Under consideration for publication in J. Functional Programming

Generalized Monoidal Effects And Handlers

RUBEN P. PIETERS, TOM SCHRIJVERS
KU Leuven, Leuven, Belgium
EXEQUIEL RIVAS
Inria, Paris, France
(*e-mail:* {ruben.pieters, tom.schrijvers}@cs.kuleuven.be, exequiel.rivas-gadda@inria.fr)

Abstract

Algebraic effects and handlers are a convenient method for structuring monadic effects with primitive effectful operations and separating the syntax from the interpretation of these operations. However, the scope of conventional handlers is limited as not all side effects are monadic in nature.

This paper generalizes the notion of algebraic effects and handlers from monads to generalized monoids, which notably covers applicative functors and arrows as well as monads. For this purpose we switch the category theoretical basis from free algebras to free monoids. In addition, we show how lax monoidal functors enable the reuse of handlers and programs across different computation classes, for example handling applicative computations with monadic handlers.

We motivate and present these handler interfaces in the context of build systems. Tasks in a build system are represented by a free computation and their interpretation as a handler. This use case is based on the work of Mokhov *et al.* (2018).

1 Introduction

Since their introduction to purely functional programming, monads (Moggi, 1991; Wadler, 1990) have monopolized modeling computational effects. This changed with the proposal of new classes of effectful computations: applicative functors (McBride & Paterson, 2008) and arrows (Hughes, 2000), which capture types of side effects amenable to static analysis at the cost of expressiveness.

In a separate development, algebraic effects and handlers (Plotkin & Pretnar, 2009) were created as a more convenient formulation of monadic effects and programs. Their success is largely due to their easier integration with impure functional and imperative languages to enable user-defined effects. This approach encodes effects as operations represented by the signature of an algebraic theory. The semantics of these effects is represented by an interpretation for the operations.

Although the conventional handlers capture monadic effects well, other computation classes such as applicative functors and arrows are not covered. To remedy this situation, Lindley (2014) presented a language design supporting handlers for the classic triad of effects: monad, arrow and applicative functor. This is backed by a type system verifying the class of expressed computations. However, Lindley's exposition lacks an extension of the category theoretical underpinnings, introduced by Plotkin and Pretnar.

This work aims to provide this extension by reviewing the definition of handlers to include non-monadic computations, notably applicative functors and arrows. For this purpose we leverage the framework of Rivas & Jaskelioff (2017) which characterizes the triad of effects in terms of generalized monoids. This is used to replace the conventional free algebra approach, with handling rules based on the unique algebra homomorphism, by a free monoid approach, with handling rules based on the unique monoid homomorphism.

Specifically our contributions are:

- We present a generic framework to derive handlers for monoids in monoidal categories.
- We give a derivation of handlers for the classes of applicative, arrow and monadic effects. Since the derived monadic handlers are equally expressive as the conventional free algebra handlers, we see the monoidal handlers as an extension to the free algebra handlers.
- We present a method for reusing handlers and programs, by employing a monoidal adjunction between the relevant monoidal categories.
- We present the build system model introduced by Mokhov *et al.* (2018) as a motivating use case for these generalized handlers.

Section 2 introduces and motivates the concept of non-monadic handlers. Section 3 introduces the relevant category theoretic background related to algebraic effects and handlers, presenting them as free algebras. Section 4 derives these handlers from the perspective of free monoids. Section 5 derives applicative and arrow handlers from the idea of free monoids. Section 6 shows an approach to reuse handlers and computations across different monoidal categories. Section 7 describes the relation with the original Haskell use case and describes some uses for arrow build systems. Section 8 presents and discusses related work.

This paper is based on two earlier publications: “Handlers for Non-Monadic Computations” (Pieters *et al.*, 2017) and “Relating Idioms, Arrows and Monads from Monoidal Adjunctions” (Rivas, 2018). The main change compared to these earlier works is the addition of the motivating use case of build systems. The build system use case has replaced the previous motivation section (Section 2). The section showcasing handler examples has been reworked to fit this new use case (Section 5). The section describing the reuse of handlers and programs using monoidal adjunctions (Section 6) has been updated and made consistent with the earlier work on monoidal adjunctions (Rivas, 2018). Section 7 covers the relation between the build systems paper (Mokhov *et al.*, 2018) and monoidal effects and handlers in more detail and shows how their work can be extended by using arrows; this section was not present in the earlier publications.

2 Motivation

The original algebraic effects and effect handlers approach by Plotkin & Pretnar (2009) covers the space of what we call monadic handlers. These monadic handlers are the conventional approach currently used when languages implement effect handlers.

This section elaborates on the difference between monadic and non-monadic handlers and presents a use case involving non-monadic handlers. The examples are given in the setting of a simple model for build systems.

2.1 Notation in Code Examples

In the rest of the paper we use code samples to convey and illustrate our ideas. Code samples are typeset in a teletype font and typically are located in separate code block. The syntax is mostly based on Haskell syntax, with differences in several aspects. There is a short overview of syntax in Table 1.

Type Signatures We deviate from Haskell syntax when giving type signatures by utilizing a single colon, for example `1 : Int`. The double colon is used for list concatenation, for example `[1, 2] :: [3, 4]`. Type variables are also denoted with uppercase characters, such as `x : A`, which mimicks the math notation.

Computations We use Haskell's `do`-notation to present more readable syntax for computations created using various constructors. We assume that the reader is familiar with the deconstruction of `do`-syntax into Haskell's `return` and `>>=` constructors.

Additionally, we use the `do`-notation for expressing computations with different constructors, which are limited in expressive power compared to monadic computations. For applicative computations this is similar to Haskell's `ApplicativeDo` extension (2016), and for arrow computations there is some similarity to Haskell's arrow syntax (2001).

Computation types are denoted with an exclamation mark and the effect set after the return value. For example, `c : A ! {operation}`, where `A` is the return value of computation `c` and `operation` is an effect that might be present in `c`. The computation class is also added as one of the effects, where `M` denotes monadic computations, `Ap` denotes applicative computations and `Ar` denotes arrow computations. Pure values, or computations with no possible effects, can either be denoted as `A ! ∅` or simply `A`.

Handlers Handlers are the consumers of effects, which in our case are introduced by computations expressed in `do`-notation. The addition of handlers induces two syntax constructs: `handler` and `handle with`.

The `handler` construct defines a new handler, it consists of several clauses which are preceded by `|`. These clauses define how certain constructors are interpreted, utilizing their inputs which are stated in brackets. For example, the clause `| op (i : Int) -> f i` interprets the `op` constructor to `f i`, which utilizes the input `i`.

Handler types are denoted with a double arrow: `=>`. The input for a handler is a computation where the handled clauses might be present, and the output is a computation where these effects are possibly removed.

The `handle comp with h` construct states that the computation `comp` is be handled by the handler `h`. Intuitively the meaning of this is that, whenever a constructor is encountered while executing the computation, the handler is consulted for the interpretation of that constructor defined. The behaviour is more precisely defined in Section 3.2.

2.2 Motivating Use Case: Build Systems

Build systems automate a series of tasks for building dependent artifacts. Typically, these artifacts are files on a file system. However, the setting in this article is the spreadsheet setting following the examples of Mokhov *et al.* (2018), where the artifacts are cells in

Table 1: Syntax Overview

Syntax Construct	Example

Creating Computations	`c = do`
	` result <- operation input`
	` return result`

Computation Types	`c : A ! {operation, M}`

Creating Handlers	`h = handler`
	`| op1 (param1: T) -> f param1`
	`| op2 (param2: T) -> f param2`

Handler Types	`h : A ! {op1, op2, ...} => A ! {...}`
Handling Computations	`handle h with c`

a spreadsheet. Throughout the paper we illustrate examples with the motivating use case of build systems. This use case is heavily inspired by the work of Mokhov *et al.* (2018), which presents a functional model of build systems.

Describing Tasks The spreadsheet setting views spreadsheets according to the build system model. In our examples, cells are uniquely identified by their name with type `String` and contain values of type `Int`. A cell can refer to values of other cells, this is represented by an operation `fetch`. The `fetch` operation takes a name as input and returns the value of that cell. The computation contained in a cell is called a *task*, an example of such a task is `taskExample`, which fetches the contents of cells `"A1"` and `"A2"` and adds them together to create the value of this cell. Note that similar to Mokhov *et al.* (2018), we do not consider cyclic calculations, and thus attaching this task to cell `"A1"` or `"A2"` is disallowed.

The computation is annotated with the type `Int ! {fetch, Ap}`, which means that it returns an `Int` and contains the `fetch` operation. Additionally it only uses a limited form of computation expressiveness, namely applicative, so `Ap` is also present in the set of effects. The difference between the computation classes is explained in more detail in Section 2.3.

```
taskExample : Int ! {fetch, Ap}
taskExample = do
  a1 <- fetch "A1"
  a2 <- fetch "A2"
  return (a1 + a2)
```

Running Tasks The operations, such as `fetch`, used in these computations have no attached meaning yet. The meaning of these operations are given by the interpretation with a handler. For example, we can interpret `taskExample` by using the handler `fetchConsole` which is defined below.

```
fetchConsole : A ! {fetch, M} => IO A
fetchConsole = handler
| val (x: A) -> return x                                    (value clause)
```

```
| fetch (cell: String, k: Int -> IO A) -> do
    print ("cell: " ++ cell)
    x <- readLn
    k x                                          (operation clause)
```

This handler requests the user to specify the values via the console for each cell. The behaviour of a handler is specified by a value clause and an operation clause for each handled operation. The value clause triggers on an evaluated computation without any operations. This handler wraps the evaluated value of type `A` to an expected value of type `IO A`. The operation clause triggers when the evaluated computation is an operation with a continuation. It takes the input arguments as parameter `p` and the continuation as parameter `k`. The former contains all data passed to the operation. The latter captures a resumption point, which resumes the computation where the operation was called and introduces a result. The computation does not resume if the continuation parameter is not invoked, resulting in behaviour similar to exceptions.

The handler uses the standard monadic interface for handling computations, and thus handles computations with monadic or lower expressiveness. This is signified in the type signature by the input computation having the `M` class in its set of effects. This should be seen as an upper bound on the input computation, since the handler is also applicable to applicative or arrow computations.

An example of handling `taskExample` with `fetchConsole` is given below. The values for `"A1"` and `"A2"` are requested and then the value of the task is given.

```
λ> handle taskExample with fetchConsole
cell: A1
5<Enter>
cell: A2
10<Enter>
15
```

Analyzing Dependencies In a build system we would like to analyze tasks, for example to compute a list of dependencies for tasks. This information enables some optimizations in our build system such as avoiding redundant building of duplicate tasks or parallelizing independent tasks. Let us see what happens if we try to compute the list of dependencies with a monadic handler.

```
handlerAnalyze : A ! {fetch, M} => [String]
handlerAnalyze = handler
| val (x: A) -> []
| fetch (cell: String, k: Int -> [String]) ->
    [cell] ++ (k ?)
```

We encounter a problem in implementing the operation clause for `fetch`: We want to combine the currently encountered dependency `cell` with the recursively computed dependencies in the continuation `k`. However, to know what these dependencies are, we have to pass the value of the current cell.

The problem is that we are implementing a monadic handler, which must be able to handle *all* monadic computations. Consider `taskDyn`, a task which fetches the cell content dynamically, meaning which cell is fetched is dependent on values from other cells.

```
taskDyn : Int ! {fetch, Ar}
taskDyn = do
  a1 <- fetch "A1"
  fetch ("B" ++ show a1)
```

For this case it is not possible to determine the list of fetched cells upfront since this is dependent on the contents of `"A1"`. Instead, we can use an applicative or arrow handler, which is limited to computations with that respective expressiveness. This paper looks at different handler interfaces derived from the category theoretical perspective of free monoids, subsuming monadic handlers.

2.3 Different Classes of Computations

Before we continue, we want to give a better intuition for the different computation classes used throughout the paper. We follow the distinction made by Lindley (2014) based on having/not having data and control flow. Data flow implies that input to operations is dependent on the results of previous operations. Control flow implies a dependency of subsequent operations on results of previous operations.

Applicative Computations We start with the applicative computations. These computations are a static list of operations which can compute a final value from each of the results. The `taskExample` computation, repeated below, is an example of an applicative computation since the exact list of operations in the computation is completely static.

```
taskExample : Int ! {fetch, Ap}
taskExample = do
  a1 <- fetch "A1"
  a2 <- fetch "A2"
  return (a1 + a2)
```

Arrow Computations Arrow computations introduce *data flow*, which means that the input to operations can depend on results from previous operations. The `taskDyn` computation, repeated below, is an example of an arrow computation since the `fetch` operation in the second line depends on the result of the previous `fetch` operation.

```
taskDyn : Int ! {fetch, Ar}
taskDyn = do
  a1 <- fetch "A1"
  fetch ("A" ++ show a1)                                  (data flow)
```

Monadic Computations Monadic computations additionally have *control flow*, which means that the choice of which subsequent operations should be invoked is dependent on the result of previous operations.

A simple example showcasing control flow is to use a conditional statement. In the example `taskControlFlow1` the contents of cell `C1` are fetched and if it contains a `1` then the contents of cell `A1` are returned, otherwise the constant `-1` is returned.

```
taskControlFlow1 : Int ! {fetch, M}
taskControlFlow1 = do
  c1 <- fetch "C1"
  if c1 == 1                                          (control flow)
    then fetch "A1"
    else return -1
```

These conditional statements have been considered separately before, such as in the `ArrowChoice` class or the `Selective` class (2019). However, conditional statements on their own do not fully capture the expressiveness of monadic computations.

The `taskControlFlow2` example below exhibits both data and control flow. The contents of cell `A1` contains the amount of times the fetch operation needs to be executed. The `for` function aggregates the results of calling `fetch` on each of the cell locations.

```
taskControlFlow2 : [Int] ! {fetch, M}
taskControlFlow2 = do
  n <- fetch "A1"
  repeat n                                      (control/data flow)
where
  repeat : Int -> [Int] ! {fetch, M}
  repeat 0 = return []
  repeat n = do
    result <- fetch ("B" ++ show n)
    return (result : repeat (n - 1))
```

Note that the above computation can technically be encoded using only infinite branching and data flow. However, we consider it as a monadic computation here as operationally it is ineffecient to evaluate massively nested branches and analysis on infinite branches is not likely give a productive result.

2.4 Analysis with Non-Monadic Handlers

Analyzing Dependencies with Applicative Handler The simplest computations to analyze are applicative computations like `taskExample`. To do this, we use use an applicative handler. The applicative handler interface exposes the continuation differently, since we have access to the recursively computed dependencies with the parameter `l`. It is possible to access these dependencies, since this is statically known when restricted to applicative computations. A simplified version of the handler implementation is given below, the full version is discussed as `iAnalyzeAp` in Example 5.1.

```
handlerAnalyzeI : A ! {fetch, Ap} => [String]
handlerAnalyzeI = handler
| val (x: A) -> []
| fetch (cell: String, l: [String]) ->
    cell :: l
```

Handling the `taskExample` computation with this handler evaluates to the list of dependencies, which is `"A1"` and `"A2"`.

```
λ> handle taskExample with handlerAnalyzeI
["A1", "A2"]
```

Analyzing Dependencies with Arrow Handler While applicatives are easy to analyze, they are also limited in what they can express. This might prevent us to describe the computation we want. For example, the `taskDyn` computation earlier does not respect the limitations of an applicative computation. Of course, arrow computations on the other hand, have extra limitations on the analysis side. It is no longer possible to determine all dependencies upfront. However, it is still possible to analyze the affected columns of dependencies. To do this we have to distinguish between statically available parts of the computation, the columns, and the dynamically available parts of the computation, the rows. A simplified version of the handler implementation is given below, the full version is discussed as `iAnalyzeAr` in Example 5.2.

```
handlerAnalyzeA : A ! {fetch, Ar} => [String]
handlerAnalyzeA = handler
| val (f: A -> B) -> []
| fetch (col: String, l: [String]) ->
    col :: l
```

Handling the `taskDyn` computation with this handler evaluates to the list of affected columns, which is `"A"` and `"B"`.

```
λ> handle taskDyn with handlerAnalyzeA
["A", "B"]
```

Of course, any analysis which does not depend on the input to the operations, for example counting the `fetch` operations, is equally applicable to applicative and arrow computations.

Conclusion Attempting to handle a computation with an inappropriate handler, for example `handle taskDyn with handlerAnalyzeI` should result in a runtime or, preferably, a type error.

We have illustrated simple use cases of analysis with a non-monadic handler. More complicated analysis includes parallelizing/batching operations, calculating a heat map of operations/parameters such as location, or other calculations on statically available information.

3 Background

This section introduces the necessary background on which the remainder of the paper is based. We assume basic familiarity with common category theoretical concepts such as functors, natural transformations and monads.

3.1 Notational Conventions

We highlight some of the more specific notation here.

Category We reserve $\mathscr{C}$ to mean the category of the programming language under consideration, with types as objects and functions between those types as morphisms. We assume that this base category has (co-)products, exponentials and (co-)ends.

Morphisms Components of natural transformations usually have a subscript mentioning their naturality, e.g. $id_A : A \to A$ is natural in A. Identity morphisms are denoted as the more compact $A : A \to A$ instead of $id_A : A \to A$.

Isomorphisms We denote an isomorphism with $f : A \cong B : g$, where $f : A \to B$ and $g : B \to A$ are part of the two-sided inverse $g \circ f = A$ and $f \circ g = B$. We leave out the names of the functions if they are not important.

(Co-)Products We use $A \times B$ to denote products, in $\mathscr{C}$ this represents the tuple type `(A,B)`. We use $A + B$ to denote coproducts, and $[f,g]$ to denote the unique morphism $A + B \to X$ constructed from $f : A \to X$ and $g : B \to X$.

Exponential Objects We use A^B to denote the exponentiation of A with B. In $\mathscr{C}$ exponential A^B is the function type `B -> A`.

Algebra of a Functor An F-algebra with *carrier* A and *action* b is denoted by $\langle A, b : FA \to A \rangle$.

(Co-)Ends We denote ends as $\int_A F(A,A)$ and co-ends as $\int^A F(A,A)$, for a bifunctor $F : \mathscr{A}^{op} \times \mathscr{A} \to \mathscr{B}$. In $\mathscr{C}$, ends can be understood as a universal type quantification $\forall$`A. F(A,A)`, while co-ends correspond to existential type quantification $\exists$`A. F(A,A)`. Usually the type quantifier $\forall$ is omitted when it is clear from context.

3.2 Algebraic Effects and Handlers

Plotkin & Pretnar's definition of algebraic effects and handlers consists of two parts: the *operations*, which introduce effects, and the *handlers*, which interpret them (Plotkin & Pretnar, 2009).

Operations as Functors Operations, such as `fetch`, is be abstracted by endofunctors of the form $\Sigma_i = P_i \times -^{N_i}$, where N_i is the *arity* of the operation, and P_i contains the *parameters* of the operation. The former refers to the type of values which the operation introduces into the computation, the latter refers to the type of values which the operation takes as input. For example, `fetch` introduces an `Int` value, the cell content, and takes a `String` value, the cell name. So, its corresponding functor is $\Sigma_{\texttt{fetch}} = \texttt{String} \times -^{\texttt{Int}}$.

The representation of all operations is obtained by constructing the coproduct of their respective functors $\Sigma = (P_0 \times -^{N_0}) + \ldots + (P_n \times -^{N_n})$.

Operation Clauses as Σ-Algebras Each of the operation clauses in a monadic handler gives an algebra for Σ_i, where i is the operation of interest. For example, the clause

```
| fetch (p: String,k: Int -> B) -> b: B                    (c_fetch)
```

is represented by $\langle \mathtt{B}, c_{\mathtt{fetch}} : \Sigma_i \mathtt{B} \to \mathtt{B} \rangle$, a Σ_i-algebra. To refer to the function defined by this clause, which is `λ(p: String,k: Int -> B). b`, we indicate its name after the clause in brackets.

The combination of all operation clauses

```
| op_i (p_i: P_i,k_i: N_i -> B) -> b: B                    (c_i)
```

forms the Σ-algebra $\langle \mathtt{B}, c = [c_0, \ldots, c_n] : \Sigma \mathtt{B} \to \mathtt{B} \rangle$.

The value clause

```
| val (a: A) -> b: B                                      (v)
```

defines the function $v : \mathtt{A} \to \mathtt{B} = \lambda\mathtt{(a: A). b}$.

Handling Rules as Equations Evaluating `handle x with h`, given a handler `h`, requires both the value and operation rules. For example, if `h` is defined as:

```
h = handler
| val (a: A) -> ...: B                                    (v)
| op_i (p_i: P_i,k_i: N_i -> B) -> ...: B                 (c_i)
```

The *value rule* triggers when no operations are left in a fully evaluated `x`, usually in the form of `return y`. The result is defined as:

```
handle (x: A) with h = v x
```

The *operation rule* triggers when the evaluated computation is an operation $\mathtt{op}_i$. The result is defined as:

```
handle (op_i (p: P_i,k: N_i -> Σ*A)) with h
= c_i p (λn. handle (k n) with h)
```

Here, the structure $\Sigma^* A$ represents a computation built from operations present in Σ, which aims to return a value of type A. The parameter `k` denotes the *continuation* of the operation call. For example, in:

```
taskExample = do
  a1 <- fetch "A1"
  a2 <- fetch "A2"
  return (a1 + a2)
```

the continuation `k` of `fetch "A1"` is the function:

```
λ(a1: Int). do
  a2 <- fetch "A2"
  return (a1 + a2)
```

Syntax Constructors Desugaring of the `do`-notation is possible with the constructors $\mathtt{val}_A$ and $\mathtt{op}_A$. The former $\mathtt{val}_A$ embeds an evaluated value of type A into $\Sigma^* A$, and the latter $\mathtt{op}_A$ embeds an operation into $\Sigma^* A$. With these constructors, the computation `taskExample` is defined as:

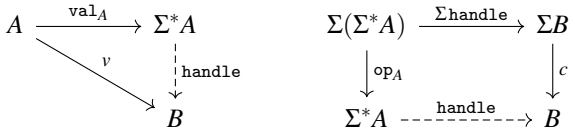
```
taskExample =
opInt (fetch ("A1",(λ(a1: Int).
  opInt (fetch ("A2",(λ(a2: Int).
    valInt (a1 + a2)
  )))
)))
```

These two constructors enable expressing the handling rules as pointfree equations. The pointfree value and operation rules are respectively: $\texttt{handle} \circ \texttt{val}_A = v$ and $\texttt{handle} \circ \texttt{op}_A = c_i \circ \Sigma\texttt{handle}$.

Handlers for Free Algebras The elements from the previous section enable viewing monadic handlers as free algebras:

Definition 3.1 (*Free* Σ*-Algebra*)
A free Σ-algebra on A in $\mathscr{C}$ consists of an object $\langle \Sigma^* A, \texttt{op}_A : \Sigma(\Sigma^* A) \to \Sigma^* A \rangle$ in Σ-$Alg(\mathscr{C})$ together with a morphism $\texttt{val}_A : A \to \Sigma^* A$ in $\mathscr{C}$ such that for any $\langle B, c : \Sigma B \to B \rangle$ in Σ-$Alg(\mathscr{C})$ and morphism $v : A \to B$ in $\mathscr{C}$, there exists a unique algebra morphism $\texttt{handle} : \langle \Sigma^* A, \texttt{op}_A \rangle \to \langle B, c \rangle$ in Σ-$Alg(\mathscr{C})$ with $\texttt{handle} \circ \texttt{val}_A = v$.

The diagrams for the conditions are:



The diagram on the left-hand side is the condition mentioned in the definition and corresponds to the value rule equation. The diagram on the right-hand side is the condition for a morphism in Σ-$Alg(\mathscr{C})$, namely a Σ-algebra homomorphism, and corresponds to the operation rule equation.

3.3 Monoids in Monoidal Categories

Rivas & Jaskelioff (2017) present a framework for different classes of side effects as (generalized) monoids in various monoidal categories. We reintroduce the relevant definitions relating to monoidal categories in the following paragraphs.

Monoidal Category A monoidal category is a category which contains a notion of monoids generalizing the monoids in *Set*, replacing the cartesian product with a general bifunctor.

Definition 3.2 (*Monoidal Category*)
A monoidal category is a tuple $(\mathscr{D}, \otimes, I, \alpha, \lambda, \rho)$, consisting of

a) a category $\mathscr{D}$
b) a bifunctor $\otimes : \mathscr{D} \times \mathscr{D} \to \mathscr{D}$ (also called the *tensor*)
c) a designated object I of $\mathscr{D}$

d) three natural isomorphisms

$$\begin{aligned} \alpha_{A,B,C} &: A \otimes (B \otimes C) \to (A \otimes B) \otimes C \\ \lambda_A &: I \otimes A \to A \\ \rho_A &: A \otimes I \to A \end{aligned}$$

such that the following diagrams commute:

$$\begin{array}{ccccc} A \otimes (B \otimes (C \otimes D)) & \xrightarrow{\alpha} & (A \otimes B) \otimes (C \otimes D) & \xrightarrow{\alpha} & ((A \otimes B) \otimes C) \otimes D \\ \downarrow{\scriptstyle A \otimes \alpha} & & & & \uparrow{\scriptstyle \alpha \otimes D} \\ A \otimes ((B \otimes C) \otimes D) & & \xrightarrow[\alpha]{} & & (A \otimes (B \otimes C)) \otimes D \end{array}$$

$$\begin{array}{ccc} A \otimes (I \otimes B) & \xrightarrow{\alpha} & (A \otimes I) \otimes B \\ & {\scriptstyle A \otimes \lambda} \searrow \quad \swarrow {\scriptstyle \rho \otimes B} & \\ & A \otimes B & \end{array}$$

When it is clear from the context, we omit α, ρ and λ.

Monoid in Monoidal Category A monoid in a monoidal category is a generalization of monoids in *Set*. The monoids from high-school algebra coincide with the notion of monoids in *Set* with cartesian product. However, from this point on, whenever we mention *monoid*, we mean the more general concept *monoid in monoidal category*.

Definition 3.3 (*Monoid in Monoidal Category*)
A monoid in a monoidal category is a tuple (M, e, m) where M is an object in a monoidal category $(\mathscr{D}, \otimes, I, \alpha, \lambda, \rho)$. The unit $e : I \to M$ and the multiplication $m : M \otimes M \to M$ are morphisms in $\mathscr{D}$ such that the following diagrams commute:

$$\begin{array}{ccccc} (M \otimes M) \otimes M & & \xrightarrow{m \otimes M} & & M \otimes M \\ \uparrow{\scriptstyle \alpha} & & & & \downarrow{\scriptstyle m} \\ M \otimes (M \otimes M) & \xrightarrow[M \otimes m]{} & M \otimes M & \xrightarrow[m]{} & M \end{array} \qquad \begin{array}{ccc} M \otimes M & \xleftarrow{M \otimes e} & M \otimes I \\ {\scriptstyle e \otimes M} \uparrow & \searrow^{m} & \downarrow{\scriptstyle \rho} \\ I \otimes M & \xrightarrow[\lambda]{} & M \end{array}$$

Exponentials for Monoidal Categories The characterizing isomorphism of exponentials can be generalized with tensors instead of products. This results in the isomorphism: $\lfloor - \rfloor : \mathscr{D}(X \otimes B, A) \cong \mathscr{D}(X, A^B) : \lceil - \rceil$. The evaluation morphism is generalized to $ev_A : A^B \otimes B \to A = \lceil A^B \rceil$.

Example 3.1
The main examples of monoidal categories we consider are:

1. The category of endofunctors, $End_\circ$, with functor composition $(F \circ G)A = F(GA)$ as tensor and the identity functor *Id* as designated object. This monoidal category is *strict*, neaning that α, λ and ρ are identities. Monoids in this monoidal category are known as monads.

2. The category of endofunctors $End_\star$, with Day convolution $(F \star G)A = \int^Z F(A^Z) \times GZ$ as tensor and the identity functor Id as designated object. Monoids in this monoidal category are known as applicative functors. We use this alternative presentation to aid readability in the code samples, it is isomorphic to the more traditional presentation $\int^{X,Y} FX \times GY \times A^{(X \times Y)}$ in our setting.
3. The category of strong profunctors, *SPro*, with profunctor composition $(P \otimes Q)(A,B) = \int^Z P(A,Z) \times Q(Z,B)$ as tensor and the *Hom* profunctor as designated object. Monoids in this monoidal category are known as arrows.

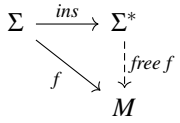
Category of Monoids For a monoidal category $(\mathscr{D}, \otimes, I)$, we have the category of monoids $Mon(\mathscr{D})$ which consists of all monoids (M,e,m) in that monoidal category and monoid homomorphisms between them.

4 Handlers for Free Monoids

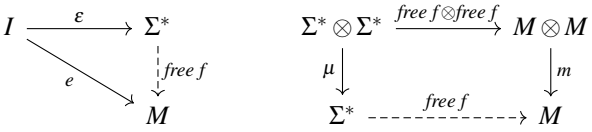
In order to include other classes of effects, this section derives a notion of handlers for free monoids. We begin by recalling the definition of the free monoid on an object:

Definition 4.1 (*Free Monoid*)
A free monoid on an object Σ in a category $\mathscr{D}$ consists of an object $(\Sigma^*, \varepsilon, \mu)$ in $Mon(\mathscr{D})$ together with a morphism $ins : \Sigma \to \Sigma^*$ in $\mathscr{D}$ such that for any (M,e,m) in $Mon(\mathscr{D})$ and morphism $f : \Sigma \to M$ in $\mathscr{D}$, there exists a unique morphism $free\, f : (\Sigma^*, \varepsilon, \mu) \to (M,e,m)$ in $Mon(\mathscr{D})$ with $free\, f \circ ins = f$.

The condition represented in a diagram is:



The diagrams corresponding to the monoid homomorphism condition, a morphism in $Mon(\mathscr{D})$, are:



More categorically, free monoids arise as the left adjoint to the forgetful functor from $Mon(\mathscr{D})$ to $\mathscr{D}$.

4.1 Monoidal Handlers

Monoidal Basis We can interpret the carrier Σ^* of a free monoid as the syntax of computations. Instead of using `val` and `op` from Section 3.2, we can construct programs in this *monoidal* syntax from the constructors ε, μ and *ins* provided by the free monoid.

Example 4.1

In the following example, we set the category $\mathscr{D}$ as $End_\circ(\mathscr{C})$, the monoidal category of monads on $\mathscr{C}$. From the general monoidal syntax (left) follows the specialized syntax (right) for this setting:

$$\begin{array}{lll} \varepsilon & : I \to \Sigma^* & \varepsilon_A \quad : A \to \Sigma^* A \\ \mu & : \Sigma^* \otimes \Sigma^* \to \Sigma^* & \rightsquigarrow \quad \mu_A \quad : \Sigma^*(\Sigma^* A) \to \Sigma^* A \\ ins & : \Sigma \to \Sigma^* & ins_A \quad : \Sigma A \to \Sigma^* A \end{array}$$

The constructor ε_A is the same as $\texttt{val}_A$: it embeds a value into a computation. Constructor ins_A embeds an operation into a computation. To embed an operation returning another computation, of type $\Sigma(\Sigma^* A)$, we use naturality of ins_A to obtain $ins_{\Sigma^* A} : \Sigma(\Sigma^* A) \to \Sigma^*(\Sigma^* A)$. Lastly, μ_A converts a computation returning a computation into a flat computation.

Consider the `taskExample` computation again:

```
taskExample = do
  a1 <- fetch "A1"
  a2 <- fetch "A2"
  return (a1 + a2)
```

which was constructed using `val`/`op` as:

```
example =
op_Int (fetch ("A1",(λ(a1: Int).
  op_Int (fetch ("A2",(λ(a2: Int).
    val_Int (a1 + a2)
  )))
)))
```

The same program can be constructed with the monoidal constructors, namely $\varepsilon/\mu/ins$:

```
μ_Int (ins_Σ*Int (fetch ("A1",(λ(a1: Int).
  μ_Int (ins_Σ*Int (fetch ("A2",(λ(a2: Int).
    ε_Int (a1 + a2)
  ))))
))))
```

Monoidal Handler Monoidal programs are interpreted by monoidal handlers. Monoidal handlers are defined by a clause for the constructors ε and μ, and one other clause for each operation; each clause evaluates programs to a monoid (M, e, m). The unit $e : I \to M$ and multiplication $m : M \otimes M \to M$ of this monoid are defined by the clauses for the ε and μ constructor respectively. This definition is expected to satisfy the monoid laws, but the notation does not enforce this. All operation clauses are combined to define the morphism $f : \Sigma \to M$, which interprets the constructor ins. This makes $free\, f$, the unique monoid morphism induced by a morphism $f : \Sigma \to M$, the handler construct for monoidal programs.

In the example, monads on $\mathscr{C}$, the clauses to define are:

```
mh = mhandler
| ε (a: A) -> ...: MA                    (e_A)
```

```
| μ (mma: M(MA)) -> ...: MA                    (m_A)
| op_i (p_i: P_i,k: N_i -> A) -> ...: MA       (f_A)
```

Notably, the handling construct is named `mhandler`, as opposed to `handler`, to signify a monoidal handler.

The evaluation rules are determined by the conditions in the free monoid definition. The ε rule is similar to the `val` rule.

```
mhandle (x: A) with mh = e_A x
```

The μ rule forwards the handling to both Σ^* structures, and then combines the result using multiplication m from the monoid (M, e, m).

```
mhandle (s: Σ*(Σ*A)) with mh
= m_A (mhandle (Σ*(mhandle _ with mh) s) with mh)
```

The *ins* rule interprets an operation $\mathtt{op}_i$ with a function f.

```
mhandle (op_i (p: P_i) (k: N_i -> A)) with mh
= f_A (p, k)
```

Example 4.2

As an example, we give the monoidal implementation for `fetchConsole`, which interprets `fetch` operations to user queries on the console. The monoid for `IO` is assumed to be defined in the internal functions $\mathtt{return}_{\mathtt{IO}}$ and $\mathtt{join}_{\mathtt{IO}}$.

```
mFetchConsole : A ! {fetch, M} => IO A
mFetchConsole = mhandler
| ε (a: A) -> return_IO a
| μ (mma: IO(IO A)) -> join_IO mma
| fetch (cell: String, k: Int -> A) -> do
    print("cell: " ++ cell)
    x <- readLn
    return (k x)
```

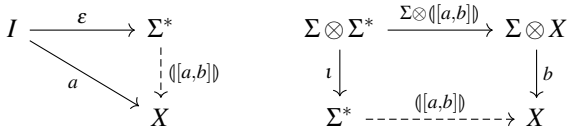
4.2 Inductive Handlers

Initial Algebra Basis The free monoid can be represented constructively as the initial algebra of the $I + \Sigma \otimes -$ functor. Concretely, this gives us an alternative set of constructors: $\varepsilon : I \to \Sigma$ and $\iota : \Sigma \otimes \Sigma^* \to \Sigma^*$. These morphisms are the two elements of the initial algebra $[\varepsilon, \iota] : I + \Sigma \otimes \Sigma^* \to \Sigma^*$.

Using this alternative syntax, the `example` computation is constructed as:

```
example =
ι_Int (fetch ("A1",(λ(a1: Int).
  ι_Int (fetch ("A2",(λ(a2: Int).
    ε_Int (a1 + a2)
  )))
)))
```

Initial Algebra Handler This alternative basis derives its handler by using the unique algebra homomorphism from the initial algebra. This unique morphism is denoted $(\!|[a,b]|\!) : \Sigma^* \to X$ for a morphism $a : I \to X$ and $b : \Sigma \otimes X \to X$. It is the unique morphism for which the following diagrams commute:



This results in a new handling construct `ihandler`. For the monads on $\mathscr{C}$ example, it requires the following clauses:

```
ih = ihandler
| ε (a: A) -> ...: XA                                   (e_A)
| op_i (p: P_i,k: N_i -> XA) -> ...: XA                 (g_A)
```

which has no laws attached.

The evaluation rules follow from the algebra homomorphism conditions.
The ε rule is unchanged:

```
ihandle (x: A) with ih = e_A x
```

The ι rule differs slightly from the *ins* rule, it handles operations returning a computation Σ^*A instead of a value A. Thus it forwards the handling before combining the results.

```
ihandle (op_i (p: P_i,k: N_i -> Σ*A)) with ih
= g_A (p, λn. ihandle (k n) with ih)
```

Example 4.3
The inductive handler for the `fetchConsole` example is implemented as:

```
iFetchConsole : A ! {fetch, M} => IO A
iFetchConsole = ihandler
| ε (a: A) -> return_IO a
| fetch (cell: String,k: Int -> IO A) -> do
    print("cell: " ++ cell)
    x <- readLn
    k x
```

4.3 Expressiveness of Monoidal and Inductive Handlers

Both the free monoid and initial algebra bases have an equal expressiveness. Either can present the interface of the other. There are also two properties to ensure the consistency between each basis: the round-trip and coherency properties. The former requires that a round-trip conversion, namely converting to one basis and then back to the other basis, is the identity. The latter requires that the handlers behave in a consistent manner in both bases.

Initial Algebra Basis from Free Monoid Basis The following definitions represent the constructor/handler from the initial algebra basis:

Table 2: Overview of Handlers

	Free Algebra	**Free Monoid (*free*)**	**Free Monoid ($(\!\lvert -\rvert\!)$)**
syntax/ computation	$\mathtt{val}_A : A \to \Sigma^* A$ $\mathtt{op}_A : \Sigma(\Sigma^* A) \to \Sigma^* A$	$\varepsilon : I \to \Sigma^*$ $ins : \Sigma \to \Sigma^*$ $\mu : \Sigma^* \otimes \Sigma^* \to \Sigma^*$	$\varepsilon : I \to \Sigma^*$ $\iota : \Sigma \otimes \Sigma^* \to \Sigma^*$
handler	Σ-algebra: $\langle B, c = [c_0, \ldots, c_n] : \Sigma B \to B \rangle$ $v : A \to B$	monoid: (M, e, m) $f = [f_0, \ldots, f_n] : \Sigma \to M$	$I + \Sigma \otimes -$-algebra: $\langle X, [e, g] : I + \Sigma \otimes X \to X \rangle$ $g = \lceil [\lfloor g_0 \rfloor, \ldots, \lfloor g_n \rfloor] \rceil$
handler (clauses)	$\mid\ \mathtt{val}\ A\ \mathtt{->}\ B \quad (v)$ $\mid\ \mathtt{op}_i\ \Sigma_i B\ \mathtt{->}\ B \quad (c_i)$	$\mid\ \varepsilon\ I\ \mathtt{->}\ M \quad (e)$ $\mid\ \mathtt{op}_i\ \Sigma_i\ \mathtt{->}\ M \quad (f_i)$ $\mid\ \mu\ M \otimes M\ \mathtt{->}\ M \quad (m)$	$\mid\ \varepsilon\ I\ \mathtt{->}\ X \quad (e)$ $\mid\ \mathtt{op}_i\ \Sigma_i \otimes X\ \mathtt{->}\ X \quad (g_i)$
handling a computation	$\mathtt{handle} : \Sigma^* A \to B$	$free\, f : \Sigma^* \to M$	$(\!\lvert [e, g] \rvert\!) : \Sigma^* \to X$
handling rules	$\mathtt{handle} \circ \mathtt{val}_A = v$ $\mathtt{handle} \circ \mathtt{op}_A = c \circ \Sigma \mathtt{handle}$	$free\, f \circ \varepsilon = e$ $free\, f \circ ins = f$ $free\, f \circ \mu =$ $m \circ (free\, f \otimes free\, f)$	$(\!\lvert [e, g] \rvert\!) \circ \varepsilon = e$ $(\!\lvert [e, g] \rvert\!) \circ \iota = g \circ (\Sigma \otimes (\!\lvert [e, g] \rvert\!))$

$$
\begin{aligned}
\iota &= \Sigma \otimes \Sigma^* \xrightarrow{ins \otimes \Sigma^*} \Sigma^* \otimes \Sigma^* \xrightarrow{\mu} \Sigma^* \\
eval_X\, e &= X^X \xrightarrow{\rho^{-1}} X^X \otimes I \xrightarrow{X^X \otimes e} X^X \otimes X \xrightarrow{ev_X} X \\
(\!\lvert [e, g] \rvert\!) &= \Sigma^* \xrightarrow{free \lfloor g \rfloor} X^X \xrightarrow{eval_X\, e} X
\end{aligned}
$$

where the use of $free \lfloor g \rfloor$ is justified, since it interprets to the endomorphism monoid $(X^X, \dot{e} : I \to X^X, \dot{m} : X^X \otimes X^X \to X^X)$.

Free Monoid Basis from Initial Algebra Basis The following definitions represent the constructors/handler from the free monoid basis:

$$
\begin{aligned}
ins &= \Sigma \xrightarrow{\rho_\Sigma^{-1}} \Sigma \otimes I \xrightarrow{\Sigma \otimes \varepsilon} \Sigma \otimes \Sigma^* \xrightarrow{\iota} \Sigma^* \\
\mu &= \lceil (\!\lvert [\lfloor I \otimes \Sigma^* \xrightarrow{\lambda_{\Sigma^*}} \Sigma^* \rfloor, \\
&\qquad \lfloor (\Sigma \otimes \Sigma^{*\Sigma^*}) \otimes \Sigma^* \xrightarrow{\alpha^{-1}} \Sigma \otimes (\Sigma^{*\Sigma^*} \otimes \Sigma^*) \\
&\qquad \xrightarrow{\Sigma \otimes ev_{\Sigma^*}} \Sigma \otimes \Sigma^* \xrightarrow{\iota} \Sigma^* \rfloor] \rceil \rvert\!) \rceil \\
free\, f &= (\!\lvert [I \xrightarrow{e} M, \Sigma \otimes M \xrightarrow{(f \otimes M)} M \otimes M \xrightarrow{m} M] \rvert\!)
\end{aligned}
$$

where (M, e, m) is a monoid.

Properties The round-trip properties are obtained by deriving the definition of the constructors, from the other basis, as a property. The proofs of these properties are in Appendix B.3, B.4, C.3, C.4 and C.5.

The coherency properties are obtained by deriving the evaluation rules of the handler, from the other basis, as a property. The proofs of these properties are in Appendix B.5 and C.6.

4.4 Expressiveness of Monoidal and Free Algebra Handlers

The monoidal handler for monads is slightly different from the original handlers based on free algebras. At first sight it seems that the carriers of the two handlers only coincide when the carrier B of the free algebra handler is of the form MA where M is the monad carrier of the monoidal handler and the free algebra handler is natural in A. This might suggest that the monoidal handler is less expressive than its free algebra counterpart, which is not restricted to this particular form of carrier. However, both handlers are equally expressive.

The continuation monad enables translating the `handle` interface in terms of `ihandle` or `mhandle`. The continuation monad is defined as X^{X^A}, which is the type `(A -> X) -> X` in $\mathscr{C}$. The translation of `handle` in terms of `ihandle` is:

```
handle x with
  (handler
   | val (a: A) -> ...: X                                   (v)
   | op_i (p_i: P_i,k: N_i -> X) -> ...: X                  (c_i)
  )
= (ihandle x with
    (ihandler
     | ε (a: A)
         -> λ(f: A -> X). f a
     | op_i (p_i: P_i,k: N_i -> ((A -> X) -> X))
         -> λ(f:A -> X). c_i (p_i, λ(n: N_i). k n f)
    )) v
```

where the `ihandler` interprets to the `(A -> X) -> X` type and is then evaluated with v.

The consistency property of this translation is proven in Appendix D.1 and D.2.

4.5 Summary

An overview of the free algebra and free monoid approach can be seen in Table 2. Since free monoid handlers are equivalent to free algebra handlers, when the former is instantiated for monads, we consider it a natural extension of the latter approach. By instantiating the free monoid handlers for other effects such as applicative functors or arrows it is possible to define handlers for non-monadic effects, which we call non-monadic handlers.

5 Non-Monadic Handlers

This section explores the monoidal handlers for applicatives and arrows. We instantiate the monoidal categories accordingly and specialize the definitions of the derived handlers.

5.1 Applicative Handlers

Applicative computations are very restricted in what they can express. They can only express computations which correspond to a list of operations to execute, returning a value combining the results of these operations.

Instantiating the syntax morphisms for $End_\star(\mathscr{C})$ with Day convolution $\star$ gives:

$$\begin{array}{lcl} \varepsilon_A & : & A \to \Sigma^* A \\ \iota_A & : & (\Sigma \star \Sigma^*)(A) \to \Sigma^* A \\ & = & (\int^Z \Sigma(Z \to A) \times \Sigma^* Z) \to \Sigma^* A \end{array}$$

or an alternative formulation for ι is $\iota_{Z,A} : \Sigma(Z \to A) \times \Sigma^* Z \to \Sigma^* A$.

This results in the following inductive handler interface:

```
ihAp = ihandler
| ε (a: A) -> ...: FA
| opᵢ (p: Pᵢ, k: Nᵢ -> Z -> A, l: FZ) -> ...: FA
```

and the following monoidal handler interface:

```
mhAp = mhandler
| ε (a: A) -> ...: FA
| μ (mza: F(Z -> A), mz: FZ) -> ...: FA
| opᵢ (pᵢ: Pᵢ, k: Nᵢ -> A) -> ...: FA
```

To show how this corresponds to the intuitive explanation, we will transform the following applicative program into the primitive syntax.

```
taskExample = do
  a1 <- fetch "A1"
  a2 <- fetch "A2"
  return (a1 + a2)
```

We introduce the `fetch "A1"` operation with $\iota_{Z,A}$. Specifically for this operation, $\iota_{Z,A}$ has the form $\texttt{String} \times (Z \to A)^{\texttt{Int}} \times \Sigma^* Z \to \Sigma^* A$. The first parameter, of type `String`, is the cell identifier `"A1"`. The second parameter, of type `Int -> Z -> A`, is a function describing how to combine the current cell value with the result from the rest of the computation. The third parameter $\Sigma^* Z$ is the rest of the computation. In this case, we take Z to be `Int`, since that is the result of the remaining `fetch "A2"` operation, and A to be `Int`, since that is the final result type of the computation. This continuation is `a1 + z`, where `a1` is the result of `fetch "A1"` and `z` is the result of the remaining computation.

The `fetch "A2"` operation follows a similar pattern. But now, Z is the type `()` since the rest of the computation does not contain any more operations. The continuation for this operation is `a2`, since it is passed to the continuation for `fetch "A1"`.

This results in the following desugaring of the applicative computation:

```
ι_int,int (fetch ("a1",λ(a1:int).λ(z:int).a1 + z)) (
  ι_(),int (fetch ("a2",λ(a2:int).λ(z:()).a2))) (
    ε_() ()
  )
)
```

Example 5.1

In the applicative syntax representation, all parameters and operations are immediately accessible, meaning that they are not inside a lambda-expression. Due to this we can express handlers which analyze computations, for example to return all dependencies of a task.

The inductive handler analyzes dependencies by starting from an empty list `[]` and adding the cell name of each `fetch` operation. Note that we interpret to the constant applicative functor Δ_A, where A must be a monoid, and use lifted operations, such as $\dot{:}$ and $\dot{+\!+}$, to operate on values inside this functor.

```
iAnalyzeAp : A ! {fetch, Ap} => Δ[String] A
iAnalyzeAp = ihandler
| ε (a: A) -> Δ[String] []
| fetch (cell: String, k: Int -> Z -> A, l: Δ[String] Z) -> cell :̇: l
```

The monoidal handler interprets each `fetch` operation to a list with the cell name as its only element, then the μ operation appends all of these lists together.

```
mAnalyzeAp : A ! {fetch, Ap} => Δ[String] A
mAnalyzeAp = mhandler
| ε (a: A) -> Δ[String] []
| μ (mza: Δ[String] Z -> A, mz: Δ[String] Z) -> mza +̇+ mz
| fetch (cell: String, k: Int -> A) -> Δ[String] [cell]
```

Handling `taskExample` with `iAnalyzeAp` or `mAnalyzeAp` gives the cells on which it depends:

```
λ> handle taskExample with iAnalyzeAp
["A1", "A2"]
λ> handle taskExample with mAnalyzeAp
["A1", "A2"]
```

Since applicative computations are a subset of monadic computations, we can adapt the handler `fetchConsole` to its applicative version. The inductive version is given below.

```
iFetchConsoleAp : A ! {fetch, Ap} => IO A
iFetchConsoleAp = ihandler
| ε (a: A) -> return a
| fetch (cell: String, k: Int -> Z -> A, r: IO Z) -> do
    print ("cell: " ++ cell)
    x <- readLn
    fmap (k x) r
```

The monoidal version is given below.

```
mFetchConsoleAp : A ! {fetch, Ap} => IO A
mFetchConsoleAp = mhandler
| ε (a: A) -> return a
| μ (mza: IO (Z -> A), mz: IO Z) -> do
    za <- mza
    z <- mz
    return (za z)
| fetch (cell: String, k: Int -> A) -> do
    print ("cell: " ++ cell)
    x <- readLn
    return (k x)
```

5.2 Arrow Handlers

Arrow computations are more restricted compared to monads, but more permissive than applicatives. They again express a static list of operations to execute, but the parameters which are passed to these operations can depend on result values from previous operations. To work with arrows, we require to change our view on operations from functors to profunctors.

Operations as Profunctors Signature functors, defined as $\Sigma_i B = P_i \times B^{N_i}$, can be extended to signature profunctors as $\vec{\Sigma}_i(A,B) = (P_i \times B^{N_i})^A$. For example, the profunctor version of `fetch` is $\vec{\Sigma}_{\texttt{fetch}}(A,B) = (\texttt{String} \times B^{\texttt{Int}})^A$.

Instantiating the syntax morphisms for $SPro(\mathscr{C})$ with Profunctor composition $\otimes$ gives:

$$\begin{array}{lcl} \varepsilon_{A,B} & : & B^A \to \vec{\Sigma}^*(A,B) \\ \iota_{A,B} & : & (\vec{\Sigma} \otimes \vec{\Sigma}^*)(A,B) \to \vec{\Sigma}^*(A,B) \\ & = & (\int^Z \vec{\Sigma}(A,Z) \times \vec{\Sigma}^*(Z,B)) \to \vec{\Sigma}^*(A,B) \end{array}$$

where the notation $\vec{\Sigma}$ denotes a profunctor signature. An alternative formulation of ι is $\iota_{Z,A,B} : \vec{\Sigma}(A,Z) \times \vec{\Sigma}^*(Z,B) \to \vec{\Sigma}^*(A,B)$. The output Z of the embedded operation $\vec{\Sigma}(A,Z)$ is linked to the input of the rest of the computation $\vec{\Sigma}^*(Z,B)$.

An example arrow program is given below. The program consists of two `fetch` operations, but the parameter of the second operation depends on the result of the first.

```
taskDyn = do
  a1 <- fetch "A1"
  fetch ("B" ++ show a1)
```

We can desugar this program with the more primitive arrow syntax. We introduce the `fetch "A1"` operation with $\iota_{Z,A,B}$. For this operation, it has the form $(\texttt{String} \times Z^{\texttt{Int}})^A \times \vec{\Sigma}^*(Z,B) \to \vec{\Sigma}^*(A,B)$. The first parameter, of type `A -> (String, Int -> Z)`, is the cell name and the continuation in a function with input A. The second parameter is the rest of the computation. The parameter A is the current input from the previous operations, so for the first operation this is `()`. The output Z is `Int` since we need to use the current result value of the output operation later in the computation.

For the second `fetch` operation, the input A from the previous operations is `Int`. This value can be used in the cell name passed as operation parameter. The output Z is again `Int`, since we only return the value from the last `fetch` operation.

This results in the following desugaring:

```
ι_Int,(),Int (fetch (λ(_:()).("A1",λ(a1:Int).a1))) (
  ι_Int,Int,Int (fetch (λ(i:Int).("B" ++ show i,λ(ax:Int).ax))) (
    ε_Int,Int (λ(v:Int).v)
  )
)
```

Static and Dynamic Parameters The earlier profunctor operation considered the whole cell name as a *dynamic* parameter, meaning that all information was behind the input parameter A. This is too restrictive for our use case. Instead, we split the information passed to the operation into a *static* and *dynamic* parameter. So, a signature functor $\Sigma_i B = P_i \times B^{N_i}$ can be extended to $\vec{\Sigma}_i(A,B) = S_i \times (D_i \times B^{N_i})^A$, where S_i, the static parameter, and D_i, the dynamic parameter, can be combined to create P_i.

With this change in operation, the desugaring for `taskDyn` is slightly different. The static parameter S_i is the cell column. The dynamic parameter D_i is the cell row, since the row is dependent on previous results in `taskDyn`. The static operation parameter moves to the outside of the lambda taking the A input parameter.

```
ι_{Int,(),Int} (fetch ("A", λ(_:()).("1",λ(a1:Int).a1))) (
  ι_{Int,Int,Int} (fetch ("B", λ(i:Int).(show i,λ(ax:Int).ax))) (
    ε_{Int,Int} (λ(v:Int).v)
  )
)
```

Considering the static and dynamic parameters, we get the following inductive handler interface as a result:

```
ihAp = ihandler
| ε (f: A -> B) -> ...: P(A,B)
| op_i (p: S_i, k: A -> (D_i, N_i -> Z), l: P(Z,B)) -> ...: P(A,B)
```

and the following monoidal handler interface:

```
mhAp = mhandler
| ε (a: A -> B) -> ...: P(A,B)
| μ (maz: P(A,Z), mzb: P(Z,B)) -> ...: P(A,B)
| op_i (p_i: S_i, f: A -> (D_i, N_i -> B)) -> ...: P(A,B)
```

Example 5.2
In the arrow syntax representation, the operations are accessible but the parameters to these operations are not. This is because these can be dependent on previous operation results. However, due to the distinction between static and dynamic parameters, we regain some applicative capabilities and can access the static part of the operation parameters. We use this to extract the column information from arrow computations where this is the static parameter.

As with the applicative handler, the inductive version builds the list by concatenating elements while the monoidal version creates one-element lists which are combined later.

```
iAnalyzeAr : A -> B ! {fetch, Ar} => Δ⃗_[String] (A,B)
iAnalyzeAr = ihandler
| ε (f: A -> B) -> Δ⃗_[String] []
| fetch (col: String, k: A -> (String, Int -> Z), l: Δ⃗_[String] (Z,B))
    -> col :: l

mAnalyzeAr : A -> B ! {fetch, Ar} => Δ⃗_[String] (A,B)
mAnalyzeAr = mhandler
| ε (f: A -> B) -> Δ⃗_[String] []
```

```
| μ (maz: Δ⃗[String] (A,Z) -> A,mzb: Δ⃗[String] (Z,B)) -> maz +̇+ mzb
| fetch (col: String, f: A -> (String, Int -> B)) -> Δ⃗[String] col
```

Handling `taskDyn` with `iAnalyzeAr` or `mAnalyzeAr` gives `["A", "B"]`, which are the columns on which `taskDyn` depends.

Arrow computations are also a subset of monadic computations, which means that we can adapt the handler `fetchConsole` to its arrow version. The arrow version makes use of the $\texttt{Kleisli}_M\texttt{(A,B)}$ arrow, where M is a monad. The constructor $\texttt{Kleisli}_M$ takes a function `A -> M B` and converts it to $\texttt{Kleisli}_M\texttt{(A,B)}$, while the function `runKleisli` takes a $\texttt{Kleisli}_M\texttt{(A,B)}$ value and converts it to `A -> M B`.

```
iFetchConsoleAr : A -> B ! {fetch, Ar} => Kleisli_IO(A,B)
iFetchConsoleAr = ihandler
| ε (f: A -> B) -> Kleisli_IO (λ(a: A). return (f a))
| fetch (col: String,f: A -> (String, Int -> Z),k: Kleisli_IO(Z,B))
    -> Kleisli_IO (λ(a: A). do
      let (row: String, g: Int -> Z) = f a
      print ("cell: " ++ col ++ row)
      x <- readLn
      runKleisli k (g x)
    )
```

The monoidal version is given below.

```
mFetchConsoleAr : A -> B ! {fetch, Ar} => Kleisli_IO(A,B)
mFetchConsoleAr = mhandler
| ε (f: A -> B) -> Kleisli_IO (λ(a: A). return (f a))
| μ (maz: Kleisli_IO(A,Z), mzb: Kleisli_IO(Z,B)) ->
    Kleisli_IO (λ(a: A). do
      z <- maz a
      mzb z
    )
| fetch (col: String, f: A -> (String, Int -> B)) ->
    Kleisli_IO (λ(a: A). do
      let (row: String, g: Int -> Z) = f a
      print ("cell: " ++ col ++ row)
      x <- readLn
      return (g x)
    )
```

6 Reusing Handlers and Programs

In section 5 we have seen handlers for different computation classes, interpreting programs to `IO`. There is no essential difference in how these handlers operate. For these cases a reuse of handler definitions across the computation classes can be useful. Dually, programs from different computation classes may express the same computation. For example, an applicative computation can be identical to a monadic computation that does not use the

full monadic expressiveness. Again raising the concern of reuse, but now for computation definitions.

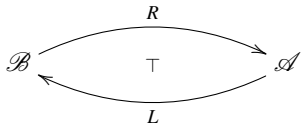
This section accomplishes both forms of reuse by means of an adjunction whose right adjoint is a lax monoidal functor. As we saw in Section 3, our model of computations is based on monoids in monoidal categories. In order to reuse monoids, we need to have a connection between the monoidal categories. The categorical ingredients for establishing such connections are twofold: *adjunctions* and *lax monoidal functors*. An adjunction is a pair of functors together with a weakened form of equivalence between the two categories. These functors are used to translate the underlying object of a monoid from one category to another, while the lax monoidal structure of the functor guarantees that the monoidal structure on top of the object is preserved. We present adjunctions and lax monoidal functors between the monoidal categories $End_\circ$, *SPro* and $End_\star$.

We will see that not every form of reuse is possible. We can only reuse programs from less expressive classes in more expressive classes, for example applicative $\rightarrow$ arrow $\rightarrow$ monad. The handler reuse opportunities are dual, that is monad $\rightarrow$ arrow $\rightarrow$ applicative.

6.1 Monoidal Functors & Adjunctions

Before explaining how to reuse handlers and programs, we introduce the two key concepts.

Adjunction The pair of functors $L : \mathscr{A} \rightarrow \mathscr{B}$ and $R : \mathscr{B} \rightarrow \mathscr{A}$ map between two monoidal categories $\mathscr{A}$ and $\mathscr{B}$. These functors are related by the adjunction $L \dashv R$:



This adjunction is characterized by a natural isomorphism on A and B:

$$[\![-]\!] : \mathscr{B}(LA, B) \cong \mathscr{A}(A, RB) : \llbracket - \rrbracket$$

(Co-)Lax Monoidal Functors If we have two monoidal categories and a functor between them, we can consider those structures that make the functor interact coherently with the monoidal structures as in the following definition.

Definition 6.1 (*Lax Monoidal Functor*)
Let $(\mathscr{A}, \otimes, I)$ and $(\mathscr{B}, \oplus, J)$ be two monoidal categories. A *lax monoidal functor* between them is

a) a functor $F : \mathscr{A} \rightarrow \mathscr{B}$
b) a morphism $\phi^0 : J \rightarrow FI$
c) a natural transformation $\phi_{A,B} : FA \oplus FB \rightarrow F(A \otimes B)$

satisfying coherence conditions with respect to unitality and associativity. There is a dual notion of *colax monoidal functor* in which the directions of ϕ^0 and $\phi_{A,B}$ are inverted.

A key property of lax monoidal functors is their mapping of monoids (M,e,m) in $\mathscr{A}$ to monoids (FM,e',m') in $\mathscr{B}$, where e' and m' are defined as:

$$\begin{array}{rcl} e' & = & J \xrightarrow{\phi^o} FI \xrightarrow{Fe} FM \\ m' & = & FM \oplus FM \xrightarrow{\phi_{M,M}} F(M \otimes M) \xrightarrow{Fm} FM \end{array}$$

Monoidal Adjunction When there is an adjunction $L \dashv R$, there is a bijection between the lax monoidal structures on R and colax monoidal structures on L, i.e. if R is lax monoidal, then L has a colax monoidal structure, and dually, if L is colax monoidal, then R has a lax monoidal functor. We refer to such adjunctions as *monoidal adjunctions* here.

6.2 Transformation-based Approach

This section presents our approach, based on transforming the programs/handlers in one category to programs/handlers in the other category.

Let $(\mathscr{A},\otimes,I)$ and $(\mathscr{B},\oplus,J)$ be two monoidal categories with an adjunction $L \dashv R$ such that $R : \mathscr{B} \longrightarrow \mathscr{A}$ is lax monoidal.

Notation To prevent confusing the category of signatures and of programs, we use the following notational convention.

Signatures that originate in category $\mathscr{A}$ are denoted Ξ, and free monoids are superscripted with $\otimes$: $\Xi^{\otimes}$, rather than Ξ^*. Signatures from category $\mathscr{B}$ are denoted Σ and free monoids have a superscript $\oplus$: $\Sigma^{\oplus}$.

Also, $\mathscr{B}$ is the category with less expressive handlers, but more expressive programs. The opposite is true for $\mathscr{A}$: it has more expressive handlers, but less expressive programs.

Algebra Conversion We now show how to convert a handler algebra $[i : J \to X, a : \Sigma \oplus X \to X]$ in $\mathscr{B}$ to a handler algebra $[i' : I \to RX, a' : R\Sigma \otimes RX \to RX]$ in $\mathscr{A}$. We precompose the R-mapped algebra morphisms with ϕ^0 and $\phi_{\Sigma,X}$ respectively to obtain the converted algebra.

$$\begin{array}{rcl} i' & = & I \xrightarrow{\phi^0} RJ \xrightarrow{Ri} RX \\ a' & = & R\Sigma \otimes RX \xrightarrow{\phi_{\Sigma,X}} R(\Sigma \oplus X) \xrightarrow{Ra} RX \end{array}$$

In other words, we can obtain an $\mathscr{A}$-handler $h' : (R\Sigma)^{\otimes} \to RX = (\!|[i',a']|\!)$ from a $\mathscr{B}$-handler $h : \Sigma^{\oplus} \to X = (\!|[i,a]|\!)$.

For example, using this algebra conversion we can convert the handler `iFetchConsoleAr` from Example 5.2 to a handler equivalent to `iFetchConsoleAp` from Example 5.1, allowing the handling of applicative computations with `iFetchConsoleAr`.

Program Conversion The dual conversion, using the same elements, takes a program written in terms of the free monoid $\Xi^{\otimes}$ on the signature Ξ in $\mathscr{A}$ to a program $R\left((L\Xi)^{\oplus}\right)$:

$$\begin{array}{rcl} [\![ins_{L\Xi}]\!] & : & \Xi \to R((L\Xi)^{\oplus}) \\ convert & = & \Xi^{\otimes} \xrightarrow{free\,[\![ins_{L\Xi}]\!]} R((L\Xi)^{\oplus}) \end{array}$$

Using *free* requires that $R((L\Xi)^{\otimes})$ induces a monoid, which it does since R is a lax monoidal functor.

For example, using this program conversion we can convert the program `taskExample` represented with applicative syntax from Section 5.1 to a representation with the arrow syntax, allowing the use of arrow handlers on `taskExample`.

Overview Now, we assume that the two signatures Ξ in $\mathscr{A}$ and Σ in $\mathscr{B}$ are related by a morphism $f : L\Xi \to \Sigma$, which through the adjunction has transpose $\lfloor\!\lfloor f \rfloor\!\rfloor = g : \Xi \to R\Sigma$. There are two conversions that enable alternative paths for handling a program $\Xi^{\otimes}$ to a result RX,

$$\begin{array}{ccccc} \Xi^{\otimes} & \xrightarrow{convert} & R((L\Xi)^{\oplus}) & \xrightarrow{R(hoist\, f)} & R(\Sigma^{\oplus}) \\ {\scriptstyle hoist\, g}\downarrow & & & & \downarrow{\scriptstyle Rh} \\ (R\Sigma)^{\otimes} & & \xrightarrow[h']{} & & RX \end{array}$$

where *hoist* is defined on a morphism $x : A \to B$ as

$$hoist\ x : A^* \to B^* = (\!|[\varepsilon, \iota \circ (x \otimes B^*)]|\!)$$

Appendix E proves that both paths of this diagram are equivalent to the fused morphism $(\!|[i', a' \circ (g \otimes RX)]|\!)$. This means that converting a handler and handling a program, or converting this program and then handling it with the handler, give the same result. The fused morphism is a likely optimization to the previous two, more intuitive, approaches.

6.3 Instances

This section instantiates the approach for three conversions: applicative $\leftrightarrow$ arrow, arrow $\leftrightarrow$ monad and applicative $\leftrightarrow$ monad.

Applicative $\leftrightarrow$ Arrow For this conversion, we are in the setting of the adjunction between the categories *SPro* and *End*. The left adjoint functor $\text{-}_! : End \to SPro$ is defined as $F_!(A,B) = F(B^A)$, which creates a (strong) profunctor by putting a contravariant argument inside the transformed functor. The right adjoint functor $\text{-}^* : SPro \to End$ is defined as $P^*(A) = P(\texttt{()}, A)$, which transforms a profunctor into a functor by putting the unit value `()` in the contravariant position[1].

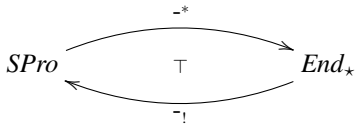
$$SPro \underset{\text{-}_!}{\overset{\text{-}^*}{\rightleftarrows}} End \quad (\top)$$

The -^* functor has lax monoidal structure, characterized by the following morphisms:

[1] In the Haskell module `Control.Arrow` it is called `ArrowMonad`, `https://hackage.haskell.org/package/base-4.10.0.0/docs/src/Control.Arrow.html#ArrowMonad`.

$$\phi_{P,Q} : P^* \star Q^* \to (P \otimes Q)^*, \qquad \phi^0 : Id \to Hom^*.$$

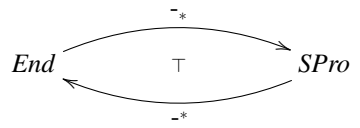
This results in the *Cayley* (monoidal) adjunction introduced by Pastro & Street (2007).



We can apply the handler and program conversion to this adjunction. Given a handler described by morphisms of type $Hom \to P$ and $\Sigma \otimes P \to P$, we obtain the converted handler in $End_\star$ of the form $Id \to P^*$ and $\Sigma^* \star P^* \to P^*$. For a signature Ξ in $End_\star$, the program conversion is implemented by the morphism $convert^\star_\Xi : \Xi^\star \to \left(\Xi^\otimes_!\right)^*$.

This monoidal adjunction is the basis to justify that an applicative functor is a *static arrow* (Lindley *et al.*, 2011): using -* and -! we can write an idempotent comonad on *SPro*. Monoids in *SPro* which carry a coalgebra structure for this comonad are applicative functors. This last affirmation is captured at the code level by an arrow `a x y` which additionally satisfies the equation `a x y = a () (x -> y)` (Rivas, 2018).

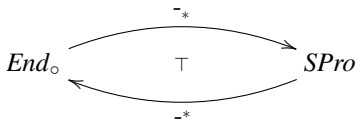
Arrow $\leftrightarrow$ Monad For this conversion, the adjunction is again between the categories *SPro* and *End*. The left adjoint is the -* functor, which was present as right adjoint in the previous paragraph. The right adjoint functor $-_* : End_\circ \to SPro$ is defined as $F_*(A,B) = (FB)^A$, which creates a profunctor by putting a contravariant argument on the transformed functor as an exponent.



The $-_*$ functor has lax monoidal structure, characterized by the following morphisms:

$$\psi_{F,G} : F_* \otimes G_* \to (F \circ G)_*, \qquad \psi^0 : Hom \to Id_*.$$

When seen as a monoidal functor, $-_*$ was called KLEISLI (Rivas & Jaskelioff, 2017). We instead use the name *Kleisli* to refer to the monoidal adjunction arising due this functor.

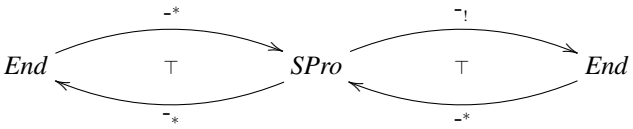


This adjunction results in a handler algebra conversion: given $Id \to X$ and $\Sigma \circ X \to X$, it forms $Hom \to X_*$ and $\Sigma_* \otimes X_* \to X_*$. It also results in a program conversion morphism $convert^\otimes_\Xi : \Xi^\otimes \to ((\Xi^*)^\circ)_*$ for a signature signature Ξ in *SPro*.

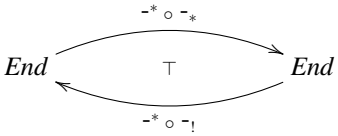
As in the previous case, this monoidal adjunction is the basis to justify that a monad is is a *higher-order arrow* (Lindley *et al.*, 2011): using -* and $-_*$ we can write an idempotent monad on *SPro*. Monoids in *SPro* which carry an algebra structure for this monad are

monads. This last affirmation is captured at the code level by an arrow `a x y` which additionally satisfies the equation `a x y = x -> a () y` (Rivas, 2018).

Applicative $\leftrightarrow$ Monad The two previous adjunctions can be composed:



It is a classical result that a composition of two adjunctions gives a new adjunction:



In this particular case, we can calculate the action of these composed functors on objects,

$$(\text{-}^* \circ \text{-}_!)(F)(Z) = (F_!)^*(Z) = F_!((),Z) = F(Z^{()}) \cong F(Z), \tag{1}$$

$$(\text{-}^* \circ \text{-}_*)(F)(Z) = (F_*)^*(Z) = F_*((),Z) = F(Z)^{()} \cong F(Z), \tag{2}$$

which means that the adjunction is just the trivial identity adjunction.

However, at the level of monoidality, the situation is more interesting. The lower morphism has a lax monoidal structure inherited from $(\psi_{F,G}, \psi^0)$ and $(\phi_{P,Q}, \phi^0)$ as $\chi = {\psi_{F,G}}^* \circ \phi_{P,Q}$ and $\chi^0 = \left(\psi^0\right)^* \circ \phi^0$. This monoidal structure on identity is not trivial, and it was called DAY previously (Rivas & Jaskelioff, 2017).

Given $i : Id \to X$ and $a : \Sigma \circ X \to X$, we have the algebra conversion

$$i' \quad = \quad Id \xrightarrow{\phi^0} Hom^* \xrightarrow{\left(\psi^0\right)^*} (Id_*)^* \xrightarrow{(i_*)^*} (X_*)^*,$$

$$a' \quad = \quad (\Sigma_*)^* \star (X_*)^* \xrightarrow{\phi_{P,Q}} (\Sigma_* \otimes X_*)^* \xrightarrow{{\psi_{F,G}}^*} ((\Sigma \circ X)_*)^* \xrightarrow{(a_*)^*} (X_*)^*,$$

and given a signature Ξ in $End_\star$, we have the program conversion

$$convert = \Xi \xrightarrow{convert^\star_\Xi} ((\Xi_!)^\otimes)^* \xrightarrow{(convert^\otimes_{\Xi_!})^*} \left(\left(\left((\Xi_!)^*\right)^\circ\right)_*\right)^*.$$

These two can be simplified using Equations 1 and 2. The handler algebra conversion transforms a monad action with components of type $Id \to X$ and $\Sigma \circ X \to X$ into an applicative action with components of type $Id \to X$ and $\Sigma \star X \to X$. The program conversion results in a morphism $\Xi^\star \to \Xi^\circ$.

7 Use of Non-Monadic Handlers in Build Systems

In this section we relate the original build system use case to the approach used in this paper. First, we relate the approach used in the original use case (Mokhov *et al.*, 2018) and the framework used in this paper. Second, we discuss possibilities enabled by arrow build systems.

Code This section contains Haskell code, which is formatted in colored teletype text.

7.1 Task in Original Build System Model

A build system consists of various `Task`s. Each task is a representation of how to build a certain artifact. Artifacts are uniquely determined by their key of type `k` and building them results in a value of type `v`. For a traditional build system based on a file system, `k` might be file paths and `v` might be file contents. In the case of spreadsheet, `k` is the cell name and `v` is the cell content.

The `Task` type itself is represented as a function:

```
newtype Task c k v = Task {
  run :: forall f. c f => (k -> f v) -> f v
}
```

This function yields a result of type `v` wrapped in type constructor `f` since building the artifact is likely to have side effects. The type constructor `f` is kept abstract to allow for interpreting the same task with different kinds of side-effects. Similarly, the constraint imposed on `f` is a parameter that can be instantiated in multiple ways, such as with `Monad` or `Applicative`. The task may depend on other artifacts. To access these, it receives a `fetch` operation of type `k -> f v`. This `fetch` operation either simply retrieves the dependency's value, if it is on hand, or first builds it, if it is not.

Task Monad A monadic task is represented by a `Task Monad`. In `testTM` we first fetch the cell `"C1"` and then depending on its value we fetch either cell `"A1"` or `"A2"`.

```
testTM :: Task Monad String Integer
testTM = Task $ \fetch -> do
  c1 <- fetch "C1"
  if c1 == 1
    then fetch "A1"
    else fetch "A2"
```

The computation `testTM` utilizes the yet unspecified effect `f` from the `Task` type. The actual effects will be determined once we decide how to run them, which is done by giving an interpretation. We give an interpretation by supplying the handler as a parameter. For example, the handler `fetchConsole` interprets each `fetch` operation in a `Task` as a request on the console to the user.

```
fetchConsole :: String -> IO Integer
fetchConsole cell = do
  print ("cell: " ++ cell)
  readLn
```

Running `run testTM fetchConsole` requests cell information twice: first the data in cell `"C1"` is requested, which then determines if we have to pass the data in cell `"A1"` or `"A2"`.

Task Applicative An applicative task is represented by a `Task Applicative`, for example `testTI`.

```
testTI :: Task Applicative String Integer
testTI = Task $ \fetch -> do
  a1 <- fetch "A1"
  a2 <- fetch "A2"
  return (a1 + a2)
```

The above example uses the `ApplicativeDo` extension to sugar applicative computations in Haskell (Marlow *et al.*, 2016). The desugared version is as follows:

```
testTI' :: Task Applicative String Integer
testTI' = Task $ \fetch ->
  (+) <$> fetch "A1" <*> fetch "A2"
```

For example, we can write the handler `fetchStatic` which gathers all static information, in this case task dependencies, into a list. This utilizes the `Applicative` instance of `Const` to fully determine the handler behaviour.

```
fetchStatic :: a -> Const [a] b
fetchStatic a = Const [a]

instance (Monoid c) => Applicative (Const c) where
  pure a = Const mempty
  (Const ma) <*> (Const mb) = Const (ma <> mb)
```

Running `getConst (run testTI fetchStatic)` evaluates to `["A1", "A2"]`, which are the dependencies of the `testTI` task.

7.2 Relation to Monoidal Effects And Handlers

This idea of different computation classes with their own interpretation interface is of course exactly the idea of the monoidal effects and handlers. In fact, this relation becomes much more apparent when we instantiate the constraint `c` with `Monad`. Then, `Task` becomes a specialization of the *Van Laarhoven Free Monad* (O'Connor, 2014), which is defined as:

```
newtype VLFree ops v = VLFree {
  unVLFree :: forall f. Monad f => ops f -> f v
}
```

where `ops` is instantiated with `data Fetch a b x = Fetch (b -> x) a`.

The Van Laarhoven representation is equivalent to the traditional free monad representation (Jaskelioff & O'Connor, 2015), used in the rest of this paper. In Haskell this traditional representation is defined as:

```
data Free ops v = Ret v | Con (ops (Free ops v))
```

This equivalent representation opens a different perspective on the `Task` type used to model build systems. A `Task` is a free computation describing how to calculate the values of a cell, which we presented in Section 2 using the `fetch` operation. In the next two paragraphs we cover the `Monad` and `Applicative` instantiations and show how the Haskell concepts instantiate the monoidal handlers and effects framework.

Monad Task A monadic task is represented by a value of type `Task Monad k v`, for example `testTM`. Its interpretation is given by a handler, which for a monoidal handler consists of three parts: f, ε and μ. The function `fetchConsole` corresponds to f, while ε and μ are the `Monad` implementations for `IO`.

Applicative Task For an applicative task, a computation is represented by a value of type `Task Applicative k v`, for example `testTI`. Its interpretation is also given by a handler, which again consists of f, ε and μ. The function `fetchStatic` corresponds to f, while the `Applicative` implementation for `Const` gives ε and μ.

7.3 Arrow Tasks

The `Task` representation nicely fits both applicative and monadic computations and their handlers, but cannot represent arrow computations and handlers. To accommodate arrow computations as well, we can adapt the `Task` type to use a profunctor operation as seen in Section 5.2. We also introduce the distinction between static and dynamic parameters.

```
newtype TaskA c si di o = TaskA {
  runA :: forall p. c p => (si -> p di o) -> p () o
}
```

TaskA Arrow Consider for example the following arrow task:

```
testA :: TaskA Arrow String String Integer
testA = TaskA $ \fetch -> proc () -> do
  a1 <- fetch "A" -< "1"
  fetch "B" -< show a1
```

This example uses the arrow notation introduced by Paterson (2001). The desugared version of this example is:

```
testA' :: TaskA Arrow String String Integer
testA' = TaskA $ \fetch ->
  arr (\_ -> "1") >>>
  fetch "A" >>>
  arr (\o -> show o) >>>
  fetch "B"
```

Defining the handler requires a definition for f:

```
fetchStaticA :: a -> ConstArr [a] b c
fetchStaticA a = ConstArr [a]
```

This `ConstArr` type is similar to the `Const` functor, but lifted to profunctors. It can implement the `Arrow` typeclass, for which we show Paterson's version (Paterson, 2001). This corresponds to the monoidal handler as follows: the ε is given by the `arr` function, while the μ is given by `(>>>)`. The `first` operation is also called *strength*, which this paper does not cover but is handled in more detail by for example Rivas & Jaskelioff (2017).

```
newtype ConstArr c i o = ConstArr { getConstArr :: c }

instance (Monoid c) => Arrow (ConstArr c) where
  arr f = ConstArr mempty
  (ConstArr a) >>> (ConstArr b) = ConstArr (a <> b)
  first (ConstArr c) = ConstArr c
```

Evaluating `getConstArr (runA testA fetchStaticA)` gives `["A", "B"]`, which are all columns used by `testA`.

TaskA ArrowChoice The computation `testTM` was shown earlier with a monadic constraint. However, as we saw in Section 2, computations with limited control flow can also be represented as a weaker computation such as `ArrowChoice`. The `ArrowChoice` class enables us to add conditionals as a control flow construct into computations expressed with `if then else` in the arrow sugar syntax.

```
testAC :: TaskA ArrowChoice String () Integer
testAC = TaskA $ \fetch -> proc () -> do
  c1 <- fetch "C1" -< ()
  if c1 == 1
    then fetch "A1" -< ()
    else fetch "A2" -< ()
```

The desugared version is shown below, using the `+++` function from the `ArrowChoice` class.

```
testAC' :: TaskA ArrowChoice String () Integer
testAC' = TaskA $ \fetch ->
  fetch "C1" >>>
  arr (\x -> if x == 1 then Left () else Right ()) >>>
  (fetch "A1" +++ fetch "A2") >>>
  arr untag
  where
    untag :: Either a a -> a
    untag (Left a) = a
    untag (Right a) = a
```

The adapted computation `testAC` can be analyzed using the previous `fetchStaticA` handler by evaluating `getConstArr (runA testAC fetchStaticA)`, giving all possible dependencies `["C1", "A1", "A2"]`. Using this handler requires an instance implementation of `ArrowChoice` for `ConstArr`.

```
instance (Monoid c) => ArrowChoice (ConstArr c) where
  (ConstArr c1) +++ (ConstArr c2) = ConstArr (c1 <> c2)
```

7.4 Overview Build Systems

This section gives an overview of the different possible and existing types of build systems corresponding to each computation class.

Monad Build System A monadic build system is the most straightforward build system. It starts from a target that needs to be built and executes the task corresponding to that target. The building of that task can execute the building of other tasks, or request the value of an input. An improvement, implemented by the Shake (Mitchell, 2012) build system, is to keep track of targets which have already been built.

Applicative Build System The applicative build system uses dependency analysis to determine an optimal building graph. This is possible because applicative tasks are restricted in such a way that it is always possible to determine their dependencies. An example of such build system is Make (Feldman, 1979), where the dependencies for each rule must be specified upfront. This allows make to construct the dependency graph before any building step is executed and enables certain optimizations such as omitting unneeded steps.

Arrow Build System An arrow build system is able to express computations where input to build tasks can depend on output from earlier build tasks. For example, in the `tempFile` example below, a user builds the file `folder/createTemp` which creates a temporary file with a random name. As the next step, the user builds the created temporary file by referring to the returned output name `tempfileName`.

```
tempFile = do
  tempfileName <- fetch "folder/createTemp"
  fetch ("folder/" ++ tempfileName)
```

It is no longer possible to statically determine every dependency, but it is possible to determine that this build task only executes tasks located in the path `folder/`, which could be used to automate the cleanup of the temporary files. This is similar to the partially static information example such as seen in `taskDyn` from Section 5.2.

ArrowChoice/Selective Build System A build system allowing conditional expressions cannot build the exact dependency graph, but it can build a pessimistic dependency graph. By aggregating all possible dependencies throughout all branching statements, it can construct all dependencies which might possibly be needed. Then, all independent dependencies can be constructed in parallel. This build system increases the overall throughput of the system since dependencies which are independent do not have to wait for a branch instruction to be built. Dune (Jane Street, 2018) is an example build system using this idea to do overapproximation of dependencies.

8 Related Work

Algebraic Effects and Handlers This paper is an exploration in the space of interfaces which a language with algebraic effects and handlers could provide. The currently developed languages and libraries in this area (such as by Bauer & Pretnar (2015), Brady (2013), Kiselyov & Ishii (2015), Leijen (2017), Lindley *et al.* (2017) and Plotkin & Pretnar (2009)) present the conventional monadic effects to the user, or only distinguish between applicative and monadic effects, resulting in an interpretation limited to these effect classes.

The interest in abstractions for effects such as applicative and arrow motivates a broader handler interface, one which allows interpretations utilizing these, and potentially more,

alternative abstractions. The motivation given at the start of the paper is a simple use case, but the overarching motivation is to port the use of these alternative abstractions to algebraic effects and handlers.

The methodology of derivation could be applied to other structures such as free near-semirings as explored by Rivas *et al.* (2015). Future work could explore the space of interfaces further to find a presentation which feels intuitive to a wide range of programmers.

Handlers for Idioms and Arrows Lindley (2014) introduces the calculus λ_{flow} which has handler constructs for monadic, applicative and arrow computations. The calculus has separate handling constructs for each of the different computation classes. We approach the same idea as a derivation from a general category theoretic framework. Lindley's and our interface slightly differ. The handler interfaces in λ_{flow} originate based on the intuition behind their behaviour. We give a short summary of the λ_{flow} interfaces below. For the full details we refer to the original work by Lindley (2014).

The λ_{flow} calculus is based on a call-by-push-value approach, and so there is a distinction between values and computations. The types of thunks are denoted with curly brackets $\{\ldots\}$ and the types of computations are denoted with square brackets `[...]`. Thunks are annotated with a list of effects and flow types, the flow types indicate whether data and/or control flow is used in the computation. We omit these annotations below, since they are not necessary for the overview given here.

The monad handler presents the traditional interface as was discussed before in the background section (Section 3).

```
| return (x: A) -> ...: C
| op_i (p: A_i, k: {B_i -> C}) -> ...: C
```

The arrow handlers are applicable to computations with an input, thus an arrow handler is applied using `handle (λz. comp) with h`. This results in the return clause taking a computation rather than a value. The continuation `k` is a thunk of type $\{\texttt{F(X} \times \texttt{B}_i\texttt{)}\}$, the functor `F` will be instantiated with the appropriate functor depending on the handler. The input to the operation `p` can not be statically inspected, since it requires an input of `X`, which is not readily available, since parameters of operations of arrow computations are not statically inspectable.

```
| return (x: X -> [A_i]) -> ...: F X
| op_i (p: {X -> [A_i]}, k: {F(X × B_i)}) -> ...: F X
```

The applicative handler is similar to the arrow handler, but the input for the parameter `p` in the operation clause does not require an input. This is because the parameters of operations in applicative computations can be statically inspected.

```
| return (x: X -> [A_i]) -> ...: F X
| op_i (p: A_i, k: {F(X × B_i)}) -> ...: F X
```

In contrast, our work derives the interfaces based on the theoretical background of free monoids. We believe that the resulting handlers are equivalent, but leave defining of the exact mapping for future work. A comparison of the interfaces by Lindley, and our derived inductive handlers can be found in Table 3. In addition, this work could also serve as a basis to give a denotational semantics for λ_{flow}.

Table 3: Handler Interface Comparison

Class	Lindley	Derived Inductive Handlers

Monad	`| return (x: A) -> ...: C`
	`| op`$_i$ `(p: A`$_i$`, k: {B`$_i$ `-> C})`
	`-> ...: C`
		`| ε (a: A) -> ...: FA`
		`| op`$_i$ `(p: P`$_i$`,k: N`$_i$ `-> FA)`
		`-> ...: FA`

Arrow	`| return (x: X -> [A`$_i$`]) -> ...: FX`
	`| op`$_i$ `(p: {X -> [A`$_i$`]},`
	`k: {F(X × B`$_i$`)}) -> ...: FX`
		`| ε (f: A -> B) -> ...: P(A,B)`
		`| op`$_i$ `(p: S`$_i$`, k: A -> (D`$_i$`, N`$_i$ `-> Z),`
		`l: P(Z,B)) -> ...: P(A,B)`

Applicative	`| return (x: X -> [A`$_i$`]) -> ...: FX`
	`| op`$_i$ `(p: A`$_i$`, k: {F(X × B`$_i$`)})`
	`-> ...: FX`
		`| ε (a: A) -> ...: FA`
		`| op`$_i$ `(p: P`$_i$`, k: N`$_i$ `-> Z -> A,`
		`l: FZ) -> ...: FA`

Free Monad/Arrow/Applicative The currently obtained interface for applicative and arrow handlers is similar to representing computations by expressing the free monad/arrow/applicative explicitly, since it is based on the same principles. These topics have been covered before, by for example Capriotti & Kaposi (2014) and Gibbons (2016). Free applicatives and arrows can be expressed in various ways, an alternative formulation is given by for example Lindley (2013). Compared to these works, we derive the handler interfaces from the general framework of monoids in monoidal categories and intend to expose this interface via specialized language syntax. The alternative formulations are an interesting avenue to explore the various intefaces which could be exposed by a language supporting handlers with generalized monoidal effects.

9 Conclusion

This paper has presented interfaces for applicative and arrow handlers derived from a unifying principle from which we also derived the conventional monadic handlers. This unifying principle is monoids in monoidal categories and was explored in detail by Rivas & Jaskelioff (2017). We have shown an equivalence between the initial algebra and free monoid syntax in the monoidal setting, as well as the initial algebra and free algebra approach in the monadic setting. We have expanded on the idea of lax monoidal functors with an adjunction to create a conversion of programs and handlers, enabling the reuse of handlers and programs across different monoidal categories. We have presented this work in the context of build systems, to motivate and illustrate the approach.

Acknowledgements

We would like to thank Nicolas Wu and the anonymous reviewers for their feedback. Exequiel Rivas was in part supported by Nomadic Labs via a grant on “Evolution, Semantics, and Engineering of the F* Verification System”. This work was partly funded by the Flemish Fund for Scientific Research (FWO), project 3E181126.

References

Bauer, A., & Pretnar, M. (2015). Programming with algebraic effects and handlers. *J. log. algebr. meth. program.*, **84**(1), 108–123.

Brady, E. (2013). Programming and reasoning with algebraic effects and dependent types. *Pages 133–144 of: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. New York, NY, USA: ACM.

Capriotti, P., & Kaposi, A. (2014). Free applicative functors. *Pages 2–30 of: Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014.*

Feldman, Stuart I. (1979). Make-a program for maintaining computer programs. *Softw., pract. exper.*, **9**(4), 255–65.

Gibbons, J. (2016). Free delivery (functional pearl). *Pages 45–50 of: Proceedings of the 9th International Symposium on Haskell*. ACM.

Hughes, J. (2000). Generalising monads to arrows. *Science of computer programming*, **37**(1-3).

Jane Street. (2018). *Dune: A composable build system*. `https://github.com/ocaml/dune`.

Jaskelioff, M., & O'Connor, R. (2015). A representation theorem for second-order functionals. *Journal of functional programming*, **25**, e13.

Kiselyov, O., & Ishii, H. (2015). Freer monads, more extensible effects. *Pages 94–105 of: Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*.

Leijen, D. (2017). Type directed compilation of row-typed algebraic effects. *Pages 486–499 of: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*.

Lindley, S. (2013). *Free idioms and free arrows in Haskell*. `https://github.com/slindley/dependent-haskell/tree/master/Free`.

Lindley, S. (2014). Algebraic effects and effect handlers for idioms and arrows. *Pages 47–58 of: Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*. WGP '14. New York, NY, USA: ACM.

Lindley, S., Wadler, P., & Yallop, J. (2011). Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electron. notes theor. comput. sci.*, **229**(5), 97–117.

Lindley, S., McBride, C., & McLaughlin, C. (2017). Do be do be do. *Pages 500–514 of: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. New York, NY, USA: ACM.

Marlow, S., Peyton Jones, S., Kmett, E., & Mokhov, A. (2016). Desugaring Haskell's do-notation into applicative operations. *Pages 92–104 of: Proceedings of the 9th International Symposium on Haskell*. Haskell 2016. New York, NY, USA: ACM.

McBride, C., & Paterson, R. (2008). Applicative programming with effects. *J. funct. program.*, **18**(1), 1–13.

Mitchell, N. (2012). Shake before building: Replacing Make with Haskell. *Pages 55–66 of: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ICFP '12. New York, NY, USA: ACM.

Moggi, E. (1991). Notions of computation and monads. *Inf. comput.*, **93**(1), 55–92.

Mokhov, A., Mitchell, N., & Peyton Jones, S. (2018). Build systems à la carte. *PACMPL*, **2**(ICFP), 79:1–79:29.

Mokhov, Andrey, Lukyanov, Georgy, Marlow, Simon, & Dimino, Jerémie. (2019). Selective applicative functors. *PACMPL*, **3**(ICFP), 90:1–90:29.

O'Connor, R. (2014). *Van Laarhoven free monad*. `http://r6.ca/blog/20140210T181244Z.html`.

Pastro, C., & Street, R. (2007). Doubles for monoidal categories. *Theory and applications of categories*, **21**(11).

Paterson, R. (2001). A new notation for arrows. *Pages 229–240 of: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*. ICFP '01. New York, NY, USA: ACM.

Pieters, R. P., Schrijvers, T., & Rivas, E. (2017). Handlers for non-monadic computations. *Pages 4:1–4:11 of: Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*. IFL 2017. New York, NY, USA: ACM.

Plotkin, G., & Pretnar, M. (2009). Handlers of algebraic effects. *Pages 80–94 of: Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*.

Rivas, E. (2018). Relating idioms, arrows and monads from monoidal adjunctions. *Pages 18–33 of: Proceedings of the 7th Workshop on Mathematically Structured Functional Programming, MSFP@FSCD 2018, Oxford, UK, 8th July 2018.*

Rivas, E., & Jaskelioff, M. (2017). Notions of computation as monoids. *J. funct. program.*, **27**, e21.

Rivas, E., Jaskelioff, M., & Schrijvers, T. (2015). From monoids to nearsemirings: The essence of MonadPlus and Alternative. *Pages 196–207 of:* Falaschi, Moreno, & Albert, Elvira (eds), *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. PPDP'15. ACM.

Wadler, P. (1990). Comprehending monads. *Pages 61–78 of: Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP '90. New York, NY, USA: ACM.

A Properties *eval*

This section proves some auxiliary properties related to *eval*.

$$\begin{aligned} \dot{e} &= \lfloor \lambda_X \rfloor && \text{(A 1)} \\ \dot{m} &= \lfloor \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ \alpha^{-1} \rfloor && \text{(A 2)} \\ eval_X\, e &= ev_X \circ (X^X \otimes e) \circ \rho_{X^X}^{-1} && \text{(A 3)} \end{aligned}$$

$$eval_X\, a \circ \lfloor \lambda_X \rfloor = a \qquad \text{(A 4)}$$

Proof

$$\begin{array}{ll} & eval_X\, a \circ \lfloor \lambda_X \rfloor \\ = & \quad \text{(def. } eval_X \text{ \& def. } ev_X\text{)} \\ & \lceil X^X \rceil \circ (X^X \otimes a) \circ \rho_{X^X}^{-1} \circ \lfloor \lambda_{X^X} \rfloor \\ = & \quad (\rho^{-1} \text{ is a natural transformation}) \\ & \lceil X^X \rceil \circ (X^X \otimes a) \circ (\lfloor \lambda_X \rfloor \otimes I) \circ \rho_I^{-1} \\ = & \quad \text{(bifunctor } \otimes\text{)} \\ & \lceil X^X \rceil \circ (\lfloor \lambda_X \rfloor \otimes X) \circ (I \otimes a) \circ \rho_I^{-1} \\ = & \quad \text{(naturality } \lceil - \rceil\text{)} \\ & \lceil \lfloor \lambda_X \rfloor \rceil \circ (I \otimes a) \circ \rho_I^{-1} \\ = & \quad \text{(inverses)} \\ & \lambda_X \circ (I \otimes a) \circ \rho_I^{-1} \\ = & \quad (\lambda \text{ is a natural transformation}) \\ & a \circ \lambda_I \circ \rho_I^{-1} \\ = & \quad \text{(def. monoidal category)} \\ & a \circ \rho_I \circ \rho_I^{-1} \\ = & \quad \text{(inverses)} \\ & a \end{array}$$

□

$$\begin{aligned} a &: I \to X \\ b &: A \otimes X \to X \\ eval_X\, a \circ \lfloor b \rfloor &= b \circ (A \otimes a) \circ \rho_A^{-1} && \text{(A 5)} \end{aligned}$$

Proof

$$\begin{array}{ll} & eval_X\, a \circ \lfloor b \rfloor \\ = & \quad \text{(def. } eval_X\text{)} \\ & \lceil X^X \rceil \circ (X^X \otimes a) \circ \rho_{X^X}^{-1} \circ \lfloor b \rfloor \\ = & \quad (\rho^{-1} \text{ is a natural transformation}) \\ & \lceil X^X \rceil \circ (X^X \otimes a) \circ (\lfloor b \rfloor \otimes I) \circ \rho_A^{-1} \\ = & \quad \text{(bifunctor } \otimes\text{)} \end{array}$$

$\lceil X^X \rceil \circ (\lfloor b \rfloor \otimes I) \circ (A \otimes a) \circ \rho_A^{-1}$
= (naturality $\lceil - \rceil$)
$\lceil \lfloor b \rfloor \rceil \circ (A \otimes a) \circ \rho_A^{-1}$
= (inverses)
$b \circ (A \otimes a) \circ \rho_A^{-1}$
□

B Initial Algebra Basis

This section proves the roundtrip and coherency properties for the initial algebra basis. first some relevant definitions are repeated, then each property related to a constructor or handler is proven in its own subsection.

B.1 Defining Properties $(\!|-|\!)$

$$(\!|\lceil a,b \rceil|\!) \circ \varepsilon = a \tag{B 1}$$
$$(\!|\lceil a,b \rceil|\!) \circ \iota = b \circ (\Sigma \otimes (\!|\lceil a,b \rceil|\!)) \tag{B 2}$$

B.2 Definition μ*/ins/free*

$$\mu = \lceil (\!|\lceil \lfloor \lambda_{\Sigma^*} \rfloor, \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \rceil|\!) \rceil \tag{B 3}$$
$$ins = \iota \circ (\Sigma \otimes \varepsilon) \circ \rho_\Sigma^{-1} \tag{B 4}$$
$$free\, f = (\!|\lceil e, m \circ (f \otimes M) \rceil|\!) \tag{B 5}$$

B.3 Roundtrip Property ι

The roundtrip property is: $\iota = \mu \circ (ins \otimes \Sigma^*)$, the definition of ι in the free monoid basis.

We use the following local definitions to save some space:

$$b_1 = \lfloor \lambda_{\Sigma^*} \rfloor$$
$$b_2 = \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor$$
$$b = \lceil b_1, b_2 \rceil$$

Proof

$\mu \circ (ins \otimes \Sigma^*)$
= (def. μ and *ins*)
$\lceil (\!|b|\!) \rceil \circ ((\iota \circ (\Sigma \otimes \varepsilon) \circ \rho_\Sigma^{-1}) \otimes \Sigma^*)$
= (naturality of $\lceil - \rceil$)
$\lceil (\!|b|\!) \circ \iota \circ (\Sigma \otimes \varepsilon) \circ \rho_\Sigma^{-1} \rceil$
= (property $(\!|-|\!)$ & bifunctor)
$\lceil \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \circ (\Sigma \otimes (\!|b|\!) \circ \varepsilon) \circ \rho_\Sigma^{-1} \rceil$
= (property $(\!|-|\!)$)
$\lceil \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \circ (\Sigma \otimes \lfloor \lambda_{\Sigma^*} \rfloor) \circ \rho_\Sigma^{-1} \rceil$

$$
\begin{aligned}
&= \quad \text{(naturality of } \lfloor - \rfloor) \\
&\quad \lceil \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \circ ((\Sigma \otimes \lfloor \lambda_{\Sigma^*} \rfloor) \otimes \Sigma^*) \rfloor \circ \rho_\Sigma^{-1} \rceil \\
&= \quad (\alpha^{-1} \text{ is a natural transformation}) \\
&\quad \lceil \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ (\Sigma \otimes (\lfloor \lambda_{\Sigma^*} \rfloor \otimes \Sigma^*)) \circ \alpha^{-1} \rfloor \circ \rho_\Sigma^{-1} \rceil \\
&= \quad \text{(bifunctor } \otimes) \\
&\quad \lceil \lfloor \iota \circ (\Sigma \otimes (ev_{\Sigma^*} \circ (\lfloor \lambda_{\Sigma^*} \rfloor \otimes \Sigma^*))) \circ \alpha^{-1} \rfloor \circ \rho_\Sigma^{-1} \rceil \\
&= \quad \text{(definition } ev_{\Sigma^*}) \\
&\quad \lceil \lfloor \iota \circ (\Sigma \otimes (\lceil \Sigma^{*^{\Sigma^*}} \rceil \circ (\lfloor \lambda_{\Sigma^*} \rfloor \otimes \Sigma^*))) \circ \alpha^{-1} \rfloor \circ \rho_\Sigma^{-1} \rceil \\
&= \quad \text{(naturality of } \lceil - \rceil) \\
&\quad \lceil \lfloor \iota \circ (\Sigma \otimes \lceil \lfloor \lambda_{\Sigma^*} \rfloor \rceil) \circ \alpha^{-1} \rfloor \circ \rho_\Sigma^{-1} \rceil \\
&= \quad \text{(inverses)} \\
&\quad \lceil \lfloor \iota \circ (\Sigma \otimes \lambda_{\Sigma^*}) \circ \alpha^{-1} \rfloor \circ \rho_\Sigma^{-1} \rceil \\
&= \quad \text{(naturality of } \lfloor - \rfloor) \\
&\quad \lceil \lfloor \iota \circ (\Sigma \otimes \lambda_{\Sigma^*}) \circ \alpha^{-1} \circ (\rho_\Sigma^{-1} \otimes \Sigma^*) \rfloor \rceil \\
&= \quad \text{(definition monoidal category, 3.2)} \\
&\quad \lceil \lfloor \iota \rfloor \rceil \\
&= \quad \text{(inverses)} \\
&\quad \iota \\
&\qquad \square
\end{aligned}
$$

B.4 Roundtrip Property $(\!|-|\!)$

The roundtrip property is: $(\!|[e,g]|\!) = eval_X\, e \circ free\, \lfloor g \rfloor$, the definition of $(\!|-|\!)$ in the free monoid basis. We show that the right-hand side is an algebra homomorphism $\Sigma^* \to X$. They are equal due to uniqueness of $(\!|[e,g]|\!)$.

Proof

$$
\begin{aligned}
&\quad eval_X\, e \circ free\, \lfloor g \rfloor \circ \varepsilon \\
&= \quad \text{(defs. } free\, \lfloor g \rfloor) \\
&\quad eval_X\, e \circ (\!|[\dot{e}, \dot{m} \circ (\lfloor g \rfloor \otimes X^X)]|\!) \circ \varepsilon \\
&= \quad \text{(property } (\!|-|\!)) \\
&\quad eval_X\, e \circ \dot{e} \\
&= \quad \text{(def. } \dot{e}) \\
&\quad eval_X\, e \circ \lfloor \lambda_X \rfloor \\
&= \quad \text{(property } eval_X) \\
&\quad e
\end{aligned}
$$

$$
\begin{aligned}
&\quad eval_X\, e \circ free\, \lfloor g \rfloor \circ \iota \\
&= \quad \text{(def. } free\, \lfloor g \rfloor) \\
&\quad eval_X\, e \circ (\!|[\dot{e}, \dot{m} \circ (\lfloor g \rfloor \otimes X^X)]|\!) \circ \iota \\
&= \quad \text{(property } (\!|-|\!)) \\
&\quad eval_X\, e \circ \dot{m} \circ (\lfloor g \rfloor \otimes X^X) \circ (\Sigma \otimes (\!|[\dot{e}, \dot{m} \circ (\lfloor g \rfloor \otimes X^X)]|\!)) \\
&= \quad \text{(introduce } \ldots \text{ to save some space)} \\
&\quad eval_X\, e \circ \dot{m} \circ (\lfloor g \rfloor \otimes X^X) \circ (\Sigma \otimes \ldots)
\end{aligned}
$$

$$
\begin{array}{ll}
= & \text{(def. } \dot{m}\text{)} \\
& eval_X\, e \circ \lfloor \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ \alpha^{-1} \rfloor \circ (\lfloor g \rfloor \otimes X^X) \\
& \quad \circ (\Sigma \otimes \ldots) \\
= & \text{(property } eval_X\text{)} \\
& \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ \alpha^{-1} \circ ((X^X \otimes X^X) \otimes e) \\
& \quad \circ \rho^{-1}_{X^X \otimes X^X} \circ (\lfloor g \rfloor \otimes X^X) \circ (\Sigma \otimes \ldots) \\
= & (\alpha^{-1} \text{ is a natural transformation}) \\
& \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ (X^X \otimes (X^X \otimes e)) \circ \alpha^{-1} \\
& \quad \circ \rho^{-1}_{X^X \otimes X^X} \circ (\lfloor g \rfloor \otimes X^X) \circ (\Sigma \otimes \ldots) \\
= & (\alpha^{-1} \text{ and } \rho^{-1} \text{ are natural transformations}) \\
& \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ (X^X \otimes (X^X \otimes e)) \\
& \quad \circ (\lfloor g \rfloor \otimes (X^X \otimes I)) \circ \alpha^{-1} \circ \rho^{-1}_{\Sigma \otimes X^X} \circ (\Sigma \otimes \ldots) \\
= & \text{(bifunctor } \otimes\text{)} \\
& \lceil X^X \rceil \circ (\lfloor g \rfloor \otimes X) \circ (\Sigma \otimes \lceil X^X \rceil) \circ (\Sigma \otimes (X^X \otimes e)) \\
& \quad \circ \alpha^{-1} \circ \rho^{-1}_{\Sigma \otimes X^X} \circ (\Sigma \otimes \ldots) \\
= & \text{(naturality } \lceil - \rceil \text{ \& inverses)} \\
& g \circ (\Sigma \otimes \lceil X^X \rceil) \circ (\Sigma \otimes (X^X \otimes e)) \circ \alpha^{-1} \circ \rho^{-1}_{\Sigma \otimes X^X} \circ (\Sigma \otimes \ldots) \\
= & \text{(property } \alpha^{-1} \circ \rho^{-1} = (id \otimes \rho^{-1})\text{)} \\
& g \circ (\Sigma \otimes \lceil X^X \rceil) \circ (\Sigma \otimes (X^X \otimes e)) \circ (\Sigma \otimes \rho^{-1}_{X^X}) \circ (\Sigma \otimes \ldots) \\
= & \text{(bifunctor } \otimes\text{)} \\
& g \circ (\Sigma \otimes (\lceil X^X \rceil \circ (X^X \otimes e) \circ \rho^{-1}_{X^X})) \circ (\Sigma \otimes \ldots) \\
= & \text{(def. } eval_X\, e\text{)} \\
& g \circ (\Sigma \otimes eval_X\, e) \circ (\Sigma \otimes \ldots) \\
= & \text{(remove } \ldots \text{ \& bifunctor } \otimes\text{)} \\
& g \circ (\Sigma \otimes (eval_X\, e \circ (\!|[\dot{e}, \dot{m} \circ (\lfloor g \rfloor \otimes X^X)]|\!))) \\
= & \text{(def. } free\, \lfloor g \rfloor\text{)} \\
& g \circ (\Sigma \otimes (eval_X\, e \circ free\, \lfloor g \rfloor))
\end{array}
$$

□

B.5 Coherency Properties free f

The *free f* morphism should have the same properties as in the free monoid basis, resulting in 3 coherency properties.

B.5.1 Property 1

We prove that $free\, f \circ \varepsilon = e$.

Proof

$$
\begin{array}{ll}
& free\, f \circ \varepsilon \\
= & \text{(def. of } free\text{, B 5)} \\
& (\!|[e, m \circ (f \otimes M)]|\!) \circ \varepsilon \\
= & \text{(B 1)} \\
& e
\end{array}
$$

□

B.5.2 Property 2

We prove that $\mathit{free}\, f \circ \mathit{ins} = f$.

Proof

$$
\begin{aligned}
& \mathit{free}\, f \circ \mathit{ins} \\
= \quad & \text{(defs. of } \mathit{ins} \text{ and } \mathit{free}\, f\text{)} \\
& (\!|[e, m \circ (f \otimes M)]|\!) \circ \iota \circ (\Sigma \otimes \varepsilon) \circ \rho_{\Sigma}^{-1} \\
= \quad & \text{(property of } (\!|-|\!)\text{)} \\
& m \circ (f \otimes M) \circ (\Sigma \otimes (\!|[e, \ldots]|\!)) \circ (\Sigma \otimes \varepsilon) \circ \rho_{\Sigma}^{-1} \\
= \quad & \text{(bifunctor } \otimes\text{)} \\
& m \circ (f \otimes M) \circ (\Sigma \otimes ((\!|[e, \ldots]|\!) \circ \varepsilon)) \circ \rho_{\Sigma}^{-1} \\
= \quad & \text{(property of } (\!|-|\!)\text{)} \\
& m \circ (f \otimes M) \circ (\Sigma \otimes e) \circ \rho_{\Sigma}^{-1} \\
= \quad & \text{(bifunctor } \otimes\text{)} \\
& m \circ (M \otimes e) \circ (f \otimes I) \circ \rho_{\Sigma}^{-1} \\
= \quad & \text{(naturality of } \rho^{-1}\text{)} \\
& m \circ (M \otimes e) \circ \rho_{M}^{-1} \circ f \\
= \quad & \text{(monoid right unit property)} \\
& \rho_M \circ \rho_{M}^{-1} \circ f \\
= \quad & \text{(inverses)} \\
& f
\end{aligned}
$$

□

B.5.3 Property 3

We prove that $\mathit{free}\, f \circ \mu = m \circ (\mathit{free}\, f \otimes \mathit{free}\, f)$. We first show that both sides are algebra homomorphisms $\Sigma^* \to M^{\Sigma^*}$, by uniqueness of $(\!|-|\!)$ both must be equal to $(\!|[\lfloor \mathit{free}\, f \circ \lambda_{\Sigma^*} \rfloor, \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor]|\!)$.

Proof
First we show that $\lfloor \mathit{free}\, f \circ \mu \rfloor = (\!|[\lfloor \mathit{free}\, f \circ \lambda_{\Sigma^*} \rfloor, \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor]|\!)$

$$
\begin{aligned}
& \lfloor \mathit{free}\, f \circ \mu \rfloor \circ \varepsilon \\
= \quad & \text{(naturality } \lfloor - \rfloor\text{)} \\
& \lfloor \mathit{free}\, f \circ \mu \circ (\varepsilon \otimes \Sigma^*) \rfloor \\
= \quad & \text{(def. } \mu \text{ \& naturality of } \lceil - \rceil\text{)} \\
& \lfloor \mathit{free}\, f \circ \lceil (\!|b|\!) \circ \varepsilon \rceil \rfloor \\
= \quad & \text{(def. } \mathit{free}\, f \text{ \& property of } (\!|-|\!)\text{)} \\
& \lfloor \mathit{free}\, f \circ \lambda_{\Sigma^*} \rfloor
\end{aligned}
$$

$$
\begin{aligned}
& \lfloor \mathit{free}\, f \circ \mu \rfloor \circ \iota \\
= \quad & \text{(naturality } \lfloor - \rfloor) \\
& \lfloor \mathit{free}\, f \circ \mu \circ (\iota \otimes \Sigma^*) \rfloor \\
= \quad & \text{(def. } \mu \text{ \& naturality of } \lceil - \rceil) \\
& \lfloor \mathit{free}\, f \circ \lceil (\!| b |\!) \circ \iota \rceil \rfloor \\
= \quad & \text{(property of } (\!| - |\!)) \\
& \lfloor \mathit{free}\, f \circ \lceil b_2 \circ (\Sigma \otimes (\!| b |\!)) \rceil \rfloor \\
= \quad & \text{(naturality of } \lceil - \rceil) \\
& \lfloor \mathit{free}\, f \circ \lceil b_2 \rceil \circ ((\Sigma \otimes (\!| b |\!)) \otimes \Sigma^*) \rfloor \\
= \quad & \text{(def. } b_2 \text{ \& inverses)} \\
& \lfloor \mathit{free}\, f \circ \iota \circ (\Sigma \otimes \mathit{ev}_{\Sigma^*}) \circ \alpha^{-1} \circ ((\Sigma \otimes (\!| b |\!)) \otimes \Sigma^*) \rfloor \\
= \quad & \text{(def. } \mathit{free}\, f \text{ \& property of } (\!| - |\!) \text{ \& bifunctor } \otimes) \\
& \lfloor m \circ (f \otimes (\mathit{free}\, f \circ \mathit{ev}_{\Sigma^*})) \circ \alpha^{-1} \circ ((\Sigma \otimes (\!| b |\!)) \otimes \Sigma^*) \rfloor \\
= \quad & \text{(def. } \mathit{ev}_{\blacksquare^*} \text{ \& naturality } \lceil - \rceil) \\
& \lfloor m \circ (f \otimes \lceil \mathit{free}\, f^{\Sigma^*} \rceil) \circ \alpha^{-1} \circ ((\Sigma \otimes (\!| b |\!)) \otimes \Sigma^*) \rfloor \\
= \quad & \text{(naturality of } \lceil - \rceil) \\
& \lfloor m \circ (f \otimes (\lceil M^{\Sigma^*} \rceil \circ (\mathit{free}\, f^{\Sigma^*} \otimes \Sigma^*)) \circ \alpha^{-1} \\
& \quad \circ ((\Sigma \otimes (\!| b |\!)) \otimes \Sigma^*) \rfloor \\
= \quad & \text{(bifunctor } \otimes) \\
& \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ (\Sigma \otimes (\mathit{free}\, f^{\Sigma^*} \otimes \Sigma^*)) \circ \alpha^{-1} \\
& \quad \circ ((\Sigma \otimes (\!| b |\!)) \otimes \Sigma^*) \rfloor \\
= \quad & \text{(naturality of } \alpha^{-1} \text{ \& bifunctor)} \\
& \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \circ ((\Sigma \otimes (\mathit{free}\, f^{\Sigma^*} \circ (\!| b |\!)) \otimes \Sigma^*) \rfloor \\
= \quad & \text{(naturality of } \lfloor - \rfloor) \\
& \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor \circ (\Sigma \otimes (\mathit{free}\, f^{\Sigma^*} \circ (\!| b |\!))) \\
= \quad & ((\!| b |\!) = \lfloor \mu \rfloor) \\
& \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor \circ (\Sigma \otimes (\mathit{free}\, f^{\Sigma^*} \circ \lfloor \mu \rfloor)) \\
= \quad & \text{(naturality of } \lfloor - \rfloor) \\
& \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor \circ (\Sigma \otimes (\lfloor \mathit{free}\, f \circ \mu \rfloor))
\end{aligned}
$$

Then we show that $\lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor = (\!| [\lfloor \mathit{free}\, f \circ \lambda_{\Sigma^*} \rfloor, \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor] |\!)$

$$
\begin{aligned}
& \lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor \circ \varepsilon \\
= \quad & \text{(naturality of } \lfloor - \rfloor) \\
& \lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \circ (\varepsilon \otimes A^*) \rfloor \\
= \quad & \text{(bifunctor } \otimes) \\
& \lfloor m \circ ((\mathit{free}\, f \circ \varepsilon) \otimes \mathit{free}\, f) \rfloor \\
= \quad & \text{(def. } \mathit{free}\, f \text{ \& property of } (\!| - |\!)) \\
& \lfloor m \circ (e \otimes \mathit{free}\, f) \rfloor \\
= \quad & \text{(bifunctor } \otimes \text{ \& monoid left unit property)} \\
& \lfloor \lambda_M \circ (I \otimes \mathit{free}\, f) \rfloor \\
= \quad & \text{(naturality of } \lambda) \\
& \lfloor \mathit{free}\, f \circ \lambda_{\Sigma^*} \rfloor
\end{aligned}
$$

$$
\begin{array}{ll}
 & \lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor \circ \iota \\
= & \quad \text{(naturality } \lfloor - \rfloor \text{ \& bifunctor } \otimes) \\
 & \lfloor m \circ ((\mathit{free}\, f \circ \iota) \otimes \mathit{free}\, f) \rfloor \\
= & \quad \text{(def. } \mathit{free}\, f \text{ \& property of } (\!|-|\!) \text{ \& bifunctor } \otimes) \\
 & \lfloor m \circ (m \otimes M) \circ ((f \otimes \mathit{free}\, f) \otimes \mathit{free}\, f) \rfloor \\
= & \quad \text{(monoid associativity property)} \\
 & \lfloor m \circ (M \otimes m) \circ \alpha^{-1} \circ ((f \otimes \mathit{free}\, f) \otimes \mathit{free}\, f) \rfloor \\
= & \quad \text{(naturality of } \alpha^{-1}) \\
 & \lfloor m \circ (M \otimes m) \circ (f \otimes (\mathit{free}\, f \otimes \mathit{free}\, f)) \circ \alpha^{-1} \rfloor \\
= & \quad \text{(bifunctor } \otimes) \\
 & \lfloor m \circ (f \otimes (m \circ (\mathit{free}\, f \otimes \mathit{free}\, f))) \circ \alpha^{-1} \rfloor \\
= & \quad \text{(inverses)} \\
 & \lfloor m \circ (f \otimes \lceil \lfloor m_M \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor \rceil) \circ \alpha^{-1} \rfloor \\
= & \quad \text{(naturality of } \lceil - \rceil) \\
 & \lfloor m \circ (f \otimes (\lceil M^{\Sigma^*} \rceil \circ (\lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor \otimes \Sigma^*))) \\
 & \quad \circ \alpha^{-1} \rfloor \\
= & \quad \text{(bifunctor } \otimes \text{ \& naturality of } \alpha^{-1}) \\
 & \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \\
 & \quad \circ ((\Sigma \otimes \lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor) \otimes \Sigma^*) \rfloor \\
= & \quad \text{(naturality } \lfloor - \rfloor) \\
 & \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor \circ (\Sigma \otimes \lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor)
\end{array}
$$

Then, using $\lfloor \mathit{free}\, f \circ \mu \rfloor = (\!|[\lfloor \mathit{free}\, f \circ \lambda_{\Sigma^*} \rfloor, \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor]|\!) = \lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor$, we show that the property holds.

$$
\begin{array}{ll}
 & \mathit{free}\, f \circ \mu \\
= & \quad \text{(inverses)} \\
 & \lceil \lfloor \mathit{free}\, f \circ \mu \rfloor \rceil \\
= & \quad \text{(proven above)} \\
 & \lceil \lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor \rceil \\
= & \quad \text{(inverses)} \\
 & m \circ (\mathit{free}\, f \otimes \mathit{free}\, f)
\end{array}
$$

□

C Free Monoid Basis

This section proves the roundtrip and coherency properties for the free monoid basis. first some relevant definitions are repeated, then each property related to a constructor or handler is proven in its own subsection.

C.1 Defining Properties free

$$\begin{aligned} free\, f \circ \varepsilon &= e && \text{(C 1)} \\ free\, f \circ ins &= f && \text{(C 2)} \\ free\, f \circ \mu &= m \circ (free\, f \otimes free\, f) && \text{(C 3)} \end{aligned}$$

, where (M, e, m) is a monoid.

C.2 Definition ι/$(\!|[e,g]|\!)$

$$\begin{aligned} \iota &= \mu \circ (ins \otimes \Sigma^*) && \text{(C 4)} \\ (\!|[e,g]|\!) &= eval_X\, e \circ free\, \lfloor g \rfloor && \text{(C 5)} \end{aligned}$$

C.3 Roundtrip Property ins

The roundtrip property is: $ins = \iota \circ (\Sigma \otimes \varepsilon) \circ \rho_\Sigma^{-1}$, the definition of *ins* in the initial algebra basis.

Proof

$\iota \circ (\Sigma \otimes \varepsilon) \circ \rho_\Sigma^{-1}$
= (def ι, C 4)
$\mu \circ (ins \otimes \Sigma^*) \circ (\Sigma \otimes \varepsilon) \circ \rho_\Sigma^{-1}$
= (bifunctor $\otimes$)
$\mu \circ (\Sigma^* \otimes \varepsilon) \circ (ins \otimes I) \circ \rho_\Sigma^{-1}$
= (monoid property)
$\rho_{\Sigma^*} \circ (ins \otimes I) \circ \rho_\Sigma^{-1}$
= (ρ is a natural transformation)
$ins \circ \rho_\Sigma \circ \rho_\Sigma^{-1}$
= (inverses)
ins
□

C.4 Roundtrip Property μ

The roundtrip property is: $\mu = \lceil (\!|[\lfloor \lambda_{\Sigma^*} \rfloor, \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor]|\!) \rceil$, the definition of μ in the initial algebra basis. We prove this by using the fact that both sides (after $\lfloor - \rfloor$) are equal to $free\, \lfloor \iota \rfloor$.

C.4.1 Left-Hand Side

First we show that $\lfloor \mu \rfloor = free\, \lfloor \iota \rfloor$. We show that it is a monoid homomorphism $\Sigma^* \to \Sigma^{*\Sigma^*}$ and that $\lfloor \mu \rfloor \circ ins = \lfloor \iota \rfloor$. They are equal due to uniqueness of *free*.

Proof

$$
\begin{array}{ll}
 & \lfloor \mu \rfloor \circ \varepsilon \\
= & \quad \text{(naturality } \lfloor - \rfloor\text{)} \\
 & \lfloor \mu \circ (\varepsilon \otimes \Sigma^*) \rfloor \\
= & \quad \text{(monoid property)} \\
 & \lfloor \lambda_{\Sigma^*} \rfloor \\
= & \quad \text{(def. } \dot{e}\text{)} \\
 & \dot{e}
\end{array}
$$

$$
\begin{array}{ll}
 & \lfloor \mu \rfloor \circ \mathit{ins} \\
= & \quad \text{(naturality } \lfloor - \rfloor\text{)} \\
 & \lfloor \mu \circ (\mathit{ins} \otimes \Sigma^*) \rfloor \\
= & \quad \text{(def } \iota\text{)} \\
 & \lfloor \iota \rfloor
\end{array}
$$

$$
\begin{array}{ll}
 & \lfloor \mu \rfloor \circ \mu \\
= & \quad \text{(naturality } \lfloor - \rfloor\text{)} \\
 & \lfloor \mu \circ (\mu \otimes \Sigma^*) \rfloor \\
= & \quad \text{(monoid property)} \\
 & \lfloor \mu \circ (\Sigma^* \otimes \mu) \circ \alpha^{-1} \rfloor \\
= & \quad \text{(inverses)} \\
 & \lfloor \lceil \lfloor \mu \rfloor \rceil \circ (\Sigma^* \otimes \lceil \lfloor \mu \rfloor \rceil) \circ \alpha^{-1} \rfloor \\
= & \quad \text{(naturality } \lceil - \rceil \text{ \& bifunctor } \otimes\text{)} \\
 & \lfloor \lceil \Sigma^{*\Sigma^*} \rceil \circ (\lfloor \mu \rfloor \otimes (\lceil \Sigma^{*\Sigma^*} \rceil \circ (\lfloor \mu \rfloor \otimes \Sigma^*))) \circ \alpha^{-1} \rfloor \\
= & \quad \text{(bifunctor } \otimes\text{)} \\
 & \lfloor \lceil \Sigma^{*\Sigma^*} \rceil \circ (\Sigma^{*\Sigma^*} \otimes \lceil \Sigma^{*\Sigma^*} \rceil) \circ (\lfloor \mu \rfloor \otimes (\lfloor \mu \rfloor \otimes \Sigma^*)) \circ \alpha^{-1} \rfloor \\
= & \quad (\alpha^{-1} \text{ is a natural transformation}) \\
 & \lfloor \lceil \Sigma^{*\Sigma^*} \rceil \circ (\Sigma^{*\Sigma^*} \otimes \lceil \Sigma^{*\Sigma^*} \rceil) \circ \alpha^{-1} \circ ((\lfloor \mu \rfloor \otimes \lfloor \mu \rfloor) \otimes \Sigma^*) \rfloor \\
= & \quad \text{(naturality } \lfloor - \rfloor\text{)} \\
 & \lfloor \lceil \Sigma^{*\Sigma^*} \rceil \circ (\Sigma^{*\Sigma^*} \otimes \lceil \Sigma^{*\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor \circ (\lfloor \mu \rfloor \otimes \lfloor \mu \rfloor) \\
= & \quad \text{(def. } \dot{m}\text{)} \\
 & \dot{m} \circ (\lfloor \mu \rfloor \otimes \lfloor \mu \rfloor)
\end{array}
$$

□

C.4.2 Right-Hand Side

We use the following local definitions for readability

$$
\begin{array}{lcl}
b_1 & = & \lfloor \lambda_{\Sigma^*} \rfloor \\
b_2 & = & \lfloor \iota \circ (\Sigma \otimes \mathit{ev}_{\Sigma^*}) \circ \alpha^{-1} \rfloor
\end{array}
$$

We show that $(\!|[b_1, b_2]|\!) = \mathit{free}\, \lfloor \iota \rfloor$. We show that it is a monoid homomorphism $\Sigma^* \to \Sigma^{*\Sigma^*}$ and that $(\!|[b_1, b_2]|\!) \circ \mathit{ins} = \lfloor \iota \rfloor$. They are equal due to uniqueness of *free*.

Proof

$$
\begin{array}{ll}
& (\!|[b_1, b_2]|\!) \circ \varepsilon \\
= & \quad \text{(def. } (\!|-|\!)\text{)} \\
& \mathit{eval}_{\Sigma^* \Sigma^*}\, b_1 \circ \mathit{free}\, \lfloor b_2 \rfloor \circ \varepsilon \\
= & \quad \text{(property } \mathit{free}\text{)} \\
& \mathit{eval}_{\Sigma^* \Sigma^*}\, b_1 \circ \lfloor \lambda_{\Sigma^* \Sigma^*} \rfloor \\
= & \quad \text{(property } \mathit{eval}\text{)} \\
& b_1 \\
= & \quad \text{(expand } b_1\text{)} \\
& \lfloor \lambda_{\Sigma^*} \rfloor \\
= & \quad \text{(def. } \dot{e}\text{)} \\
& \dot{e}
\end{array}
$$

$$
\begin{array}{ll}
& (\!|[b_1, b_2]|\!) \circ \mathit{ins} \\
= & \quad \text{(def. } (\!|-|\!)\text{)} \\
& \mathit{eval}_{\Sigma^* \Sigma^*}\, b_1 \circ \mathit{free}\, \lfloor b_2 \rfloor \circ \mathit{ins} \\
= & \quad \text{(property } \mathit{free}\text{)} \\
& \mathit{eval}_{\Sigma^* \Sigma^*}\, b_1 \circ \lfloor b_2 \rfloor \\
= & \quad \text{(property } \mathit{eval}\text{)} \\
& b_2 \circ (\Sigma \otimes b_1) \circ \rho_\Sigma^{-1} \\
= & \quad \text{(expand } b_2\text{)} \\
& \lfloor \iota \circ (\Sigma \otimes \mathit{ev}_{\Sigma^*}) \circ \alpha^{-1} \rfloor \circ (\Sigma \otimes b_1) \circ \rho_\Sigma^{-1} \\
= & \quad \text{(naturality } \lfloor - \rfloor\text{)} \\
& \lfloor \iota \circ (\Sigma \otimes \mathit{ev}_{\Sigma^*}) \circ \alpha^{-1} \circ ((\Sigma \otimes b_1) \otimes \Sigma^*) \circ (\rho_\Sigma^{-1} \otimes \Sigma^*) \rfloor \\
= & \quad (\alpha^{-1} \text{ is a natural transformation}) \\
& \lfloor \iota \circ (\Sigma \otimes \mathit{ev}_{\Sigma^*}) \circ (\Sigma \otimes (b_1 \otimes \Sigma^*)) \circ \alpha^{-1} \circ (\rho_\Sigma^{-1} \otimes \Sigma^*) \rfloor \\
= & \quad \text{(def. } \mathit{ev}\text{)} \\
& \lfloor \iota \circ (\Sigma \otimes \lceil \Sigma^{*\Sigma^*} \rceil) \circ (\Sigma \otimes (b_1 \otimes \Sigma^*)) \circ \alpha^{-1} \circ (\rho_\Sigma^{-1} \otimes \Sigma^*) \rfloor \\
= & \quad \text{(bifunctor } \otimes \text{ \& naturality } \lceil - \rceil\text{)} \\
& \lfloor \iota \circ (\Sigma \otimes \lceil b_1 \rceil) \circ \alpha^{-1} \circ (\rho_\Sigma^{-1} \otimes \Sigma^*) \rfloor \\
= & \quad \text{(expand } b_1 \text{ \& inverses)} \\
& \lfloor \iota \circ (\Sigma \otimes \lambda_{\Sigma^*}) \circ \alpha^{-1} \circ (\rho_\Sigma^{-1} \otimes \Sigma^*) \rfloor \\
= & \quad \text{(def. monoidal category)} \\
& \lfloor \iota \rfloor
\end{array}
$$

$$
\begin{array}{ll}
& (\!|[b_1, b_2]|\!) \circ \mu \\
= & \quad \text{(def. } (\!|-|\!)\text{)} \\
& \mathit{eval}_{\Sigma^* \Sigma^*}\, b_1 \circ \mathit{free}\, \lfloor b_2 \rfloor \circ \mu \\
= & \quad (\mathit{free}\, \lfloor b_2 \rfloor = \lfloor \dot{m} \rfloor \circ \mathit{free}\, \lfloor \iota \rfloor) \\
& \mathit{eval}_{\Sigma^* \Sigma^*}\, b_1 \circ \lfloor \dot{m} \rfloor \circ \mathit{free}\, \lfloor \iota \rfloor \circ \mu \\
= & \quad (\mathit{eval}_{\Sigma^* \Sigma^*}\, b_1 \circ \lfloor \dot{m} \rfloor = \Sigma^{*\Sigma^*}) \\
& \mathit{free}\, \lfloor \iota \rfloor \circ \mu \\
= & \quad \text{(property } \mathit{free}\text{)} \\
& \dot{m} \circ (\mathit{free}\, \lfloor \iota \rfloor \otimes \mathit{free}\, \lfloor \iota \rfloor)
\end{array}
$$

$$
\begin{array}{ll}
= & (eval_{\Sigma^*\Sigma^*}\ b_1 \circ \lfloor \dot{m} \rfloor = \Sigma^{*\Sigma^*}) \\
& \dot{m} \circ ((eval_{\Sigma^*\Sigma^*}\ b_1 \circ \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor) \otimes \\
& \quad (eval_{\Sigma^*\Sigma^*}\ b_1 \circ \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor)) \\
= & (free\ \lfloor b_2 \rfloor = \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor) \\
& \dot{m} \circ ((eval_{\Sigma^*\Sigma^*}\ b_1 \circ free\ \lfloor b_2 \rfloor) \otimes (eval_{\Sigma^*\Sigma^*}\ b_1 \circ free\ \lfloor b_2 \rfloor)) \\
= & (\text{def. } (\!|-|\!)) \\
& \dot{m} \circ ((\!|[b_1, b_2]|\!) \otimes (\!|[b_1, b_2]|\!))
\end{array}
$$

Equality $free\ \lfloor b_2 \rfloor = \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor$ holds since

$$
\begin{array}{ll}
& \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor \circ ins \\
= & (\text{property } free) \\
& \lfloor \dot{m} \rfloor \circ \lfloor \iota \rfloor \\
= & (\text{def. } \dot{m}) \\
& \lfloor \lfloor ev_{\Sigma^*} \circ (\Sigma^{*\Sigma^*} \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \rfloor \circ \lfloor \iota \rfloor \\
= & (\text{naturality } \lfloor - \rfloor) \\
& \lfloor \lfloor ev_{\Sigma^*} \circ (\Sigma^{*\Sigma^*} \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \circ (\lfloor \iota \rfloor \otimes \Sigma^{*\Sigma^*}) \rfloor \\
= & (\text{naturality } \lfloor - \rfloor) \\
& \lfloor \lfloor ev_{\Sigma^*} \circ (\Sigma^{*\Sigma^*} \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \circ ((\lfloor \iota \rfloor \otimes \Sigma^{*\Sigma^*}) \otimes \Sigma^*) \rfloor \rfloor \\
= & (\alpha^{-1} \text{ is a natural transformation}) \\
& \lfloor \lfloor ev_{\Sigma^*} \circ (\Sigma^{*\Sigma^*} \otimes ev_{\Sigma^*}) \circ (\lfloor \iota \rfloor \otimes (\Sigma^{*\Sigma^*} \otimes \Sigma^*)) \circ \alpha^{-1} \rfloor \rfloor \\
= & (\text{bifunctor } \otimes) \\
& \lfloor \lfloor ev_{\Sigma^*} \circ (\lfloor \iota \rfloor \otimes \Sigma^*) \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \rfloor \\
= & (\text{def. } ev_{\Sigma^*}) \\
& \lfloor \lfloor \lceil \Sigma^{*\Sigma^*} \rceil \circ (\lfloor \iota \rfloor \otimes \Sigma^*) \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \rfloor \\
= & (\text{naturality } \lceil - \rceil) \\
& \lfloor \lfloor \lceil \lfloor \iota \rfloor \rceil \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \rfloor \\
= & (\text{inverses}) \\
& \lfloor \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \rfloor \\
= & (\text{def. } b_2) \\
& \lfloor b_2 \rfloor
\end{array}
$$

, and $\lfloor \dot{m} \rfloor$ and $free\ \lfloor \iota \rfloor$ are both monoid homorphisms and thus their composition is a monoid homomorphism, meaning $\lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor \circ \varepsilon = \ddot{e}$ and $\lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor \circ \mu = \ddot{m} \circ (\lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor \otimes \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor)$. Since $free\ \lfloor b_2 \rfloor$ is the unique monoid homomorphism, $free\ \lfloor b_2 \rfloor = \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor$.

Where $\dot{m}$ and $\ddot{m}$ are specialized to $\ddot{m} : (\Sigma^{*\Sigma^*})^{(\Sigma^{*\Sigma^*})} \otimes (\Sigma^{*\Sigma^*})^{(\Sigma^{*\Sigma^*})} \to (\Sigma^{*\Sigma^*})^{(\Sigma^{*\Sigma^*})}$ and $\dot{m} : \Sigma^{*\Sigma^*} \otimes \Sigma^{*\Sigma^*} \to \Sigma^{*\Sigma^*}$ for this case.

□

C.4.3 Property Proof

Then, using $(\!|[b_1, b_2]|\!) = free\ \lfloor \iota \rfloor = \lfloor \mu \rfloor$, we show that the roundtrip property holds.

Proof

$\lceil (\!|[\lfloor \lambda_{\Sigma^*} \rfloor, \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor]|\!) \rceil$
= (proven above)
$\lceil \lfloor \mu \rfloor \rceil$
= (inverses)
μ
□

C.5 Roundtrip Property free

The roundtrip property is: $free\, f = (\!|[e, m \circ (f \otimes M)]|\!)$, the definition of $free\, f$ in the initial algebra basis. We show that the right-hand side is a monoid homomorphism $\Sigma^* \to M$ and $(\!|[e, m \circ (f \otimes M)]|\!) \circ ins = f$. They are equal due to uniqueness of *free*.

Proof

$(\!|[e, m \circ (f \otimes M)]|\!) \circ \varepsilon$
= (def. $(\!|-|\!)$)
$eval_M\, e \circ free \lfloor m \circ (f \otimes M) \rfloor \circ \varepsilon$
= (property *free*)
$eval_M\, e \circ \dot{e}$
= (def. $\dot{e}$ & property *eval*)
e

$(\!|[e, m \circ (f \otimes M)]|\!) \circ ins$
= (def. $(\!|-|\!)$)
$eval_M\, e \circ free \lfloor m \circ (f \otimes M) \rfloor \circ ins$
= (property *free*)
$eval_M\, e \circ \lfloor m \circ (f \otimes M) \rfloor$
= (property *eval*)
$m \circ (f \otimes M) \circ (\Sigma \otimes e) \circ \rho_\Sigma^{-1}$
= (bifunctor $\otimes$)
$m \circ (M \otimes e) \circ (f \otimes I) \circ \rho_\Sigma^{-1}$
= (monoid property)
$\rho_M \circ (f \otimes I) \circ \rho_\Sigma^{-1}$
= (ρ is a natural transformation)
$f \circ \rho_\Sigma \circ \rho_\Sigma^{-1}$
= (inverses)
f

$(\!|[e, m \circ (f \otimes M)]|\!) \circ \mu$
= (def. $(\!|-|\!)$)
$eval_M\, e \circ free \lfloor m \circ (f \otimes M) \rfloor \circ \mu$
= ($free \lfloor m \circ (f \otimes M) \rfloor = \lfloor m \rfloor \circ free\, f$)

$$
\begin{array}{ll}
& eval_M\, e \circ \lfloor m \rfloor \circ free\, f \circ \mu \\
= & \quad (eval_M\, e \circ \lfloor m \rfloor = M) \\
& free\, f \circ \mu \\
= & \quad (\text{property } free) \\
& m \circ (free\, f \otimes free\, f) \\
= & \quad (eval_M\, e \circ \lfloor m \rfloor = M) \\
& m \circ ((eval_M\, e \circ \lfloor m \rfloor \circ free\, f) \otimes (eval_M\, e \circ \lfloor m \rfloor \circ free\, f)) \\
= & \quad (free\, \lfloor m \circ (f \otimes M) \rfloor = \lfloor m \rfloor \circ free\, f) \\
& m \circ ((eval_M\, e \circ free\, \lfloor m \circ (f \otimes M) \rfloor) \otimes (eval_M\, e \circ free\, \lfloor m \circ (f \otimes M) \rfloor)) \\
= & \quad (\text{def. } (\!|-|\!)) \\
& m \circ ((\!|[e, m \circ (f \otimes M)]|\!) \otimes (\!|[e, m \circ (f \otimes M)]|\!))
\end{array}
$$

Equality $free\, \lfloor m \circ (f \otimes M) \rfloor = \lfloor m \rfloor \circ free\, f$ holds since

$$
\begin{array}{ll}
& \lfloor m \rfloor \circ free\, f \circ ins \\
= & \quad (\text{property } free) \\
& \lfloor m \rfloor \circ f \\
= & \quad (\text{naturality } \lfloor - \rfloor) \\
& \lfloor m \circ (f \otimes M) \rfloor
\end{array}
$$

, and $\lfloor m \rfloor$ and $free\, f$ are both monoid homorphisms and thus their composition is a monoid homomorphism, meaning $\lfloor m \rfloor \circ free\, f \circ \varepsilon = \dot{e}$ and $\lfloor m \rfloor \circ free\, f \circ \mu = \dot{m} \circ (\lfloor m \rfloor \circ free\, f \otimes \lfloor m \rfloor \circ free\, f)$. Since $free\, \lfloor m \circ (f \otimes M) \rfloor$ is the unique monoid homomorphism, $free\, \lfloor m \circ (f \otimes M) \rfloor = \lfloor m \rfloor \circ free\, f$.

□

C.6 Coherency Properties $(\!|-|\!)$

The $(\!|[a,b]|\!)$ morphism should have the same properties as in the initial algebra basis, resulting in 2 coherency properties.

C.6.1 Property 1

We prove that $(\!|[a,b]|\!) \circ \varepsilon = a$.

Proof

$$
\begin{array}{ll}
& (\!|[a,b]|\!) \circ \varepsilon \\
= & \quad (\text{def. } (\!|[a,b]|\!)) \\
& eval_X\, a \circ free\, \lfloor b \rfloor \circ \varepsilon \\
= & \quad (\text{property } free) \\
& eval_X\, a \circ \lfloor \lambda_X \rfloor \\
= & \quad (\text{property } eval) \\
& a
\end{array}
$$

□

C.6.2 Property 2

We prove that $(\!|[a,b]|\!) \circ \iota = b \circ (\Sigma \otimes (\!|[a,b]|\!))$.

Proof

$$
\begin{aligned}
& (\!|[a,b]|\!) \circ \iota \\
= \quad & \text{(def. of } \iota \text{ and } (\!|[a,b]|\!)) \\
& \mathit{eval}_X\, a \circ \mathit{free}\, \lfloor b \rfloor \circ \mu \circ (\mathit{ins} \otimes \Sigma^*) \\
= \quad & \text{(property } \mathit{free}) \\
& \mathit{eval}_X\, a \circ \dot{m} \circ (\mathit{free}\, \lfloor b \rfloor \otimes \mathit{free}\, \lfloor b \rfloor) \circ (\mathit{ins} \otimes \Sigma^*) \\
= \quad & \text{(bifunctor } \otimes \text{ \& } \mathit{free}\ f \circ \mathit{ins} = f) \\
& \mathit{eval}_X\, a \circ \dot{m} \circ (\lfloor b \rfloor \otimes \mathit{free}\, \lfloor b \rfloor) \\
= \quad & \text{(def. } \mathit{eval}_X\, a) \\
& \mathit{ev}_X \circ (X^X \otimes a) \circ \rho^{-1}_{X^X} \circ \dot{m} \circ (\lfloor b \rfloor \otimes \mathit{free}\, \lfloor b \rfloor) \\
= \quad & (\rho^{-1} \text{ natural transformation}) \\
& \mathit{ev}_X \circ (X^X \otimes a) \circ (\dot{m} \otimes I) \circ \rho^{-1}_{X^X \otimes X^X} \circ (\lfloor b \rfloor \otimes \mathit{free}\, \lfloor b \rfloor) \\
= \quad & \text{(bifunctor } \otimes) \\
& \mathit{ev}_X \circ (\dot{m} \otimes X) \circ ((X^X \otimes X^X) \otimes a) \circ \rho^{-1}_{X^X \otimes X^X} \\
& \quad \circ (\lfloor b \rfloor \otimes \mathit{free}\, \lfloor b \rfloor) \\
= \quad & \text{(def. } \mathit{ev}_X \text{ and } \dot{m}) \\
& \lceil X^X \rceil \circ (\lfloor \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ \alpha^{-1} \rfloor \otimes X) \\
& \quad \circ ((X^X \otimes X^X) \otimes a) \circ \rho^{-1}_{X^X \otimes X^X} \circ (\lfloor b \rfloor \otimes \mathit{free}\, \lfloor b \rfloor) \\
= \quad & \text{(naturality } \lceil - \rceil \text{ \& inverses)} \\
& \lceil X^X \rceil \circ (X^X \otimes \mathit{ev}_X) \circ \alpha^{-1} \circ ((X^X \otimes X^X) \otimes a) \circ \rho^{-1}_{X^X \otimes X^X} \\
& \quad \circ (\lfloor b \rfloor \otimes \mathit{free}\, \lfloor b \rfloor) \\
= \quad & (\alpha^{-1} \text{ natural transformation}) \\
& \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ (X^X \otimes (X^X \otimes a)) \circ \alpha^{-1} \circ \rho^{-1}_{X^X \otimes X^X} \\
& \quad \circ (\lfloor b \rfloor \otimes \mathit{free}\, \lfloor b \rfloor) \\
= \quad & \text{(property } \alpha^{-1} \circ \rho^{-1} = \mathit{id} \otimes \rho^{-1}) \\
& \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ (X^X \otimes (X^X \otimes a)) \circ (X^X \otimes \rho^{-1}_{X^X}) \\
& \quad \circ (\lfloor b \rfloor \otimes \mathit{free}\, \lfloor b \rfloor) \\
= \quad & \text{(bifunctor } \otimes) \\
& \lceil X^X \rceil \circ (X^X \otimes (\lceil X^X \rceil \circ (X^X \otimes a) \circ \rho^{-1}_{X^X} \circ \mathit{free}\, \lfloor b \rfloor)) \\
= \quad & \text{(def. of } \mathit{eval}_X\, a) \\
& \lceil X^X \rceil \circ (X^X \otimes (\mathit{eval}_X\, a \circ \mathit{free}\, \lfloor b \rfloor)) \circ\ (\lfloor b \rfloor \otimes \Sigma^*) \\
= \quad & \text{(def. of } (\!|[a,b]|\!)) \\
& \lceil X^X \rceil \circ (X^X \otimes (\!|[a,b]|\!)) \circ (\lfloor b \rfloor \otimes \Sigma^*) \\
= \quad & \text{(bifunctor } \otimes) \\
& \lceil X^X \rceil \circ (\lfloor b \rfloor \otimes X^X) \circ (\Sigma \otimes (\!|[a,b]|\!)) \\
= \quad & \text{(naturality } \lceil - \rceil \text{ \& inverses)} \\
& b \circ (\Sigma \otimes (\!|[a,b]|\!))
\end{aligned}
$$

□

D Coherency Properties `handle`

The `handle _ with h` operation in the initial algebra basis should have the same properties as in the free algebra basis. This results in 2 coherency properties for the operation and value rule respectively.

In the following proofs, we assume the following is defined:

```
h =
  handler
  | val (a: A) -> ...: X                                        (v)
  | op_i (p_i: P_i,k: N_i -> X) -> ...: X                       (c_i)

ih =
   ihandler
   | ε (a: A)
       -> λ(f: A -> X). f a                                     (e)
   | op_i (p_i: P_i,k: N_i -> ((A -> X) -> X))
       -> λ(f:A -> X). c_i (p_i, λ(n: N_i). k n f)              (g_i)
```

D.1 Coherency Property: Value Rule

```
  handle (x: A) with h
= (def. handle)
  (ihandle (x: A) with ih) v
= (ε rule)
  (e x) v
= (def. e)
  ((λ a. λf. f a) x) v
= (application)
  (λf. f x) v
= (application)
  v x
```

D.2 Coherency Property: Operation Rule

```
  handle (op_i (p: P_i, ◇: N_i -> Σ*A)) with h
= (def. handle)
  (ihandle (op_i (p: P_i, ◇: N_i -> Σ*A)) with ih) v
= (ι rule)
  (g_i (p, λn. ihandle (◇ n) with ih)) v
= (def. g_i)
  ((λ(p_i,k). λf. c_i (p_i, λn. k n f))
      (p, λn. ihandle (◇ n) with ih)) v
= (α-renaming & application)
  (λf. c_i (p, λn. (λx. ihandle (◇ x) with ih) n f)) v
= (application)
  c_i (p, (λn. λx. ihandle (◇ x) with ih) n v))
= (application)
```

```
  c_i (p, (λn. ihandle (◇ n) with ih) v)
= (def. handle)
  c_i (p, λn. handle (◇ n) with h)
```

E Conversion Diagram

We show that both paths of the diagram are equal to $(\!|[i', a' \circ (g \oplus FX)]|\!)$. First we repeat some relevant definitions, then prove the algebra conversion path and lastly prove the program conversion path.

E.1 Definitions

E.1.1 Signature Conversion

$$\begin{aligned} f &: L\Xi \to \Sigma \\ g &: \Xi \to R\Sigma = [\![f]\!] \end{aligned}$$

E.1.2 Algebra Conversion

The original algebra components are:

$$\begin{aligned} i &: J \to X \\ a &: \Sigma \oplus X \to X \end{aligned}$$

The transformed algebra components are:

$$\begin{aligned} i' &: I \to RX = Ri \circ \phi^0 \\ a' &: R\Sigma \otimes RX \to RX = Ra \circ \phi \end{aligned}$$

E.2 Algebra Conversion

We have to show that $h' \circ hoist\, g = (\!|[i', a' \circ (g \otimes RX)]|\!)$.

Proof
Both $h' = (\!|[i', a']|\!)$ and $hoist\, g = (\!|[\varepsilon, \iota \circ (g \otimes (R\Sigma)^{\otimes})]|\!)$ are algebra homomorphisms. Their composition is equal to $(\!|[i', a' \circ (g \otimes RX)]|\!)$ since it is the unique algebra homomorphism. □

E.3 Program Conversion

Both $h = (\!|[i, a]|\!)$ and $hoist\, f = (\!|[\varepsilon, \iota \circ (f \oplus \Sigma^{\oplus})]|\!)$ are algebra homomorphisms. Their composition is an algebra homomorphism and thus it is equal to $ah = (\!|[i, a \circ (f \oplus X)]|\!)$ since it is unique.

$$\begin{aligned} ah &= (\!|[i, a \circ (f \oplus X)]|\!) \\ convert &= free\, [\![ins_{L\Xi}]\!] \\ &= (\!|[R\varepsilon \circ \phi^0, R\mu \circ \phi \circ ([\![ins_{L\Xi}]\!] \oplus R((L\Xi)^{\oplus}))]|\!) \end{aligned}$$

We have to show that $Rh \circ R(hoist\, f) \circ convert = R(ah) \circ convert = (\!|[i', a' \circ (g \otimes RX)]|\!)$. We show that $R(ah) \circ convert$ is an algebra homomorphism $\Xi^{\otimes} \to RX$, then it is equal to $(\!|[i', a' \circ (g \otimes RX)]|\!)$ due to its uniqueness.

Proof

$$
\begin{aligned}
& R(ah) \circ convert \circ \varepsilon \\
= \quad & \text{(def. } convert \text{ \& property } (\!|-|\!)) \\
& R(ah) \circ R\varepsilon \circ \phi^0 \\
= \quad & \text{(functor } R) \\
& R(ah \circ \varepsilon) \circ \phi^0 \\
= \quad & \text{(def. } ah \text{ \& property } (\!|-|\!)) \\
& Ri \circ \phi^0 \\
= \quad & \text{(def. } i') \\
& i'
\end{aligned}
$$

$$
\begin{aligned}
& R(ah) \circ convert \circ \iota \\
= \quad & \text{(def. } convert \text{ \& property } (\!|-|\!)) \\
& R(ah) \circ R\mu \circ \phi \circ (\llbracket ins_{L\Xi} \rrbracket \otimes R((L\Xi)^{\oplus})) \circ (\Xi \otimes convert) \\
= \quad & \text{(bifunctor } \otimes) \\
& R(ah) \circ R\mu \circ \phi \circ (\llbracket ins_{L\Xi} \rrbracket \otimes convert) \\
= \quad & \text{(naturality } \llbracket - \rrbracket) \\
& R(ah) \circ R\mu \circ \phi \circ ((R(ins_{L\Xi}) \circ \llbracket L\Xi \rrbracket) \otimes convert) \\
= \quad & \text{(bifunctor } \otimes) \\
& R(ah) \circ R\mu \circ \phi \circ (R(ins_{L\Xi}) \otimes R((L\Xi)^{\oplus})) \circ (\llbracket L\Xi \rrbracket \otimes convert) \\
= \quad & (\phi \text{ is a natural transformation}) \\
& R(ah) \circ R\mu \circ R(ins_{L\Xi} \oplus (L\Xi)^{\oplus}) \circ \phi \circ (\llbracket L\Xi \rrbracket \otimes convert) \\
= \quad & \text{(def. } \iota) \\
& R(ah) \circ R\iota \circ \phi \circ (\llbracket L\Xi \rrbracket \otimes convert) \\
= \quad & \text{(functor } R) \\
& R(ah \circ \iota) \circ \phi \circ (\llbracket L\Xi \rrbracket \otimes convert) \\
= \quad & \text{(def. } ah \text{ \& property } (\!|-|\!)) \\
& R(a \circ (f \oplus X) \circ (L\Xi \oplus ah)) \circ \phi \circ (\llbracket L\Xi \rrbracket \otimes convert) \\
= \quad & \text{(functor } R) \\
& Ra \circ R(f \oplus X) \circ R(L\Xi \oplus ah) \circ \phi \circ (\llbracket L\Xi \rrbracket \otimes convert) \\
= \quad & (\phi \text{ is a natural transformation}) \\
& Ra \circ \phi \circ (Rf \otimes RX) \circ (R(L\Xi) \otimes R(ah)) \circ (\llbracket L\Xi \rrbracket \otimes convert) \\
= \quad & \text{(bifunctor } \otimes) \\
& Ra \circ \phi \circ ((Rf \circ \llbracket L\Xi \rrbracket) \otimes RX) \circ (\Xi \otimes (R(ah) \circ convert)) \\
= \quad & \text{(naturality } \llbracket - \rrbracket) \\
& Ra \circ \phi \circ (\llbracket f \rrbracket \otimes RX) \circ (\Xi \otimes (R(ah) \circ convert)) \\
= \quad & \text{(def. } g \text{ \& def. } a') \\
& a' \circ (g \otimes RX) \circ (\Xi \otimes (R(ah) \circ convert))
\end{aligned}
$$

□