton Jones's proposal [Erwig, Peyton Jones 2001] to treat pattern matching failure as an exception that can be caught in the same *or in another* `case` expression.

In this paper, we provide a *semantic* basis for the exploration of these and further pattern matching calculi by giving a compositional monadic semantics for the core PMC syntax. The interesting aspect is that the two syntactic categories of PMC correspond to two separate monads that are, in general, only relatively lightly coupled. As we fundamentally use the separate notions of "computation" in each syntactic category, it is very natural to use a monadic formalism, and from there to continue using a categorical setting throughout for our semantics. It has been suggested to us that using a metalanguage like that of [Moggi 1991b] could clarify our presentation; while we agree with this, we do not yet know how to model the necessary "pointwise extensions" we need (see Sect. 4.1) in the metalanguage. Thus we have opted to stay with a purely categorical presentation.

The main contributions of this paper are the clean separation of concerns between the (monadic) semantics of *expressions* and of *matchings*, the crucial observation that the interpretation of function types must be different for matchings and expressions, and a clean isolation of the design choices available when considering pattern-matching semantics. Another important technical ingredient was the need to create appropriate "pointwise extensions" of operations in the base monads to function types — something routinely done in mathematics, but seldom done in statically typed programming languages.[1]

After presenting and explaining the abstract syntax of simply typed PMC in the next section, we fix some category-theoretical notation and terminology in Sect. 3.1 before defining the bimonadic PMC semantics in Sect. 4. In Sect. 5 we give the soundness theorem for the core reduction rules from [Kahl 2004] (listed in Appendix B) with respect to our semantics without further constraints on the two monads, and explain the core of its proof steps. We then start an exploration of possible bimonadic constellations for alternative interpretations of PMC in Sect. 6.

Acknowledgement. We would like to thank the anonymous referees of draft versions of this report for their valuable comments.

[1] Maple™ and Mathematica™ both overload arithmetic operators so that $f + g$ means pointwise addition, but both of these languages are dynamically typed.

2 Abstract PMC Syntax

PMC has two major syntactic categories, *expressions* and *matchings.* These are defined by mutual recursion.

When considering the analogy to functional programs, only *expressions* of the pattern matching calculus correspond to expressions of functional languages.

Matchings can be seen as a generalisation of (groups of) case alternatives. Matchings can expose patterns to be matched against arguments; we say such matchings are *waiting for argument supply,* and give them function types. Complete case expressions correspond to expressions formed from matchings that already have an argument supplied to their outermost patterns; matchings that have arguments supplied to all their open patterns are called *saturated.* Argument supply to patterns is separated from performing pattern matching itself; depending on the outcomes of the involved pattern matchings, saturated matchings can *succeed* and then *return* an expression, or they can *fail.*

Patterns form a separate, auxiliary syntactic category that will be used to construct pattern matchings.

In this paper, we will consider a class of simply-typed pattern matching calculi with common syntax; the abstract syntax of these calculi is defined by the following grammar:

$$
\begin{array}{lcll}
\mathsf{Pat} & ::= & \mathsf{Var} & \text{variable} \\
 & \mid & \mathsf{Constr}(\mathsf{Pat}, \ldots, \mathsf{Pat}) & \text{constructor pattern} \\[1ex]
\mathsf{Expr} & ::= & \mathsf{Var} & \text{variable} \\
 & \mid & \mathsf{Constr}(\mathsf{Expr}, \ldots, \mathsf{Expr}) & \text{constructor application} \\
 & \mid & \mathsf{Expr}\ \mathsf{Expr} & \text{function application} \\
 & \mid & \{\!|\ \mathsf{Match}\ |\!\} & \text{(result) extraction} \\
 & \mid & \oslash_{\mathsf{Type}} & \text{empty expression} \\[1ex]
\mathsf{Match} & ::= & \lceil \mathsf{Expr} \rceil & \text{lifting} \\
 & \mid & \lightning_{\mathsf{Type}} & \text{failure} \\
 & \mid & \mathsf{Pat} \Mapsto \mathsf{Match} & \text{pattern matching} \\
 & \mid & \mathsf{Expr} \rhd \mathsf{Match} & \text{argument supply} \\
 & \mid & \mathsf{Match}\ \|\ \mathsf{Match} & \text{alternative}
\end{array}
$$

Since this syntax has a number of unusual aspects, we explain the intuition behind it in more detail below.

Throughout this paper, we will use the following conventions for meta-level variables:

- α. α_i, β, ... are types; τ, τ_i are constructed types.
- v, v_i, w_i, x, x_i, y, y_i are variables; c, d are constructors.
- p, p_1, p_2, ..., q are patterns; m, m_1, m_2, ... are matchings,
- a, b, e, e_1, e_2, ..., f are expressions,
- i, k, n are natural numbers,

Types are generated from data type constructors and the function type constructor. Technically, we assume a family $(\mathsf{TConstr}_k)_{k\in\mathbb{N}}$ of disjoint countable sets of data type constructors of arity k, and types are generated by:

$$
\begin{array}{lcll}
\mathsf{Type} & ::= & \mathsf{TConstr}_k(\mathsf{Type}_1, \ldots, \mathsf{Type}_k) & \text{constructed types} \\
 & \mid & \mathsf{Type} \rightarrow \mathsf{Type} & \text{function types}
\end{array}
$$

For the sake of simplicity, we do not consider polymorphism in this paper, so there are no type variables, and therefore no concept of principal types; each well-typed expression or matching has exactly one type.

We then assume that the set of constructors is organised as a family of disjoint countable sets $\mathsf{Constr}_{\alpha_1\times\cdots\times\alpha_n\rightarrow\tau}$ for all types $\alpha_1, \ldots, \alpha_n, \tau$.

We also assume that the set of (expression) variables is organised as a family of disjoint countable sets

$$\mathsf{Var} = \biguplus_{\alpha\in\mathsf{Type}} \mathsf{Var}_\alpha$$

and a function $\mathsf{type} : \mathsf{Var} \rightarrow \mathsf{Type}$ to be given such that $v \in \mathsf{Var}_{\mathsf{type}\ v}$ for each variable $v \in \mathsf{Var}$. Type judgements then need no context.

For the purpose of our examples, all literals, like numbers and characters, are assumed to be constructors of appropriate types, and are used only in zero-ary constructions (which are written without parentheses). Constructors will, as usual, be used to build both patterns and expressions. Indeed, one might consider Pat as a subset of Expr.

Typing judgements expressing that pattern p, expression e, respectively matching m are well-typed of type α are written in the following way:

$$\underset{\mathsf{P}}{\vdash}\ p : \alpha \qquad\qquad \underset{\mathsf{E}}{\vdash}\ e : \alpha \qquad\qquad \underset{\mathsf{M}}{\vdash}\ m : \alpha$$

Patterns are built from variables and constructor applications. All variables occurring in a pattern are *free* in that pattern; for every pattern $p : \mathsf{Pat}$, we denote its set of free variables by $\mathsf{FV}(p)$. In the following, we silently restrict *all* patterns to be *linear*, i.e., not to contain more than one occurrence of any variable. The pattern typing rules are shown in Fig. 1.

$$\underset{\mathsf{P}}{\vdash}\ v : \mathsf{type}\ v \qquad\qquad \frac{c \in \mathsf{Constr}_{\alpha_1\times\cdots\times\alpha_n\rightarrow\tau} \qquad \underset{\mathsf{P}}{\vdash}\ p_1 : \alpha_1 \quad \cdots \quad \underset{\mathsf{P}}{\vdash}\ p_n : \alpha_n}{\underset{\mathsf{P}}{\vdash}\ c(p_1, \ldots, p_n) : \tau}$$

Figure 1: Pattern typing rules

Expressions are the syntactic category that embodies the term construction aspects; besides variables, constructor application and function application, we also have the following special kinds of expressions: Every matching m gives rise to the *(result) extraction* $\{\!|\ m\ |\!\}$. If the type of matching m is a function type, then $\{\!|\ m\ |\!\}$ extracts a function from m. If m is not a pattern matching again, then it can either succeed or fail; if it succeeds, then $\{\!|\ m\ |\!\}$ extracts the value(s) "returned" by m; otherwise, $\{\!|\ m\ |\!\}$ extracts "nothing", which can also be expressed as the expression $\oslash$, which is henceforth called the *empty expression.*

We use this somewhat uncommitted name "empty expression" since we shall consider two interpretations of $\oslash$:

- It can be a "manifestly undefined" expression equivalent to non-termination, following the common view that divergence is semantically equivalent to run-time errors.
- It can be a special "error" value, propagating matching failure considered as an "exception" through the syntactic category of expressions.

None of the expression constructors binds any variables; we overload the $\mathsf{FV}(\_)$ notation and use it to denote the set of free variables $\mathsf{FV}(e)$ for an expression $e : \mathsf{Expr}$. Expressions are typed according to the rules in Fig. 2.

$$\vdash_{\mathsf{E}} v : \mathsf{type}\ v \qquad\qquad \vdash_{\mathsf{E}} \varnothing_\alpha : \alpha \qquad\qquad \frac{\vdash_{\mathsf{M}} m : \alpha}{\vdash_{\mathsf{E}} \{\!| m |\!\} : \alpha}$$

$$\frac{c \in \mathsf{Constr}_{\alpha_1 \times \cdots \times \alpha_n \to \tau} \qquad \vdash_{\mathsf{E}} e_1 : \alpha_1 \quad \cdots \quad \vdash_{\mathsf{E}} e_n : \alpha_n}{\vdash_{\mathsf{E}} c(e_1, \ldots, e_n) : \tau} \qquad\qquad \frac{\vdash_{\mathsf{E}} e_1 : \alpha \to \beta \qquad \vdash_{\mathsf{E}} e_2 : \alpha}{\vdash_{\mathsf{E}} (e_1\ e_2) : \beta}$$

Figure 2: Expression typing rules

Matchings are the syntactic category that embodies the pattern analysis aspects:

- For an expression $e : \mathsf{Expr}$, the *lifting* or *expression embedding* $\upharpoonleft\mathsf{Expr}\upharpoonright$ can be seen as the matching that always succeeds and attempts to lift the result e into the enclosing expression, so we propose to read it "*lift* e".
- Failure $\lightning$ is the matching that always fails.
- The *pattern matching* $p \mapsto m$ waits for supply of one argument more than m; this pattern matching can be understood as succeeding on instances of the (linear) pattern $p : \mathsf{Pat}$ and then continuing to behave as the resulting instance of the matching $m : \mathsf{Match}$. It roughly corresponds to a single case alternative in languages with case expressions, or to pattern-binding λ-abstractions.
- *argument supply* $a \rhd m$ is the matching-level incarnation of function application, with the argument on the left and the matching it is supplied to on the right. It saturates the first argument m is waiting for. "$a \rhd m$" can be read "*a into m*" or "*a feeds m*". The inclusion of argument supply into the calculus is an important source of flexibility. By separating the aspects of traversing the boundary between expressions and matchings, and matching patterns against the right arguments, the design of the reduction system is made more modular.
- the *alternative* $m_1 \,\mathbf{|}\, m_2$ combines the possible matching results of m_1 and m_2 in some way that can usefully be understood as "alternative". In instances corresponding to conventional functional programming, it has to be understood sequentially: $m_1 \,\mathbf{|}\, m_2$ then behaves like m_1 until this fails, and then (and only then) it behaves like m_2.

The typing rules for matchings are again straight-forward, and are shown in Fig. 3.

$$\vdash_{\mathsf{M}} \lightning_\alpha : \alpha \qquad\qquad \frac{\vdash_{\mathsf{E}} e : \alpha}{\vdash_{\mathsf{M}} \upharpoonleft e \upharpoonright : \alpha} \qquad\qquad \frac{\vdash_{\mathsf{M}} m_1 : \alpha \qquad \vdash_{\mathsf{M}} m_2 : \alpha}{\vdash_{\mathsf{M}} (m_1 \,\mathbf{|}\, m_2) : \alpha}$$

$$\frac{\vdash_{\mathsf{E}} e : \alpha \qquad \vdash_{\mathsf{M}} m : \alpha \to \beta}{\vdash_{\mathsf{M}} (e \rhd m) : \beta} \qquad\qquad \frac{\vdash_{\mathsf{P}} p : \alpha \qquad \vdash_{\mathsf{M}} m : \beta}{\vdash_{\mathsf{M}} (p \mapsto m) : \alpha \to \beta}$$

Figure 3: Matching typing rules

Pattern matching $p \mapsto m$ binds all variables occurring in p, so $\mathsf{FV}(p \mapsto m) = \mathsf{FV}(m) - \mathsf{FV}(p)$, letting $\mathsf{FV}(m)$ denote the set of free variables of a matching m. Pattern matching is the only variable binder in this calculus — taking this into account, the definitions of free variables, bound variables, and substitution are as usual. Note that there are no "matching variables"; variables can only occur as patterns or as expressions.

We will omit the parentheses in matchings of the shape $a \rhd (p \mapsto m)$ since there is only one way to parse $a \rhd p \mapsto m$ in PMC.

As-Patterns and Irrefutable Patterns

Several "more advanced" pattern matching facilities have been proposed in the literature; Haskell98 defines two of those, namely as-patterns and irrefutable patterns. Both are defined via syntactic translations in the Haskell98 report. For as-patterns, the following translation is used:

$$\mathsf{case}\ v\ \mathsf{of}\{x@p \to e;\ \_ \to e'\} \qquad = \qquad \mathsf{case}\ v\ \mathsf{of}\{p \to (\lambda\, x \to e)\ v;\ \_ \to e'\}$$

In PMC, we can arrange this slightly more economically, thanks to the possibility to have sequential matchings — in Haskell with pattern guards, the same approach would be possible:

$$x@p \mapsto m \qquad = \qquad x \mapsto x \rhd p \mapsto m$$

Although irrefutable patterns appear to be much more intricate, the Haskell98 report formally defines these using a straight-forward translation:

$$\mathsf{case}\ v\ \mathsf{of}\{\tilde{}p \to e;\ \_ \to e'\} \quad = \quad (\lambda\, x_1 \ldots x_n \to e)\ (\mathsf{case}\ v\ \mathsf{of}\{p \to x_1\})\ \ldots\ (\mathsf{case}\ v\ \mathsf{of}\{p \to x_n\})$$
$$\text{where } x_1, \ldots, x_n \text{ are all the variables in } p$$

Non-strictness implies that matching (with potential failure) is only performed when evaluation of e requires one of the x_i. We can follow the same approach:

$$\tilde{}p \mapsto m \quad = \quad y \mapsto (y \rhd p \mapsto x_1) \rhd x_1 \mapsto \ldots (y \rhd p \mapsto x_n) \rhd x_n \mapsto m$$
$$\text{where } x_1, \ldots, x_n \text{ are all the variables in } p \text{ and } y \text{ is a new variable.}$$

The above two translations could be used as reduction rules. Another option is to restrict ourselves to a core calculus where only variables can be arguments of constructors in patterns; then the above two translations turn into expansion rules and we can consider as-patterns and irrefutable patterns as syntactic sugar. This approach also requires an expansion rule for nested patterns, considering them as just an abbreviation for sequential matchings:

$$c(p_1, \ldots, p_n) \mapsto m \qquad := \qquad c(y_1, \ldots, y_n) \mapsto y_1 \rhd p_1 \mapsto \cdots y_n \rhd p_n \mapsto m$$
$$\text{where } y_1, \ldots, y_n \text{ are distinct new variables.}$$

With this, pattern semantics becomes slightly easier to formulate, but nothing else really changes.

3 PMC Monads

We will define the semantics for PMC in an abstract categorical setting; we assume some "standard" familiarity with category theory basics (some more details are supplied in the appendix). We quickly introduce the notations we need in Sect. 3.1, then characterise the bi-monadic setting for PMC-semantics in Sect. 3.2, and also explain some simple instances of this setting.

3.1 Summary of Categorical Notation

We will define the semantics for PMC in an abstract categorical setting; in this section we assume some "standard" familiarity with category theory basics, and quickly introduce the notations we need; some more details are supplied in Appendix A.

We write $f : a \to b$ for a morphism with *source* object a and *target* object b. The identity on object a is id_a, and composition of morphisms $f : a \to b$ and $g : b \to c$ is written $f \mathbin{⨾} g$.

We assume a choice $\times$ of binary products with projections $\mathsf{fst}_{a,b} : a \times b \to a$ and $\mathsf{snd}_{a,b} : a \times b \to b$, and morphism pairing $\langle f, g \rangle : c \to a \times b$ for morphisms $f : c \to a$ and $g : c \to b$. We will denote by $\mathsf{term}_a : a \to \mathbb{1}$ the unique morphism into the terminal object $\mathbb{1}$.

As we restrict ourselves to cartesian closed categories, for every two objects a and b, there are an exponential object (for "functions from a to b") written $[a \to b]$, a "function application" morphism $\mathsf{eval}_{[a \to b]} : [a \to b] \times a \to b$, and a currying operation Λ that maps every morphism $f : c \times a \to b$ to the unique morphism $\Lambda f : c \to [a \to b]$ such that $(\Lambda f \times \mathsf{id}_a) \mathbin{⨾} \mathsf{eval}_{[a \to b]} = f$.

We essentially follow Barr and Wells [Barr, Wells 1999] in adopting the following notations: we write $\Pi i : \mathcal{I} \bullet a(i)$ for the indexed (but not necessarily ordered) product over the *finite* index set $\mathcal{I}$, with component $a(i)$ for index i; the projection to the sub-product indexed by elements of a subset $\mathcal{J} \subseteq \mathcal{I}$ is

$$\mathsf{proj}^a_{I \succ J} : (\Pi i : \mathcal{I} \bullet a(i)) \to (\Pi i : \mathcal{J} \bullet a(i)) \enspace .$$

We identify singleton products with their components: $(\Pi i : \{j\} \bullet a(i)) = a(j)$.

We will follow category theoretic usage in writing both the object mapping and the morphism mapping of a *functor* as an application of the functor name (Haskell uses the the $\mathsf{Functor}$ class member fmap for the morphism mapping). So for a functor H and a morphism $f : a \to b$, we have $H\ f : H\ a \to H\ b$.

Given two functors H and K between the same two categories, a *natural transformation* $t : H \to K$ is a function mapping each object a in the source category of H and K to a morphism $t_a : H\ a \to K\ a$ in the target category, such that for $f : a \to b$, the following diagram commutes (the *naturality* condition):

$$\begin{tikzcd} K\ a \arrow[r, "K\ f"] & K\ b \\ H\ a \arrow[u, "t_a"] \arrow[r, "H\ f"] & H\ b \arrow[u, "t_b"'] \end{tikzcd} \qquad \text{i.e.,} \qquad H\ f \mathbin{⨾} t_b = t_a \mathbin{⨾} K\ f$$

A *monad* is a triple $(M, \mathsf{return}^M, \mathsf{join}^M)$ consisting of an endofunctor M together with two natural transformations which, for readability, we also present as polymorphic morphisms:

$$\begin{array}{lcllllcl} \mathsf{return}^M & : & \mathsf{id} \to M & , \text{i.e.,} & & \mathsf{return}^M_a & : & a \to M\ a \\ \mathsf{join}^M & : & M \mathbin{⨾} M \to M & , \text{i.e.,} & & \mathsf{join}^M_a & : & M\ (M\ a) \to M\ a \end{array}$$

(The required laws are listed in Appendix A.3.)

Every monad M gives rise to a so-called Kleisli category; it has return^M morphisms as identities, and for arrows $f : a \to M\ b$ and $g : b \to M\ c$, composition is defined as:

$$\begin{array}{lcl} f \odot_{\mathsf{M}} g & : & a \to M\ c \\ f \odot_{\mathsf{M}} g & = & f \mathbin{⨾} M\ g \mathbin{⨾} \mathsf{join}^M_c \end{array}$$

An *additive monad* in addition has two natural transformations

$$\begin{array}{lcllllcl} \mathsf{zero}^M & : & \mathbb{1} \to M & , \text{i.e.,} & & \mathsf{zero}^M_a & : & \mathbb{1} \to M\ a \\ \mathsf{plus}^M & : & M \times M \to M & , \text{i.e.,} & & \mathsf{plus}^M_a & : & M\ a \times M\ a \to M\ a \end{array}$$

with zero^M being (up to the canonical isomorphisms) a right and left unit for plus^M, and plus^M associative.

As Moggi explains in [Moggi 1991b], we need *strong monads* for being able to deal with expressions with more than one free variable; a strong monad M has a natural transformation

$$\mathsf{strengthL}^M_{a,b} : a \times M\ b \to M\ (a \times b)$$

called *tensorial strength* satisfying several properties listed in the appendix A.4. Using the isomorphism ($\mathsf{swap}_{a,b}$ from $a \times b$ to $b \times a$), we can define the "swapped version" $\mathsf{strengthR}^M_{a,b} : M\ a \times b \to M\ (a \times b)$. This allows us to define

$$\otimes_{\mathsf{M}} : (M\ a \times M\ b) \to M(a \times b)$$

as well as an n-ary version of the same via folding over ordered tuples, still denoted $\otimes_{\mathsf{M}}$.

For additive strong monads, we also demand $(\mathsf{id}_a \times \mathsf{zero}^{\mathsf{M}}_b) \mathbin{⨾} \mathsf{strengthL}^M_{a,b} = \mathsf{zero}^{\mathsf{M}}_{a \times b}$.

3.2 The Bi-Monadic Setting

We need a monad E for the expression semantics, and an additive monad M for the matching semantics, so we have $\mathsf{zero}^{\mathsf{M}}$ and $\mathsf{plus}^{\mathsf{M}}$. In addition, there should be two natural transformations

$$\begin{array}{rcllrcl} \mathsf{extract} & : & \mathsf{M} \to \mathsf{E} & , \text{i.e.,} & \mathsf{extract}_a & : & \mathsf{M}\ a \to \mathsf{E}\ a \\ \mathsf{lift} & : & \mathsf{E} \to \mathsf{M} & , \text{i.e.,} & \mathsf{lift}_a & : & \mathsf{E}\ a \to \mathsf{M}\ a \end{array}$$

satisfying the following additional laws:

$$\begin{array}{rcllrcll} \mathsf{lift} \mathbin{⨾} \mathsf{extract} & = & \mathsf{id}_{\mathsf{E}} & , \text{i.e.,} & \mathsf{lift}_a \mathbin{⨾} \mathsf{extract}_a & = & \mathsf{id}_{\mathsf{E}\ a} & (\mathsf{lift} \mathbin{⨾} \mathsf{extract}) \\ \mathsf{return}^{\mathsf{E}} \mathbin{⨾} \mathsf{lift} & = & \mathsf{return}^{\mathsf{M}} & , \text{i.e.,} & \mathsf{return}^{\mathsf{E}}_a \mathbin{⨾} \mathsf{lift}_a & = & \mathsf{return}^{\mathsf{M}}_a & (\mathsf{return}^{\mathsf{E}} \mathbin{⨾} \mathsf{lift}) \end{array}$$

The law $(\mathsf{lift} \mathbin{⨾} \mathsf{extract})$ ensures that lift_a is injective; a further consequences of these laws is:

$$\mathsf{return}^{\mathsf{M}}_a \mathbin{⨾} \mathsf{extract}_a = \mathsf{return}^{\mathsf{E}}_a \mathbin{⨾} \mathsf{lift}_a \mathbin{⨾} \mathsf{extract}_a = \mathsf{return}^{\mathsf{E}}_a \enspace .$$

Although it is tempting to demand that lift and $\mathsf{extract}$ should be monad homomorphisms, i.e., not only preserve return, but also join, we have not found it necessary to make that assumption for proving that the core PMC reduction rules are sound with respect to the semantics given in Sect. 4.

Two particularly simple patterns of binmonadic settings will cover most of the examples discussed in Sect. 6:

Setting 3.2.1 $(\mathsf{M} = \mathsf{E})$

We can use the same monad in both rôles of matching monad and expression monad, with identical natural transformations for $\mathsf{extract}$ and lift. Such a setting trivially satisfies the laws $(\mathsf{lift} \mathbin{⨾} \mathsf{extract})$ and $(\mathsf{return}^{\mathsf{E}} \mathbin{⨾} \mathsf{lift})$.

Setting 3.2.2 $(\mathsf{M} = \mathsf{E} + \mathbb{1})$

More interesting is the case where the image of $\mathsf{zero}^{\mathsf{M}}_a$ is disjoint from the image of lift_a.

On the first class of settings we consider, the matching monad M is the monad coproduct (see [Lüth, Ghani 2002]) of the expression monad E and the constant monad $\mathbb{1}$. This gives us as the two monad coproduct injections the natural transformations $\mathsf{lift} : \mathsf{E} \to \mathsf{M}$ and $\mathsf{zero}^{\mathsf{M}} : \mathbb{1} \to \mathsf{M}$; since these commute by definition with return and join, the law $(\mathsf{return}^{\mathsf{E}} \mathbin{⨾} \mathsf{lift})$ is automatically satisfied.

For any choice of monad homomorphism $\mathsf{empty}^{\mathsf{E}} : \mathbb{1} \to \mathsf{E}$, we then define $\mathsf{extract} : \mathsf{M} \to \mathsf{E}$ as the mediating morphism $\mathsf{extract} := [\mathsf{id}_{\mathsf{E}}, \mathsf{empty}^{\mathsf{E}}]$, and from the coproduct definition we immediately obtain ($\mathsf{lift} \fatsemi \mathsf{extract}$), and:

$$\mathsf{zero}^{\mathsf{M}} \fatsemi \mathsf{extract} = \mathsf{zero}^{\mathsf{M}} \fatsemi [\mathsf{id}_{\mathsf{E}}, \mathsf{empty}^{\mathsf{E}}] = \mathsf{empty}^{\mathsf{E}} \qquad (\mathsf{zero} \fatsemi \mathsf{extract})$$

For the additive part of M, we still need to define $\mathsf{plus}^{\mathsf{M}} : \mathsf{M} \times \mathsf{M} \to \mathsf{M}$. To be able to essentially follow the additive pattern of the Maybe monad, we restrict ourselves to cases where there is a distribution isomorphism $\mathsf{distrL}_{\mathsf{E},\mathbb{1}}$ from $\mathsf{M} \times \mathsf{M} = (\mathsf{E} + \mathbb{1}) \times \mathsf{M}$ to $(\mathsf{E} \times \mathsf{M}) + (\mathbb{1} \times \mathsf{M})$, so we can define:

$$\mathsf{plus}^{\mathsf{M}} = \mathsf{distrL}_{\mathsf{E},\mathbb{1}} \fatsemi [\mathsf{fst}_{\mathsf{E},\mathsf{M}} \fatsemi \mathsf{lift}\ ,\ \mathsf{snd}_{\mathbb{1},\mathsf{M}}]$$

4 Monadic PMC Semantics

The semantics in this section is very much influenced by previous work, more specifically [Moggi 1991a; Moggi 1991b; Jung, Fiore+ 1996].

4.1 Type Semantics

The interpretation of function types is different for matchings and expressions. Therefore, for defining type semantics in the setting of the two monads E and M, we will use $\mathsf{K} \in \{\mathsf{E}, \mathsf{M}\}$ as a meta-variable to unify treatment of expression and matching semantics.

For each of our two syntactic categories of expressions and matchings, we will define below *two* different type semantics (both parameterised with a monad K) for each type α:

- the "raw" type semantics $\llbracket\alpha\rrbracket_{\mathsf{K}}$, and
- the "standard" type semantics $\llbracket\alpha\rrbracket^{\mathsf{K}}$.

As a mnemonic rule, one could remember that "superscript semantics" $\llbracket\alpha\rrbracket^{\mathsf{K}} = \mathsf{K}\ \llbracket\alpha\rrbracket_{\mathsf{K}}$ is, in a generalised way, "in" the monad, while subscript $\llbracket\alpha\rrbracket_{\mathsf{K}}$ semantics only "involves" the monad, where "involving" means that the type typically is a container of items "in" the monad.

Constructed Types

For each constructed type τ, the type semantics is obtained from the raw type semantics by application of the corresponding monad:

$$\llbracket\tau\rrbracket^{\mathsf{K}} = \mathsf{K}\ \llbracket\tau\rrbracket_{\mathsf{K}}\ ,$$

and the "raw" semantics $\llbracket\tau\rrbracket_{\mathsf{K}}$ is the direct sum (over all constructors producing this type) of the direct products of the corresponding constructor argument types.[2] Since constructor applications take expressions as arguments, these argument types have to be wrapped in the expression semantics monad E — the raw semantics of constructed types τ does indeed not depend on K.

$$\llbracket\tau\rrbracket_{\mathsf{K}} = \biguplus_{n \in \mathbb{N}, (c : \alpha_1 \times \cdots \times \alpha_n \to \tau) \in \mathsf{Constr}} \llbracket\overline{\alpha}\rrbracket_{\mathsf{E}}$$

where we use $\llbracket\overline{\alpha}\rrbracket_{\mathsf{E}}$ to denote

$$\llbracket\alpha_1\rrbracket^{\mathsf{E}} \times \cdots \times \llbracket\alpha_n\rrbracket^{\mathsf{E}}$$

[2] Since, normally, only finite sets of constructors are considered, practical applications only require finite sums to exist in the underlying category.

A constructor $c : \alpha_1 \times \cdots \times \alpha_n \to \tau$ is then interpreted by the corresponding *constructor injection* $c^{\mathsf{E}} : \llbracket\overline{\alpha}\rrbracket_{\mathsf{E}} \to \llbracket\tau\rrbracket_{\mathsf{E}}$ together with the corresponding *destructor morphism*:

$$\widetilde{c}^{\mathsf{E}} : \llbracket\tau\rrbracket_{\mathsf{E}} \to \mathsf{M}\ (\llbracket\overline{\alpha}\rrbracket_{\mathsf{E}})$$

such that $c^{\mathsf{E}} \fatsemi \widetilde{c}^{\mathsf{E}} = \mathsf{return}^{\mathsf{M}}_{\llbracket\overline{\alpha}\rrbracket_{\mathsf{E}}}$ and, if $c \neq d$, with $d : \beta_1 \times \cdots \times \beta_n \to \tau_d$, then:

$$d^{\mathsf{E}} \fatsemi \widetilde{c}^{\mathsf{E}} = \mathsf{term}^{\mathcal{C}}_{\llbracket\overline{\beta}\rrbracket_{\mathsf{E}}} \fatsemi \mathsf{zero}^{\mathsf{M}}_{\llbracket\overline{\alpha}\rrbracket_{\mathsf{E}}}$$

Function Types

Now consider the raw semantics of function types. Since *all* application constructs of the PMC syntax (constructor application, function application, and argument supply) take *expressions* as arguments, the argument type in the K-semantics will always be the *expression* type semantics of the argument type β. The result type however depends on the context, and will therefore be the (raw) K-semantics of the result type γ:

$$\begin{aligned} \llbracket\beta \to \gamma\rrbracket_{\mathsf{K}} &= [\llbracket\beta\rrbracket^{\mathsf{E}} \to \llbracket\gamma\rrbracket_{\mathsf{K}}] \\ \llbracket\beta \to \gamma\rrbracket^{\mathsf{K}} &= [\llbracket\beta\rrbracket^{\mathsf{E}} \to \llbracket\gamma\rrbracket^{\mathsf{K}}] \end{aligned}$$

In order to ease analysis of our semantics, we provide essentially full type information, but this tends to blow up our notation. We therefore incorporate the function type semantics pattern into a variant notation for eval:

$$\begin{aligned} \mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma} &: \llbracket\beta \to \gamma\rrbracket^{\mathsf{K}} \times \llbracket\beta\rrbracket^{\mathsf{E}} \to \llbracket\gamma\rrbracket^{\mathsf{K}} \\ \mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma} &:= \mathsf{eval}_{[\llbracket\beta\rrbracket^{\mathsf{E}} \to \llbracket\gamma\rrbracket^{\mathsf{K}}]} \end{aligned}$$

Point-wise Extension Combinators

Since the semantics of function types is not directly monadic, we need a generalisation of Kleisli composition: If $f : q \to \mathsf{K}\ r$ and $g : r \to \llbracket\alpha\rrbracket^{\mathsf{K}}$, then $(f \boxdot^{\mathsf{K}}_{\alpha} g) : q \to \llbracket\alpha\rrbracket^{\mathsf{K}}$ is defined by:

$$\begin{aligned} f \boxdot^{\mathsf{K}}_{\tau} g &= f \odot_{\mathsf{K}} g \\ f \boxdot^{\mathsf{K}}_{\beta \to \gamma} g &= \Lambda\Big(((f \times \mathsf{id}_{\llbracket\beta\rrbracket^{\mathsf{E}}}) \fatsemi \mathsf{strengthR}^{\mathsf{K}}_{r,\llbracket\beta\rrbracket^{\mathsf{E}}}) \boxdot^{\mathsf{K}}_{\gamma} ((g \times \mathsf{id}_{\llbracket\beta\rrbracket^{\mathsf{E}}}) \fatsemi \mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma})\Big) \end{aligned}$$

This behaves "mostly like" Kleisli composition: we have $(f \fatsemi g) \boxdot^{\mathsf{K}}_{\alpha} h = f \fatsemi (g \boxdot^{\mathsf{K}}_{\alpha} h)$ (Lemma C.2), so we can omit those parentheses, and we also have $\mathsf{return}^{\mathsf{K}}_{r} \boxdot^{\mathsf{K}}_{\alpha} g = g$ (Lemma C.1).

Because of the way we treat of function types, we shall frequently need a construction that corresponds to "point-wise extension to function types" of the composition $f \fatsemi t_{\llbracket\tau\rrbracket}$ of a morphism $f : q \to \llbracket\tau\rrbracket^{\mathsf{K}}$ with a transformation $t : \mathsf{K} \to \mathsf{H}$. For this purpose, we define the following "generalised composition" operation inductively over the function type structure.

If $f : q \to [\![\alpha]\!]^{\mathsf{K}}$, then $[_q\; f \boxed{\fatsemi}^{\alpha}\; t\;] : q \to [\![\alpha]\!]^{\mathsf{H}}$ is defined by:[3]

$$\begin{aligned}
[_q\; f \boxed{\fatsemi}^{\tau}\; t\;] &= f \fatsemi t_{[\![\tau]\!]} \\
[_q\; f \boxed{\fatsemi}^{\beta\to\gamma}\; t\;] &= \Lambda[_{q\times[\![\beta]\!]^{\mathsf{E}}}\; (f \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}}) \fatsemi \mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma} \boxed{\fatsemi}^{\gamma}\; t\;]
\end{aligned}$$

This even works for $\mathsf{K} = \mathbb{1}$ since the constant functor $\mathbb{1}$ is trivially a strong monad; in this case all f arguments are morphisms to the terminal object $\mathbb{1}$.

Similarly, we need a "pointwise extension" of $\mathsf{plus}^{\mathsf{M}}$ to function types: For two morphisms $g_1, g_2 : q \to [\![\alpha]\!]^{\mathsf{M}}$, we define $g_1 \boxplus_\alpha g_2 : q \to [\![\alpha]\!]^{\mathsf{M}}$ as:

$$g_1 \boxplus_\tau g_2 = \langle g_1, g_2 \rangle \fatsemi \mathsf{plus}^{\mathsf{M}} \qquad\qquad g_1 \boxplus_{\beta\to\gamma} g_2 = \Lambda \left(\begin{array}{l} (g_1 \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}}) \fatsemi \mathsf{Eval}^{\mathsf{M}}_{\beta,\gamma} \\ \boxplus_\gamma \\ (g_2 \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}}) \fatsemi \mathsf{Eval}^{\mathsf{M}}_{\beta,\gamma} \end{array} \right)$$

The following properties enable high-level reasoning using the point-wise extension combinators defined above; the proofs of these and more can be found in Appendix C:

$$\begin{aligned}
([_q\; f \boxed{\fatsemi}^{\beta\to\gamma}\; t\;] \times g) \fatsemi \mathsf{Eval}^{\mathsf{H}}_{\beta,\gamma} &= [_{q\times r}\; (f \times g) \fatsemi \mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma} \boxed{\fatsemi}^{\gamma}\; t\;] \\
f \fatsemi [_r\; g \boxed{\fatsemi}^{\alpha}\; t\;] &= [_q\; f \fatsemi g \boxed{\fatsemi}^{\alpha}\; t\;] \\
[_q\; [_q\; f \boxed{\fatsemi}^{\alpha}\; t\;] \boxed{\fatsemi}^{\alpha}\; u\;] &= [_q\; f \boxed{\fatsemi}^{\alpha}\; t \fatsemi u\;] \\
f \fatsemi (g_1 \boxplus_\alpha g_2) &= f \fatsemi g_1 \boxplus_\alpha f \fatsemi g_2 \\
((g_1 \boxplus_{\beta\to\gamma} g_2) \times h) \fatsemi \mathsf{Eval}^{\mathsf{H}}_{\beta,\gamma} &= ((g_1 \times h) \fatsemi \mathsf{Eval}^{\mathsf{H}}_{\beta,\gamma}) \boxplus_\gamma ((g_2 \times h) \fatsemi \mathsf{Eval}^{\mathsf{H}}_{\beta,\gamma})
\end{aligned}$$

Examples

Throughout the following examples, we assume an arbitrary but fixed expression monad E.

For zero-ary constructors, the product $[\![\overline{\alpha}]\!]_{\mathsf{E}} = [\![\alpha_1]\!]^{\mathsf{E}} \times \cdots \times [\![\alpha_0]\!]^{\mathsf{E}}$ of argument types is a zero-ary product, and therefore the (more precisely: a) terminal object $\mathbb{1}$.

With Bool defined by $\mathsf{data\ Bool = False \mid True}$, we therefore obtain for any monad K:

$$[\![\mathsf{Bool}]\!]_{\mathsf{K}} = \mathbb{1} + \mathbb{1} \qquad \text{and} \qquad [\![\mathsf{Bool}]\!]^{\mathsf{K}} = \mathsf{K}\ (\mathbb{1} + \mathbb{1})\ .$$

For the function type $\mathsf{Bool} \to \mathsf{Bool}$ we obtain:

$$[\![\mathsf{Bool} \to \mathsf{Bool}]\!]_{\mathsf{K}} = [[\![\mathsf{Bool}]\!]^{\mathsf{E}} \to [\![\mathsf{Bool}]\!]_{\mathsf{K}}] \qquad \text{and} \qquad [\![\mathsf{Bool} \to \mathsf{Bool}]\!]^{\mathsf{K}} = [[\![\mathsf{Bool}]\!]^{\mathsf{E}} \to [\![\mathsf{Bool}]\!]^{\mathsf{K}}]$$

If we define $\mathsf{data\ BoolFun = BF\ (Bool \to Bool)}$, we obtain:

$$[\![\mathsf{BoolFun}]\!]_{\mathsf{K}} = [\![\mathsf{Bool} \to \mathsf{Bool}]\!]^{\mathsf{E}} = [[\![\mathsf{Bool}]\!]^{\mathsf{E}} \to [\![\mathsf{Bool}]\!]^{\mathsf{E}}]$$

[3] Note that the transformation t has to be mentioned in $\left[_q\; f \boxed{\fatsemi}^{\alpha}\; t\;\right]$ without type argument, since it will be instantiated as $t_a : \mathsf{K}\ a \to \mathsf{H}\ a$ at different types a.
Also note that we put the subscript q not close to the box, but after the opening parenthesis, since q is the type "before f".

and

$$[\![\mathsf{BoolFun}]\!]^{\mathsf{K}} = \mathsf{K}\ [[\![\mathsf{Bool}]\!]^{\mathsf{E}} \to [\![\mathsf{Bool}]\!]^{\mathsf{E}}]\ .$$

Expanding these one step further:

$$\begin{aligned}
[\![\mathsf{Bool} \to \mathsf{Bool}]\!]_{\mathsf{K}} &= [[\![\mathsf{Bool}]\!]^{\mathsf{E}} \to [\![\mathsf{Bool}]\!]_{\mathsf{K}}] &= [\mathsf{E}\ (\mathbb{1}+\mathbb{1}) \to \quad (\mathbb{1}+\mathbb{1})] \\
[\![\mathsf{BoolFun}]\!]_{\mathsf{K}} &= [[\![\mathsf{Bool}]\!]^{\mathsf{E}} \to [\![\mathsf{Bool}]\!]^{\mathsf{E}}] &= [\mathsf{E}\ (\mathbb{1}+\mathbb{1}) \to \mathsf{E}\ (\mathbb{1}+\mathbb{1})]
\end{aligned}$$

and:

$$\begin{aligned}
[\![\mathsf{Bool} \to \mathsf{Bool}]\!]^{\mathsf{K}} &= [[\![\mathsf{Bool}]\!]^{\mathsf{E}} \to [\![\mathsf{Bool}]\!]^{\mathsf{K}}] &= [\mathsf{E}\ (\mathbb{1}+\mathbb{1}) \to \mathsf{K}\ (\mathbb{1}+\mathbb{1})] \\
[\![\mathsf{BoolFun}]\!]^{\mathsf{K}} &= \mathsf{K}\ [[\![\mathsf{Bool}]\!]^{\mathsf{E}} \to [\![\mathsf{Bool}]\!]^{\mathsf{E}}] &= \mathsf{K}\ [\mathsf{E}\ (\mathbb{1}+\mathbb{1}) \to \mathsf{E}\ (\mathbb{1}+\mathbb{1})]
\end{aligned}$$

4.2 Organisation of the Semantic Functions

While in strict languages, in the rewriting semantics only values can be substituted for variables, and analogously only values need to be bound to variables by the valuations in the denotational semantics, we are here targeting non-strict languages, where the operational semantics can substitute arbitrary expressions for variables, and therefore, analogously, the type of the denotational variable semantics has to coincide with that of the expression semantics. The object associated with a variable is therefore the E-image of the object that interprets the variable's type.

For the sake of conciseness and readability, we abbreviate the object corresponding to the type of a variable v by

$$v^{\mathsf{E}} := [\![\mathsf{type}(v)]\!]^{\mathsf{E}}$$

and also introduce similar notation for each set $\mathcal{V}$ of variables:

$$\mathcal{V}^{\mathsf{E}} := \Pi v : \mathcal{V} \bullet [\![\mathsf{type}(v)]\!]^{\mathsf{E}}$$

Since we want the reduction rules to translate into semantic equations, both sides of a rule always have to be interpreted in a compatible way; since the reduction rules do not preserve all free variables, we have to externally impose a source object for the semantic morphisms.

Therefore, given a variable set $\mathcal{V}$, we define the semantics of an expression e of type α with $\mathsf{FV}(e) \subseteq \mathcal{V}$ as a morphism from the product corresponding to the variable set $\mathcal{V}$ to the object corresponding to α:

$$[\![e]\!]^{\mathsf{E}}_{\mathcal{V},\alpha} : \mathcal{V}^{\mathsf{E}} \to [\![\alpha]\!]^{\mathsf{E}}$$

(When the type α is clear from the context, we write $[\![e]\!]^{\mathsf{E}}_{\mathcal{V}}$ instead of $[\![e]\!]^{\mathsf{E}}_{\mathcal{V},\alpha}$, and analogously for the other semantics functions.)

For each matching m of type α, we define its semantics as a morphism in the Kleisli category for M from the variables to the result type:

$$[\![m]\!]^{\mathsf{M}}_{\mathcal{V},\alpha} : \mathcal{V}^{\mathsf{E}} \to [\![\alpha]\!]^{\mathsf{M}}$$

Finally, to each pattern p of type α, we associate a morphism in the Kleisli category of M from the object used for expression semantics of type α to the object corresponding to the set of free variables of the pattern:

$$[\![p]\!]^{\mathsf{P}}_{\alpha} : [\![\alpha]\!]^{\mathsf{E}} \to \mathsf{M}\ (\mathsf{FV}(p)^{\mathsf{E}})$$

Constructor pattern semantics have to be "strict" as can be seen from the first occurrence of $\odot_{\mathsf{M}}$ in the corresponding clause in Fig. 4.

The definitions for all three semantics functions are listed in Fig. 4.

Pattern semantics:

$$
\begin{array}{lcl}
\llbracket p \rrbracket^{\mathsf{P}}_{\alpha} & : & \llbracket \alpha \rrbracket^{\mathsf{E}} \to \mathsf{M}\ (\mathsf{FV}(p)^{\mathsf{E}}) \\
\llbracket v \rrbracket^{\mathsf{P}} & = & \mathsf{return}^{\mathsf{M}}_{v^{\mathsf{E}}} \\
\llbracket c(p_1, \dots, p_n) \rrbracket^{\mathsf{P}} & = & \mathsf{lift}_{\llbracket \tau \rrbracket} \odot_{\mathsf{M}} \widetilde{c}^{\mathsf{E}} \odot_{\mathsf{M}} ((\llbracket p_1 \rrbracket^{\mathsf{P}} \times \cdots \times \llbracket p_n \rrbracket^{\mathsf{P}}) ; \bigotimes)
\end{array}
$$

for $c : \alpha_1 \times \cdots \times \alpha_n \to \tau$.
The target type here is isomorphic to $\mathsf{M}\ (\Pi v : \mathsf{FV}(p) \bullet \llbracket \mathsf{type}(v) \rrbracket^{\mathsf{E}})$; for the sake of conciseness we consider these two types as identified.

Expression semantics:

$$
\begin{array}{lcl}
\llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\alpha} & : & \mathcal{V}^{\mathsf{E}} \to \llbracket \alpha \rrbracket^{\mathsf{E}} \\
\llbracket v \rrbracket^{\mathsf{E}}_{\mathcal{V}} & = & \mathsf{proj}^{\mathsf{E}}_{\mathcal{V} \succ \{v\}} \\
\llbracket c(e_1, \dots, e_n) \rrbracket^{\mathsf{E}}_{\mathcal{V}} & = & \langle \llbracket e_1 \rrbracket^{\mathsf{E}}_{\mathcal{V}}, \dots, \llbracket e_n \rrbracket^{\mathsf{E}}_{\mathcal{V}} \rangle ; c^{\mathsf{E}} ; \mathsf{return}^{\mathsf{E}}_{\llbracket \alpha \rrbracket_{\mathsf{E}}} \\
\llbracket f\ a \rrbracket^{\mathsf{E}}_{\mathcal{V},\gamma} & = & \langle \llbracket f \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta \to \gamma}, \llbracket a \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle ; \mathsf{Eval}^{\mathsf{E}}_{\beta,\gamma} \\
\llbracket \{\!|\ m\ |\!\} \rrbracket^{\mathsf{E}}_{\mathcal{V},\alpha} & = & [_{\mathcal{V}^{\mathsf{E}}}\ \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha}\ \boxed{;}^{\alpha}\ \mathsf{extract}\ ] \\
\llbracket \oslash_{\alpha} \rrbracket^{\mathsf{E}}_{\mathcal{V}} & = & [_{\mathcal{V}^{\mathsf{E}}}\ \mathsf{term}^{\mathcal{C}}_{\mathcal{V}^{\mathsf{E}}}\ \boxed{;}^{\alpha}\ \mathsf{zero}^{\mathsf{M}} ; \mathsf{extract}\ ]
\end{array}
$$

Matching semantics:

$$
\begin{array}{lcl}
\llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} & : & \mathcal{V}^{\mathsf{E}} \to \llbracket \alpha \rrbracket^{\mathsf{M}} \\
\llbracket \rceil e \lceil \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} & = & [_{\mathcal{V}^{\mathsf{E}}}\ \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V}}\ \boxed{;}^{\alpha}\ \mathsf{lift}\ ] \\
\llbracket \lightning_{\alpha} \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} & = & [_{\mathcal{V}^{\mathsf{E}}}\ \mathsf{term}^{\mathcal{C}}_{\mathcal{V}^{\mathsf{E}}}\ \boxed{;}^{\alpha}\ \mathsf{zero}^{\mathsf{M}}\ ] \\
\llbracket a \rhd m \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} & = & \langle \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V},\beta \to \gamma}, \llbracket a \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle ; \mathsf{Eval}^{\mathsf{M}}_{\beta,\gamma} \\
\llbracket m_1 \mathbin{\boldsymbol{|}} m_2 \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} & = & \llbracket m_1 \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} \boxplus_{\alpha} \llbracket m_2 \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} \\
\llbracket p \Mapsto m \rrbracket^{\mathsf{M}}_{\mathcal{V},\beta \to \gamma} & = & \Lambda \left((\mathsf{proj}^{\mathsf{E}}_{\mathcal{V} \succ \mathcal{U}} \times \llbracket p \rrbracket^{\mathsf{P}}_{\beta}) ; \mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(p)^{\mathsf{E}}}\ \boxdot^{\mathsf{M}}_{\gamma}\ \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U} \oplus \mathsf{FV}(p),\gamma} \right)
\end{array}
$$

where $\mathcal{U} = \mathcal{V} \setminus \mathsf{FV}(p)$, and a product rearrangement morphism is again omitted

Figure 4: PMC semantics

5 Soundness of the Core Reduction Rules

For the core reduction rules of PMC listed in Fig. 5 (see [Kahl 2004] and Appendix B for more explanation), we prove the following soundness result in Appendix D:

Theorem 5.1 All core reduction rules listed in Fig. 5 are sound at arbitrary types. □

Here is a quick summary of which assumptions were crucial for the proofs to succeed; the detailed proofs, to be found in Appendix D, mostly proceed at the level of the semantics definitions of Fig. 4, thanks to the properties of the "pointwise extensions" operators $\boxed{;}^{\alpha}$ and $\boxplus_{\alpha}$ listed in Sect. 4.1.

- ($\lightning \mathbin{\boldsymbol{|}}$) relies on zero^{M}_{τ} being a left-unit for plus^{M}_{τ}.
- ($\rhd \mathbin{\boldsymbol{|}}$) relies crucially on the type-dependent, recursive definition of $\boxplus$.

$$
\begin{array}{rcll}
\lightning \mathbin{\boldsymbol{|}} m & \xrightarrow[\mathsf{M}]{} & m & (\lightning \mathbin{\boldsymbol{|}}) \\
\{\!|\ \lightning\ |\!\} & \xrightarrow[\mathsf{E}]{} & \oslash & (\{\!|\ \lightning\ |\!\}) \\
\oslash\ e & \xrightarrow[\mathsf{E}]{} & \oslash & (\oslash @) \\
e \rhd \lightning & \xrightarrow[\mathsf{M}]{} & \lightning & (\rhd \lightning) \\
e \rhd (m_1 \mathbin{\boldsymbol{|}} m_2) & \xrightarrow[\mathsf{M}]{} & (e \rhd m_1) \mathbin{\boldsymbol{|}} (e \rhd m_2) & (\rhd \mathbin{\boldsymbol{|}}) \\
\{\!|\ \rceil e \lceil\ |\!\} & \xrightarrow[\mathsf{E}]{} & e & (\{\!|\ \rceil \lceil\ |\!\}) \\
\{\!|\ m\ |\!\}\ a & \xrightarrow[\mathsf{E}]{} & \{\!|\ a \rhd m\ |\!\} & (\{\!|\ |\!\} @) \\
a \rhd \rceil e \lceil & \xrightarrow[\mathsf{M}]{} & \rceil e\ a \lceil & (\rhd \rceil \lceil) \\
a \rhd v \Mapsto m & \xrightarrow[\mathsf{M}]{} & m[v \backslash a] & (\rhd v)
\end{array}
$$

$$
\begin{array}{rcll}
d(e_1, \dots, e_k) \rhd c(p_1, \dots, p_n) \Mapsto m & \xrightarrow[\mathsf{M}]{} & \lightning \qquad \text{if } c \neq d \text{ or } k \neq n & (d \rhd c) \\
c(e_1, \dots, e_n) \rhd c(p_1, \dots, p_n) \Mapsto m & \xrightarrow[\mathsf{M}]{} & e_1 \rhd p_1 \Mapsto \cdots e_n \rhd p_n \Mapsto m & (c \rhd c)
\end{array}
$$

$$\text{if } \mathsf{FV}(c(e_1, \dots, e_n)) \cap \mathsf{FV}(c(p_1, \dots, p_n)) = \{\}$$

Figure 5: PMC core reduction rules

- ($\{\!|\ \lightning\ |\!\}$) relies on compositionality for $\boxed{;}^{\alpha}$.
- ($\{\!|\ \rceil \lceil\ |\!\}$) is because $\mathsf{lift}_{\tau} ; \mathsf{extract}_{\tau} = \mathsf{id}_{\tau}$
- ($\{\!|\ |\!\} @$) and ($\rhd \rceil \lceil$) are both a reflection of the symmetry of the rules for supply and application, as well as commutativity of $\boxplus$ and Eval.
- ($\oslash @$) and ($\rhd \lightning$) rely on the same properties as ($\{\!|\ |\!\} @$) and ($\rhd \rceil \lceil$), but also on the definition of $\oslash$ and $\lightning$ at function types, which reflect their being defined "pointwise".
- ($\rhd v$) corresponds to β-reduction in λ-calculi, and relies on standard categorical and monadic properties.
- ($c \rhd c$) relies crucially on the fact that $c^{\mathsf{E}} ; \widetilde{c}^{\mathsf{E}} = \mathsf{return}^{\mathsf{M}}_{\mathsf{E}\ \llbracket \overline{\alpha} \rrbracket_{\mathsf{E}} \to \llbracket \tau \rrbracket}$, as well on $\mathsf{return}^{\mathsf{M}}_{a} ; \mathsf{extract}_{a} = \mathsf{return}^{\mathsf{E}}_{a}$, and on Λ being able to curry multiple variables.
- ($d \rhd c$) relies on $d^{\mathsf{E}} ; \widetilde{c}^{\mathsf{E}} = \mathsf{term}^{\mathcal{C}}_{\llbracket \overline{\beta} \rrbracket_{\mathsf{E}}} ; \mathsf{zero}^{\mathsf{M}}_{\llbracket \overline{\alpha} \rrbracket_{\mathsf{E}}}$ for $d \neq c$, and on propagation of $\mathsf{zero}^{\mathsf{M}}$ by strength.

6 Using Different Monad Instances

Depending on the choice of monads E and M, additional rules become sound. For deterministic functional programming, [Kahl 2004] proposes a rule that turns expression matchings into left-zeros for alternative, and so essentially prohibits backtracking (and non-deterministic choice):

$$\rceil e \lceil \mathbin{\boldsymbol{|}} m \xrightarrow[\mathsf{M}]{} \rceil e \lceil \qquad (\rceil \lceil \mathbin{\boldsymbol{|}})$$

For the case where an empty expression is matched against a constructor pattern, [Kahl 2004] offers two different right-hand sides:

- The first rule corresponds to interpreting the empty expression as equivalent to non-termination, as usual in Haskell:

$$\oslash \rhd c(p_1, \dots, p_n) \Mapsto m \xrightarrow[\mathsf{M}]{} \rceil \oslash \lceil \qquad (\oslash \rhd c \to \oslash)$$

- The second rule corresponds to interpreting the empty expression as propagating the exception of matching failure as in the approach proposed in [Erwig, Peyton Jones 2001], this rule "resurrects" that failure:

$$\oslash \rhd c(p_1, \dots, p_n) \Mapsto m \xrightarrow[\mathsf{M}]{} \lightning \qquad (\oslash \rhd c \to \lightning)$$

For each of these two variants of deterministic functional programming, we now show a corresponding bimonadic semantics, and then go on to explore more general monads.

6.1 Preliminaries: The Haskell Monad H

Haskell uses a non-strict cpo semantics where all objects have a least element $\mathsf{undefined}$, but morphisms need not preserve this least element.

Since we need access to that least element from our semantics, it makes sense factor this out and consider Haskell as based on an appropriate category of "potentially unboxed types". Dcpos, i.e., directed complete posets (which need not have a least element) form a cartesian closed category that is also closed under finite sums [Gunter 1985], and therefore serve our purpose nicely.

Types considered as expression types in Haskell do have the least element $\mathsf{undefined}$, and we consider this to be added by the "Haskell monad" H, which we therefore define to be the lifting monad, a special case of monad coproduct:

$$\mathsf{H} := (\_)^{\perp} = \mathsf{id} + \mathbb{1}$$

We denote the two natural coproduct injections as $\mathsf{return}^{\mathsf{H}}$ and bottom, so we have:

$$\begin{array}{rcl} \mathsf{return}^{\mathsf{H}}_{\alpha} & : & \alpha \to \mathsf{H}\ \alpha \\ \mathsf{bottom}_{\alpha} & : & \mathbb{1} \to \mathsf{H}\ \alpha \end{array}$$

6.2 Haskell

For standard Haskell semantics, we choose the above Haskell monad H as the expression monad $\mathsf{E} := \mathsf{H}$, and complete this to a bimonadic setting as in Setting 3.2.2 with $\mathsf{M} = \mathsf{E} + \mathbb{1}$, choosing $\mathsf{empty}^{\mathsf{E}} : \mathbb{1} \to \mathsf{E}$ as $\mathsf{empty}^{\mathsf{E}} = \mathsf{bottom}$, so that it maps failure to $\perp$. As described in Setting 3.2.2, lift is then the first monad coproduct injection, from E to M, and $\mathsf{zero}^{\mathsf{M}}$ is the second monad coproduct injection.

Since $\mathsf{E} = \mathsf{H} = \mathsf{id} + \mathbb{1}$, undefined computations propagate as a second left-zero through the matching monad, i.e., for any $F : \alpha \to \mathsf{M}\ \beta$ and $G : \alpha \to \beta$:

$$\begin{array}{rcl} (\mathsf{bottom}_{\alpha} ; \mathsf{lift}_{\alpha}) \odot_{\mathsf{M}} F & = & \mathsf{bottom}_{\beta} ; \mathsf{lift}_{\beta} \qquad (1) \\ \langle G, \mathsf{term}_{\alpha} ; \mathsf{bottom}_{\gamma} ; \mathsf{lift}_{\gamma} \rangle ; \mathsf{strengthL}^{\mathsf{M}}_{\beta,\gamma} & = & \mathsf{term}_{\alpha} ; \mathsf{bottom}_{\beta\times\gamma} ; \mathsf{lift}_{\beta\times\gamma} \qquad (2) \\ (\mathsf{bottom}_{\alpha} ; \mathsf{lift}_{\alpha}) \boxdot^{\mathsf{M}}_{\alpha} F & = & \mathsf{bottom}_{\beta} ; \mathsf{lift}_{\beta} \qquad (3) \end{array}$$

This corresponds to the approaches used by Tullsen [Tullsen 2000] and Harrison *et al.* [Harrison, Sheard+ 2002; Harrison, Kieburtz 2005] which all essentially employ the Maybe monad for this kind of purpose.

This setting also makes the rules $(\upharpoonleft\upharpoonright \pmb{|})$ and $(\oslash \triangleright c \to \oslash)$ sound (proofs are in Appendix D.4), which proves that $\mathsf{PMC}_{\oslash}$ as defined in [Kahl 2004] appropriately implements the semantics of Haskell.

6.3 Matching Failure as Exception

To achieve a semantics that is consistent with Erwig and Peyton Jones' proposal to treat matching failure as exception that can be caught by other matching alternatives [Erwig, Peyton Jones 2001], $\oslash$ needs to be a zero for the expression monad E, which we can chose as $\mathsf{E} = \mathsf{H} + \mathbb{1}$.

If we complete this via Setting 3.2.1 with $\mathsf{M} = \mathsf{E}$, then this equates the semantics of $\lightning$ and $\upharpoonleft \oslash \upharpoonright$, and makes rule $(\oslash \triangleright c \to \lightning)$ sound, see Appendix D.5 for the proof.

But this also makes the rule $(\upharpoonleft\upharpoonright \pmb{|})$, which corresponds to (deterministic) functional programming, unsound, since it introduces inconsistencies, e.g., with semantics-level equations:

$$\oslash = \{\!|\upharpoonleft \oslash \upharpoonright|\!\} = \{\!|\upharpoonleft \oslash \upharpoonright \pmb{|} \upharpoonleft 42 \upharpoonright|\!\} \overset{!}{=} \{\!|\lightning \pmb{|} \upharpoonleft 42 \upharpoonright|\!\} = \{\!|\upharpoonleft 42 \upharpoonright|\!\} = 42$$

So with this semantics, we cannot use the general rule $(\upharpoonleft\upharpoonright \pmb{|})$, but only restricted rules, e.g.:

$$\upharpoonleft c(e_q, \dots, e_n) \upharpoonright \pmb{|}\, m \quad \underset{\mathsf{M}}{\longrightarrow} \quad \upharpoonleft c(e_q, \dots, e_n) \upharpoonright \qquad (\upharpoonleft c \upharpoonright \pmb{|})$$

Note that the (mechanised) confluence proof described in [Kahl 2003; Kahl 2004] implies that $\lightning = \upharpoonleft \oslash \upharpoonright$ is *not* a consequence of $\mathsf{PMC}_{\lightning}$, which results from adding $(\upharpoonleft\upharpoonright \pmb{|})$ and $(\oslash \triangleright c \to \lightning)$ to the core rules. This shows that the semantics considered here is not fully abstract for $\mathsf{PMC}_{\lightning}$. On the other hand, the calculus resulting from using $(\upharpoonleft c \upharpoonright \pmb{|})$ instead of $(\upharpoonleft\upharpoonright \pmb{|})$ (and possibly adding also $\upharpoonleft \oslash \upharpoonright \underset{\mathsf{M}}{\longrightarrow} \lightning$) seems to be a more natural fit to the understanding behind Erwig and Peyton Jones' proposal [Erwig, Peyton Jones 2001], since it really makes sure that exceptions are "caught" in the closest available alternative.

6.4 Functional-Logic Programming

Lazy functional-logic programming (FLP) extends lazy functional programming with logic variables and non-deterministic choice and the ability that any expression evaluates to "multiple values". This kind of choice can be modelled for example using a list monad, a tree monad, or the LogicT monad of Kiselyov *et al.* [Kiselyov, Shan+ 2005].

By using this kind of monad both for M, where choice originates in PMC, and, following Setting 3.2.1 with $\mathsf{E} = \mathsf{M}$, also for the expression monad, to which it needs to propagate in FLP, we obtain an appropriate semantics for the fragment of FLP that can be expressed with the syntax of PMC as presented here. The pointwise extension behaviour of alternative in our semantics actually exactly corresponds to the way choice is treated in the functional-logic programming language Curry [Hanus 1997; Hanus+ 2006]. (To obtain the full expressive power of FLP, we need to extend the pattern syntax with the third alternative of call-by-value variables — the details are beyond the scope of the present paper.)

6.5 Choice

A particularly interesting situation arises when the list monad is chosen for M, but just partiality for E. Then we get all possible matches, yet we must then return only a single valid result. This can be very useful in some situations where we have either an intrinsic measurement of "better" choices, or where choice is inevitable but inessential. The same algorithm, Gaussian Elimination, can serve as an example of both of these situations. [Carette 2006] shows how for many different domains, there is an intrinsic notion of "better than" for the purposes of pivot choice. On the other hand, [Tucker, Zucker 2004] shows that either multi-valuedness or non-determinism are necessary ingredients even for single-valued functions (like Gaussian Elimination) if one wishes to be fully abstract, in other words representation-independent. Correspondingly, the "better than" notions of [Carette 2006] are generally representation-dependent. Having a convenient programming language where we can disentangle these issues would clearly be beneficial. We believe that this could allow versions of some numerical algorithms, in the style of [Carette, Kiselyov 2005], to be made even more generic.

7 Conclusion and Outlook

Using a monad, most typically Maybe, for the semantics of pattern matching in Haskell-like languages has been proposed previously [Tullsen 2000; Harrison, Sheard+ 2002; Harrison, Kieburtz 2005].

Since PMC offers a finer-grained, more systematic separation of pattern matching aspects from other expression evaluation aspects, choosing to interpret the two syntactic categories with separate monads is an obvious choice — the alternative of using a monad transformer deserves further exploration.

From this starting point, defining a general, monadic semantics for PMC required the resolution of two fine technical points:

- the necessity to use different definitions of the function type semantics for expressions and matchings, and
- the necessity to provide the corresponding "pointwise extensions" to the operations in the base monads.

As a result, the soundness of all the core reduction rules of the two PMC calculi defined in [Kahl 2004] is obtained assuming only remarkably light coupling of the two monads through the laws assumed for extract and lift.

The only common rule of the two calculi defined in [Kahl 2004] that is left out of the set of core rules is the rule ($\rceil\upharpoonright\Vert$) expressing that the first success of a matching will be its only result; this obviously would exclude monads with a non-determinism or backtracking component from being used for matching semantics. By not including such an assumption, we keep our bi-monadic PMC semantics open to uses also in functional-logic programming, which is one of the topics we plan to explore in more depth in the future, and we are extending our current prototype Haskell implementation of PMC reduction and of the semantics presented in the paper to serve as a test-bed for exploration in this direction.

It is also quite intriguing that by just taking two list monads, one gets "all" answers out of programs written as pure functions, if the patterns turned out to be overlapping. Generalising this further, to say tree monads, is definitely worth exploring.

Finally we would like to use the given semantics as justification for transformation rules that are useful for compilation of non-strict pattern matching.

References

[Barr, Wells 1999] M. Barr, C. Wells. *Category Theory for Computing Science.* Centre de recherches mathématiques (CRM), Université de Montréal, 3rd edition, 1999.

[Carette, Kiselyov 2005] J. Carette, O. Kiselyov. *Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code.* In R. Glück, M. R. Lowry, eds., Generative Programming and Component Engineering, GPCE 2005, LNCS **3676**, pp. 256–274. Springer, 2005.

[Carette 2006] J. Carette. *Gaussian Elimination: a Case Study in Efficient Genericity with MetaOCaml.* Science of Computer Programming , 2006. accepted.

[Erwig, Peyton Jones 2001] M. Erwig, S. Peyton Jones. *Pattern Guards and Transformational Patterns.* In: Proc. Haskell Workshop 2000, Montreal, Electronic Notes in Computer Science **41 no. 1**, pp. 12.1–12.27, 2001.

[Gunter 1985] C. A. Gunter. *Comparing Categories of Domains.* In A. Melton, ed., Mathematical Foundations of Programming Semantics, MFPS 1985, LNCS **239**, pp. 101–121, 1985.

[Hanus 1997] M. Hanus. *A Unified Computation Model for Declarative Programming.* In: Proc. of the 1997 Joint Conference on Declarative Programming (APPIA-GULP-PRODE'97), pp. 9–24, 1997.

[Hanus+ 2006] M. Hanus et al. *Curry — An Integrated Functional Logic Language, Version 0.8.2,* 2006. http://www.informatik.uni-kiel.de/~curry/report.html.

[Harrison, Sheard+ 2002] W. L. Harrison, T. Sheard, J. Hook. *Fine Control of Demand in Haskell.* In: Mathematics of Program Construction, MPC 2002, LNCS. Springer, 2002.

[Harrison, Kieburtz 2005] W. L. Harrison, R. B. Kieburtz. *The logic of demand in Haskell.* J. Functional Programming **15** 837–891, 2005.

[Jung, Fiore+ 1996] A. Jung, M. Fiore, E. Moggi, P. O'Hearn, J. Riecke, G. Rosolini, I. Stark. *Domains and Denotational Semantics: History, Accomplishments and Open Problems,* 1996.

[Kahl 2003] W. Kahl. *Basic Pattern Matching Calculi: Syntax, Reduction, Confluence, and Normalisation.* SQRL Report 16, Software Quality Research Laboratory, McMaster Univ., 2003. available from http://sqrl.mcmaster.ca/sqrl_reports.html.

[Kahl 2004] W. Kahl. *Basic Pattern Matching Calculi: A Fresh View on Matching Failure.* In Y. Kameyama, P. Stuckey, eds., Functional and Logic Programming, Proceedings of FLOPS 2004, LNCS **2998**, pp. 276–290. Springer, 2004.

[Kahl, Carette+ 2006] W. Kahl, J. Carette, X. Ji. *Bimonadic Semantics for Basic Pattern Matching Calculi.* In T. Uustalu, ed., Mathematics of Program Construction, MPC 2006, LNCS **4014**, pp. 253–273. Springer, 2006.

[Kiselyov, Shan+ 2005] O. Kiselyov, C.-c. Shan, D. P. Friedman, A. Sabry. *Backtracking, Interleaving, and Terminating Monad Transformers.* In: ICFP 2005, Intl. Conf. on Functional Programming, ACM Sigplan Notices **40**(9), pp. 192–203. ACM, 2005.

[Lüth, Ghani 2002] C. Lüth, N. Ghani. *Composing Monads Using Coproducts.* In: ICFP 2002, Intl. Conf. on Functional Programming, ACM Sigplan Notices **37**(9), pp. 133–144. ACM, 2002.

[Moggi 1991a] E. Moggi. *A Modular Approach to Denotational Semantics.* In D. H. Pitt et al., eds., Category Theory and Computer Science, LNCS **530**, pp. 138–139. Springer, 1991.

[Moggi 1991b] E. Moggi. *Notions of Computation and Monads.* Information and Computation **93** 55–92, 1991.

[Peyton Jones+ 2003] S. Peyton Jones et al. *The Revised Haskell 98 Report.* Cambridge Univ. Press, 2003. Also on http://haskell.org/.

[Plasmeijer, van Eekelen 1993] R. Plasmeijer, M. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* International Computer Science Series. Addison-Wesley, 1993.

[Tucker, Zucker 2004] J. V. Tucker, J. I. Zucker. *Abstract Versus Concrete Computation on Metric Partial Algebras.* ACM Trans. Comput. Logic **5** 611–668, 2004.

[Tullsen 2000] M. Tullsen. *First Class Patterns.* In E. Pontelli, V. Santos Costa, eds., PADL 2000, LNCS **1753**, pp. 1–15. Springer, 2000.

A Monad Laws

A.1 Categories

A *category* consists of a class of object, a class of morphisms, a partial composition operation on morphisms, and an operation id assigning each object an *identity* morphism.

Each morphism has exactly one *source* and *target* object; we write $f : a \to b$ for a morphism from a to b.

Composition of morphisms $f : a \to b$ and $g : b' \to c$ is defined iff $b = b'$, and then $(f \mathbin{;} g) : a \to c$; composition is associative.

id produces identities for composition: $\mathsf{id}_a \mathbin{;} f = f = f \mathbin{;} \mathsf{id}_b$.

A.2 Endofunctors

An endofunctor F maps objects to objects and morphisms to morphisms and preserves composition and identities; if $f : a \to b$, then $F\ f : F\ a \to F\ b$.

A natural transformation t from functor F to functor G is a "polymorphic morphism" $t_a : F\ a \to G\ a$ satisfying the *naturality* condition $t_a \mathbin{;} G\ f = F\ f \mathbin{;} t_b$.

A.3 Monads

A *monad* is a triple $(M, \mathsf{return}^M, \mathsf{join}^M)$ consisting of an endofunctor M together with two natural transformations

$$\mathsf{return}^M_a \quad : \quad a \to M\ a \qquad\qquad \mathsf{join}^M_a \quad : \quad M\ (M\ a) \to M\ a$$

satisfying the following additional laws:

$$\begin{aligned} \mathsf{join}^M_a \mathbin{;} \mathsf{return}^M_a &= \mathsf{id}_{M\ a} \\ M\ \mathsf{return}^M_a \mathbin{;} \mathsf{join}^M_a &= \mathsf{id}_{M\ a} \end{aligned} \qquad\qquad M\ \mathsf{join}^M_a \mathbin{;} \mathsf{join}^M_a = \mathsf{join}^M_{M\ a} \mathbin{;} \mathsf{join}^M_a$$

A.4 Strong Monads

Assuming a cartesian category with terminal object $\mathbb{1}$, there are natural isomorphisms

$$\begin{aligned} r_a &\ :\ (\mathbb{1} \times a) \to a \\ \mathsf{assoc}_{a,b,c} &\ :\ (a \times b) \times c \to a \times (b \times c) \end{aligned}$$

A strong monad M is a monad in a cartesian category that has a natural transformation

$$\mathsf{strengthL}^M_{a,b} : a \times M\ b \to M\ (a \times b)$$

satisfying (see [Moggi 1991b] for diagrams and more explanation):

$$\begin{aligned} r_{M\ a} &= \mathsf{strengthL}^M_{\mathbb{1},a} \mathbin{;} M\ r_a \\ \mathsf{strengthL}^M_{a\times b,c} \mathbin{;} M\ \mathsf{assoc}_{a,b,c} &= \mathsf{assoc}_{a,b,M\ c} \mathbin{;} (\mathsf{id}_a \times \mathsf{strengthL}^M_{b,c}) \mathbin{;} \mathsf{strengthL}^M_{a,b\times c} \\ \mathsf{return}^M_{a\times b} &= (\mathsf{id}_a \times \mathsf{return}^M_b) \mathbin{;} \mathsf{strengthL}^M_{a,b} \\ (\mathsf{id}_a \times \mathsf{join}^M_b) \mathbin{;} \mathsf{strengthL}^M_{a,b} &= \mathsf{strengthL}^M_{a,M\ b} \mathbin{;} M\ \mathsf{strengthL}^M_{a,b} \mathbin{;} \mathsf{join}^M_{a\times b} \end{aligned}$$

Using the isomorphism ($\mathsf{swap}_{a,b}$ from $a \times b$ to $b \times a$), we can define the "swapped version"

$$\begin{aligned} \mathsf{strengthR}^M_{a,b} &\ :\ M\ a \times b \to M\ (a \times b) \\ \mathsf{strengthR}^M_{a,b} &\ =\ \mathsf{swap}_{M\ a,M\ b} \mathbin{;} \mathsf{strengthL}^M_{b,a} \mathbin{;} M\ (\mathsf{swap}_{M\ b,a}) \end{aligned}$$

This allows us to define $\otimes_{\mathsf{M}} : (M\ a \times M\ b) \to M(a \times b)$ via

$$\otimes_{\mathsf{M}} = \mathsf{strengthR}^M_{a,b} \odot_{\mathsf{M}} \mathsf{strengthL}^M_{a,b}$$

Notice that this chooses to "execute the first computation first" — this is in general different from proceeding the other way round.

This can be folded over ordered tuples to give

$$\begin{aligned} \bigotimes &\ :\ (M\ a_1 \times \cdots \times M\ a_n) \to M\ (a_1 \times \cdots \times a_n) \\ \bigotimes &\ =\ (\otimes \times \mathsf{id}_{(M\ a_3 \times \cdots \times M\ a_n)}) \mathbin{;} \cdots \mathbin{;} \otimes \end{aligned}$$

B PMC Core Reduction Rules

Here, we repeat from [Kahl 2004] the set of rules that implement the usual pattern matching semantics of non-strict functional programming languages by allowing corresponding reduction of PMC expressions as they arise from translating functional programs. In particular, we do not include extensionality rules.

Formally, we define two *redex reduction* relations:

- for expressions, $\underset{\mathsf{E}}{\longrightarrow} : \mathsf{Expr} \leftrightarrow \mathsf{Expr}$, and
- for matchings, $\underset{\mathsf{M}}{\longrightarrow} : \mathsf{Match} \leftrightarrow \mathsf{Match}$.

These are the smallest relations including the rules listed below and ($\upharpoonleft\upharpoonright\mathbin{\|}$) and either ($\oslash \triangleright c \to \oslash$) or ($\oslash \triangleright c \to \lightning$) (mentioned in Sect. 6). The resulting rewriting system contains a mix of first-order rules, rule schemata, and second-order rules; the first author described a direct confluence proof mechanised in Isabelle and a deterministic normalising strategy (via reduction to strong head normal form) in [Kahl 2004]. (That proof has since also be performed for the core rule set.)

B.1 Failure and Returning

Failure is the (left) unit for $\mathbin{\|}$:

$$\lightning \mathbin{\|} m \quad \underset{\mathsf{M}}{\longrightarrow} \quad m \qquad\qquad (\lightning\mathbin{\|})$$

A matching abstraction where all alternatives fail can be understood as representing an ill-defined case — this is reduced to the "empty expression":

$$\{\!|\, \lightning \,|\!\} \quad \underset{\mathsf{E}}{\longrightarrow} \quad \oslash \qquad\qquad (\{\!|\lightning|\!\})$$

Matching abstractions built from expression matchings are equivalent to the contained expression:

$$\{\!|\, \upharpoonleft e \upharpoonright \,|\!\} \quad \underset{\mathsf{E}}{\longrightarrow} \quad e \qquad\qquad (\{\!|\upharpoonleft\upharpoonright|\!\})$$

B.2 Application and Argument Supply

Application of a matching abstraction reduces to argument supply inside the abstraction:

$$\{\!|\, m \,|\!\}\ a \quad \xrightarrow[\mathsf{E}]{} \quad \{\!|\, a \rhd m \,|\!\} \qquad (\{\!|\,|\!\}@)$$

Argument supply to an expression matching reduces to function application inside the expression matching:

$$a \rhd \upharpoonleft e \upharpoonright \quad \xrightarrow[\mathsf{M}]{} \quad \upharpoonleft e\ a \upharpoonright \qquad (\rhd\upharpoonleft\upharpoonright)$$

No matter which of our two interpretations of the empty expression we choose, it absorbs arguments when used as function in an application:

$$\oslash\ e \quad \xrightarrow[\mathsf{E}]{} \quad \oslash \qquad (\oslash @)$$

Analogously, failure absorbs argument supply:

$$e \rhd ⇯ \quad \xrightarrow[\mathsf{M}]{} \quad ⇯ \qquad (\rhd ⇯)$$

Argument supply distributes into alternatives:

$$e \rhd (m_1 \mathbin{\pmb{|}} m_2) \quad \xrightarrow[\mathsf{M}]{} \quad (e \rhd m_1) \mathbin{\pmb{|}} (e \rhd m_2) \qquad (\rhd \pmb{|})$$

B.3 Pattern Matching

Everything matches a variable pattern; this matching gives rise to substitution:

$$a \rhd v \mapsto m \quad \xrightarrow[\mathsf{M}]{} \quad m[v \backslash a] \qquad (\rhd v)$$

Matching constructors match, and the proviso in the following rule can always be ensured via α-conversion (for this rule to make sense, linearity of patterns is important):

$$c(e_1, \dots, e_n) \rhd c(p_1, \dots, p_n) \mapsto m \quad \xrightarrow[\mathsf{M}]{} \quad e_1 \rhd p_1 \mapsto \cdots e_n \rhd p_n \mapsto m$$
$$\text{if } \mathsf{FV}(c(e_1, \dots, e_n)) \cap \mathsf{FV}(c(p_1, \dots, p_n)) = \{\} \qquad (c \rhd c)$$

Matching of different constructors fails:

$$d(e_1, \dots, e_k) \rhd c(p_1, \dots, p_n) \mapsto m \quad \xrightarrow[\mathsf{M}]{} \quad ⇯ \qquad \text{if } c \neq d \text{ or } k \neq n \qquad (d \rhd c)$$

C Combinator Lemmas

Lemma C.1 For $g : r \to [\![\alpha]\!]^{\mathsf{M}}$, we have: $\mathsf{return}^{\mathsf{K}}_r \boxdot^{\mathsf{K}}_\alpha g = g$

Proof: For constructed types, we have: $\mathsf{return}^{\mathsf{K}}_r \boxdot^{\mathsf{K}}_\tau g = \mathsf{return}^{\mathsf{K}}_r \odot_{\mathsf{M}} g = g$.

For function types:

$$
\begin{array}{lll}
& \mathsf{return}^{\mathsf{K}}_r \boxdot^{\mathsf{M}}_{\beta\to\gamma} g & \\
= & \Lambda\left(((\mathsf{return}^{\mathsf{K}}_r \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{strengthR}^{\mathsf{K}}_{r,[\![\beta]\!]^{\mathsf{E}}}) \boxdot^{\mathsf{K}}_\gamma ((g \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma})\right) & \text{def. } \boxdot_{\beta\to\gamma} \\
= & \Lambda\left(\mathsf{return}^{\mathsf{K}}_{r\times\mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}}} \boxdot^{\mathsf{K}}_\gamma ((g \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma})\right) & \text{strength preserves } \mathsf{return} \\
= & \Lambda\left(((g \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma})\right) & \text{induction hyp.} \\
= & g & \Lambda \text{ is nat. bijection} \quad \square
\end{array}
$$

Lemma C.2 For $f : p \to q$ and $g : q \to \mathsf{K}\ r$ and $h : r \to [\![\alpha]\!]^{\mathsf{M}}$, we have $(f;g) \boxdot^{\mathsf{K}}_\alpha h = f;(g \boxdot^{\mathsf{K}}_\alpha h)$.

Proof: For constructed types: $(f;g) \boxdot^{\mathsf{K}}_\tau h = (f;g) \odot_{\mathsf{M}} h = f;(g \odot_{\mathsf{M}} h) = f;(g \boxdot^{\mathsf{K}}_\tau h)$

For function types:

$$
\begin{array}{lll}
& (f;g) \boxdot^{\mathsf{M}}_{\beta\to\gamma} h & \\
= & \Lambda\left(((f;g \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{strengthR}^{\mathsf{K}}_{r,[\![\beta]\!]^{\mathsf{E}}}) \boxdot^{\mathsf{K}}_\gamma ((h \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma})\right) & \text{def. } \boxdot_{\beta\to\gamma} \\
= & \Lambda\left(((f \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});(g \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{strengthR}^{\mathsf{K}}_{r,[\![\beta]\!]^{\mathsf{E}}}) \boxdot^{\mathsf{K}}_\gamma ((h \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma})\right) & \text{functoriality of } \times \\
= & f;\Lambda\left(((g \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{strengthR}^{\mathsf{K}}_{r,[\![\beta]\!]^{\mathsf{E}}}) \boxdot^{\mathsf{K}}_\gamma ((h \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma})\right) & \text{naturality of } \Lambda \\
= & f;(g \boxdot^{\mathsf{K}}_{\beta\to\gamma} h) & \text{def. } \boxdot_{\beta\to\gamma} \quad \square
\end{array}
$$

Lemma C.3 $f;[_r\ g \boxed{;}^{\alpha}\ t\ ] = [_q\ f;g \boxed{;}^{\alpha}\ t\ ]$ for $f : q \to r$.

Proof: By induction over the number of argument types in α.

Base case (constructed types): using the definition of $[\ \boxed{;}^{\tau}\ ]$: $f;[_r\ g \boxed{;}^{\tau}\ t\ ] = f;g;t_{[\![\tau]\!]} = [_q\ f;g \boxed{;}^{\tau}\ t\ ]$.

Induction step: Assume $\alpha = \beta \to \gamma$:

$$
\begin{array}{lll}
& f;[_r\ g \boxed{;}^{\beta\to\gamma}\ t\ ] & \\
= & f;\Lambda[_{r\times[\![\beta]\!]^{\mathsf{E}}}\ (g \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma} \boxed{;}^{\gamma}\ t\ ] & \text{definition of } [\ \boxed{;}^{\beta\to\gamma}\ ] \\
= & \Lambda\left((f \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});[_{q\times[\![\beta]\!]^{\mathsf{E}}}\ (g \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma} \boxed{;}^{\gamma}\ t\ ]\right) & \text{naturality of } \Lambda \\
= & \Lambda[_{q\times[\![\beta]\!]^{\mathsf{E}}}\ (f \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});(g \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma} \boxed{;}^{\gamma}\ t\ ] & \text{induction hypothesis} \\
= & \Lambda[_{q\times[\![\beta]\!]^{\mathsf{E}}}\ ((f;g) \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma} \boxed{;}^{\gamma}\ t\ ] & \text{functoriality of } \times \\
= & [_q\ f;g \boxed{;}^{\beta\to\gamma}\ t\ ] & \text{definition of } [\ \boxed{;}^{\beta\to\gamma}\ ] \quad \square
\end{array}
$$

Lemma C.4 Assume $f : q \to [\![\beta \to \gamma]\!]^{\mathsf{K}}$ and $g : r \to [\![\beta]\!]^{\mathsf{E}}$, and let t be a transformation from K to H. Then:

$$([_q\ f \boxed{;}^{\beta\to\gamma}\ t\ ] \times g);\mathsf{Eval}^{\mathsf{H}}_{\beta,\gamma} = [_{q\times r}\ (f \times g);\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma} \boxed{;}^{\gamma}\ t\ ]$$

Proof:

$$
\begin{array}{lll}
& ([_q\ f \boxed{;}^{\beta\to\gamma}\ t\ ] \times g);\mathsf{Eval}^{\mathsf{H}}_{\beta,\gamma} & \\
= & (\Lambda[_{q\times[\![\beta]\!]^{\mathsf{E}}}\ (f \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma} \boxed{;}^{\gamma}\ t\ ] \times g);\mathsf{Eval}^{\mathsf{H}}_{\beta,\gamma} & \text{def. of } [\ \boxed{;}^{\beta\to\gamma}\ ] \\
= & (\mathsf{id}_q \times g);(\Lambda[_{q\times[\![\beta]\!]^{\mathsf{E}}}\ (f \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma} \boxed{;}^{\gamma}\ t\ ] \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{Eval}^{\mathsf{H}}_{\beta,\gamma} & \text{properties of } \times \\
= & (\mathsf{id}_q \times g);[_{q\times[\![\beta]\!]^{\mathsf{E}}}\ (f \times \mathsf{id}_{[\![\beta]\!]^{\mathsf{E}}});\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma} \boxed{;}^{\gamma}\ t\ ] & \text{definition of } \mathsf{Eval} \\
= & [_{q\times r}\ (f \times g);\mathsf{Eval}^{\mathsf{K}}_{\beta,\gamma} \boxed{;}^{\gamma}\ t\ ] & \text{Lemma C.3} \quad \square
\end{array}
$$

Lemma C.5 $f;(g_1 \boxplus_\alpha g_2) = f;g_1 \boxplus_\alpha f;g_2$

Proof: By induction over the number of argument types in α.

Base case: If α is a constructed type τ, the definition of $\boxplus_\tau$ and properties of $\times$ immediately give us:

$$f;(g_1 \boxplus_\tau g_2) = f;\langle g_1, g_2\rangle;\mathsf{plus}^{\mathsf{M}} = \langle f;g_1, f;g_2\rangle;\mathsf{plus}^{\mathsf{M}} = f;g_1 \boxplus_\tau f;g_2$$

Induction step: Assume $\alpha = \beta \to \gamma$:

$$\begin{aligned}
& f;(g_1 \boxplus_{\beta\to\gamma} g_2) \\
={}& f;\Lambda\left((g_1 \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma} \boxplus_\gamma (g_2 \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma}\right) && \text{definition of } \boxplus_{\beta\to\gamma} \\
={}& \Lambda((f \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\left((g_1 \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma} \boxplus_\gamma (g_2 \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma}\right)) && \text{naturality of } \Lambda \\
={}& \Lambda\left((f \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});(g_1 \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma} \boxplus_\gamma (f \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});(g_2 \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma}\right) && \text{induction hypothesis} \\
={}& \Lambda\left((f;g_1 \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma} \boxplus_\gamma (f;g_2 \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma}\right) && \text{functoriality of } \times \\
={}& (f;g_1) \boxplus_{\beta\to\gamma} (f;g_2) && \text{definition of } \boxplus_{\beta\to\gamma} \quad \square
\end{aligned}$$

Lemma C.6 If $f : q \to [\![\alpha]\!]^\mathsf{K}$, then $[_q\ f \boxed{;}^\alpha\ \mathsf{id}\ ] = f$.

Proof: By induction over the number of argument types in α.

Base case: If α is a constructed type τ, the definition of $[\ \boxed{;}^\tau\ ]$ immediately gives us:

$$[_q\ f \boxed{;}^\tau\ \mathsf{id}\ ] = f;\mathsf{id}_{[\![\tau]\!]} = f$$

Induction step: Assume $\alpha = \beta \to \gamma$:

$$\begin{aligned}
[_q\ f \boxed{;}^{\beta\to\gamma}\ \mathsf{id}\ ] ={}& \Lambda[_{q\times[\![\beta]\!]^\mathsf{E}}\ (f \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{K}_{\beta,\gamma} \boxed{;}^\gamma\ \mathsf{id}\ ] && \text{definition of } [\ \boxed{;}^{\beta\to\gamma}\ ] \\
={}& \Lambda((f \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{K}_{\beta,\gamma}) && \text{induction hypothesis} \\
={}& \Lambda((f \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{eval}_{[[\![\beta]\!]^\mathsf{E} \to [\![\gamma]\!]^\mathsf{K}]}) && \text{definition of } \mathsf{Eval}, \\
={}& f && \Lambda \text{ is nat. bijection} \quad \square
\end{aligned}$$

Lemma C.7 Assume two transformations t with $t_a : \mathsf{K}\ a \to \mathsf{H}\ a$ and u with $u_a : \mathsf{H}\ a \to \mathsf{G}\ a$.

$$[_q\ [_q\ f \boxed{;}^\alpha\ t\ ] \boxed{;}^\alpha\ u\ ] = [_q\ f \boxed{;}^\alpha\ t;u\ ]$$

Proof: By induction over the number of argument types in α.

Base case: assume α is a constructed type τ:

$$\begin{aligned}
[_q\ [_q\ f \boxed{;}^\tau\ t\ ] \boxed{;}^\tau\ u\ ] ={}& [_q\ f;t_{[\![\tau]\!]} \boxed{;}^\tau\ u\ ] && \text{definition of } [\ \boxed{;}^\tau\ ] \\
={}& f;t_{[\![\tau]\!]};u_{[\![\tau]\!]} && \text{definition of } [\ \boxed{;}^\tau\ ] \\
={}& f;(t;u)_{[\![\tau]\!]} && \text{composition of transformations} \\
={}& [_q\ f \boxed{;}^\tau\ t;u\ ] && \text{definition of } [\ \boxed{;}^\tau\ ]
\end{aligned}$$

Induction step: Assume $\alpha = \beta \to \gamma$:

$$\begin{aligned}
& [_q\ [_q\ f \boxed{;}^{\beta\to\gamma}\ t\ ] \boxed{;}^{\beta\to\gamma}\ u\ ] \\
={}& \Lambda[_{q\times[\![\beta]\!]^\mathsf{E}}\ ([_q\ f \boxed{;}^{\beta\to\gamma}\ t\ ] \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{H}_{\beta,\gamma} \boxed{;}^\gamma\ u\ ] && \text{definition of } [\ \boxed{;}^{\beta\to\gamma}\ ] \\
={}& \Lambda[_{q\times[\![\beta]\!]^\mathsf{E}}\ [_{q\times[\![\beta]\!]^\mathsf{E}}\ (f \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{K}_{\beta,\gamma} \boxed{;}^\gamma\ t\ ] \boxed{;}^\gamma\ u\ ] && \text{Lemma C.4} \\
={}& \Lambda[_{q\times[\![\beta]\!]^\mathsf{E}}\ (f \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{K}_{\beta,\gamma} \boxed{;}^\gamma\ t;u\ ] && \text{induction hypothesis} \\
={}& [_q\ f \boxed{;}^{\beta\to\gamma}\ (t;u)\ ] && \text{definition of } [\ \boxed{;}^{\beta\to\gamma}\ ] \quad \square
\end{aligned}$$

Lemma C.8 For $m : r \to [\![\alpha]\!]^\mathsf{M}$, we have $\mathsf{term}^\mathcal{C}_q;\mathsf{zero}^\mathsf{M}_r \boxdot^\mathsf{M}_\alpha m = [_q\ \mathsf{term}^\mathcal{C}_q \boxed{;}^\alpha\ \mathsf{zero}^\mathsf{M}\ ]$.

Proof: By definition of $[\ \boxed{;}\ ]$, we have for constructed types:

$$\mathsf{term}^\mathcal{C}_q;\mathsf{zero}^\mathsf{M}_r \boxdot^\mathsf{M}_\tau m = \mathsf{term}^\mathcal{C}_q;\mathsf{zero}^\mathsf{M}_r \odot_\mathsf{M} m = \mathsf{term}^\mathcal{C}_q;\mathsf{zero}^\mathsf{M}_{[\![\tau]\!]_\mathsf{M}} = [_q\ \mathsf{term}^\mathcal{C}_q \boxed{;}^\tau\ \mathsf{zero}^\mathsf{M}\ ]$$

For function types:

$$\begin{aligned}
& \mathsf{term}^\mathcal{C}_q;\mathsf{zero}^\mathsf{M}_r \boxdot^\mathsf{M}_{\beta\to\gamma} m \\
={}& \Lambda\left(((\mathsf{term}^\mathcal{C}_q;\mathsf{zero}^\mathsf{M}_r \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{strengthR}^\mathsf{K}_{r,[\![\beta]\!]^\mathsf{E}}) \boxdot^\mathsf{K}_\gamma ((m \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{K}_{\beta,\gamma})\right) && \text{def. } \boxdot_{\beta\to\gamma} \\
={}& \Lambda\left((\mathsf{term}^\mathcal{C}_{q\times[\![\beta]\!]^\mathsf{E}};\mathsf{zero}^\mathsf{M}_{r\times[\![\beta]\!]^\mathsf{E}}) \boxdot^\mathsf{K}_\gamma ((m \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{K}_{\beta,\gamma})\right) && \text{strength preserves } \mathsf{zero}^\mathsf{M} \\
={}& \Lambda[_{q\times[\![\beta]\!]^\mathsf{E}}\ \mathsf{term}^\mathcal{C}_{q\times[\![\beta]\!]^\mathsf{E}} \boxed{;}^\gamma\ \mathsf{zero}^\mathsf{M}\ ] && \text{induction hyp.} \\
={}& \Lambda[_{q\times[\![\beta]\!]^\mathsf{E}}\ (\mathsf{term}^\mathcal{C}_q \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^{\mathbb{1}}_{\beta,\gamma} \boxed{;}^\gamma\ \mathsf{zero}^\mathsf{M}\ ] && \text{terminality} \\
={}& [_q\ \mathsf{term}^\mathcal{C}_q \boxed{;}^{\beta\to\gamma}\ \mathsf{zero}^\mathsf{M}\ ] && \text{def. } [\ \boxed{;}^{\beta\to\gamma}\ ] \quad \square
\end{aligned}$$

Lemma C.9 $[_q\ \mathsf{term}^\mathcal{C}_q \boxed{;}^\alpha\ \mathsf{zero}^\mathsf{M}\ ] \boxplus_\alpha g \quad = \quad g$

Proof: The base case follows directly from zero^M being a unit for plus:

$$[_q\ \mathsf{term}^\mathcal{C}_q \boxed{;}^\tau\ \mathsf{zero}^\mathsf{M}\ ] \boxplus_\tau g = \langle \mathsf{term}^\mathcal{C}_q;\mathsf{zero}^\mathsf{M}_{[\![\tau]\!]}, g\rangle;\mathsf{plus}^\mathsf{M} = g$$

Induction step:

$$\begin{aligned}
& [_q\ \mathsf{term}^\mathcal{C}_q \boxed{;}^{\beta\to\gamma}\ \mathsf{zero}^\mathsf{M}\ ] \boxplus_{\beta\to\gamma} g \\
={}& \Lambda\left(([_q\ \mathsf{term}^\mathcal{C}_q \boxed{;}^{\beta\to\gamma}\ \mathsf{zero}^\mathsf{M}\ ] \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma} \boxplus_\gamma (g \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma}\right) && \text{definition of } \boxplus_{\beta\to\gamma} \\
={}& \Lambda\left([_{q\times[\![\beta]\!]^\mathsf{E}}\ (\mathsf{term}^\mathcal{C}_q \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^{\mathbb{1}}_{\beta,\gamma} \boxed{;}^\gamma\ \mathsf{zero}^\mathsf{M}\ ] \boxplus_\gamma (g \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma}\right) && \text{Lemma C.4} \\
={}& \Lambda\left([_{q\times[\![\beta]\!]^\mathsf{E}}\ \mathsf{term}^\mathcal{C}_{q\times[\![\beta]\!]^\mathsf{E}} \boxed{;}^\gamma\ \mathsf{zero}^\mathsf{M}\ ] \boxplus_\gamma (g \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma}\right) && \text{terminality} \\
={}& \Lambda\left((g \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma}\right) && \text{induction hypothesis} \\
={}& g && \Lambda \text{ is nat. bijection} \quad \square
\end{aligned}$$

Lemma C.10 Assume $g_1, g_2 : q_1 \to [\![\beta \to \gamma]\!]^\mathsf{M}$ and $h : q_2 \to [\![\beta]\!]^\mathsf{E}$. Then:

$$((g_1 \boxplus_{\beta\to\gamma} g_2) \times h);\mathsf{Eval}^\mathsf{H}_{\beta,\gamma} \quad = \quad ((g_1 \times h);\mathsf{Eval}^\mathsf{H}_{\beta,\gamma}) \boxplus_\gamma ((g_2 \times h);\mathsf{Eval}^\mathsf{H}_{\beta,\gamma})$$

Proof:

$$\begin{aligned}
& ((g_1 \boxplus_{\beta\to\gamma} g_2) \times h);\mathsf{Eval}^\mathsf{H}_{\beta,\gamma} \\
={}& (\Lambda\left((g_1 \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma} \boxplus_\gamma (g_2 \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma}\right) \times h);\mathsf{Eval}^\mathsf{H}_{\beta,\gamma} && \text{definition of } \boxplus_{\beta\to\gamma} \\
={}& (\mathsf{id}_{q_1} \times h);\left((g_1 \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma} \boxplus_\gamma (g_2 \times \mathsf{id}_{[\![\beta]\!]^\mathsf{E}});\mathsf{Eval}^\mathsf{M}_{\beta,\gamma}\right) && \text{props. of } \times, \mathsf{Eval} \\
={}& \left((g_1 \times h);\mathsf{Eval}^\mathsf{M}_{\beta,\gamma} \boxplus_\gamma (g_2 \times h);\mathsf{Eval}^\mathsf{M}_{\beta,\gamma}\right) && \text{Lemma C.5} \quad \square
\end{aligned}$$

D Correctness of the Core Reduction Rules

For showing the correctness of the core reduction rules, we rely heavily on the lemmas shown in appendix C; they allow us to perform the proofs as direct calculation at the level of the combinators used in the semantics definitions.

D.1 Failure and Returning

Rule $(\{\!|\lightning|\!\})$:

$$\begin{aligned}
\llbracket \{\!| \lightning |\!\} \rrbracket^{\mathsf{E}}_{\mathcal{V},\alpha} &= [_{\mathcal{V}^{\mathsf{E}}}\ \llbracket \lightning \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha}\ \boxed{;}^{\alpha}\ \mathsf{extract}\ ] && \text{semantics of } \{\!|\ |\!\} \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ [_{\mathcal{V}^{\mathsf{E}}}\ \mathsf{term}^{\mathcal{C}}_{\mathcal{V}^{\mathsf{E}}}\ \boxed{;}^{\alpha}\ \mathsf{zero}^{\mathsf{M}}\ ]\ \boxed{;}^{\alpha}\ \mathsf{extract}\ ] && \text{semantics of } \lightning \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ \mathsf{term}^{\mathcal{C}}_{\mathcal{V}^{\mathsf{E}}}\ \boxed{;}^{\alpha}\ \mathsf{zero}^{\mathsf{M}} \mathbin{;} \mathsf{extract}\ ] && \text{Lemma C.7} \\
&= \llbracket \oslash \rrbracket^{\mathsf{E}}_{\mathcal{V},\alpha} && \text{semantics of } \oslash
\end{aligned}$$

Rule $(\lightning\,|)$:

$$\begin{aligned}
\llbracket \lightning \mid m \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} &= \llbracket \lightning \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} \boxplus_{\alpha} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} && \text{semantics of } | \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ \mathsf{term}^{\mathcal{C}}_{\mathcal{V}^{\mathsf{E}}}\ \boxed{;}^{\alpha}\ \mathsf{zero}^{\mathsf{M}}\ ] \boxplus_{\alpha} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} && \text{semantics of } \lightning \\
&= \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} && \text{Lemma C.9}
\end{aligned}$$

Rule $(\{\!|\upharpoonleft\upharpoonright|\!\})$:

$$\begin{aligned}
\llbracket \{\!| \upharpoonleft e \upharpoonright |\!\} \rrbracket^{\mathsf{E}}_{\mathcal{V},\alpha} &= [_{\mathcal{V}^{\mathsf{E}}}\ \llbracket \upharpoonleft e \upharpoonright \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha}\ \boxed{;}^{\alpha}\ \mathsf{extract}\ ] && \text{semantics of } \{\!|\ |\!\} \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ [_{\mathcal{V}^{\mathsf{E}}}\ \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V}}\ \boxed{;}^{\alpha}\ \mathsf{lift}\ ]\ \boxed{;}^{\alpha}\ \mathsf{extract}\ ] && \text{semantics of } \upharpoonleft\ \upharpoonright \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V}}\ \boxed{;}^{\alpha}\ \mathsf{lift} \mathbin{;} \mathsf{extract}\ ] && \text{Lemma C.7} \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V}}\ \boxed{;}^{\alpha}\ \mathsf{id}\ ] && (\mathsf{lift} \mathbin{;} \mathsf{extract}) \\
&= \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\alpha} && \text{Lemma C.6}
\end{aligned}$$

D.2 Application and Argument Supply

Rule $(\{\!|\ |\!\}@)$:

$$\begin{aligned}
\llbracket \{\!| m |\!\}\ a \rrbracket^{\mathsf{E}}_{\mathcal{V},\gamma} &= \langle \llbracket \{\!| m |\!\} \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta\to\gamma}, \llbracket a \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{E}}_{\beta,\gamma} && \text{semantics of appl.} \\
&= \langle [_{\mathcal{V}^{\mathsf{E}}}\ \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V},\beta\to\gamma}\ \boxed{;}^{\beta\to\gamma}\ \mathsf{extract}\ ], \llbracket a \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{E}}_{\beta,\gamma} && \text{semantics of } \{\!|\ |\!\} \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ \langle \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V},\beta\to\gamma}, \llbracket a \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{M}}_{\beta,\gamma}\ \boxed{;}^{\gamma}\ \mathsf{extract}\ ] && \text{Lemma C.4} \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ \llbracket a \rhd m \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma}\ \boxed{;}^{\gamma}\ \mathsf{extract}\ ] && \text{semantics of } \rhd \\
&= \llbracket \{\!| a \rhd m |\!\} \rrbracket^{\mathsf{E}}_{\mathcal{V},\gamma} && \text{semantics of } \{\!|\ |\!\}
\end{aligned}$$

Rule $(\oslash @)$:

$$\begin{aligned}
\llbracket \oslash\ e \rrbracket^{\mathsf{E}}_{\mathcal{V},\gamma} &= \langle \llbracket \oslash \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta\to\gamma}, \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{E}}_{\beta,\gamma} && \text{semantics of appl.} \\
&= \langle [_{\mathcal{V}^{\mathsf{E}}}\ \mathsf{term}^{\mathcal{C}}_{\mathcal{V}^{\mathsf{E}}}\ \boxed{;}^{\beta\to\gamma}\ \mathsf{zero}^{\mathsf{M}} \mathbin{;} \mathsf{extract}\ ], \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{E}}_{\beta,\gamma} && \text{semantics of } \oslash \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ \langle \mathsf{term}^{\mathcal{C}}_{\mathcal{V}^{\mathsf{E}}}, \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathbb{1}}_{\beta,\gamma}\ \boxed{;}^{\gamma}\ \mathsf{zero}^{\mathsf{M}} \mathbin{;} \mathsf{extract}\ ] && \text{Lemma C.4} \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ \mathsf{term}^{\mathcal{C}}_{\mathcal{V}^{\mathsf{E}}}\ \boxed{;}^{\gamma}\ \mathsf{zero}^{\mathsf{M}} \mathbin{;} \mathsf{extract}\ ] && \text{terminality} \\
&= \llbracket \oslash \rrbracket^{\mathsf{E}}_{\mathcal{V},\gamma} && \text{semantics of } \oslash
\end{aligned}$$

Rule $(\rhd\upharpoonleft\upharpoonright)$:

$$\begin{aligned}
\llbracket a \rhd \upharpoonleft e \upharpoonright \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} &= \langle \llbracket \upharpoonleft e \upharpoonright \rrbracket^{\mathsf{M}}_{\mathcal{V},\beta\to\gamma}, \llbracket a \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{M}}_{\beta,\gamma} && \text{semantics of } \rhd \\
&= \langle [_{\mathcal{V}^{\mathsf{E}}}\ \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V}}\ \boxed{;}^{\beta\to\gamma}\ \mathsf{lift}\ ], \llbracket a \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{M}}_{\beta,\gamma} && \text{semantics of } \upharpoonleft\ \upharpoonright \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ \langle \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta\to\gamma}, \llbracket a \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{E}}_{\beta,\gamma}\ \boxed{;}^{\gamma}\ \mathsf{lift}\ ] && \text{Lemma C.4} \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ \llbracket e\ a \rrbracket^{\mathsf{E}}_{\mathcal{V}}\ \boxed{;}^{\gamma}\ \mathsf{lift}\ ] && \text{semantics of application} \\
&= \llbracket \upharpoonleft e\ a \upharpoonright \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} && \text{semantics of } \upharpoonleft\ \upharpoonright
\end{aligned}$$

Rule $(\rhd\lightning)$:

$$\begin{aligned}
\llbracket e \rhd \lightning \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} &= \langle \llbracket \lightning \rrbracket^{\mathsf{M}}_{\mathcal{V},\beta\to\gamma}, \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{M}}_{\beta,\gamma} && \text{semantics of } \rhd \\
&= \langle [_{\mathcal{V}^{\mathsf{E}}}\ \mathsf{term}^{\mathcal{C}}_{\mathcal{V}^{\mathsf{E}}}\ \boxed{;}^{\beta\to\gamma}\ \mathsf{zero}^{\mathsf{M}}\ ], \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{M}}_{\beta,\gamma} && \text{semantics of } \lightning \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ \langle \mathsf{term}^{\mathcal{C}}_{\mathcal{V}^{\mathsf{E}}}, \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathbb{1}}_{\beta,\gamma}\ \boxed{;}^{\gamma}\ \mathsf{zero}^{\mathsf{M}}\ ] && \text{Lemma C.4} \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ \mathsf{term}^{\mathcal{C}}_{\mathcal{V}^{\mathsf{E}}}\ \boxed{;}^{\gamma}\ \mathsf{zero}^{\mathsf{M}}\ ] && \text{terminality} \\
&= \llbracket \lightning \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} && \text{semantics of } \lightning
\end{aligned}$$

Rule $(\rhd\,|)$:

$$\begin{aligned}
\llbracket e \rhd (m_1 \mid m_2) \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} &= \langle \llbracket m_1 \mid m_2 \rrbracket^{\mathsf{M}}_{\mathcal{V},\beta\to\gamma}, \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{M}}_{\beta,\gamma} && \text{semantics of } \rhd \\
&= \langle (\llbracket m_1 \rrbracket^{\mathsf{M}}_{\mathcal{V},\beta\to\gamma} \boxplus_{\beta\to\gamma} \llbracket m_2 \rrbracket^{\mathsf{M}}_{\mathcal{V},\beta\to\gamma}), \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{M}}_{\beta,\gamma} && \text{semantics of } | \\
&= (\langle \llbracket m_1 \rrbracket^{\mathsf{M}}_{\mathcal{V},\beta\to\gamma}, \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{H}}_{\beta,\gamma}) \boxplus_{\gamma} && \text{Lemma C.10 and C.5} \\
&\quad\ (\langle \llbracket m_2 \rrbracket^{\mathsf{M}}_{\mathcal{V},\beta\to\gamma}, \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{H}}_{\beta,\gamma}) && \\
&= \llbracket e \rhd m_1 \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} \boxplus_{\gamma} \llbracket e \rhd m_2 \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} && \text{semantics of } \rhd \\
&= \llbracket (e \rhd m_1) \mid (e \rhd m_2) \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} && \text{semantics of } |
\end{aligned}$$

D.3 Pattern Matching

First we show a simplification of the semantics of pattern supply; this will abbreviate the correctness proofs of the pattern matching rules:

Lemma D.1 With $\mathcal{U} = \mathcal{V} \setminus \mathsf{FV}(p)$,

$$\llbracket e \rhd p \Mapsto m \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} = \langle \overline{\mathsf{proj}}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}},\ \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \mathbin{;} \llbracket p \rrbracket^{\mathsf{P}}_{\beta} \rangle \mathbin{;} \mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(p)^{\mathsf{E}}}\ \boxdot^{\mathsf{M}}_{\gamma}\ \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(p),\gamma}\ .$$

Proof:

$$\begin{aligned}
&\llbracket e \rhd p \Mapsto m \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} && \\
= {}& \langle \llbracket p \Mapsto m \rrbracket^{\mathsf{M}}_{\mathcal{V},\beta\to\gamma}, \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{M}}_{\beta,\gamma} && \text{semantics of } \rhd \\
= {}& \langle \Lambda((\overline{\mathsf{proj}}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}} \times \llbracket p \rrbracket^{\mathsf{P}}_{\beta}) \mathbin{;} \mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(p)^{\mathsf{E}}}\ \boxdot^{\mathsf{M}}_{\gamma}\ \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(p),\gamma}), \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{Eval}^{\mathsf{M}}_{\beta,\gamma} && \text{semantics of } \Mapsto \\
= {}& \langle \mathsf{id}_{\mathcal{V}^{\mathsf{E}}}, \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} (\overline{\mathsf{proj}}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}} \times \llbracket p \rrbracket^{\mathsf{P}}_{\beta}) \mathbin{;} \mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(p)^{\mathsf{E}}}\ \boxdot^{\mathsf{M}}_{\gamma}\ \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(p),\gamma} && \text{props. of } \times,\ \mathsf{Eval} \\
= {}& \langle \overline{\mathsf{proj}}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}},\ \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \mathbin{;} \llbracket p \rrbracket^{\mathsf{P}}_{\beta} \rangle \mathbin{;} \mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(p)^{\mathsf{E}}}\ \boxdot^{\mathsf{M}}_{\gamma}\ \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(p),\gamma} && \text{properties of } \times \quad \square
\end{aligned}$$

Rule $(\rhd v)$: Let $\mathcal{U} = \mathcal{V} \setminus \{v\}$), and assume $\mathcal{U}^{\mathsf{E}} \times \llbracket \beta \rrbracket^{\mathsf{E}} = \mathcal{V}^{\mathsf{E}}$:

$$\begin{aligned}
&\llbracket a \rhd v \Mapsto m \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} && \\
= {}& \langle \overline{\mathsf{proj}}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}},\ \llbracket a \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \mathbin{;} \llbracket v \rrbracket^{\mathsf{P}}_{\beta} \rangle \mathbin{;} \mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(v)^{\mathsf{E}}}\ \boxdot^{\mathsf{M}}_{\gamma}\ \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(v),\gamma} && \text{Lemma D.1} \\
= {}& \langle \overline{\mathsf{proj}}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}},\ \llbracket a \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \mathbin{;} \mathsf{return}^{\mathsf{M}}_{\llbracket \beta \rrbracket^{\mathsf{E}}} \rangle \mathbin{;} \mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\llbracket \beta \rrbracket^{\mathsf{E}}}\ \boxdot^{\mathsf{M}}_{\gamma}\ \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\{v\},\gamma} && \text{variable pattern semantics} \\
= {}& \langle \overline{\mathsf{proj}}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}},\ \llbracket a \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \mathsf{return}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}}\times\llbracket \beta \rrbracket^{\mathsf{E}}}\ \boxdot^{\mathsf{M}}_{\gamma}\ \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} && \mathsf{strengthL}\text{, properties} \\
= {}& \langle \overline{\mathsf{proj}}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}}, \llbracket a \rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} \rangle \mathbin{;} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} && \text{Lemma C.1} \\
= {}& \llbracket m[v \backslash a] \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} && \text{induction over constr. of } m
\end{aligned}$$

Rule $(c \triangleright c)$: For simplifying the presentation, we restrict ourselves to two constructor arguments in the matching rule (let $\mathcal{U} = \mathcal{V} \setminus \mathsf{FV}(c(p_1,p_2))$ and $\mathcal{W} = \mathcal{V} \setminus \mathsf{FV}(p_1)$ and $\mathcal{U}' = (\mathcal{W} \oplus \mathsf{FV}(p_1)) \setminus \mathsf{FV}(p_2)$):

$$
\begin{aligned}
& \llbracket c(e_1,e_2) \triangleright c(p_1,p_2) \Mapsto m \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} \\
= \; & \langle \mathsf{proj}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}},\ \llbracket c(e_1,e_2)\rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} ; \llbracket c(p_1,p_2)\rrbracket^{\mathsf{P}}_{\beta} \rangle && \text{Lemma D.1} \\
& ;\mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(c(p_1,p_2))^{\mathsf{E}}} \boxdot^{\mathsf{M}}_{\gamma} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(c(p_1,p_2)),\gamma} \\
= \; & \langle \mathsf{proj}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}} && \text{semantics of } c \\
& ,\langle \llbracket e_1\rrbracket^{\mathsf{E}}_{\mathcal{V}}, \llbracket e_2\rrbracket^{\mathsf{E}}_{\mathcal{V}}\rangle ; c^{\mathsf{E}} ; \mathsf{return}^{\mathsf{E}}_{\llbracket\beta\rrbracket_{\mathsf{E}}} ; \mathsf{lift}_{\llbracket\tau\rrbracket} \odot_{\mathsf{M}} \widetilde{c}^{\mathsf{E}} \odot_{\mathsf{M}} ((\llbracket p_1\rrbracket^{\mathsf{P}} \times \llbracket p_2\rrbracket^{\mathsf{P}}) ; \otimes_{\mathsf{M}}) \\
& \rangle ; \mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(c(p_1,p_2))^{\mathsf{E}}} \boxdot^{\mathsf{M}}_{\gamma} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(c(p_1,p_2)),\gamma} \\
= \; & \langle \mathsf{proj}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}}, \langle \llbracket e_1\rrbracket^{\mathsf{E}}_{\mathcal{V}}, \llbracket e_2\rrbracket^{\mathsf{E}}_{\mathcal{V}}\rangle ; c^{\mathsf{E}} ; \widetilde{c}^{\mathsf{E}} \odot_{\mathsf{M}} ((\llbracket p_1\rrbracket^{\mathsf{P}} \times \llbracket p_2\rrbracket^{\mathsf{P}}) ; \otimes_{\mathsf{M}})\rangle && (\mathsf{return}^{\mathsf{E}};\mathsf{lift}),\ \text{unit of } \odot \\
& ;\mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(c(p_1,p_2))^{\mathsf{E}}} \boxdot^{\mathsf{M}}_{\gamma} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(c(p_1,p_2)),\gamma} \\
= \; & \langle \mathsf{proj}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}}, \langle \llbracket e_1\rrbracket^{\mathsf{E}}_{\mathcal{V}}, \llbracket e_2\rrbracket^{\mathsf{E}}_{\mathcal{V}}\rangle ; (\llbracket p_1\rrbracket^{\mathsf{P}} \times \llbracket p_2\rrbracket^{\mathsf{P}}) ; \otimes_{\mathsf{M}}\rangle && \text{destructor axiom, } \odot \text{ unit} \\
& ;\mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(c(p_1,p_2))^{\mathsf{E}}} \boxdot^{\mathsf{M}}_{\gamma} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(c(p_1,p_2)),\gamma} \\
= \; & \langle \mathsf{proj}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}}, \langle \llbracket e_1\rrbracket^{\mathsf{E}}_{\mathcal{V}} ; \llbracket p_1\rrbracket^{\mathsf{P}}, \llbracket e_2\rrbracket^{\mathsf{E}}_{\mathcal{V}} ; \llbracket p_2\rrbracket^{\mathsf{P}}\rangle ; \otimes_{\mathsf{M}}\rangle && \text{properties of } \times \\
& ;\mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(c(p_1,p_2))^{\mathsf{E}}} \boxdot^{\mathsf{M}}_{\gamma} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(c(p_1,p_2)),\gamma} \\
= \; & \langle \mathsf{proj}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{W}},\ \llbracket e_1\rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} ; \llbracket p_1\rrbracket^{\mathsf{P}}_{\beta}\rangle ; \mathsf{strengthL}^{\mathsf{M}}_{\mathcal{W}^{\mathsf{E}},\mathsf{FV}(p_1)^{\mathsf{E}}} \odot_{\mathsf{M}} && \text{case analysis,} \\
& \langle \mathsf{proj}^{\mathsf{E}}_{\mathcal{W}\oplus\mathsf{FV}(p_1)\succ\mathcal{U}'},\ \llbracket e_2\rrbracket^{\mathsf{E}}_{\mathcal{W}\oplus\mathsf{FV}(p_1),\beta} ; \llbracket p_2\rrbracket^{\mathsf{P}}_{\beta}\rangle && \mathsf{strengthL}\text{ laws,} \\
& ;\mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}'^{\mathsf{E}},\mathsf{FV}(p_2)^{\mathsf{E}}} \boxdot^{\mathsf{M}}_{\gamma} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}'\oplus\mathsf{FV}(p_2),\gamma} && \text{suppressed product isom.} \\
= \; & \langle \mathsf{proj}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{W}},\ \llbracket e_1\rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} ; \llbracket p_1\rrbracket^{\mathsf{P}}_{\beta}\rangle && \text{Lemma D.1} \\
& ;\mathsf{strengthL}^{\mathsf{M}}_{\mathcal{W}^{\mathsf{E}},\mathsf{FV}(p_1)^{\mathsf{E}}} \boxdot^{\mathsf{M}}_{\gamma} \llbracket e_2 \triangleright p_2 \Mapsto m \rrbracket^{\mathsf{M}}_{\mathcal{W}\oplus\mathsf{FV}(p_1),\gamma} \\
= \; & \llbracket e_1 \triangleright p_1 \Mapsto e_2 \triangleright p_2 \Mapsto m \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} && \text{Lemma D.1}
\end{aligned}
$$

Rule $(d \triangleright c)$:

$$
\begin{aligned}
& \llbracket d(e_1,\dots,e_k) \triangleright c(p_1,\dots,p_n) \Mapsto m \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} \\
= \; & \langle \mathsf{proj}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}},\ \llbracket d(e_1,\dots,e_k)\rrbracket^{\mathsf{E}}_{\mathcal{V},\beta} ; \llbracket c(p_1,\dots,p_n)\rrbracket^{\mathsf{P}}_{\beta}\rangle && \text{Lemma D.1} \\
& ;\mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(c(\vec{p}))^{\mathsf{E}}} \boxdot^{\mathsf{M}}_{\gamma} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(c(\vec{p})),\gamma} \\
= \; & \langle \mathsf{proj}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}} && \text{semantics of } c \text{ and } d \\
& ,\langle \llbracket e_1\rrbracket^{\mathsf{E}}_{\mathcal{V}},\dots,\llbracket e_k\rrbracket^{\mathsf{E}}_{\mathcal{V}}\rangle ; d^{\mathsf{E}} ; \mathsf{return}^{\mathsf{E}}_{\llbracket\beta\rrbracket_{\mathsf{E}}} ; \mathsf{lift}_{\llbracket\tau\rrbracket} \odot_{\mathsf{M}} \widetilde{c}^{\mathsf{E}} \\
& \odot_{\mathsf{M}} ((\llbracket p_1\rrbracket^{\mathsf{P}} \times \dots \times \llbracket p_n\rrbracket^{\mathsf{P}}) ; \bigotimes) \\
& \rangle ; (\mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(c(\vec{p}))^{\mathsf{E}}}) \boxdot^{\mathsf{M}}_{\gamma} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(c(\vec{p})),\gamma} \\
= \; & \langle \mathsf{proj}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}}, \langle \llbracket e_1\rrbracket^{\mathsf{E}}_{\mathcal{V}},\dots,\llbracket e_k\rrbracket^{\mathsf{E}}_{\mathcal{V}}\rangle ; d^{\mathsf{E}} ; \widetilde{c}^{\mathsf{E}} && (\mathsf{return}^{\mathsf{E}};\mathsf{lift}),\ \text{unit of } \odot \\
& \odot_{\mathsf{M}} ((\llbracket p_1\rrbracket^{\mathsf{P}} \times \dots \times \llbracket p_n\rrbracket^{\mathsf{P}}) ; \bigotimes)\rangle \\
& ;(\mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(c(\vec{p}))^{\mathsf{E}}}) \boxdot^{\mathsf{M}}_{\gamma} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(c(\vec{p})),\gamma} \\
= \; & \langle \mathsf{proj}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}}, \langle \llbracket e_1\rrbracket^{\mathsf{E}}_{\mathcal{V}},\dots,\llbracket e_k\rrbracket^{\mathsf{E}}_{\mathcal{V}}\rangle ; \mathsf{term}^{\mathcal{C}}_{\llbracket\vec{\beta}\rrbracket_{\mathsf{E}}} ; \mathsf{zero}^{\mathsf{M}}_{\llbracket\vec{\alpha}\rrbracket_{\mathsf{E}}} && \text{destructor axiom} \\
& \odot_{\mathsf{M}} ((\llbracket p_1\rrbracket^{\mathsf{P}} \times \dots \times \llbracket p_n\rrbracket^{\mathsf{P}}) ; \bigotimes) \\
& \rangle ; (\mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(c(\vec{p}))^{\mathsf{E}}}) \boxdot^{\mathsf{M}}_{\gamma} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(c(\vec{p})),\gamma} \\
= \; & \langle \mathsf{proj}^{\mathsf{E}}_{\mathcal{V}\succ\mathcal{U}}, \mathsf{term}^{\mathcal{C}}_{\mathcal{V}^{\mathsf{E}}} ; \mathsf{zero}^{\mathsf{M}}_{\mathsf{FV}(c(\vec{p}))^{\mathsf{E}}} && \text{terminality, } \mathsf{zero}^{\mathsf{M}} \text{ property} \\
& \rangle ; (\mathsf{strengthL}^{\mathsf{M}}_{\mathcal{U}^{\mathsf{E}},\mathsf{FV}(c(\vec{p}))^{\mathsf{E}}}) \boxdot^{\mathsf{M}}_{\gamma} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(c(\vec{p})),\gamma} \\
= \; & \mathsf{term}^{\mathcal{C}}_{\mathcal{V}^{\mathsf{E}}} ; \mathsf{zero}^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(c(\vec{p}))^{\mathsf{E}}} \boxdot^{\mathsf{M}}_{\gamma} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{U}\oplus\mathsf{FV}(c(\vec{p})),\gamma} && \mathsf{zero}^{\mathsf{M}} \text{ interaction with } \mathsf{strengthL} \\
= \; & [_{\mathcal{V}^{\mathsf{E}}}\ \mathsf{term}^{\mathcal{C}}_{\mathcal{V}^{\mathsf{E}}}\ \boxed{;}^{\gamma}\ \mathsf{zero}^{\mathsf{M}}\ ] && \text{Lemma C.8} \\
= \; & \llbracket \lightning \rrbracket^{\mathsf{M}}_{\mathcal{V},\gamma} && \text{semantics of } \lightning
\end{aligned}
$$

D.4 Haskell Semantics

The following proofs assume the setting of Sect. 6.2, using the Haskell monad for expressions, i.e., $\mathsf{E} = \mathsf{H}$, adding, through combination with Setting 3.2.2, only failure for matchings, i.e., $\mathsf{M} = \mathsf{E} + \mathbb{1}$, and extracting failure to the bottom provided by the Haskell monad, i.e., $\mathsf{empty}^{\mathsf{E}} = \mathsf{bottom}$.

Rule $(\upharpoonleft\upharpoonright \mathbin{\|})$:

$$
\begin{aligned}
\llbracket \upharpoonleft e \upharpoonright \mathbin{\|} m \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} &= \llbracket \upharpoonleft e \upharpoonright \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} \boxplus_{\alpha} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} && \text{semantics of } \mathbin{\|} \\
&= [_{\mathcal{V}^{\mathsf{E}}}\ \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V}}\ \boxed{;}^{\alpha}\ \mathsf{lift}\ ] \boxplus_{\alpha} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha} && \text{semantics of } \upharpoonleft\upharpoonright
\end{aligned}
$$

To prove that this equals $\llbracket \upharpoonleft e \upharpoonright \rrbracket^{\mathsf{M}}_{\mathcal{V},\alpha}$, we show the following generalisation:

$$
[_q\ f\ \boxed{;}^{\alpha}\ \mathsf{lift}\ ] \boxplus_{\alpha} g = [_q\ f\ \boxed{;}^{\alpha}\ \mathsf{lift}\ ]
$$

For constructed types, this is easy:

$$\begin{aligned}
[_q\ f\ \boxed{;}^{\tau}\ \mathsf{lift}\ ] \boxplus_\tau g &= (f ; \mathsf{lift}_\tau) \boxplus_\tau g && \text{definition of } [\ \boxed{;}^{\tau}\ ] \\
&= \langle (f;\mathsf{lift}_\tau), g\rangle ; \mathsf{plus}^{\mathsf{M}} && \text{definition of } \boxplus_\tau \\
&= f;\mathsf{lift}_\tau && \text{definition of } \mathsf{plus}^{\mathsf{E}+1}\text{, Setting 3.2.2} \\
&= [_q\ f\ \boxed{;}^{\tau}\ \mathsf{lift}\ ] && \text{definition of } [\ \boxed{;}^{\tau}\ ]
\end{aligned}$$

For function types, we obtain:

$$\begin{aligned}
& [_q\ f\ \boxed{;}^{\beta\to\gamma}\ \mathsf{lift}\ ] \boxplus_{\beta\to\gamma} g \\
={}& \Lambda[_{q\times[\![\beta]\!]^{\mathsf E}}\ (f\times \mathsf{id}_{[\![\beta]\!]^{\mathsf E}});\mathsf{Eval}^{\mathsf K}_{\beta,\gamma}\ \boxed{;}^{\gamma}\ \mathsf{lift}\ ] \boxplus_{\beta\to\gamma} g && \text{definition of } [\ \boxed{;}^{\beta\to\gamma}\ ] \\
={}& \Lambda\left(\begin{array}{l} (\Lambda[_{q\times[\![\beta]\!]^{\mathsf E}}\ (f\times \mathsf{id}_{[\![\beta]\!]^{\mathsf E}});\mathsf{Eval}^{\mathsf K}_{\beta,\gamma}\ \boxed{;}^{\gamma}\ \mathsf{lift}\ ]\times \mathsf{id}_{[\![\beta]\!]^{\mathsf E}});\mathsf{Eval}^{\mathsf M}_{\beta,\gamma} \\ \boxplus_\gamma \\ (g\times \mathsf{id}_{[\![\beta]\!]^{\mathsf E}});\mathsf{Eval}^{\mathsf M}_{\beta,\gamma} \end{array}\right) && \text{definition of } \boxplus_{\beta\to\gamma} \\
={}& \Lambda\left(\begin{array}{l} [_{q\times[\![\beta]\!]^{\mathsf E}}\ (f\times \mathsf{id}_{[\![\beta]\!]^{\mathsf E}});\mathsf{Eval}^{\mathsf K}_{\beta,\gamma}\ \boxed{;}^{\gamma}\ \mathsf{lift}\ ] \\ \boxplus_\gamma \\ (g\times \mathsf{id}_{[\![\beta]\!]^{\mathsf E}});\mathsf{Eval}^{\mathsf M}_{\beta,\gamma} \end{array}\right) && \text{definition of } \mathsf{Eval} \\
={}& \Lambda[_{q\times[\![\beta]\!]^{\mathsf E}}\ (f\times \mathsf{id}_{[\![\beta]\!]^{\mathsf E}});\mathsf{Eval}^{\mathsf K}_{\beta,\gamma}\ \boxed{;}^{\gamma}\ \mathsf{lift}\ ] && \text{induction hyp.} \\
={}& [_q\ f\ \boxed{;}^{\beta\to\gamma}\ \mathsf{lift}\ ] && \text{definition of } [\ \boxed{;}^{\beta\to\gamma}\ ]
\end{aligned}$$

Rule $(\oslash \triangleright c \to \oslash)$:

$$\begin{aligned}
& [\![\oslash \triangleright c(p_1,\ldots,p_n) \Mapsto m]\!]^{\mathsf M}_{\mathcal V,\gamma} \\
={}& \langle \mathsf{proj}^{\mathsf E}_{\overline{\mathcal V}\succ\mathcal U},\ [\![\oslash]\!]^{\mathsf E}_{\mathcal V,\tau};[\![c(p_1,\ldots,p_n)]\!]^{\mathsf P}_{\tau}\rangle && \text{Lemma D.1} \\
& ;\mathsf{strengthL}^{\mathsf M}_{\mathcal U^{\mathsf E},\mathsf{FV}(c(\vec p))^{\mathsf E}}\ \boxdot^{\mathsf M}_{\gamma}\ [\![m]\!]^{\mathsf M}_{\mathcal U\oplus\mathsf{FV}(c(\vec p)),\gamma} \\
={}& \langle \mathsf{proj}^{\mathsf E}_{\overline{\mathcal V}\succ\mathcal U} && \text{semantics of } c \text{ and } \oslash \\
& ,[_{\mathcal V^{\mathsf E}}\ \mathsf{term}^{\mathcal C}_{\mathcal V^{\mathsf E}}\ \boxed{;}^{\tau}\ \mathsf{zero}^{\mathsf M};\mathsf{extract}\ ];\mathsf{lift}_{[\![\tau]\!]}\odot_{\mathsf M}\widetilde{c}^{\mathsf E} \\
& \quad\odot_{\mathsf M}(([\![p_1]\!]^{\mathsf P}\times\cdots\times[\![p_n]\!]^{\mathsf P});\bigotimes) \\
& \rangle;(\mathsf{strengthL}^{\mathsf M}_{\mathcal U^{\mathsf E},\mathsf{FV}(c(\vec p))^{\mathsf E}})\ \boxdot^{\mathsf M}_{\gamma}\ [\![m]\!]^{\mathsf M}_{\mathcal U\oplus\mathsf{FV}(c(\vec p)),\gamma} \\
={}& \langle \mathsf{proj}^{\mathsf E}_{\overline{\mathcal V}\succ\mathcal U} && (\mathsf{zero};\mathsf{extract}) \\
& ,[_{\mathcal V^{\mathsf E}}\ \mathsf{term}^{\mathcal C}_{\mathcal V^{\mathsf E}}\ \boxed{;}^{\tau}\ \mathsf{empty}^{\mathsf E}\ ];\mathsf{lift}_{[\![\tau]\!]}\odot_{\mathsf M}\widetilde{c}^{\mathsf E} \\
& \quad\odot_{\mathsf M}(([\![p_1]\!]^{\mathsf P}\times\cdots\times[\![p_n]\!]^{\mathsf P});\bigotimes) \\
& \rangle;(\mathsf{strengthL}^{\mathsf M}_{\mathcal U^{\mathsf E},\mathsf{FV}(c(\vec p))^{\mathsf E}})\ \boxdot^{\mathsf M}_{\gamma}\ [\![m]\!]^{\mathsf M}_{\mathcal U\oplus\mathsf{FV}(c(\vec p)),\gamma} \\
={}& \langle \mathsf{proj}^{\mathsf E}_{\overline{\mathcal V}\succ\mathcal U} && \text{definition of } [\ \boxed{;}^{\tau}\ ] \text{ and } \mathsf{empty}^{\mathsf E} \\
& ,\mathsf{term}^{\mathcal C}_{\mathcal V^{\mathsf E}};\mathsf{bottom}_{[\![\tau]\!]};\mathsf{lift}_{[\![\tau]\!]}\odot_{\mathsf M}\widetilde{c}^{\mathsf E} \\
& \quad\odot_{\mathsf M}(([\![p_1]\!]^{\mathsf P}\times\cdots\times[\![p_n]\!]^{\mathsf P});\bigotimes) \\
& \rangle;(\mathsf{strengthL}^{\mathsf M}_{\mathcal U^{\mathsf E},\mathsf{FV}(c(\vec p))^{\mathsf E}})\ \boxdot^{\mathsf M}_{\gamma}\ [\![m]\!]^{\mathsf M}_{\mathcal U\oplus\mathsf{FV}(c(\vec p)),\gamma} \\
={}& \langle \mathsf{proj}^{\mathsf E}_{\overline{\mathcal V}\succ\mathcal U} && \text{(1) from page 17} \\
& ,\mathsf{term}^{\mathcal C}_{\mathcal V^{\mathsf E}};\mathsf{bottom}_{\mathsf{FV}(c(\vec p))^{\mathsf E}};\mathsf{lift}_{\mathsf{FV}(c(\vec p))^{\mathsf E}} \\
& \rangle;(\mathsf{strengthL}^{\mathsf M}_{\mathcal U^{\mathsf E},\mathsf{FV}(c(\vec p))^{\mathsf E}})\ \boxdot^{\mathsf M}_{\gamma}\ [\![m]\!]^{\mathsf M}_{\mathcal U\oplus\mathsf{FV}(c(\vec p)),\gamma}
\end{aligned}$$

$$\begin{aligned}
={}& [_{\mathcal V^{\mathsf E}}\ \mathsf{term}^{\mathcal C}_{\mathcal V^{\mathsf E}}\ \boxed{;}^{\gamma}\ \mathsf{bottom};\mathsf{lift}\ ] && \text{(2), (3) from page 17} \\
={}& [_{\mathcal V^{\mathsf E}}\ [_{\mathcal V^{\mathsf E}}\ \mathsf{term}^{\mathcal C}_{\mathcal V^{\mathsf E}}\ \boxed{;}^{\gamma}\ \mathsf{zero}^{\mathsf M};\mathsf{extract}\ ]\ \boxed{;}^{\gamma}\ \mathsf{lift}\ ] && \text{Lemma C.7, def. of } \mathsf{empty}^{\mathsf E} \\
={}& [_{\mathcal V^{\mathsf E}}\ [\![\oslash]\!]^{\mathsf E}_{\mathcal V}\ \boxed{;}^{\gamma}\ \mathsf{lift}\ ] && \text{semantics of } \oslash \\
={}& [\![\upharpoonleft \oslash \upharpoonright]\!]^{\mathsf M}_{\mathcal V,\gamma} && \text{semantics of } \upharpoonleft\upharpoonright
\end{aligned}$$

D.5 The Setting $\mathsf M = \mathsf E$

The following proof relies on $\mathsf M = \mathsf E$, or, more precisely, on $\mathsf{extract};\mathsf{lift}_\tau = \mathsf{id}$, or even just the following for all constructed types τ:

$$\mathsf{zero}^{\mathsf M}_{[\![\tau]\!]};\mathsf{extract}_{[\![\tau]\!]};\mathsf{lift}_{[\![\tau]\!]} = \mathsf{zero}^{\mathsf M}_{[\![\tau]\!]}\ .$$

Rule $(\oslash \triangleright c \to \lightning)$: We assume that c is a constructor with $c : \alpha_1 \times \cdots \alpha_n \to \tau$.

$$\begin{aligned}
& [\![\oslash \triangleright c(p_1,\ldots,p_n) \Mapsto m]\!]^{\mathsf M}_{\mathcal V,\gamma} \\
={}& \langle \mathsf{proj}^{\mathsf E}_{\overline{\mathcal V}\succ\mathcal U},\ [\![\oslash]\!]^{\mathsf E}_{\mathcal V,\tau};[\![c(p_1,\ldots,p_n)]\!]^{\mathsf P}_{\tau}\rangle && \text{Lemma D.1} \\
& ;\mathsf{strengthL}^{\mathsf M}_{\mathcal U^{\mathsf E},\mathsf{FV}(c(\vec p))^{\mathsf E}}\ \boxdot^{\mathsf M}_{\gamma}\ [\![m]\!]^{\mathsf M}_{\mathcal U\oplus\mathsf{FV}(c(\vec p)),\gamma} \\
={}& \langle \mathsf{proj}^{\mathsf E}_{\overline{\mathcal V}\succ\mathcal U} && \text{semantics of } c \text{ and } \oslash \\
& ,[_{\mathcal V^{\mathsf E}}\ \mathsf{term}^{\mathcal C}_{\mathcal V^{\mathsf E}}\ \boxed{;}^{\tau}\ \mathsf{zero}^{\mathsf M};\mathsf{extract}\ ];\mathsf{lift}_{[\![\tau]\!]}\odot_{\mathsf M}\widetilde{c}^{\mathsf E} \\
& \quad\odot_{\mathsf M}(([\![p_1]\!]^{\mathsf P}\times\cdots\times[\![p_n]\!]^{\mathsf P});\bigotimes) \\
& \rangle;(\mathsf{strengthL}^{\mathsf M}_{\mathcal U^{\mathsf E},\mathsf{FV}(c(\vec p))^{\mathsf E}})\ \boxdot^{\mathsf M}_{\gamma}\ [\![m]\!]^{\mathsf M}_{\mathcal U\oplus\mathsf{FV}(c(\vec p)),\gamma} \\
={}& \langle \mathsf{proj}^{\mathsf E}_{\overline{\mathcal V}\succ\mathcal U} && \text{definition of } [\ \boxed{;}^{\tau}\ ] \\
& ,\mathsf{term}^{\mathcal C}_{\mathcal V^{\mathsf E}};\mathsf{zero}^{\mathsf M}_{[\![\tau]\!]};\mathsf{extract}_{[\![\tau]\!]};\mathsf{lift}_{[\![\tau]\!]}\odot_{\mathsf M}\widetilde{c}^{\mathsf E}\odot_{\mathsf M}(([\![p_1]\!]^{\mathsf P}\times\cdots\times[\![p_n]\!]^{\mathsf P});\bigotimes) \\
& \rangle;(\mathsf{strengthL}^{\mathsf M}_{\mathcal U^{\mathsf E},\mathsf{FV}(c(\vec p))^{\mathsf E}})\ \boxdot^{\mathsf M}_{\gamma}\ [\![m]\!]^{\mathsf M}_{\mathcal U\oplus\mathsf{FV}(c(\vec p)),\gamma} \\
={}& \langle \mathsf{proj}^{\mathsf E}_{\overline{\mathcal V}\succ\mathcal U} && \mathsf M = \mathsf E \\
& ,\mathsf{term}^{\mathcal C}_{\mathcal V^{\mathsf E}};\mathsf{zero}^{\mathsf M}_{[\![\tau]\!]}\odot_{\mathsf M}\widetilde{c}^{\mathsf E}\odot_{\mathsf M}(([\![p_1]\!]^{\mathsf P}\times\cdots\times[\![p_n]\!]^{\mathsf P});\bigotimes) \\
& \rangle;(\mathsf{strengthL}^{\mathsf M}_{\mathcal U^{\mathsf E},\mathsf{FV}(c(\vec p))^{\mathsf E}})\ \boxdot^{\mathsf M}_{\gamma}\ [\![m]\!]^{\mathsf M}_{\mathcal U\oplus\mathsf{FV}(c(\vec p)),\gamma} \\
={}& \langle \mathsf{proj}^{\mathsf E}_{\overline{\mathcal V}\succ\mathcal U} && \mathsf{zero}^{\mathsf M} \text{ is zero of } \odot \\
& ,\mathsf{term}^{\mathcal C}_{\mathcal V^{\mathsf E}};\mathsf{zero}^{\mathsf M}_{\mathsf{FV}(c(\vec p))^{\mathsf E}}\rangle;(\mathsf{strengthL}^{\mathsf M}_{\mathcal U^{\mathsf E},\mathsf{FV}(c(\vec p))^{\mathsf E}})\ \boxdot^{\mathsf M}_{\gamma}\ [\![m]\!]^{\mathsf M}_{\mathcal U\oplus\mathsf{FV}(c(\vec p)),\gamma} \\
={}& \mathsf{term}^{\mathcal C}_{\mathcal V^{\mathsf E}};\mathsf{zero}^{\mathsf M}_{\mathcal U^{\mathsf E}\times\mathsf{FV}(c(\vec p))^{\mathsf E}}\ \boxdot^{\mathsf M}_{\gamma}\ [\![m]\!]^{\mathsf M}_{\mathcal U\oplus\mathsf{FV}(c(\vec p)),\gamma} && \text{strength preserves } \mathsf{zero} \\
={}& [_{\mathcal V^{\mathsf E}}\ \mathsf{term}^{\mathcal C}_{\mathcal V^{\mathsf E}}\ \boxed{;}^{\gamma}\ \mathsf{zero}^{\mathsf M}\ ] && \text{Lemma C.8} \\
={}& [\![\lightning]\!]^{\mathsf M}_{\mathcal V,\gamma} && \text{semantics of } \lightning
\end{aligned}$$