Under consideration for publication in J. Functional Programming

Not by Equations Alone

Reasoning with Extensible Effects

OLEG KISELYOV
Tohoku University, Japan
SHIN-CHENG MU
Academia Sinica, Taiwan
AMR SABRY
Indiana University, USA

Abstract

The challenge of reasoning about programs with (multiple) effects such as mutation, jumps or IO dates back to the inception of program semantics in the works of Strachey and Landin. Using monads to represent individual effects and the associated equational laws to reason about them proved exceptionally effective. Even then it is not always clear what laws are to be associated with a monad – for a good reason, as we show for non-determinism. Combining expressions using different effects brings challenges not just for monads, which do not compose, but also for equational reasoning: the interaction of effects may invalidate their individual laws, as well as induce emerging properties that are not apparent in the semantics of individual effects. Overall, the problems are judging the adequacy of a law; determining if or when a law continues to hold upon addition of new effects; and obtaining and easily verifying emergent laws.

We present a solution relying on the framework of (algebraic, extensible) effects, which already proved itself for writing programs with multiple effects. Equipped with a fairly conventional denotational semantics, this framework turns useful, as we demonstrate, also for reasoning about and optimizing programs with multiple interacting effects. Unlike the conventional approach, equational laws are not imposed on programs/effect handlers, but induced from them: our starting point hence is a program (model), whose denotational semantics, besides being used directly, suggests and justifies equational laws and clarifies side-conditions. The main technical result is the introduction of the notion of *equivalence modulo handlers* ('modulo observation') or a particular combination of handlers – and proving it to be a *congruence*. It is hence usable for reasoning in any context, not just evaluation contexts – provided particular conditions are met.

Concretely, we describe several realistic handlers for non-determinism and elucidate their laws (some of which hold in the presence of any other effect). We demonstrate appropriate equational laws of non-determinism in the presence of global state, which have been a challenge to state and prove before.

1 Introduction

Although the algebraic approach to effects was proposed back in 2003 (Plotkin & Power, 2003) – and the denotational one about a quarter century ago (Cartwright & Felleisen, 1994) – only recently it has entered the mainstream. There are implementations of algebraic/extensible effects in almost every popular language, from C and JavaScript to OCaml,

Scala, Haskell and Idris. There are full-fledged languages built around effects, such as PureScript, Koka, Eff, Links, Fran(k) and Multicore OCaml. Algebraic/extensible effects are being increasingly used in industry. At long last it is becoming clear that algebraic/extensible effects deliver what monad transformers have struggled (Kiselyov *et al.*, 2013) to do: combine, in the same program, independently developed effectful components.

In many presentations on the topic of effects, one sort of question comes over and over again. Granting that algebraic/extensible effects may be useful in practice, how to reason about them? For example, what sort of equational laws – program transformations/optimizations – may be expected to hold? Our first motivation is therefore to understand how do the laws established for one effect change when another effect is added?

1.1 Equational reasoning for effectful functional programs: A critical view

The second motivation for the present paper is dissatisfaction with the current state of equational reasoning in functional programming with effects. To emphasize, there is no denying of the deserving popularity of equational reasoning and its many successes. It is enough to point out the exemplary (Gibbons & Hinze, 2011) for both points. Equational reasoning seems like the only game in town as far as monadic programming is concerned. Furthermore, the approach of algebraic effects (Plotkin & Power, 2003; Plotkin & Pretnar, 2009; Pretnar, 2010) is also motivated by and based on equational reasoning. The semantics of the effect operations is specified through an algebra: the handler deals with an implementation that satisfies the algebraic identities and the requester relies on such identities to reason about the client-side code.

Yet one cannot evade the questions:

1. Where do the equations – equalities on terms containing effectful operations – come from? When one defines a new effect, as often recommended[1], how does one find out which equations it satisfies, or ought to?
2. Most of the time the equations are given a priori, as a specification of an effectful operation. How does one then verify, with as little hand waving as possible, that a particular implementation (program) satisfies them?
3. In fact, how does one ensure that a set of equalities, taken as a specification, is implementable? Is there a non-trivial model of the desired equational theory? This is not an idle concern: we have seen first-hand postulating so many equations to make the reasoning come through that the only model is trivial.
4. Even if the equational theory turns out to have a model, how practical is it? A rich equational theory does little good if it exerts unrealistic demands on its implementations. This turns out to be the case for non-determinism, as we discuss below.
5. How to realistically combine equational theories for several effects? Although there exist theoretical approaches (Plotkin & Power, 2003; Hyland *et al.*, 2006) how useful are they in practice? That is, how to avoid ending up with an excessively weak, and hence, useless combined theory?

[1] For example, in the tutorial `http://okmij.org/ftp/Haskell/extensible/tutorial.html`

1.2 Contributions

To emphasize, we do not reject equational reasoning. On the contrary, we want more equational laws – but worry about their adequacy. Therefore, we do not postulate laws and do not take them as a specification for writing handlers, which may turn out useless. We start with already proven useful handler implementations and ask what equations/properties they provide/assure, what laws can be distilled from them. Thus, to us, the equational laws come a posteriori rather than a priori – reminding one of 'definability in a model' in model theory.

One can see the close parallel with Dijkstra's and Harel-Pratt's dueling approaches to program correctness – the argument pursued in the late 1970s and early 1980s. Dijkstra advocated a priori correctness: start from a specification, formulated around the notion of a weakest precondition, and design the program that meets it. Harel and Pratt argued for, as they put it, "a much healthier 'bottom up approach"'. Their argument (cited from Armoni & Ben-Ari (2009, §7.1)) strongly resonates with us:

> *"[F]irst defining the semantics objects (states and binary relations), and only then introducing the logical language and assigning meaning. . . . At this point the truth of the formulae of the logical language has already been determined (and in a plausible way!), and the 'axioms' of Dijkstra's definition can then be verified as mere theorems."*
> *Harel & Pratt (1978, p. 210)*

We aspire to pursue the same approach, but for arbitrary algebraic effects. Our contributions are twofold.

Theoretically, we introduce the notion of equational laws (term congruences) *modulo a particular handler* or a combination of handlers. Such equivalence modulo (handler, observation) is often invoked, informally (e.g., Hinze (2000), Fischer et al. (2011)). To be useful, e.g., to optimize within function bodies, it has to be a congruence, applicable within arbitrary contexts. That fact is difficult to prove, partly because it is generally not true. It becomes true only under certain conditions. We state these conditions and demonstrate the congruence.

Practically, we show which equational laws hold for frequently used handlers for nondeterminism (the List Monad, the Maybe Monad, the 'once' observation, the general depth-first search strategy handler), and determine if these laws continue to hold (unconditionally or under some conditions) when other effects, or just State, are added. In the case nondeterminism is combined with global state, we demonstrate why some 'obvious' (even to us, at some point) equivalences do not in fact hold, and non-obvious equivalences that do (Lemma 12 and 13). In the Appendix, we show how to concretely use the derived laws to reason about and optimize interesting programs with multiple effects.

Coming back to questions in §1.1, we answer Q1 and Q2 by computing denotations of handlers and finding expressions whose denotations, in arbitrary evaluation contexts, become equal when composed with the handler denotation. The other questions do not even come up: we start with the handlers or a particular combination of them. Therefore, we already have a practically useful model of the laws.

1.3 Paper Structure

The next section tells the whole story, but on an exasperatingly simple example. It explains the main theoretical contribution of the equivalence modulo handlers, and shows the easy-to-understand proof that it is really an equivalence. One could potentially stop reading at that point. Presumably, however, one is interested if this notion can be applied to realistic and interesting examples – which is the topic of the rest of the paper.

We start §3 with a model of a realistic programming language – higher-order calculus with arbitrary effect operations and arbitrary ways of handling them – and describe its type system and a fairly conventional denotational semantics. §3.4 introduces equational laws that are valid for any interpretation of effects; §3.5 shows an example of such 'free' laws for non-determinism.

The central §4 introduces the notion of equivalence modulo handler, and proves, under some conditions, that it is context compatible and is hence a congruence. The following §5 gives an example, the familiar Get-Put laws of state, but in a more general form – which hold only modulo (the commonly used) state effect handler. The Appendix shows the proof, and also the practical use to justify an implementation of a scanl list operation as a state-mutating foldr.

Handlers for non-determinism are introduced in §6. We clearly see which handlers validate which of the commonly considered so-called semi-lattice laws (idempotence, commutativity, and associativity), and what is the price to pay for supporting all of the semi-lattice laws. One of the surprising results is the proof that a handler that always picks the first choice satisfies (Idem) and (Assoc) (but not (Comm)) – in the presence of any other effect.

Finally, §7 combines non-determinism and state, which can be done in two ways. §7.1 explores the laws that arise from one combination, so-called 'local state'. §7.2 describes the challenges and successes of reasoning with non-determinism and global state.

The Appendix gives detailed derivations and practical examples, taken from the project of deriving efficient *n*-queens solvers.

As an overarching notational convention, we use sans-serif – as in **let** x=get **in** x+1 and foldl (⩒x.⩒y.x+y) 0 – to talk about programs/terms; we use the symbol $\equiv$ (possibly adorned with subscripts) to talk about term equivalences, as defined in §3 and §4. Term denotations are written in the mathematical font – e.g., $\mathtt{E}(\mathsf{Get}, \star, \lambda x.\, \mathtt{V}(x+1))$ and $\mathit{foldl}\ (\lambda xy.\, x+y)\ 0$ – with the ordinary (mathematical) $=$ to equate denotations.

The present paper is of theoretical nature, and our calculus of algebraic effects and handlers was developed as an abstraction of real implementations for ease of reasoning. Yet we have implemented the calculus itself, by embedding it into OCaml in the tagless-final style. The code is available at `http://okmij.org/ftp/Computation/eff-calculus.ml`. We have used the implementation to run all the examples in the paper; the equational laws discussed in the paper are not only proven but also tested.

We assume a small familiarity with the denotational semantics, at least at the level of Winskel (1993).

2 Preview

This section introduces all the ideas and tells the whole story, but on an exceedingly simple example. The rest of the paper re-tells the story in a quite more complicated way – but also with quite more interesting and realistic examples.

2.1 Handling Variables

To show the intuition for our approach we use an analogy between variables and effect operations.[2]

Let us consider the trivial calculus with integer constants $\mathsf{i} \in \mathbb{N}$, variables $\mathsf{x} \in \mathbb{V}$ and addition. A naïve semantics that maps terms to $\mathbb{N}$ has the obvious trouble with open terms like $\mathsf{x} + 1$: a variable has no definite mapping to an integer.

The common solution to deal with variables is to generalize the semantics so that terms denote maps from an *environment* to $\mathbb{N}$. The environment (denoted ρ) is a variable assignment, giving a meaning to each variable:

$$\begin{aligned}
[\![\mathsf{i}]\!]^{env} &= \lambda\rho.\,\mathsf{i} \\
[\![\mathsf{x}]\!]^{env} &= \lambda\rho.\,\rho(\mathsf{x}) \\
[\![\mathsf{e}_1 + \mathsf{e}_2]\!]^{env} &= \lambda\rho.\,[\![\mathsf{e}_1]\!]^{env}\,\rho + [\![\mathsf{e}_2]\!]^{env}\,\rho
\end{aligned}$$

This approach has become standard for a reason. One should however beware of its hardwired assumption that the meaning of a variable depends solely on its name. Such semantics therefore becomes inadequate if we want to use the calculus to model a (modern) computational system where accessing variables may take drastically different amount of time which we want to account for, may fail, or depend on access history or external factors. An alternative – general and compositional – approach has been developed in the Vienna school and culminated in the work of Cartwright & Felleisen (1994). The key idea is to disentangle the action of asking for the value of a variable from the action of answering with the value. The denotation of a variable simply asks the question and this question propagates until it encounters an appropriate *handler*. The handler, being an independent entity, may use other parameters (elapsed time, internal counters, etc.) to respond appropriately.

Formally, we use a more sophisticated semantic domain that can represent simple values (natural numbers) as well as collections of questions and answers. Such a domain, call it D, is the set of well-founded infinitely-branching trees of finite height. It can be concretely defined by the following (OCaml) inductive data type declaration

$$\textbf{type}\ \mathsf{d} = \mathsf{V}\ \textbf{of}\ \mathsf{int} \mid \mathsf{E}\ \textbf{of}\ \mathsf{var} * (\mathsf{int} \to \mathsf{d})$$

[2] This is not a mere simile: it can be traced to Reynolds' Idealized Algol (Reynolds, 1981) and explored in more detail by Kiselyov (2017).

(where var is the type of variable names $\mathbb{V}$), or, mathematically, as the minimal[3] solution to the following domain equation

$$D = \mathbb{N} + (\mathbb{V} \times D^{\mathbb{N}})$$

The pair $\mathbb{V} \times D^{\mathbb{N}}$ is a *question*, about the meaning of the variable given in the first component of the pair. The second component is a specification of what to do with the answer (we see an example shortly). Unlike the domain equation, the data type declaration explicitly spells out the constructors V and E, which can be used to construct the tree d and deconstruct/pattern-match on it. Analogously, we will be using `V` and `E` to denote inclusions/retracts into/from the two disjoint union components of the domain D.

Using this more sophisticated domain of denotations, the denotational semantics is then

$$[\![\mathsf{i}]\!] = \mathtt{V}(\mathsf{i}) \tag{1}$$

$$[\![\mathsf{x}]\!] = \mathtt{E}(\mathsf{x}, \lambda v.\, \mathtt{V}(v)) \tag{2}$$

$$[\![\mathsf{e}_1 + \mathsf{e}_2]\!] = (\lambda v_1.\, (\lambda v_2.\, \mathtt{V}(v_1 + v_2))^\dagger\, [\![\mathsf{e}_2]\!])^\dagger\, [\![\mathsf{e}_1]\!] \tag{3}$$

It maps integer literals to $\mathbb{N}$, and variables to questions about their value as a natural number. When this natural number is provided, it is injected in the domain of answers. In Eq. (3), the left subexpression gets to ask questions first. The notation $f^\dagger$ means lifting of a map $f : \mathbb{N} \to D$ to $D \to D$. It is inductively defined by the following two cases:

$$f^\dagger\ \mathtt{V}(n) = fn$$
$$f^\dagger\ \mathtt{E}(x,k) = \mathtt{E}(x, f^\dagger \circ k)$$

In other words, if the argument is a proper value, f applies to it. If the argument is a question, it is propagated upwards applying f again once the answer is received. All in all, Eqs. (1)-(3) map a term to a question-answer tree with integer leaves, which specifies the meaning of the term for each possible answer about the meaning of its variables.

Although we have not yet described how to answer, or handle, questions, we can already use the semantics to verify that $\mathsf{x}+1$ may always be replaced with $1+\mathsf{x}$ preserving meaning, in symbols:

$$\mathsf{x} + 1 \equiv 1 + \mathsf{x} \tag{4}$$

Indeed, the denotations of both sides are equal:

$$[\![x+1]\!] \quad = \quad \mathtt{E}(\mathsf{x}, (\lambda v.\, \mathtt{V}(v+1))) \quad = \quad \mathtt{E}(\mathsf{x}, (\lambda v.\, \mathtt{V}(1+v))) \quad = \quad [\![1+x]\!]$$

The middle equality holds because the domain-theoretic addition is commutative. We have thus validated an equational law, which holds for *any* question-answering strategy, i.e., for any handler.

The semantics can be used to verify many more such "free identities":

$$\begin{gathered} \mathsf{e} + \mathsf{i} \equiv \mathsf{i} + \mathsf{e} \\ (\mathsf{e}_1 + \mathsf{e}_2) + \mathsf{e}_3 \equiv \mathsf{e}_1 + (\mathsf{e}_2 + \mathsf{e}_3) \end{gathered} \tag{5}$$

[3] In the sense of possessing the minimal invariant property (Pitts, 1996)

for any integer i and expressions e, e_1, etc. This gives a hint about our answer to Question 1 in §1.1, about a way to derive algebraic identities. We start with a model, and choose identities that reflect its salient features.

Interestingly, the semantics so far does not validate an equation like:

$$\mathsf{x}_1 + \mathsf{x}_2 \equiv \mathsf{x}_2 + \mathsf{x}_1 \tag{6}$$

because it is not "free": it does not hold universally. For example, it is violated by a handler that maintains a global counter of the received questions, adding the counter to the answer. Our most important contribution is that we show how to reason about equations modulo particular handlers. For example, given a handler that always provides the same value to each variable, (6) holds. We may even show that for such handler, the general semantics coincides with $[\![-]\!]^{env}$.

Indeed, the environment semantics is a particular case of the tree semantics (1)-(3):

$$[\![\mathsf{e}]\!]^{env}\,\rho = \hbar\;\rho\;\;[\![\mathsf{e}]\!]$$

where $\hbar$, to be called the handler, is inductively defined by the two cases:

$$\hbar\;\rho\;\mathtt{V}(n) = n$$
$$\hbar\;\rho\;\mathtt{E}(x,k) = ((\hbar\;\rho)\circ k)\;\rho(x)$$

In other words, the handler takes a question-answer tree and a variable assignment ρ and uses the latter to answer all the questions in the former. The question-answering strategy provided by $\hbar$ (and hence, by the environment semantics) can verify the identity (6). We show the proof method next.

This is the taste of things to come: throughout the paper we will be encountering two sorts of semantics. One is the question-answer–tree semantics like $[\![-]\!]$ that presupposes no question-answering strategy. We often call such semantics 'free'; the identities (equational laws) it verifies are to be called 'free' as well. The other semantics, such as $[\![-]\!]^{env}$, rely on a particular question-answering strategy, and hence can verify more identities.

2.2 Induction principle

Let us look again at the domain D, which was inductively defined as $D = \mathbb{N} + (\mathbb{V} \times D^{\mathbb{N}})$. It is the domain of trees of finite height (but with ω branching factor). Therefore, it lets us prove properties and define functions and relations, by the familiar structural/height induction.

For example, we can prove that $(\lambda x.\,\mathtt{V}(x))^\dagger = \lambda y.\,y$, which is the analogue of the right unit monad law (see also Lemma 1). Indeed, in the base case, $(\lambda x.\,\mathtt{V}(x))^\dagger\mathtt{V}(v) = \mathtt{V}(v)$. The inductive case $(\lambda x.\,\mathtt{V}(x))^\dagger\mathtt{E}(\mathsf{x},k)$ is also the straightforward application of the induction hypothesis $(\lambda x.\,\mathtt{V}(x))^\dagger \circ k = k$.

A more interesting (and related to the central contribution) example is the inductive construction of the following relation $\approx$ on semantic domains and maps between the domains.

$$\frac{n \in \mathbb{N}}{n \approx n} \qquad \frac{\forall i \in \mathbb{N}.\;k_1\;i \approx k_2\;i}{\mathtt{E}(\mathsf{x},k_1) \approx \mathtt{E}(\mathsf{x},k_2)} \qquad \frac{v_1 \approx v_2}{\mathtt{V}(v_1) \approx \mathtt{V}(v_2)}$$

$$\frac{\forall i \in \mathbb{N}.\ k_1\ i \approx k_2\ i}{\mathtt{E}(\mathtt{x}_1, \lambda x.\, \mathtt{E}(\mathtt{x}_2, \lambda y.\, k_1\ (x+y))) \approx \mathtt{E}(\mathtt{x}_2, \lambda x.\, \mathtt{E}(\mathtt{x}_1, \lambda y.\, k_2\ (x+y)))}$$

$$\frac{\forall x_1 x_2 \in X.\, x_1 \approx x_2 \text{ implies } f_1 x_1 \approx f_2 x_2}{f_1 \approx f_2}\ f_1, f_2 \in Y^X$$

Here, $\mathtt{x}_1$ and $\mathtt{x}_2$ are distinguished variable names. The relation $\approx$ is clearly symmetric and transitive, but not reflexive on all domains and maps. For example, if a map $f : D^D$ could distinguish $\mathtt{E}(\mathtt{x}_1, \lambda x.\, \mathtt{E}(\mathtt{x}_2, \lambda y.\, \mathtt{V}(x+y)))$ from $\mathtt{E}(\mathtt{x}_2, \lambda x.\, \mathtt{E}(\mathtt{x}_1, \lambda y.\, \mathtt{V}(x+y)))$ then $f \not\approx f$. One can easily prove that if two trees $d_1, d_2 \in D$ are such that $d_1 \approx d_2$, their heights are the same.

The height induction on D lets us prove that if $f_1, f_2 \in D^{\mathbb{N}}$ such that $f_1 \approx f_2$ then $f_1^\dagger \approx f_2^\dagger$. In other words, for all $d_1, d_2 \in D$ such that $d_1 \approx d_2$, $f_1^\dagger d_1 \approx f_2^\dagger d_2$. In the base case of height zero, $d_1 = \mathtt{V}(v_1)$. It must then be $d_2 = \mathtt{V}(v_2)$ where $v_1 \approx v_2$, and the conclusion immediately follows from the premise. In the inductive case, suppose $f_1^\dagger d_1' \approx f_2^\dagger d_2'$. for all related d_1', d_2' of height less than some m. Let d_1 be $\mathtt{E}(\mathtt{x}, k_1)$ of height m. There are two cases to consider. In the general case, the related d_2 is also of the form $\mathtt{E}(\mathtt{x}, k_2)$, and for all $i \in \mathbb{N}$, $k_1\ i \approx k_2\ i$. The trees $k_1\ i$ and $k_2\ i$ are of height at most $m-1$. The induction hypothesis applies and gives $f_1^\dagger (k_1\ i) \approx f_2^\dagger (k_2\ i)$. The conclusion immediately follows from the definition of lifting and $\approx$. In the special case of $d_1 = \mathtt{E}(\mathtt{x}_1, \lambda x.\, \mathtt{E}(\mathtt{x}_2, \lambda y.\, k_1\ (x+y)))$ the related d_2 may also be of the form $d_2 = \mathtt{E}(\mathtt{x}_2, \lambda x.\, \mathtt{E}(\mathtt{x}_1, \lambda y.\, k_2\ (x+y)))$. Then $f_1^\dagger d_1 = \mathtt{E}(\mathtt{x}_1, \lambda x.\, \mathtt{E}(\mathtt{x}_2, \lambda y.\, f_1^\dagger (k_1\ (x+y))))$ and similarly for $f_2^\dagger d_2$. Again, the induction hypothesis applies and the conclusion follows.

The just proven property and (3) shows that if $[\![\mathsf{e}_1]\!] \approx [\![\mathsf{e'}_1]\!]$ and $[\![\mathsf{e}_2]\!] \approx [\![\mathsf{e'}_2]\!]$ then $[\![\mathsf{e}_1 + \mathsf{e}_2]\!] \approx [\![\mathsf{e'}_1 + \mathsf{e'}_2]\!]$. Let us write $\mathsf{e} \equiv_h \mathsf{e'}$ just in case $[\![\mathsf{e}]\!] \approx [\![\mathsf{e'}]\!]$. What we have proven is that $\equiv_h$ is context-compatible. Then the fact that $\approx$ is an equivalence relation on D gives us that $\equiv_h$ is a congruence. Such result manifests the central contribution of the paper: equivalence modulo handler is a congruence (in the just considered case, unconditionally). Therefore, if there is a handler $\hbar$ that does not distinguish $\mathsf{x}_1 + \mathsf{x}_2$ from $\mathsf{x}_2 + \mathsf{x}_1$, then we may replace the former expression with the later in *any context* without affecting the program result, provided the program is handled by $\hbar$.

3 Basic Calculus with Effects

We now extend the toy calculus of §2 to a model of a realistic, higher-order programming language supporting arbitrary effects, beyond the 'get the value of a variable'. The full calculus is inspired and greatly influenced by Eff 3.1 and its formalizations (Bauer & Pretnar, 2015; Bauer & Pretnar, 2014), with many big and small differences noted in §8. The next section presents the syntax; §3.2 the type system, §3.3 the denotational semantics, §3.4 examples of equational reasoning, and §3.5 reasoning about an example effect: nondeterminism. The calculus is designed to model typical programs written in Haskell, Scala, Multicore OCaml, etc., using one of many algebraic effect libraries.

The salient feature is the separation of effectful and pure operations. For example, in many programming languages total function applications such as `odd 42`, and invocations of procedures on potentially effectful expressions like `print x[1]` look the same, as juxtapositions. We distinguish them syntactically: the effectful application is denoted as $\circledast$.

The calculus is bare, with just a dash of syntax sugar. We tried to strike the balance between minimality (so it is easy to reason about) and convenience (so one can easily write – and read – interesting programs in it).

3.1 Syntax

Figure 1 presents the basic calculus: specific effects are added later, by specifying the inhabitants of the sort o, of effect constants; see §3.5 for an example. The rest of this section explains the figure line by line. The calculus is simply-typed.

Variables	x,y,z,u,k,l...
Constants	c ::= (), **true**, **false**, 0, 1, ... +, −, [], ...
Effects	o, to be defined as needed
Types	t,u,a,b ::= unit | int | bool | t list | t → t | t ⇒ t | t ↪ t
Values	v ::= x | c | ⩤x.v | **rec** f x. e | v v | **op** o
Expressions (Computations)	e ::= **val** v | e ⊛ e | **if** e **then** e **else** e | **handle** e **with** {**val** x → e ∣ o x k → e ...}
Evaluation Contexts	C ::= [] | C ⊛ e | **val** v ⊛ C | **if** C **then** e **else** e | **handle** C **with** {**val** x → e ∣ o x k → e ...}
Syntax sugar	**fn** x.e ≔ **rec** f x.e (f not free **in** e) **let** x = e_1 **in** e_2 ≔ **val** (**fn** x. e_2) ⊛ e_1 v ⊙ e ≔ **val** (**fn** x. **val** (v x)) ⊛ e e_1; e_2 ≔ **let** x = e_1 **in** e_2 (x not free **in** e_2) **fn** (). e ≔ **fn** (x:unit). e (x not free **in** e)

Fig. 1. The Basic Eff

The characteristic of the calculus is a thorough separation of computations that may have an effect or diverge, from non-divergent, non-effectful – so-called 'pure' – computations. The separation is evident already in types: $t_1 \to t_2$ is the type of pure functions, whose applications always terminate with no effect. In contrast, $t_1 \Rightarrow t_2$ is the type of general, effectful functions. The other types are basic (such as unit and integer), and data types such as list types.

The separation is also clear in the syntax of expressions and computations. Pure computations are represented by value terms v: variable references, built-in constants such as integer and boolean literals and addition, applications v_1 v_2 and pure abstractions ⩤x.v. The latter represent effect-free, total functions. Values of the type $t_1 \Rightarrow t_2$ are created by general abstractions and effect operations. The former, **rec** f x. e, represent (potentially) recursive and effectful functions; **op** o creates a function that does the effect o (we delay the description of effects and their types $u \hookrightarrow t$ till §3.2).[4] A frequent particular case of **rec** f x.e is of f not occurring free in e. We adopt an abbreviation **fn** x.e for such non-recursive, but still potentially effectful, functions.

[4] Like effect libraries in Haskell and other languages – and unlike Eff – we do not specifically distinguish effect instances: i.e., distinct and independently handled realizations of the same effect, created at run-time.

Expression terms e describe general, potentially effectful and non-terminating computations. A value v may be promoted to the (trivial) computation as **val** v. This promotion is ubiquitous, and easy to see from the context. Therefore, as a rule we shall elide **val**, except for emphasis. The conditional expressions are standard; the effect-handling form handle is described in §3.3.

Befitting the separation of pure and effectful computations, there are two sorts of functional applications, both left-associative. First is the pure application of values v_1 v_2 mentioned earlier. It is guaranteed by the type system to be effect-free and terminating. Next is the general application of expressions $e_1 \circledast e_2$: Either expression or the application itself may have an effect or diverge. We also introduce the notation for two particular forms of $\circledast$: one is let-expression and the other is the application $v \odot e$ of a pure, terminating function to an effectful expression; it corresponds to Haskell's fmap. Thus, **val** v $\circledast$ e (often written as v $\circledast$ e per our convention) is the call-by-value application of a potentially effectful function v. On the other hand, v $\odot$ e is the call-by-value application of a pure function v. Therefore, $v_1 \odot v_2$ is just v_1 v_2, as we shall see later. An evaluation context is an expression with the hole [] marking the next reducible expression. Filling the hole in the context C with some expression e is written C[e].

For notational convenience we use name : type $\coloneqq$ term to give a name to a term for easy reference in text and other terms. The type may be omitted. Notational definitions may also be parameterized by metavariables.

Our programs, like typical functional programs, use a number of data types: products, sums (variants), lists, etc. We only describe lists, the others are similar. Each data type is represented by its type constructor, such as t list, and the constructor and deconstructor constants. For lists, the constructors are the familiar [] (for the empty list), and :: of the type t $\to$ t list $\to$ t list. We write its applications in infix, as v :: vs (and the partial application as (v::)). The list deconstructor is the constant decons of the type (unit $\Rightarrow$ t') $\to$ (t $\to$ t list $\Rightarrow$ t') $\to$ t list $\Rightarrow$ t'. Using it directly is too cumbersome, so we adopt a bit of syntax sugar, writing the application

decons (**fn** (). onnil) (**fn** (x:t). (**fn** (xs:t list). oncons)) $\circledast$ xs

(where onnil and oncons are some terms) as:

match xs **with**
| [] $\to$ onnil
| (x::xs) $\to$ oncons

We will silently add other list-related constants such as map, foldr, etc. – and other data types such as pairs as needed.

3.2 Type System

The type system, Fig. 2, also draws the sharp distinction between pure terminating and generally effectful diverging computations, by using two different type judgements $\Gamma \vdash_v$ v:t for values and $\Gamma \vdash_e$ e:t for general expressions. In both judgements, the type environment Γ is a possibly empty sequence of associations x:t of variables with types.

Like Eff 3.1 or Multicore OCaml (but unlike the extensible-effects library in Haskell (Kiselyov & Ishii, 2015), for example) we do not further distinguish in types what sort of

$$\frac{\mathsf{x{:}t} \in \Gamma}{\Gamma \vdash_v \mathsf{x{:}\ t}} \qquad \frac{}{\Gamma \vdash_v + \mathsf{:\ int{\to}int{\to}int}}\ \text{Const} \qquad \frac{\Gamma \vdash_v \mathsf{v_1{:}\ t_1{\to}t_2} \quad \Gamma \vdash_v \mathsf{v_2{:}\ t_1}}{\Gamma \vdash_v \mathsf{v_1\ v_2{:}\ t_2}}$$

$$\frac{\vdash_o \mathsf{o{:}\ t_1{\hookrightarrow}t_2}}{\Gamma \vdash_v \mathbf{op}\ \mathsf{o{:}\ t_1{\Rightarrow}t_2}}\ \text{Op} \qquad \frac{\Gamma,\ \mathsf{x{:}t_1} \vdash_v \mathsf{v{:}\ t_2}}{\Gamma \vdash_v \mathsf{\lambda x.v{:}\ t_1{\to}t_2}} \qquad \frac{\Gamma,\ \mathsf{x{:}t_1,\ f{:}t_1{\Rightarrow}t_2} \vdash_e \mathsf{e{:}\ t_2}}{\Gamma \vdash_v \mathbf{rec}\ \mathsf{f\ x.e{:}\ t_1{\Rightarrow}t_2}}$$

$$\frac{\Gamma \vdash_v \mathsf{v{:}\ t}}{\Gamma \vdash_e \mathbf{val}\ \mathsf{v{:}\ t}} \qquad \frac{\Gamma \vdash_e \mathsf{e_1{:}t_1{\Rightarrow}t_2} \quad \Gamma \vdash_e \mathsf{e_2{:}\ t_1}}{\Gamma \vdash_e \mathsf{e_1} \circledast \mathsf{e_2{:}\ t_2}} \qquad \frac{\Gamma \vdash_e \mathsf{e{:}\ bool} \quad \Gamma \vdash_e \mathsf{e_1{:}\ t} \quad \Gamma \vdash_e \mathsf{e_2{:}\ t}}{\Gamma \vdash_e \mathbf{if}\ \mathsf{e}\ \mathbf{then}\ \mathsf{e_1}\ \mathbf{else}\ \mathsf{e_2{:}\ t}}$$

$$\frac{\Gamma \vdash_e \mathsf{e{:}\ t} \quad \Gamma,\ \mathsf{x_v{:}t} \vdash_e \mathsf{e_v{:}t'} \quad \vdash_o \mathsf{o_i{:}t_i{\hookrightarrow}u_i} \quad \Gamma,\ \mathsf{x_i{:}t_i,k{:}u_i{\Rightarrow}t'} \vdash_e \mathsf{e_i{:}\ t'}}{\Gamma \vdash_e \mathbf{handle}\ \mathsf{e}\ \mathbf{with}\ \{\mathbf{val}\ \mathsf{x_v} \to \mathsf{e_v} \mid \mathsf{o_i\ x_i\ k_i} \to \mathsf{e_i}\ \ldots\ \}\mathsf{:\ t'}}\ \text{Hnd}$$

Fig. 2. Type System

effect the computation may have. That is, our type system is not a type-and-effect system. Type-and-effect systems are a large area with significant progress in recent years (Bauer & Pretnar, 2014; Brady, 2013; Leijen, 2017). For the present paper about semantics, the simply-typed effect-less type system suffices.

Most of the typing rules are standard, adjusted to carefully distinguish pure and effectful computations and the corresponding judgements. The rule (Const) shows the typing judgement only for the addition constant; the others are analogous. Effects are described by constants of a special sort o, which are associated with (or, indexed by) a pair of types. We write this association as $\vdash_o \mathsf{o{:}\ t_1 \hookrightarrow t_2}$. As seen from the typing rule (Op), $\mathsf{t_1}$ is the type of the argument of the effect operation and $\mathsf{t_2}$ is the type of the result it may produce. §3.5 will show an example and explain the most complex typing rule (Hnd).

3.3 Denotational Semantics

The section extends the denotational semantics of the toy calculus of §2.1 to the full calculus, which is typed. We adopt the Church-style semantics, assigning meaning to typing derivations (represented by the judgement in their conclusions). The full calculus also permits generally recursive expressions. Correspondingly, we adopt CPOs as computational domains, with a least element denoted as $\bot$. Overall, the semantics is quite standard. In fact, it is almost the same as the (skeletal) semantics of Bauer & Pretnar (2014, §§5.1-5.3), to which we refer for details.

Our calculus clearly separates expressions (representing potentially effectful and divergent computations) from values (inert or total computations). Correspondingly, we use domains to give the semantics to the former, and pre-domains (without $\bot$) to the latter. Both domains and pre-domains are indexed by the type of the denoted expressions/values. Fig. 3 presents the assignment $\mathscr{T}[\mathsf{t}]$ of pre-domains and the assignment $\mathscr{R}[\mathsf{t}]$ of domains to types. If D_1, D_2 are (pre)domains, we write $D_1 \to D_2$ for a continuous map between them, which is also a domain. For pre-domains D_1, D_2 we write $\mathsf{l_1}{:}D_1 \times \mathsf{l_2}{:}D_2$ for the labeled product pre-domain: the set of pairs $<\mathsf{l_1}{:}d_1, \mathsf{l_2}{:}d_2>$, $d_1 \in D_1, d_2 \in D_2$. The components of the pair are identified by their labels l_i rather than their position. If p is such a labeled pair, we write $p.\mathsf{l}_i$ to access the l_i-th component, and $p \times \mathsf{l_3}{:}d_3$ to extend the pair with a new component. The set of all effect constants is denoted as $\mathbb{O}$.

$$\begin{array}{ll} \mathscr{T}[\mathsf{int}] = \mathbb{Z} & \mathscr{T}[\mathsf{t}_1 \to \mathsf{t}_2] = \mathscr{T}[\mathsf{t}_1] \to \mathscr{T}[\mathsf{t}_2] \\ \mathscr{T}[\mathsf{bool}] = \{\mathit{true}, \mathit{false}\} & \mathscr{T}[\mathsf{t}_1 \Rightarrow \mathsf{t}_2] = \mathscr{T}[\mathsf{t}_1] \to \mathscr{R}[\mathsf{t}_2] \\ \mathscr{T}[\mathsf{unit}] = \{\star\} & \mathscr{T}[\mathsf{x}_1{:}\mathsf{t}_1, \ldots, \mathsf{x}_n{:}\mathsf{t}_n] = \mathsf{x}_1{:}\mathscr{T}[\mathsf{t}_1] \times \cdots \times \mathsf{x}_n{:}\mathscr{T}[\mathsf{t}_n] \end{array}$$

$$\mathscr{R}[\mathsf{t}] = (\mathscr{T}[\mathsf{t}] + \{\ (\mathsf{o},\mathsf{v},\mathsf{k}) \mid \exists \mathsf{t}_1 \mathsf{t}_2.\ \mathsf{o} \in \mathbb{O}, \vdash_o \mathsf{o}{:}\mathsf{t}_1 \hookrightarrow \mathsf{t}_2, \mathsf{v} \in \mathscr{T}[\mathsf{t}_1],\ \mathsf{k} \in \mathscr{T}[\mathsf{t}_2] \to \mathscr{R}[\mathsf{t}]\ \})_\perp$$

Fig. 3. Semantic (pre)domains and the interpretation of types

The right-hand-side of $\mathscr{R}[\mathsf{t}]$ is to be read as a disjoint union, with the separately added $\perp$, of $\mathscr{T}[\mathsf{t}]$ and the triple product of $\mathbb{O}$, $\mathscr{T}[\mathsf{t}_1]$ and $\mathscr{T}[\mathsf{t}_2] \to \mathscr{R}[\mathsf{t}]$. As in §2.1, we write $\mathtt{V}$ and $\mathtt{E}$ for the 'tags' of that disjoint union (canonical retracts/inclusions).

Overall, Fig. 3 presents a set of mutually recursive equations for the domains and predomains. Such equations have already been considered (Bauer & Pretnar, 2014; Pitts, 1996) and the solutions exist. Furthermore, there exists a minimal solution (in the sense of possessing the minimal invariant property (Pitts, 1996)), which we adopt.[5] In the minimal solution, $\mathscr{R}[\mathsf{t}]$ may be regarded as a domain of trees (quite like those in §2.1), whose leaves may have $\perp$ and which are not necessarily of finite height. Nevertheless, there exists an induction principle, as in §2.2, which lets us define functions and relations on $\mathscr{R}[\mathsf{t}]$ and prove properties by induction (see (Bauer & Pretnar, 2014) for more discussion).

The semantic function $[\![\Gamma \vdash_v \mathsf{v}{:}\mathsf{t}]\!]\ \rho$ in Fig. 4 maps a type derivation and an environment to an element of $\mathscr{T}[\mathsf{t}]$. Here ρ is an element of the labeled product $\mathscr{T}[\Gamma]$. The semantics of built-in constants is standard: $\mathsf{0}$ means the integer zero, $+$ means integer addition, etc. In addition, the semantics of the list data type and its constants are presented in Fig. 5.

We also need a language to write semantics functions in. We use the simply-typed lambda-calculus (abstractions are written in Math font, as $\lambda x.x$) with added constants such as $\perp$, integers, booleans (*true* and *false*), the least fix-point operator fix and the function *if* defined as

$$\mathit{if}(b,x,y) \coloneqq \begin{cases} x & \text{if } b = \mathit{true} \\ y & \text{otherwise} \end{cases}$$

$$\begin{array}{ll} [\![\Gamma \vdash_v \mathsf{c}{:}\ \mathsf{t}]\!]\ \rho & \in \mathscr{T}[\mathsf{t}] \\ [\![\Gamma \vdash_v \mathsf{x}{:}\ \mathsf{t}]\!]\ \rho & = \rho.\mathsf{x} \\ [\![\Gamma \vdash_v \mathsf{v}_1\ \mathsf{v}_2{:}\ \mathsf{t}]\!]\ \rho & = ([\![\Gamma \vdash_v \mathsf{v}_1]\!]\rho)\ ([\![\Gamma \vdash_v \mathsf{v}_2]\!]\rho) \\ [\![\Gamma \vdash_v \diagdown\!\!\diagdown \mathsf{x}.\ \mathsf{v}_2{:}\ \mathsf{t}_1 \to \mathsf{t}_2]\!]\ \rho & = \lambda v.\ [\![\Gamma,\ \mathsf{x}{:}\mathsf{t}_1 \vdash_v \mathsf{v}_2{:}\mathsf{t}_2]\!]\ (\rho \times \mathsf{x}{:}v) \\ [\![\Gamma \vdash_v \mathbf{fn}\,\mathsf{x}.\ \mathsf{e}{:}\ \mathsf{t}_1 \Rightarrow \mathsf{t}_2]\!]\ \rho & = \lambda v.\ [\![\Gamma,\ \mathsf{x}{:}\mathsf{t}_1 \vdash_e \mathsf{e}{:}\mathsf{t}_2]\!]\ (\rho \times \mathsf{x}{:}v) \\ [\![\Gamma \vdash_v \mathbf{rec}\ \mathsf{f}\ \mathsf{x}.\mathsf{e}{:}\ \ \mathsf{t}_1 \Rightarrow \mathsf{t}_2]\!]\ \rho & = \mathsf{fix}\lambda f.\lambda v.\ [\![\Gamma,\mathsf{f}{:}\mathsf{t}_1 \Rightarrow \mathsf{t}_2,\mathsf{x}{:}\mathsf{t}_1 \vdash_e \mathsf{e}{:}\mathsf{t}_2]\!]\ (\rho \times \mathsf{f}{:}f \times \mathsf{x}{:}v) \\ [\![\Gamma \vdash_v \mathbf{op}\ \mathsf{o}{:}\ \ \mathsf{t}_1 \Rightarrow \mathsf{t}_2]\!]\ \rho & = \lambda v.\mathtt{E}(\mathsf{o}, v, \lambda x.\mathtt{V}(x)) \end{array}$$

Fig. 4. Denotational semantics of the value fragment

The semantics of expressions is shown in Fig. 6. For every $f\colon \mathscr{T}[\mathsf{u}] \to \mathscr{R}[\mathsf{t}]$ there exists a strict map $\mathscr{R}[\mathsf{u}] \to \mathscr{R}[\mathsf{t}]$, which we write as $f^\dagger$ and call 'lift'. It depends on f continuously and is defined as the least fixed point of F:

$$F\,g \coloneqq \lambda x. \begin{cases} f\ v & \text{if } x = \mathtt{V}(v) \\ \mathtt{E}(o, v, g \circ k) & \text{if } x = \mathtt{E}(o, v, k) \end{cases}$$

(We do not write the obvious, for a strict map, case $f^\dagger \perp = \perp$.) Frequently occurring is a special case of lift: lifting a pure function $f\colon \mathscr{T}[\mathsf{u}] \to \mathscr{T}[\mathsf{t}]$ to $\mathscr{R}[\mathsf{u}] \to \mathscr{R}[\mathsf{t}]$. It is

[5] The construction can be generalized by introducing monads (furthermore, free monads) – in the fruitful approach to denotational semantics pioneered by Moggi (1989).

$\mathscr{T}[\![\mathsf{t\ list}]\!] = [x_1, x_2, \ldots x_n],\ n \geq 0$
set of all finite sequences of elements of type t
$[\![\vdash_v [] \colon \mathsf{t\ list}]\!]\rho = []$
$[\![\vdash_v (::) \colon \mathsf{t} \to \mathsf{t\ list} \to \mathsf{t\ list}]\!]\rho$ = right-associative infix operation :: defined as
$x :: [x_1, x_2, \ldots x_n] \coloneqq [x, x_1, x_2, \ldots x_n]$
$[\![\vdash_v \mathsf{map} \colon (\mathsf{t}_1 \to \mathsf{t}_2) \to \mathsf{t}_1\ \mathsf{list} \to \mathsf{t}_2\ \mathsf{list}]\!]\rho$ = *map* where $\quad map\ f\ l \coloneqq [f\,x \mid x \in l]$

Fig. 5. Semantics of the list data type

denoted as $f^{\ddagger}$ and defined as $(\lambda v.\, \mathtt{V}(f\ v))^{\dagger}$. Yet another special case is the strict map from $g \colon \mathscr{R}[\mathsf{u} \Rightarrow \mathsf{t}]$ to $\mathscr{T}[\mathsf{u}] \to \mathscr{R}[\mathsf{t}]$ defined as $g^{\P} \coloneqq \lambda x.\, (\lambda r.\, r\ x)^{\dagger} g$. Whereas lifting is used to compute the denotation of a function to an effectful expression, $g^{\P}$ describes the application of an effectful expression to a value. The following two lemmas describe the often used properties of lifting. The lifting identity is trivially proven using the induction principle, similarly to §2.2. The proof of lifting composition is shown in the Appendix, and can serve as an illustration of proofs in denotational semantics.

Lemma 1 (*Lifting identity*)
$(\lambda x.\, \mathtt{V}(x))^{\dagger} = (\lambda x.\, x)^{\ddagger} = \lambda x.\, x$

Lemma 2 (*Lift compositions*)
$f^{\dagger} \circ g^{\dagger} = (f^{\dagger} \circ g)^{\dagger} \qquad f^{\ddagger} \circ g^{\dagger} = (f^{\ddagger} \circ g)^{\dagger} \qquad f^{\ddagger} \circ g^{\ddagger} = (f \circ g)^{\ddagger}$

$[\![\Gamma \vdash_e \mathbf{val}\ \mathsf{v} \colon \mathsf{t}]\!]\ \rho = \mathtt{V}([\![\Gamma \vdash_v \mathsf{v} \colon \mathsf{t}]\!]\rho)$
$[\![\Gamma \vdash_e \mathsf{e}_1 \circledast \mathsf{e}_2 \colon \mathsf{t}_2]\!]\ \rho = (\lambda f.\, f^{\dagger}\ [\![\Gamma \vdash_e \mathsf{e}_2 \colon \mathsf{t}_1]\!]\rho)^{\dagger}\ [\![\Gamma \vdash_e \mathsf{e}_1 \colon \mathsf{t}_1 \Rightarrow \mathsf{t}_2]\!]\rho$
$[\![\Gamma \vdash_e \mathbf{if}\ \mathsf{e}\ \mathbf{then}\ \mathsf{e}_1\ \mathbf{else}\ \mathsf{e}_2 \colon \mathsf{t}]\!]\ \rho = (\lambda b.\, \mathit{if}(b, [\![\Gamma \vdash_e \mathsf{e}_1 \colon \mathsf{t}]\!]\rho, [\![\Gamma \vdash_e \mathsf{e}_2 \colon \mathsf{t}]\!]\rho))^{\dagger}\ [\![\Gamma \vdash_e \mathsf{e} \colon \mathsf{bool}]\!]\rho$
$[\![\Gamma \vdash_e \mathbf{handle}\ \mathsf{e}\ \mathbf{with}\ \ldots]\!]\ \rho$ see text

$[\![\Gamma \vdash_e \mathbf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2 \colon \mathsf{t}_2]\!]\ \rho = (\lambda v.\ [\![\Gamma, \mathsf{x} \colon \mathsf{t}_1 \vdash_e \mathsf{e}_2 \colon \mathsf{t}_2]\!](\rho \times \mathsf{x} \colon v))^{\dagger}\ [\![\Gamma \vdash_e \mathsf{e}_1 \colon \mathsf{t}_1]\!]\rho$
$[\![\Gamma \vdash_e \mathsf{v} \odot \mathsf{e} \colon \mathsf{t}_2]\!]\ \rho = ([\![\Gamma \vdash_v \mathsf{v} \colon \mathsf{t}_1 \to \mathsf{t}_2]\!]\rho)^{\ddagger}\ [\![\Gamma \vdash_e \mathsf{e} \colon \mathsf{t}_1]\!]\rho$
$[\![\Gamma \vdash_e \mathsf{e}_1; \mathsf{e}_2 \colon \mathsf{t}_2]\!]\ \rho = (\lambda\_.\ [\![\Gamma \vdash_e \mathsf{e}_2 \colon \mathsf{t}_2]\!]\rho)^{\dagger}\ [\![\Gamma \vdash_e \mathsf{e}_1 \colon \mathsf{unit}]\!]\rho$

Fig. 6. Denotational semantics of expressions

For handlers, $[\![\Gamma \vdash_e \mathbf{handle}\ \mathsf{e}\ \mathbf{with}\ \{\mathbf{val}\ \mathsf{x}_v \to \mathsf{e}_v \mid \mathsf{o}_i\ \mathsf{x}_i\ \mathsf{k}_i \to \mathsf{e}_i\ \ldots\ \} \colon \mathsf{t}']\!]\ \rho$ is defined to be $\hbar'\ (\rho_h\ \rho)\ [\![\Gamma \vdash_e \mathsf{e} \colon \mathsf{t}]\!]\rho$ where, as in §2, $\hbar'$ is the handler and ρ_h is a handling environment. The former is defined generically whereas the latter is effect-specific and constructed from the **handle with** $\{\ldots\}$ expression. To simplify the notation, we shall call $\hbar'(\rho_h\ \rho)$ the handler and denote as $\hbar$. Which particular effects it handles (that is, which handling environment it uses) should be clear from the context.

Thus the handler $\hbar$ for $[\![\Gamma \vdash_e \mathbf{handle}\ (\mathsf{e} \colon \mathsf{t})\ \mathbf{with}\ \{\mathbf{val}\ \mathsf{x}_v \to \mathsf{e}_v \mid \mathsf{o}_i\ \mathsf{x}_i\ \mathsf{k}_i \to \mathsf{e}_i\ \ldots\ \} \colon \mathsf{t}']\!]\ \rho$ is the strict $\mathscr{R}[\mathsf{t}] \to \mathscr{R}[\mathsf{t}']$ map inductively defined as:

$\hbar\ \mathtt{V}(v) = [\![\Gamma, \mathsf{x}_v \colon \mathsf{t} \vdash_e \mathsf{e}_v \colon \mathsf{t}']\!](\rho \times \mathsf{x}_v \colon v)$
$\hbar\ \mathtt{E}(o, v, k) = [\![\Gamma, \mathsf{x}_i \colon \mathsf{t}_1, \mathsf{k}_i \colon \mathsf{t}_2 \Rightarrow \mathsf{t}' \vdash_e \mathsf{e}_i \colon \mathsf{t}']\!](\rho \times \mathsf{x}_i \colon v \times \mathsf{k}_i \colon \hbar \circ k)$
o is equal to o_i for some i
$\hbar\ \mathtt{E}(o, v, k) = \mathtt{E}(o, v, \hbar \circ k)$ o is not equal to any o_i

This definition generalizes the intuitive handlers in §2. The denotation makes it clear that our handlers, like those in Eff (Bauer & Pretnar, 2015), are 'deep' (that is, the k_i passed

to a handler clause includes the handler $\hbar$ itself to deal with the result of the suspended computation). We see an example in §5.

To lighten the notation, we may abbreviate $[\![\Gamma \vdash_v \mathsf{v}:\mathsf{t}]\!]$ and $[\![\Gamma \vdash_e \mathsf{e}:\mathsf{t}]\!]$ to just $[\![\mathsf{v}]\!]$ and $[\![\mathsf{e}]\!]$, resp., if the typing environment and the type can be easily guessed from the context.

By inspection, $[\![-]\!]$ is well-defined. In other words, for all (well-typed) values v and expressions e, $[\![\Gamma \vdash_v \mathsf{v}:\mathsf{t}]\!]\rho \in \mathscr{T}[\![\mathsf{t}]\!]$ and $[\![\Gamma \vdash_e \mathsf{e}:\mathsf{t}]\!]\rho \in \mathscr{R}[\![\mathsf{t}]\!]$ for any $\rho \in \mathscr{T}[\![\Gamma]\!]$. This is the statement of type soundness of the calculus.

3.4 Equational Laws

One of the motivations for denotational semantics is to justify program equivalences, to be able to say when two expressions are equivalent and inter-replaceable. Following Mosses (§3.1) (1990), we say that two expressions of the same type $\Gamma \vdash_e \mathsf{e}_1:\mathsf{t}$ and $\Gamma \vdash_e \mathsf{e}_2:\mathsf{t}$ for some Γ and t are contextually equivalent just in case that for any full context $\mathsf{D}[-]$ such that $\mathsf{D}[\mathsf{e}_1]$ and $\mathsf{D}[\mathsf{e}_2]$ are closed well-typed expressions of the type $()$, $[\![\mathsf{D}[\mathsf{e}_1]]\!]() = \star$ iff $[\![\mathsf{D}[\mathsf{e}_2]]\!]() = \star$. (Contextual equivalence of values is analogous.) Here, by full context $\mathsf{D}[-]$ we mean an expression with a single occurrence of the distinct free variable written as $[-]$. The notation $\mathsf{D}[\mathsf{e}]$ means a possibly capturing substitution of an expression e for the distinct variable in the full context $\mathsf{D}[-]$. Because of the quantification over all full contexts, contextual equivalence is difficult to prove. Denotational semantics helps by offering a sufficient condition for contextual equivalence: equality of denotations.

Definition 1 (*Term equivalence*)
The equivalences $\Gamma \vdash \mathsf{v}_1 \equiv \mathsf{v}_2$ and $\Gamma \vdash \mathsf{e}_1 \equiv \mathsf{e}_2$ (also written as $\mathsf{v}_1 \equiv \mathsf{v}_2$ and $\mathsf{e}_1 \equiv \mathsf{e}_2$, if Γ is easily guessed) on possibly open values or expressions of the same type are defined as follows:

$$\frac{[\![\Gamma \vdash_v \mathsf{v}_1:\mathsf{t}]\!] = [\![\Gamma \vdash_v \mathsf{v}_2:\mathsf{t}]\!]}{\Gamma \vdash \mathsf{v}_1 \equiv \mathsf{v}_2} \qquad \frac{[\![\Gamma \vdash_e \mathsf{e}_1:\mathsf{t}]\!] = [\![\Gamma \vdash_e \mathsf{e}_2:\mathsf{t}]\!]}{\Gamma \vdash \mathsf{e}_1 \equiv \mathsf{e}_2}$$

It is immediate from the definition that $\equiv$ is an equivalence relation. The compositionality of the denotational semantics ensures that $\equiv$ is a congruence (that is, closed under term construction). Two expressions related by $\mathsf{e1} \equiv \mathsf{e2}$ are hence equivalent in all contexts, including under lambda. §4 defines a more refined version of $\equiv$ modulo a handler.

Figure 7 shows a sample of term equivalences (or, as we often say, equational laws), easily proven by computing $[\![-]\!]$ of both sides and observing they are identical. These laws are valid for any effect and handler (like the 'free' laws in §2). In contrast, proving such laws using operational semantics is very difficult. In many so-called proofs of equational laws one sees, say, in Haskell community, congruence is not even mentioned let alone demonstrated. In the denotational approach, congruence is automatic: once we prove two expressions have the same denotations, the expressions are substitutable in any context, whether the context is an evaluation context or a general one; applying laws in the general binding context (function bodies) is common when optimizing programs.

3.5 Basic Reasoning with Non-determinism

We finish §3 by giving examples of using the just introduced formalism. Although the formalism is not yet complete (see §4 for the final part), it is already useful.

Map laws	$\mathsf{map\ f\ []} \equiv \mathsf{[]}$
	$\mathsf{map\ f\ (v::vs)} \equiv \mathsf{(f\ v)::(map\ f\ vs)}$
Substitution laws	$(\mathbf{fn}\,x.\ e) \circledast v \equiv e[x \leftarrow v]$
	$\mathbf{rec}\ f\ x.\ e \equiv \mathbf{fn}\,x.\ e[f \leftarrow \mathbf{rec}\ f\ x.e]$
Lifting laws	$f \odot v \equiv f\ v$
	$f \odot e \equiv \mathbf{let}\ x = e\ \mathbf{in}\ f \odot x$
	$e \circledast e' \equiv \mathbf{let}\ x = e\ \mathbf{in}\ x \circledast e'$
	$v \circledast e \equiv \mathbf{let}\ x = e\ \mathbf{in}\ v \circledast x$
Monad laws: left unit, right unit, associativity	$(\mathbf{fn}\,x.\ e) \circledast \mathbf{val}\ v \equiv e[x \leftarrow v]$
	$\mathbf{val}\ (\mathbf{fn}\,x.\ \mathbf{val}\ x) \circledast e \equiv e$ or, alternatively $(\nearrow x.x) \odot e \equiv e$
	$\mathbf{let}\ x = (\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)\ \mathbf{in}\ e_3 \equiv \mathbf{let}\ y = e_1\ \mathbf{in}\ \mathbf{let}\ x = e_2\ \mathbf{in}\ e_3$ where y does not occur free in e_3
	$\mathbf{let}\ x = e_1\ \mathbf{in}\ f \odot e_2 \equiv f \odot (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2)$
	$e_1; f \odot e_2 \equiv f \odot (e_1; e_2)$

Fig. 7. Basic equational laws. Here, v is an arbitrary value, e is an arbitrary term, vs is an arbitrary list, and $[x \leftarrow e]$ denotes substitution.

We introduce[6] the first effect: the effect constant Coin with the type $\vdash_o$ Coin: unit$\hookrightarrow$bool. Intuitively, the corresponding effect is tossing a coin. The following abbreviation saves a bit of tedium:

$$\mathsf{coin} \coloneqq \mathbf{op}\ \mathsf{Coin} \circledast ()$$

Actually, the meaning of an effect operation like Coin is entirely determined by a handler – just as the meaning of the name n in the definition **let** n=e **in** ... is entirely determined by e; n is just a name. Handlers for non-determinism are discussed in §6. Yet we can write code with the Coin effect right away, without waiting for handlers. Moreover, we can even prove some properties of the Coin code, which thus hold for every interpretation of the Coin effect, similar to the 'free' laws in §2. Such laws, valid for any interpretation of an effect, are quite rare. One may also think they are 'content-free' because of their generality. Here we demonstrate the free laws that are interesting and useful, for reasoning and program optimization.

The first free law is the one that is indispensable in reasoning about non-determinism, letting us 'pull out' non-determinism out of a computation:

Lemma 3 (*Left distributivity*)

$$C_C[\mathbf{if}\ \mathsf{coin}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2] \equiv \mathbf{if}\ \mathsf{coin}\ \mathbf{then}\ C_C[e_1]\ \mathbf{else}\ C_C[e_2]$$

where C_C is the evaluation context not containing a handler for the Coin effect.

As an example of writing programs in our minimalistic language and using the Coin effect, consider the following function to non-deterministically insert a value x at some position in the given list xs (which can be used, for example, to compute list permutations):

[6] In real programming language systems and effect libraries, effects are introduced by special declarations.

```
insert : t → t list ⇒ t list ≔
 ⧹x. rec loop xs.
   match xs with
   | []    → x::[]
   | y::ys → if coin then x::y::ys else (y::) ⊙ (loop ⊛ ys)
```

The code non-deterministically inserts x either at the head of xs, or somewhere inside xs (if it is non-empty).

The first question to ask about a list-processing function is how it behaves with respect to map:

Lemma 4 (*'Free' theorems of insert*)

1. insert (f x) ⊛ map f xs ≡ map f ⊙ (insert x ⊛ xs)
2. insert (f x) ⊛ (map f ⊙ e) ≡ map f ⊙ (insert x ⊛ e)

for all x:t, xs:t list, f:t→t', and an arbitrary expression e:t list of arbitrary effects.

Free theorems come to mind. However, insert is an effectful function, and the free theorem no longer comes for free. Still, it holds for the 'free' non-determinism, regardless of the specific meaning of the Coin effect. No matter how Coin effect may be handled, we are *always* justified, in all possible contexts including under lambda, to replace the left-hand-side term of the lemma with the right-hand-side, or vice-versa. Such a replacement may be profitable, if the moved map f is later fused.

The most interesting is part 1. of the lemma (the other is a consequence). To prove it, we need the denotation for insert, to be called *insert*, which we easily compute compositionally as below (it is clearly parametric in the type t of list elements).

$$\begin{array}{rl} [\![\vdash_v \mathsf{insert{:}t{\Rightarrow}t\ list{\Rightarrow}t\ list}]\!]() & = \mathit{insert} \qquad \text{where} \\ \mathit{insert}\ x & \coloneqq \mathsf{fix}(\mathit{insF}\ x) \\ \mathit{insF}\ x\ u & \coloneqq \lambda l.\begin{cases} \mathtt{V}(x::[]) & \text{if } l = [] \\ \mathtt{E}(\mathsf{Coin},\star,g\,u\,x\,y\,l') & \text{if } l = y :: l' \end{cases} \\ g\,u\,x\,y\,l' & \coloneqq \lambda b.\mathit{if}(b,\mathtt{V}(x :: y :: l'),(y ::)^{\ddagger}(u\,l')) \end{array}$$

We thus aim to show that

$$\mathit{insert}\ (f\,x)\ (\mathit{map}f\ l) = (\mathit{map}f)^{\ddagger}(\mathit{insert}\ x\ l)$$

where x is the denotation of x, f is of f, l of ys. The proof proceeds very similarly to the proof of Lemma 2. It uses induction, but *not* on the length of the list; rather, it is an induction on the approximants to the fixpoint. The Appendix shows the complete proof.

The Appendix also elaborates a extended example based on the second author's study (Mu, 2019b) on modeling and reasoning about Spark, a popular platform for distributed data-parallel computation (Zaharia *et al.*, 2010).

4 Equivalence modulo handler

We now complete the formalism, introducing the equivalence modulo handler, which in turn relies on the denotation of evaluation contexts.

One of the important intuitions about effects is that they 'bubble up' through evaluation contexts that contain no handler for them. Indeed, Fig. 6 shows that $[\![\mathsf{C[e]}\ ]\!]\rho$ can be written

in the form of an application of a strict map to $[\![\mathsf{e}]\!]\rho$; that strict map depends only on the structure of the evaluation context C and ρ and does not depend on e. Such a factorization is the manifestation of compositionality of the semantics. We write this strict map as $[\![\mathsf{C}]\!]\rho$ and call $[\![\mathsf{C}]\!]$ the *context denotation*. Fig. 6 lets us make further observations:

Lemma 5 (*Evaluation context congruence*)

1. $[\![\mathsf{C[C']}]\!] = \lambda\rho.[\![\mathsf{C}]\!]\rho\ ([\![\mathsf{C'}]\!]\rho)$, where $\mathsf{[C[C']\ ]}$ is the composition of contexts
2. If C has no handlers at all, $[\![\mathsf{C}]\!]$ has the form $\lambda\rho.\,f^{\dagger}$ for some f
3. If C has no handler for the effect o, then $[\![\mathsf{C}]\!]\rho\ \mathtt{E}(o,v,k) = \mathtt{E}(o,v,[\![\mathsf{C}]\!]\rho \circ k)$

Part 3. of the lemma formally expresses the 'bubbling up' of the effects. In other words, our effects are algebraic, in the sense of Plotkin & Power (2003). Left distributivity for non-determinism (Lemma 3) is a particular case of algebraicity. Part 3. of Lemma 5 lets us view the denotation $\mathtt{E}(o,v,k)$ of an effectful expression intuitively as the denotation of an effectful operation $\mathbf{op}\ \mathsf{o} \circledast \mathsf{v}$ in the evaluation context whose denotation is k.[7]

We now move to contextual equivalence modulo handler. Let o_0 be the effect dealt with by handler in question[8] and let H be a *closed* evaluation context that contains that handler; that is, H handles o_0.[9] We shall write C_h for an evaluation context that has no handlers for o_0.

Let e_1 and e_2 be two possibly open expressions of the same type, both performing the effect o_0. If their denotations differ, we may not, in general, replace e_1 with e_2 without affecting the meaning of the whole program. Suppose, however, we prove that for all contexts C_h, $[\![\Gamma \vdash_e \mathsf{H[C}_h\mathsf{[e}_1\mathsf{]\ ]}]\!] = [\![\Gamma \vdash_e \mathsf{H[C}_h\mathsf{[e}_2\mathsf{]\ ]}]\!]$. That will let us substitute e_1 with e_2 in an expression $\mathsf{H[C}_h\mathsf{[e}_1\mathsf{]\ ]}$ for an arbitrary C_h, preserving the program's meaning. One may say that e_1 and e_2 are equivalent *provided* they occur in an evaluation context that contains H but no other handler for o_0. That notion of equivalence (used, for example, in Bauer & Pretnar (2014)) is rather weak, however. It does not us replace e_1 with e_2 in general contexts, for example, $\mathbf{let}\ \mathsf{f} = \mathbf{fn}\,\mathsf{x}.\ \mathsf{C}_h\mathsf{[e}_1\mathsf{]}\ \mathbf{in}\ \mathsf{H[C}_h\mathsf{'[f} \circledast \mathsf{e']\ ]}$, which, one may feel, should be justifiable if there are no handlers for o_0 other than H. We now justify this intuition, closely following §2.2.

Let H be a handler for o_0 (whose denotation $[\![\mathsf{H}]\!]()$ we write as $\hbar$) subject to conditions specified below. Let $\approx$ be a relation defined on all semantic domains and maps between domains, as follows:

$$\frac{\hbar\mathtt{E}(\mathsf{o}_0,v_1,k_1) \approx \hbar\mathtt{E}(\mathsf{o}_0,v_2,k_2)}{\mathtt{E}(\mathsf{o}_0,v_1,k_1) \approx \mathtt{E}(\mathsf{o}_0,v_2,k_2)} \qquad \frac{v_1 \approx v_2 \quad k_1 \approx k_2}{\mathtt{E}(\mathsf{o},v_1,k_1) \approx \mathtt{E}(\mathsf{o},v_2,k_2)}\ \mathsf{o} \neq \mathsf{o}_0 \qquad \frac{v_1 \approx v_2}{\mathtt{V}(v_1) \approx \mathtt{V}(v_2)} \qquad \frac{}{\bot \approx \bot}$$

$$\frac{v \in \mathscr{T}[\![\mathsf{t}]\!]}{v \approx v}\ \mathsf{t} \text{ is primitive type} \qquad \frac{\forall x_1 x_2.\, x_1 \approx x_2 \text{ implies } f_1 x_1 \approx f_2 x_2}{f_1 \approx f_2}\ f_1, f_2 \text{ domain maps}$$

[7] Not every expression whose denotation is $\mathtt{E}(o,v,k)$ is of the form $\mathsf{C[\mathbf{op}\ o} \circledast \mathsf{v]}$; for example: $\mathbf{if}\ \mathsf{true}\ \mathbf{then}\ \mathbf{op}\ \mathsf{o} \circledast \mathsf{v}\ \mathbf{else}\ \mathsf{e'}$. But every expression with such denotation is $\equiv$ to an expression of that form.

[8] We use the single effect for clarity. Also, without loss of generality several effects can always we combined into one, at the expense of modularity and code bloat.

[9] Requiring H to be closed is not limiting. In fact, the state handler immediately below shows an example of parametrization.

The condition on $\hbar$, which we demand from now on, is: For all context denotations C_1, C_2 such that $\hbar \circ C_1$ and $\hbar \circ C_2$ are defined and $C_1 \approx C_2$

$$\hbar \circ C_1 \approx \hbar \circ C_2 \tag{7}$$

Clearly, $\approx$ is symmetric and transitive (and also admissible: it relates $\bot$ to itself and closed under least-upper-bounds of chains). It is generally not reflexive though: if a map f: $\mathscr{R}[\mathsf{t}] \to \mathscr{R}[\mathsf{t}']$ could examine the elements of the form $\mathtt{E}(\mathsf{o}, v, k)$, it could distinguish the related $\mathtt{E}(\mathsf{o}_0, v_1, k_1)$ and $\mathtt{E}(\mathsf{o}_0, v_2, k_2)$ when $v_1 \neq v_2$. Our goal is to identify the subset of domains on which $\approx$ is an equivalence relation.

First we note several simple facts about $\approx$:

$\approx$ is preserved by applications and compositions (8)

On domains $\mathscr{T}[\mathsf{t}]$ where t does not contain $\Rightarrow$, the relation $\approx$ is the identity (9)

If $f_1 \approx f_2$ then $f_1^\dagger \approx f_2^\dagger$ (10)

$\hbar \approx \hbar$ (11)

Here, (11) follows from (7). That $\approx$ is preserved by lifting, (10), is proved like the similar property in §2.2, relying on the induction principle.

Theorem 1
For all expressions e and values v such that the only handler for o_0 (if present) is H, we have $[\![\mathsf{e}]\!] \approx [\![\mathsf{e}]\!]$ and $[\![\mathsf{v}]\!] \approx [\![\mathsf{v}]\!]$.

The proof is by structural induction. Here are two typical cases. Let e be an application $\mathsf{e}_1 \circledast \mathsf{e}_2$. Then $[\![\mathsf{e}_1 \circledast \mathsf{e}_2]\!]$ is $\lambda\rho.(\lambda f.\, f^\dagger\ [\![\mathsf{e}_2]\!]\,\rho)^\dagger\,[\![\mathsf{e}_1]\!]\,\rho$, and the conclusion follows from (10) and the induction hypothesis. Let v be $\mathbf{fn}\,\mathsf{x}.\ \mathsf{e}$, whose denotation is $\lambda\rho.\lambda v.\,[\![\mathsf{e}]\!]\,(\rho \times \mathsf{x} : v)$. The conclusion follows from the definition of $\approx$ and the inductive hypothesis.

The very similar proof shows that if e_1 and e_2 are such that $[\![\mathsf{e}_1]\!] \approx [\![\mathsf{e}_2]\!]$ then we have $[\![\mathsf{D}[\mathsf{e}_1]]\!] \approx [\![\mathsf{D}[\mathsf{e}_2]]\!]$, for any general context $\mathsf{D}[-]$ whose handlers for o_0, if any, is H.

Finally, consider two possibly open expressions e_1 and e_2 of the same type such that $[\![\mathsf{e}_1]\!] \approx [\![\mathsf{e}_2]\!]$. Let $\mathsf{D}[-]$ be a context such that $\mathsf{D}[\mathsf{e}_1]$ and $\mathsf{D}[\mathsf{e}_2]$ are both closed programs of a primitive type. Further assume that the only handler for o_0 in $\mathsf{D}[-]$ is H. We have just shown that $[\![\mathsf{D}[\mathsf{e}_1]]\!] \approx [\![\mathsf{D}[\mathsf{e}_2]]\!]$. Since $\mathsf{D}[\mathsf{e}_1]$ and $\mathsf{D}[\mathsf{e}_2]$ are both closed programs of a primitive type, it then follows that if $[\![\mathsf{D}[\mathsf{e}_1]]\!] = \mathtt{V}(v)$ for some v then $[\![\mathsf{D}[\mathsf{e}_2]]\!] = \mathtt{V}(v)$, and vice versa. In other words, for programs in which the only handler for o_0 is H, satisfying (7), the relation $\approx$ is the contextual equivalence. Therefore, for such related e_1 and e_2 we adopt the notation $\mathsf{e}_1 \equiv_h \mathsf{e}_2$ and call such expressions equivalent *modulo handler H*.

5 State

This section, expanding on §2.1, formally introduces the State effect and its handler and demonstrates reasoning about the State-ful code *relative* to the handler.

We introduce two effect constants: Get and Put with the types $\vdash_o \mathsf{Get}\colon \mathsf{unit} \hookrightarrow \mathsf{t}_s$ and $\vdash_o \mathsf{Put}\colon \mathsf{t}_s \hookrightarrow \mathsf{unit}$ for some type t_s, and define the short names for the primitive State operations:

$\mathsf{get} \coloneqq \mathbf{op}\ \mathsf{Get} \circledast ()$ $\qquad\qquad$ $\mathsf{put}\ \mathsf{v} \coloneqq \mathbf{op}\ \mathsf{Put} \circledast \mathsf{v}$

The closed handler

```
handleS s0 e :=
 (handle e with {val x → fn s. x
  | Get x k → fn s. k ⊛ s ⊛ s
  | Put x k → fn s. k ⊛ () ⊛ x
 }) ⊛ s0
```

will be referred to as handleS s_0 e where e:t is the computation to handle and the value s_0 is the initial state (of the type t_s). We define C_s as an evaluation with no handlers for Get and Put.

The following is a sample State computation

```
handleS 0 (put 10; let x = get in put 20; let y = get in (x+y))
```

whose denotation is $\mathtt{V}(30)$.

The handler $\hbar$ corresponding to handleS is the $\mathscr{R}[t] \to \mathscr{R}[t_s \Rightarrow t]$ map, defined as the least fixed point of the following recursive definition by cases. It is a particular case of the general definition in §3.3:

$$\begin{aligned}
\hbar\ \mathtt{V}(v) &= \mathtt{V}(\lambda s.\, \mathtt{V}(v)) \\
\hbar\ \mathtt{E}(\mathsf{Get}, v', k') &= \mathtt{V}(\lambda s.\, (\hbar(k'\ s))^{¶} s) \\
\hbar\ \mathtt{E}(\mathsf{Put}, v', k') &= \mathtt{V}(\lambda s.\, (\hbar(k'\ \star))^{¶} v') \\
\hbar\ \mathtt{E}(o, v, k) &= \mathtt{E}(o, v, \hbar \circ k) \qquad o \text{ is neither Get nor Put}
\end{aligned}$$

It is easy to see that the condition (7) is satisfied, from the induction principle.

Assuming the State handler $\hbar$ we derive the following equivalences modulo that handler:

Lemma 6 (*Get-Put Laws*)

(PutPut) put v'; C_s[put v] $\equiv_\hbar$ C_s[put v]
(PutGet) put v; C_s[get] $\equiv_\hbar$ put v; C_s[**val** v]
(GetPut) **let** x = get **in** C_s[put x] $\equiv_\hbar$ **let** x = get **in** C_s[**val** ()]
(GetGet) **let** x = get **in** C_s[get] $\equiv_\hbar$ **let** x = get **in** C_s[**val** x]

(The proof of the (GetPut) law is in the Appendix. The others are analogous, and simpler.) Actually, the familiar get-put laws (Pretnar, 2010; Gibbons & Hinze, 2011) are the particular case of our laws, when C_s is the empty context (that is, when the get/put operations are performed right one after the other). Our laws are a bit more general, saving us trouble of bringing the effectful operations next to each other. The Appendix shows an extended derivation where this generality comes useful. The derivation (part of a larger project of deriving an efficient *n*-queens solver) aims to show that an accumulating list mapping can be done as a right fold, using mutable state as an accumulator. It includes the following step, justified by the general (PutPut) law without further ado.

```
put s0; (f s0 x::) ⊙ (put (f s0 x); e)   ≡h   (f s0 x::) ⊙ (put (f s0 x); e)
```

where e is some expression and s_0, f and x are some variables.

We should stress our State laws are congruences. They apply even if C_s or s contain free variables, as the just shown example has demonstrated. We can use the laws to simplify even **fn** x. put x; put x; e to just **fn** x. put x; e, assuming that it is handleS that will handle the Put effects when the expression will eventually be executed.

6 Non-determinism: different handlers, different laws, different use cases

The non-determinism effect Coin was already introduced in §3.5, where we proved 'free' equivalences of non-deterministic computations, in particular, left distributivity. This section considers non-free equivalences, specific to particular Coin effect handlers. It will become clear that the equivalences (equational laws) do indeed vary with the handler. However, some of the equational laws we prove here turn out semi-free, so to speak: although they hold only for a particular non-determinism handler, they still allow arbitrary other effects. Completely 'non-free' equivalences, that is, reasoning with interacting multiple effect handlers is the topic of §7.

6.1 Laws of Non-determinism

Let us first recall the laws commonly presupposed for non-determinism, so called 'semi-lattice' laws. The already mentioned Coin effect that non-deterministically yields either true or false is one way to express a binary non-deterministic choice. Alternatively, a binary choice can be thought of as a program composition operator, denoted as $\oplus$ below, that combines two expressions e_1 and e_2 into the choice expression, with e_1 and e_2 as the alternatives. The two ways are equivalent and inter-expressible (see Plotkin & Power (2003) for details). However, $\oplus$ lets us state the algebraic laws of non-determinism in a particularly elegant way; that is why it is used particularly in the algebraic effect tradition (see Pretnar (2010) for the collection of references).

$$\mathsf{e} \oplus \mathsf{e} \equiv \mathsf{e} \tag{Idem}$$

$$\mathsf{e}_1 \oplus \mathsf{e}_2 \equiv \mathsf{e}_2 \oplus \mathsf{e}_1 \tag{Comm}$$

$$\mathsf{e}_1 \oplus (\mathsf{e}_2 \oplus \mathsf{e}_3) \equiv (\mathsf{e}_1 \oplus \mathsf{e}_2) \oplus \mathsf{e}_3 \tag{Assoc}$$

Non-determinism is hence treated as a theory of semi-lattices.

Non-determinism as a theory of semi-lattices is indeed helpful when non-determinism is used as a *specification* and for reasoning, not necessarily constructive, about the algorithm. The eventual implementation is supposed to be deterministic.[10] However, non-determinism is useful not only for writing specifications of programs but also for writing programs themselves. For example, Thompson's patent and the implementation of the regular expression matching (Thompson, 1968) described the running of NFA directly, on a specially designed non-deterministic (virtual) machine. Therefore, it is rather common to see non-deterministic choice offered as a programming language facility, as an effect.

When $\oplus$, or Coin, is used for writing programs (rather than program specifications), the semi-lattice theory is no longer adequate or becomes too expensive to support, which we are to show below. (The semi-lattice laws take another hit when other effects are present, in particularly, state. This is the topic of §7.) In our calculus it is more convenient to use the Coin effect. The semi-lattice laws take the following form:

[10] This is exactly how non-determinism was used in Rabin and Scott's pioneering paper (1959). See Armoni & Ben-Ari (2009) for detailed discussion and historic overview.

$$\textbf{if}\ \mathsf{coin}\ \textbf{then}\ \mathsf{e}\ \textbf{else}\ \mathsf{e} \equiv_h \mathsf{e} \qquad \text{(idem)}$$
$$\textbf{if}\ \mathsf{coin}\ \textbf{then}\ \mathsf{e}_1\ \textbf{else}\ \mathsf{e}_2 \equiv_h \textbf{if}\ \mathsf{coin}\ \textbf{then}\ \mathsf{e}_2\ \textbf{else}\ \mathsf{e}_1 \qquad \text{(comm)}$$
$$\textbf{if}\ \mathsf{coin}\ \textbf{then}\ (\textbf{if}\ \mathsf{coin}\ \textbf{then}\ \mathsf{e}_1\ \textbf{else}\ \mathsf{e}_2)\ \textbf{else}\ \mathsf{e}_3 \equiv_h \textbf{if}\ \mathsf{coin}\ \textbf{then}\ \mathsf{e}_1\ \textbf{else}\ (\textbf{if}\ \mathsf{coin}\ \textbf{then}\ \mathsf{e}_2\ \textbf{else}\ \mathsf{e}_3) \qquad \text{(assoc)}$$

Which particular handler $\hbar$ should be had in mind when speaking about each of these laws is described next.

6.2 Laws and Handlers

This section describes three concrete handlers that validate some or all of the semi-lattice laws. We will also clearly see the price to pay for supporting all the laws.

We start with one of the simplest, and perhaps the most familiar handleList: the 'list' monad. Unlike Haskell lists, which are streams in disguise, our lists are finite. The handler for the Coin-effect expression returns all of its possible choices in order, in a list.

$$\begin{array}{l}\mathsf{handleList\ e} \coloneqq \\ \ \textbf{handle}\ \mathsf{e}\ \textbf{with}\ \{\textbf{val}\ \mathsf{x} \to \mathsf{x}::[]\ \mid\ \mathsf{Coin\ z\ k} \to \\ \quad \textbf{let}\ \mathsf{x} = \mathsf{k} \circledast \textbf{true}\ \textbf{in let}\ \mathsf{y} = \mathsf{k} \circledast \textbf{false}\ \textbf{in}\ \mathsf{concat\ x\ y}\}\end{array}$$

where $\mathsf{concat} : \mathsf{t\ list} \to \mathsf{t\ list} \to \mathsf{t\ list}$ is the list concatenation function (list append). As an example, handleList (**if** coin **then** 1 **else** 2) has $\mathtt{V}([1;2])$ as its denotation.

The handler denotation $\hbar$ for handleList is the strict inductively defined $\mathscr{R}[\mathsf{t}] \to \mathscr{R}[\mathsf{t\ list}]$ map, a particular case of the general definition in §3.3:

$$\begin{array}{ll}\hbar\ \mathtt{V}(v) & = \mathtt{V}([v]) \\ \hbar\ \mathtt{E}(\mathsf{Coin}, v', k') & = \mathit{concat}''\ (\hbar(k'\ \mathit{true}))\ (\hbar(k'\ \mathit{false})) \\ \hbar\ \mathtt{E}(o, v, k) & = \mathtt{E}(o, v, \hbar \circ k) \qquad o \text{ is not Coin} \\ \mathit{concat}'' & = \lambda x_1 x_2.\ ((\lambda v_1 v_2.\ \mathtt{V}(\mathit{concat}\ v_1\ v_2))^{\ddagger} x_1)^{\dagger} x_2\end{array}$$

We now demonstrate that with respect to such an $\hbar$, (assoc) equivalence holds. The key step is as follows, obtained by straightforward calculation. Let e_1, e_2 and e_3 be arbitrary (possibly open) expressions of the same type t', ρ the environment, C_C the evaluation context not containing a Coin handler, and $\Gamma \vdash_e \mathsf{C}_C[\mathsf{e}_i]$: t holds. We write $[\![\mathsf{e}]\!]'$ to stand for $\hbar\ [\![\mathsf{C}_C[\mathsf{e}]]\!]\rho$.

$$[\![\textbf{if}\ \mathsf{coin}\ \textbf{then}\ (\textbf{if}\ \mathsf{coin}\ \textbf{then}\ \mathsf{e}_1\ \textbf{else}\ \mathsf{e}_2)\ \textbf{else}\ \mathsf{e}_3]\!]' = \mathit{concat}''(\mathit{concat}''[\![\mathsf{e}_1]\!]'\ [\![\mathsf{e}_2]\!]')[\![\mathsf{e}_3]\!]' \qquad (12)$$
$$[\![\textbf{if}\ \mathsf{coin}\ \textbf{then}\ \mathsf{e}_1\ \textbf{else}\ (\textbf{if}\ \mathsf{coin}\ \textbf{then}\ \mathsf{e}_2\ \textbf{else}\ \mathsf{e}_3]\!]' = \mathit{concat}''[\![\mathsf{e}_1]\!]'(\mathit{concat}''[\![\mathsf{e}_2]\!]'\ [\![\mathsf{e}_3]\!]') \qquad (13)$$

The two denotations are the same because list concatenation is associative (and lifting composition laws preserve associativity).

We stress that expressions e_i may diverge or have arbitrary effects, which may either be handled by the context C_C or propagate past handleList. The above equalities hold nonetheless. We have thus established the associativity of non-determinism modulo handleList – *in the presence of arbitrary other effects*.

Computing the denotations for handleList (**if** coin **then** 1 **else** 1) and handleList (**if** coin **then** 1 **else** 2) easily shows that neither (idem) nor (comm) holds for handleList.

The next handler for non-determinism is even simpler: it returns the first choice of the result out of many that a non-deterministic expression may have:

handleFirst e ≔ **handle** e **with** {**val** x → x ∣ Coin z k → k ⊛ **true**}

Although this handler seems silly, it is the right handler to use for 'don't care non-determinism'; it corresponds to the operation once, offered in many (Functional) Logic Programming languages. The straightforward calculations similar to those for handleList show that (assoc) *and* (idem) – but not (comm) – equivalences now hold, modulo handleFirst. They hold in the presence of any other possible effects besides Coin, including non-termination.

The final handler, like handleList, collects all choices for the result of a non-deterministic expression in a list. In addition, it also sorts the list, in the order specified by a total order predicate le: t→t→bool, and removes duplicates:

handleSet le e ≔ sortUniq le ⊙ handleList e

Now all three (assoc), (idem) and (comm) equivalences hold. However, whereas (assoc) holds in the presence of any other effect, (idem) and (comm) do not. We see this point more clearly below, Lemma 7.

There is a price to pay for maintaining all three (assoc), (idem) and (comm) equivalences. First, there is the expense of sorting, which requires the existence of a total order on the results of a handled computation. Second, handleList can be easily extended to return only a fixed number of choices for the expression result – or return the choices incrementally. It will then cope with computations with an unbounded number of non-deterministic choices; the trivial, yet practically useful, example is

iota ≔ **rec** self x. **if** coin **then** x **else** self ⊛ (x+1)

Such computations are beyond handleSet or any other handler that maintains (comm). There is a compromise between expressivity and the richness of the equivalences for a handler. Lastly, (comm) and (idem) for handleSet are generally broken if the handled expression also does other effects.

6.3 General Depth-first Non-determinism Handler

The three handlers handleList, handleFirst and handleSet are instances of a general handler for Coin effects, to be described below.[11] The general handler handleGen for non-deterministic computations of type t is parameterized by three operations: inj: $t \to t_i$, extr: $t_i \Rightarrow t'$ and comb: $t_i \to (\mathsf{unit} \Rightarrow t_i) \Rightarrow t_i$. Here, t' is the type of the handler result (which may be different from t: see handleList) and t_i is the type of the intermediate result accumulator. The inj operation[12] puts the result of the handled expression into the accumulator; extr extracts the final result from the accumulator, and comb combines the intermediate results for the two choices of Coin. For example, handleSet is the instance of handleGen in which t' and t_i are t list, inj makes a singleton list, comb concatenates the lists of choices so far, and extr sorts.

[11] It generally expresses depth-first–like non-deterministic search strategies, to which we limit our attention.

[12] We gave it a pure type for simplicity; in general, it could be allowed to have side-effects, e.g., state.

```
handleGen inj extr comb e :=
  extr ⊛ handle e with {
          val x → inj x
        | Coin z k → comb ⊙ (k ⊛ true) ⊛ (fn (). k ⊛ false)}
```

The general handler can also be instantiated to return the result alternatives incrementally, or return up to a certain maximum number of alternatives. Therefore, it could also cope with expressions like iota 0 that involve an unbounded number of non-deterministic choices.

We can relate the term equivalences with the properties of *inj*, *comb* and *extr*, which are the meanings of the corresponding parameters of handleGen. The associativity (assoc) equivalence holds if *comb* is associative (modulo *extr*: for instance, *comb* may maintain an intermediate tree data structure, which is fast to adjoin. Provided that *extr* flattens the tree at the end, (assoc) is preserved). Likewise, (comm) and (idem) are related to the properties of *comb* and *extr*.

Lemma 7 (*Generic handler–specific equivalences of non-determinism*)
The equivalences (assoc), (idem) and (comm) holds provided the following equalities are satisfied, resp.

$$\text{(assoc)}\quad extr^\dagger\ \left(comb^\dagger\ (comb^\dagger\ z_1\ \lambda().z_2)\ \lambda().z_3\right) = extr^\dagger\ \left(comb^\dagger\ z_1\ (\lambda().comb^\dagger\ z_2\ \lambda().z_3)\right)$$

$$\text{(idem)}\quad extr^\dagger\ (comb^\dagger\ z\ \lambda().z) = extr^\dagger\ z$$

$$\text{(comm)}\quad extr^\dagger\ (comb^\dagger\ z_1\ \lambda().z_2) = extr^\dagger\ (comb^\dagger\ z_2\ \lambda().z_1)$$

where $\{z, z_1, z_2, z_3\} \subset \mathscr{R}[t_i]$ (but with no Coin effect).

If there are no effects other than Coin – that is, z_i are either $\bot$ or $\mathtt{V}(\ldots)$ – whether (comm) holds or not is up to the properties of *extr* and *comb*. However, when other effects are involved, i.e., z may be of the form $\mathtt{E}(o,v,l)$, we need to know how exactly the operation o is handled to decide if (comm) still holds.

There is still something we can say about handleGen that holds regardless of how other effects are handled. The following lemma states a commutativity-like property, for any instance of handleGen in the presence of any other effect.

Lemma 8 (*Commuting Coin with other effects*)
Let $\hbar$ be handleGen inj extr comb for any choice of its parameters, and let o be an effect operation not handled by it and v any suitable argument to it, then

```
let x = op o ⊛ v in if coin then e1 else e2  ≡h
    if coin then (let x = op o ⊛ v in e1) else e2
```

where e_1 and e_2 are arbitrary expressions; x may occur free in e_1 but not in e_2.

The proof is the straightforward computation of denotations of the two sides (pre-composed with $\hbar \circ g_C$) and comparing them.

6.4 Non-determinism with Failure

Along with the non-deterministic choice, Rabin and Scott – who introduced non-deterministic finite automata and non-determinism in general into computer science (Rabin & Scott,

1959) – also acknowledged the possibility of a failure. We have avoided failure so far. Now, we bring it back.

We introduce the effect operation Fail of the type $\vdash_o$ Fail: unit↪empty (where empty is the empty type) and the convenient abbreviation for using it:

```
fail : t ≔ match op Fail ⊛ () with
```

Since the empty type has no constructors, matching on it needs no clauses and may be assigned an arbitrary type. One may think of Fail as an exception, and of fail as raising that exception.

Failure is indeed very useful in non-determinism. Suppose we have a function p: int → bool that expresses some property on natural numbers. To find the number or numbers that make p true, we literally write

```
comprehension p ≔ let x = iota 0 in if p x then x else fail
```

where, recall, iota 0 non-deterministically gives a natural number. This one-liner is not merely the specification of the comprehension principle. We can really compute with it, given the appropriate handler that embodies a particular search strategy. We see an interesting example shortly.

To handle the Fail operation, the earlier handleFirst is changed to

```
handleOne e ≔ handle e with {val x → x::[]
  | Fail z k → []
  | Coin z k → match k ⊛ true with
                | []   → k ⊛ false
                | h::l → h::l}
```

which returns the singleton list with the result of one (of possibly many) non-deterministic choices – the choice that succeeded. The handler returns the empty list if all choices failed. This new handler is what Haskell calls 'the Maybe monad'. Adding the clause Fail z k → [] and removing concatenation are the only changes to the earlier handleList.

As an example, of constructively using the comprehension principle, consider finding (the first) perfect number greater than 28. If not practical, it is at least an interesting example: the reader likely cannot tell the answer offhand.

```
handleOne
  (let i = iota ⊛ 0 in
    if leq i 28 then fail else
     let l = factors ⊛ i in
      if ieq i (sum (1::l)) then i else fail)
where
factors : int ⇒ int list ≔
 fn n. handleList (let i = upto n ⊛ 2 in if divisible n i then i else fail)
upto : int → int ⇒ int ≔
 ⩛z. rec self n.
   if leq z n then fail else if coin then n else self ⊛ (add 1 n)
sum : int list → int ≔ foldl add 0
```

Law	handleList			handleFirst			handleSet			handleOne		
	None	Any	Fail	None	Any	Fail	None	Any	Fail	None	Any	Fail
(idem)	−	−	−	+	+	+	+	−	+	+	−	+
(comm)	−	−	−	−	−	−	+	−	+	−	−	−
(assoc)	+	+	+	+	+	+	+	+	+	+	+	+

Table 1. *Non-determinism handlers and their laws. 'None' means no other effects beside Coin; 'Fail' means the Fail effect may also be present; 'Any' means arbitrary other effect.*

where divisible: int→int→bool, ordering leq: int→int→bool and equality ieq:int→int→bool are primitives with the obvious meaning. The example highlights the advantages of first-class handlers, which may appear within the program rather than being permanently stationed 'outside' (as in Cartwright & Felleisen (1994) and the original algebraic effect approach of Plotkin & Power (2003)): the body of the non-deterministic expression deciding if a number is perfect embeds another non-deterministic expression, finding out if the number is divisible, and collecting, using handleList, all of the divisors (except 1 and the number itself).

The addition of the Fail effect does not invalidate the earlier equational properties of handleList, and, consequently, handleSet: they still verify (assoc), whether there are other effects or not. Lemma 8 continues to hold, after a trivial extension to handleGen. If there are no other effects, handleSet still supports (comm) and (idem) equivalences. The extended handleOne, like its original handleFirst, validates (assoc) and (idem). However, whereas handleFirst supported idempotence 'for free' (in the presence of arbitrary other effects), this is no longer the case for handleOne.

The Fail effect also adds new equivalences, specific to this effect. The proof is an easy exercise.

Lemma 9 (*Laws of failure*)
Let $\hbar$ be handleOne or handleList (extended for the Fail effect) or handleSet. Then

$$\mathsf{C}_C[\mathsf{fail}] \equiv_h \mathsf{fail}$$
$$\textbf{if}\ \mathsf{coin}\ \textbf{then}\ \mathsf{fail}\ \textbf{else}\ \mathsf{e} \equiv_h \mathsf{e}$$
$$\textbf{if}\ \mathsf{coin}\ \textbf{then}\ \mathsf{e}\ \textbf{else}\ \mathsf{fail} \equiv_h \mathsf{e}$$

where e is any expression and C_C is a context with no non-determinism handlers.

6.5 Laws and Handlers: Summary

Table 1 overviews which handler validates which law, in the presence of no other effect, or any other effect, or just Fail as the other effect, beside Coin. It should be stressed again there is a trade-off between expressivity and performance, on one hand, and the richness of the equivalences for a handler on the other hand. The handler handleSet validates all semi-lattice laws. It also cannot deliver answers incrementally, cannot cope with non-deterministic computations with unbounded number of answers, takes extra time for sorting the answers, and requires the existence of the total order on answers (which may be expensive to compute).

Yet another challenge comes from searching over large (and often, infinite) search space – a very common application of non-determinism. A search procedure that always finds an

answer if it exists, without getting trapped, either has to validate both (comm) and (assoc), or deny both.[13] If the search is to deliver results incrementally, neither (comm) nor (assoc) may hold.

7 Non-determinism and State

One of the motivations of this work is to study the interaction between effects. We have reasoned with multiple effects already: in §6 we proved handler-specific equational laws of non-determinism that are valid in the presence of any other effect. This section describes the true interaction of effects – or, to be precise, the interaction of their handlers, since it is the handlers that give effects their meaning. We shall see how a handler for one effect may invalidate or preserve term equivalences established modulo another handler; we shall also see the equivalences that arise from a particular combination of several handlers. Our running example will be the interplay between non-determinism and state.

To account for the equational laws that arise from the interaction of two handlers, we extend the development in §4 in a rather simple way. Instead of one distinguished effect o_0 we now have two: o_1 and o_2, which are to be handled by H_1 and H_2. The expressions e_1 and e_2 are to be considered equivalent modulo H_1 and H_2 if $\mathsf{H}_1[\mathsf{C}_1[\mathsf{H}_2[\mathsf{C}_2[\mathsf{e}_1]\ ]\ ]\ ]$ and $\mathsf{H}_1[\mathsf{C}_1[\mathsf{H}_2[\mathsf{C}_2[\mathsf{e}_2]\ ]\ ]\ ]$ have related denotations, where the contexts C_1 and C_2 have no handlers for the effects of H_1 and H_2. In other words, we consider the equality of the denotations of e_1 and e_2 modulo $\hbar_1 \circ g_1 \circ \hbar_2 \circ g_2$ where g_1 and g_2 are arbitrary context denotations (corresponding to the appropriately typed but otherwise arbitrary C_1 and C_2). Clearly equalities modulo $\hbar_1 \circ g_1 \circ \hbar_2 \circ g_2$ are different from equalities modulo $\hbar_2 \circ g_2 \circ \hbar_1 \circ g_1$, as we shall see soon on concrete examples. We attach the subscript N when talking about non-determinism handlers (one of those described in §6 – which one should be clear from the context) and the subscript S when referring to handleS from §5; thus $\equiv_{NS}$ is the equivalence modulo $\hbar_N \circ g_N \circ \hbar_S \circ g_S$. Throughout this section we assume that expressions have no effects other than non-determinism and state.

7.1 Local State

The first case of combining non-determinism and state is handling the state before non-determinism effects: the context $\mathsf{handleN}(\mathsf{C}_N[\mathsf{handleS}\ \mathsf{s}_0\ \mathsf{C}_S[]\ ])$. The Get-Put laws of state from §5 were proved without making any assumptions about the the other effects. When a non-determinism handler is outermost, its handled expression $\mathsf{C}_N[\mathsf{handleS}\ \mathsf{s}_0\ \mathsf{C}_S[\mathsf{e}]\ ]$ may only have non-determinism effects (since state is dealt with by handleS); therefore, we may use the laws from §6 on the assumption of no other effects. Furthermore, the State effects in non-deterministic branches are handled independently from each other. One may think of it as that each non-deterministic branches has it own *local* copy of the state. This is in contrast to the *global state* situation to be discussed in the next section, where the state

[13] Laws of MonadPlus `http://okmij.org/ftp/Computation/monads.html#monadplus`

is shared among non-deterministic branches, and changes made to the state in one branch carries over to the next.[14]

Lemma 10
If $\mathsf{e}_1 \equiv_S \mathsf{e}_2$ or $\mathsf{e}_1 \equiv_N \mathsf{e}_2$ (regarding e_1 and e_2 as if they may have only fail and Coin effects) then $\mathsf{e}_1 \equiv_{NS} \mathsf{e}_2$

Thus when state is handled locally to non-deterministic choice, we may reason about state and non-determinism independently. The independence is actually stronger than what is implied by Lemma 10:

Lemma 11 (*Early Failure*)

$$\mathsf{put\ s;\ fail} \equiv_{NS} \mathsf{fail} \qquad \mathsf{get;\ fail} \equiv_{NS} \mathsf{fail}$$

In other words, if we are going to fail eventually, we may as well fail right away (and save ourselves from doing useless computations). Compared to Lemma 9, the fail operation here is *not* in the evaluation context: put s and get are. The proof of the lemma is a straightforward calculation.

Lemma 11 says that Fail and state effects do in a sense 'commute', which is of great help in equational reasoning. Failing early is also the most important optimization on non-deterministic programs. The Appendix shows an extended example, a part of the derivation of an efficient *n*-queens solver.

7.2 Global State

When the handlers for non-determinism and state are stacked as $\mathsf{handleS\ s_0}(\mathsf{C}_S[\mathsf{handleN}(\mathsf{C}_N[])])$ (i.e., handleS is outermost), reasoning becomes more complicated. The non-determinism handlers no longer may assume that Coin and Fail are the only effects in their handled expression. Although the analogue of Lemma 10 still holds, we can use only those $\equiv_N$ equivalences that are valid in the presence of other effects: that is, only (assoc). The useful early failure Lemma 11 no longer holds: put s; fail and fail are distinct because the former affects the global state that the context C_S may observe.

To better appreciate the subtlety of reasoning with global state, we consider two simple examples, distilled from realistic derivations. Let s be a value of the state type handled by handleS and v an arbitrary value.

$$\mathsf{e}_1 \coloneqq \mathsf{put\ s;\ v}$$
$$\mathsf{e}_2 \coloneqq \mathbf{let}\ \mathsf{s}_0 = \mathsf{get}\ \mathbf{in\ if}\ \mathsf{coin}\ \mathbf{then}\ (\mathsf{put\ s;\ v})\ \mathbf{else}\ (\mathsf{put\ s}_0\mathsf{;\ fail})$$

In the case of the local state, discussed in §7.1, the two expressions are equivalent: $\mathsf{e}_1 \equiv_{NS} \mathsf{e}_2$. One can easily see it by first applying Lemma 11 and simplifying $(\mathsf{put\ s}_0\mathsf{;\ fail})$ to just fail; The laws of failure Lemma 9 prove the equivalence. For the global state Lemma 11 does not hold and hence the two expressions are distinct: whereas e_1 changes the global

[14] The terminologies "local/global state" were used in this sense by, for example, Wu et al. (2014). This is different from, for example, Plotkin and Power (2003), where "local state" denotes the state effect that allows one to create new local variables.

state to s, e_2 appears to leave it, effectively intact. One may even think that e_2 is equivalent to just v. However, the following calculation of the denotation of e_2 in the context handleS s_0 (C_S[handleN(C_N[])]) shows this not to be the case:

$$\begin{aligned}&\hbar_S(g_S(\hbar_N(g_N\mathtt{E}(\mathsf{Get},\star,\lambda s_o.\,\mathtt{E}(\mathsf{Coin},\star,\\&\qquad\lambda b.\mathit{if}(b,\mathtt{E}(\mathsf{Put},s,\lambda().\,\mathtt{V}(v)),\mathtt{E}(\mathsf{Put},s_0,\lambda().\,\mathtt{E}(\mathsf{Fail},\star,\bot))))))))\ s_0\\&=\hbar_S(g_S(\hbar_N\mathtt{E}(\mathsf{Coin},\star,\lambda b.\mathit{if}(b,\mathtt{E}(\mathsf{Put},s,\lambda().\,g_N\ \mathtt{V}(v)),\mathtt{E}(\mathsf{Put},s_0,\lambda().\,\mathtt{E}(\mathsf{Fail},\star,\bot)))))))\ s_0\end{aligned}$$

If we put just v in the above context, the application of the context denotation $g_N\ \mathtt{V}(v)$ occurs in the context of the global state s_0; whereas in the case of e_2 the same application occurs within $\mathtt{E}(\mathsf{Put},s,\lambda().\,g_N\ \mathtt{V}(v))$, that is, after we have requested to change the state to s. The context denotation g_N may query the state and hence distinguish the two cases. It may also appear that when $\hbar_N$ is handleOne from 6.4, e_2 is equivalent to e_1. Again, the above denotation shows this not to be the case. Recall, handleOne may also evaluate the second branch of the non-deterministic choice, if the first branch produced a failure, that is, if $g_N\ \mathtt{V}(v)$ turns out to have $\mathtt{E}(\mathsf{Fail},\star,\bot)$ as a denotation (which may happen for the appropriately chosen g_N).

The next example is taken verbatim from a real derivation. It concerns the following pair of expressions, where again s is some value of the appropriate state type, v is an arbitrary value and e is an arbitrary expression.

$e_1 \coloneqq$ **if** coin **then** (put s; v) **else** e
$e_2 \coloneqq$ **if** coin **then** v **else** (put s; e)

It has truly appeared that e_1 and e_2 are equivalent. But let's look at the denotations of the two expressions in the context handleS s_0 (C_S[handleN(C_N[])])

$$\hbar_S(g_S(\hbar_N\mathtt{E}(\mathsf{Coin},\star,\lambda b.\mathit{if}(b,\mathtt{E}(\mathsf{Put},s,g_N\ \mathtt{V}(v)),g_N\ e))))\ s_0$$
$$\hbar_S(g_S(\hbar_N\mathtt{E}(\mathsf{Coin},\star,\lambda b.\mathit{if}(b,g_N\ \mathtt{V}(v),\mathtt{E}(\mathsf{Put},s,g_N\ e))))\ s_0$$

The difference becomes easy to see: the application of the context denotation $g_N\ \mathtt{V}(v)$ appears under different global states.

Interestingly, we find the equivalence for a simple modification of e_1 and e_2 (e and e' are arbitrary expressions):

Lemma 12

put s; (**if** coin **then** e **else** e') $\equiv_{SN}$ **if** coin **then** (put s; e) **else** e'

which is the immediate consequence of Lemma 8. Therefore, the equivalence actually holds for any other state handler, besides handleS. It is not 'free', however: it is specific to handleGen or its instantiations such as handleList. (Informally, it holds only for handlers that, when dealing with a Coin operation, pursue the **true** branch first.) Clearly the above equivalence is incompatible with (comm).

Another equivalence following from Lemma 8 is in the spirit of Lemma 11, letting us commute state and failure, to some extent:

Lemma 13

if coin **then** (e_S; fail) **then** e $\equiv_{SN}$ e_S; e

where e_S is an expression that only performs state effects; e is arbitrary.

8 Related Work

8.1 Algebraic Effects

The algebraic effects approach has originated in the work of Plotkin & Power ((2003), among others), aiming for a uniform denotational treatment of actual effects such as mutation of memory cells, hardware exceptions, etc. Effects are represented (and hence, described) by equational theories. Multiple effects are likewise treated axiomatically, as compositions of equational theories. The effect operations hence have a meaning of their own, specified through algebraic identities. Plotkin and Power also elucidated and developed the correspondence between algebraic operations and generic effects. Effect handlers did not come until about decade later, in the work of Plotkin & Pretnar (2010). A handler is taken to be an algebra homomorphism, and hence has to respect the identities that are associated with the effects they are to handle. Since the pioneering work of Plotkin and Power, the approach has developed significantly; in fact, it has become rather difficult to cite all the relevant literature. A significant portion of the literature deals with types – far more sophisticated than the ones used in our calculus – that are to reflect possible effects of an expression.

8.2 Extensible Denotational Language Specifications

Overcoming the ad hoc treatment of various effects was also the motivation of Cartwright & Felleisen (1994). In their approach, however, an effectful operation such as 'store' has no inherent meaning; it is the handler, of the 'central authority', which executes the 'store' and other requests that gives the operation its meaning. As they wrote in 1994:

> *An effect is most easily understood as an interaction between a sub-expression and a central authority that administers the global resources of a program. [. . .] Given an administrator, an effect can be viewed as a message to the central authority plus enough information to resume the suspended calculation.*

The denotational model of effects developed by Cartwright and Felleisen uses one global trusted handler such that the semantics of every effectful operation is denoted by a message to this handler. The two salient features of this model are (i) that the denotation of an expression does not need to change if more effects are added to the language: only the global handler needs to change, and (ii) interactions of effects are straightforward to model as there is exactly one semantic component that is responsible for managing all the effects.

Cartwright and Felleisen's 'central authority' (the global handler) may be thought of as the 'hardware' that executes the 'actual' effect operations. It does not take a big leap to think of that handler to be part of a program, rather than being stationed outside it. One may think of such embedded handler as a virtual machine, or a hypervisor. Our approach to effects comes from this view.

8.3 Bauer and Pretnar's Eff and its Formalizations

One can see the clear influence of Eff 3.4 (Bauer & Pretnar, 2015) and especially of its formalization in Bauer & Pretnar (2014) in the design of our calculus and its denotational semantics. In fact, our denotational semantics is essentially the same as the skeletal semantics of Bauer & Pretnar. Our calculus however is more core than Bauer and Pretnar's Core Eff: We do not distinguish between effects and operations because each effect has only one operation. Kiselyov & Sivaramakrishnan (2018) demonstrate why this is adequate. We also introduce pure computations and take $\circledast$ as a primitive operation. The thrust of Bauer & Pretnar (2014) is the effect system. We, in contrast, focus on reasoning and get by with a simple type system.

One can see the origin of equivalence modulo handlers in Bauer and Pretnar's work (§7.1)(2014): *"With handlers the workings of a computation may be inspected in a highly intensional way. Consequently, there are few generally valid observational equivalences. However, when known handlers are used to handle operations, we may derive equivalences that describe the behavior of operations. The situation is opposite to that of [19], where we start with an equational theory for operations and require that the handlers respect it."* This quote characterizes our work as well.

Bauer & Pretnar (2014) derive several equivalences modulo particular state effect handlers, including the Get-Put laws. However, their laws are not actually equational laws because they are not congruences and do not apply in general contexts. Although their laws can be used to simplify handler[put x; put x] to just handler[put x], they do not apply to simplify the body of the function f in **let** f = **fn** x. put x; put x **in** ... (even if we assume a particular handler that will eventually handle all state effects in a program). The present paper introduces truly equational reasoning modulo handlers, which can be carried out in any context.

8.4 Equational Reasoning with Monads

Most research on monads and effects are either on the theoretical side or are about, as libraries, implementations of various monads and their combinations. Curiously, it is less common seeing people reason about actual monadic programs. Hutton & Fulger (2007) proved the correctness of a tree labelling algorithm that uses a monad. Their approach, however, essentially unfolds the definitions of the monad and the effect operators, proving everything for this specific monad.

Gibbons & Hinze (2011) is an archetypal example of monadic equational reasoning. It showed that correctness of monadic programs, for example a backtracking program solving the *n*-queens problem using non-determinism and state, can be proved using the monad laws and properties the effect operators are supposed to satisfy, and independently from the actual implementation of the monad. They classified effects into hierarchical classes, and identified various properties these effects are supposed to have.

Affeldt & Nowak (2018) formally verified the results of Hinze and Gibbons in Coq, using the features of Coq to modularly organize the hierarchy of effects and their properties. The properties are used to verify effectful algorithms. A separate proof shows that a given monad, with a monolithic implementation, satisfies the properties.

8.5 Other Related Work

Forster et al. (2019) demonstrate an alternative denotational framework, based on a call-by-push-value calculus enhanced with effect operations, with set-theoretic (algebraic) semantics. The authors of that work had put the semantics to a different, from ours, use: to rigorously prove or disprove expressibility of different ways of realizing computational effects (algebraic effects, delimited control and monad reflection). It seems likely our development of equivalence-modulo-handlers can also be done in their denotational framework.

Deriving, rather than postulating equational laws was also the motivation of Johann, Simpson and Voigtländer (2010). Their approach, however, is in some sense opposite from ours: syntactic rather than semantics, operational rather than denotational. Yet trees (and preorders on trees) play just as essential role.

9 Conclusions

This paper champions, develops and applies the idea that a handler determines the equational theory of the effects, rather than the equational theory taken as given. Equational laws emerge as a distillation of the behavior of a handler or a combination thereof. In this, "bottom-up" approach harking back to Harel and Pratt (1978), we are assured that our reasoning, and laws, are grounded, in a real and practical implementation. Our main technical tool is denotational semantics, which we hence also champion and apply, for reasoning about and optimizing interesting programs involving multiple, interleaving effects.

Technically, 'equational theory from handler' takes the form of equivalence modulo handler, which we introduce and prove it to be congruence and an equivalence relation. We apply this notion to investigate the equational theory of popular handlers for non-determinism (List and Maybe monads), and also two combinations of non-determinism and state (Local and Global state).

Acknowledgements

Conversations with Gordon Plotkin are gratefully acknowledged. We thank the anonymous reviewers for many helpful comments and suggestions.

This work was partially supported by JSPS KAKENHI Grants Number 18H03218 and 17K00091.

References

Affeldt, Reynald, & Nowak, David. (2018). Experimenting with monadic equational reasoning in Coq. *The 35th Conference of the Japan Society for Software Science and Technology*.

Armoni, Michal, & Ben-Ari, Mordechai. (2009). The concept of nondeterminism: its development and implications for teaching. *Sigcse bulletin*, **41**(2), 141–160.

Bauer, Andrej, & Pretnar, Matija. (2014). An effect system for algebraic effects and handlers. *Logical methods in computer science*, **1**(4).

Bauer, Andrej, & Pretnar, Matija. (2015). Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming*, **84**(1), 108–123.

Brady, Edwin. (2013). Programming and reasoning with algebraic effects and dependent types. *Pages 133–144 of: ICFP '13*. ACM Press.

Cartwright, Robert, & Felleisen, Matthias. (1994). Extensible denotational language specifications. *Pages 244–272 of:* Hagiya, Masami, & Mitchell, John C. (eds), *Theor. Aspects of Comp. Soft.* LNCS, no. 789. Springer.

Chen, Yu-Fang, Hong, Chih-Duo, Lengál, Ondřej, Mu, Shin-Cheng, Sinha, Nishant, & Wang, Bow-Yaw. (2017). An executable sequential specification for Spark aggregation. *International Conference on Networked Systems*. Springer-Verlag.

Fischer, Sebastian, Kiselyov, Oleg, & Shan, Chung-Chieh. (2011). Purely functional lazy nondeterministic programming. *Journal of functional programming*, **21**, 413–465.

Forster, Yannick, Kammar, Ohad, Lindley, Sam, & Pretnar, Matija. (2019). On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. funct. program*, **29**, e15.

Gibbons, Jeremy, & Hinze, Ralf. (2011). Just do it: Simple monadic equational reasoning. *Pages 2–14 of: ICFP '11*. ACM Press.

Harel, David, & Pratt, Vaughan R. 1978 (Jan.). Nondeterminism in logics of programs (preliminary report). *Pages 203–213 of: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*.

Hinze, Ralf. (2000). Deriving backtracking monad transformers. *Pages 186–197 of: ICFP '00*. ACM Press.

Hutton, Graham, & Fulger, Diana. (2007). Reasoning about effects: seeing the wood through the trees. *Symposium on Trends in Functional Programming*.

Hyland, Martin, Plotkin, Gordon, & Power, John. (2006). Combining effects: Sum and tensor. *Theoretical computer science*, **357**(1–3), 70–99.

Johann, Patricia, Simpson, Alex, & Voigtländer, Janis. (2010). A generic operational metatheory for algebraic effects. *Pages 209–218 of: LICS*. IEEE Press.

Kiselyov, Oleg. 2017 (3 Sept.). *Higher-order programming is an effect*. HOPE 2017 at ICFP 2017.

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

Kiselyov, Oleg, & Sivaramakrishnan, KC. (2018). Eff directly in OCaml. *Electronic proceedings in theoretical computer science*, **285**, 23–58.

Kiselyov, Oleg, Sabry, Amr, & Swords, Cameron. (2013). Extensible effects: an alternative to monad transformers. *Pages 59–70 of: Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*. ACM.

Leijen, Daan. (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. New York, NY, USA: ACM.

Moggi, Eugenio. (1989). *An abstract view of programming languages*. Tech. rept. ECS-LFCS-90-113. Edinburgh Univ.

Mosses, Peter D. (1990). Denotational semantics. *Chap. 11, pages 577–631 of:* van Leewen, J. (ed), *Handbook of Theoretical Computer Science*, vol. B: Formal Models and Semantics. New York, NY: The MIT Press.

Mu, Shin-Cheng. (2019a). *Calculating a backtracking algorithm: an exercise in monadic program derivation*. Tech. rept. TR-IIS-19-003. Institute of Information Science, Academia Sinica.

Mu, Shin-Cheng. (2019b). *Equational reasoning for non-determinism monad: the case of Spark aggregation*. Tech. rept. TR-IIS-19-002. Institute of Information Science, Academia Sinica.

Pitts, Andrew M. (1996). Relational properties of domains. *Information and computation*, **127**(2), 66–90.

Plotkin, Gordon, & Power, John. (2003). Algebraic operations and generic effects. *Applied categorical structures*, **11**(1), 69–94.

Plotkin, Gordon, & Pretnar, Matija. (2009). Handlers of algebraic effects. *Pages 80–94 of:* Castagna, Giuseppe (ed), *Programming Languages and Systems*. Lecture Notes in Computer Science, vol. 5502. Springer.

Pretnar, Matija. (2010). *The logic and handling of algebraic effects*. Ph.D. thesis, The University of Edinburgh.

Rabin, Michael O., & Scott, Dana. (1959). Finite automata and their decision problems. *IBM journal of research and development*, **3**, 114–125.

Reynolds, John C. (1981). The essence of Algol. *Pages 345–372 of:* de Bakker, Jacobus Willem, & van Vliet, J. C. (eds), *Algorithmic Languages*. Amsterdam: North-Holland.

Thompson, Ken. (1968). Programming techniques: Regular expression search algorithm. *Commun. acm*, **11**(6), 419–422.

Winskel, Glynn. (1993). *Formal semantics of programming languages*. MIT Press.

Wu, Nicolas, Schrijvers, Tom, & Hinze, Ralf. (2014). Effect handlers in scope. *Pages 1–12 of: Proceedings of the 7th ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. ACM.

Zaharia, Matei, Chowdhury, Mosharaf, Franklin, Michael J., Shenker, Scott, & Stoica, Ion. (2010). Spark: Cluster computing with working sets. *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Berkeley, CA, USA: USENIX Association.

A Proofs of properties of lift

Lemma 14 (*Lift compositions*)

$$f^\dagger \circ g^\dagger = (f^\dagger \circ g)^\dagger \qquad f^\ddagger \circ g^\dagger = (f^\ddagger \circ g)^\dagger \qquad f^\ddagger \circ g^\ddagger = (f \circ g)^\ddagger$$

Proof
To prove the first law, we introduce the following functionals (higher-order functions)

$$Gu \coloneqq \lambda x. \begin{cases} g\ v & \text{if } x = \mathtt{V}(v) \\ E(o,v,u \circ k) & \text{if } x = \mathtt{E}(o,v,k) \end{cases}$$

$$Hu \coloneqq \lambda x. \begin{cases} f^\dagger(g\ v) & \text{if } x = \mathtt{V}(v) \\ E(o,v,u \circ k) & \text{if } x = \mathtt{E}(o,v,k) \end{cases}$$

and define the following sequences of functions:

$$G^0 \coloneqq \bot \qquad\qquad G^{n+1} \coloneqq G\ G^n$$
$$H^0 \coloneqq \bot \qquad\qquad H^{n+1} \coloneqq H\ H^n$$

Clearly, $G^n \sqsubseteq G^{n+1}$ and the same for H^n, thus G^n and H^n form ascending chains. We now show that $f^\dagger \circ G^n = H^n$. Clearly, $f^\dagger \circ G^0 = f^\dagger \circ \bot = \bot = H^0$. Assuming $f^\dagger \circ G^n = H^n$, we have

$$\begin{aligned} & f^\dagger \circ G^{n+1} = f^\dagger \circ G G^n \\ & = \lambda x. \begin{cases} f^\dagger(gv) & \text{if } x = \mathtt{V}(v) \\ E(o,v,f^\dagger \circ G^n \circ k) & \text{if } x = \mathtt{E}(o,v,k) \end{cases} \\ & = \lambda x. \begin{cases} f^\dagger(gv) & \text{if } x = \mathtt{V}(v) \\ E(o,v,H^n \circ k) & \text{if } x = \mathtt{E}(o,v,k) \end{cases} \quad \text{by inductive hypothesis} \\ & = H^{n+1} \end{aligned}$$

The two chains $f^\dagger \circ G^n$ and H^n are hence identical and so have the same least-upper bounds (the existence of the least-upper bounds is guaranteed by the domain). Keeping in mind

that $\mathsf{lub}\, H^n = (f^\dagger \circ g)^\dagger$ and, by continuity of functional composition, $\mathsf{lub}(f^\dagger \circ G^n) = f^\dagger \circ \mathsf{lub}\, G^n = f^\dagger \circ g^\dagger$ we complete the proof. □

The other composition laws of lift are easy consequences, keeping in mind that $f^\ddagger$ is the abbreviation for $(\lambda x.\, \mathtt{V}(f\,x))^\dagger$.

B Free Reasoning with Non-determinism

Recall, we have introduced the effect constant Coin with the associated type $\vdash_o$ Coin: unit↪bool, and the convenient abbreviation

```
coin ≔ op Coin ⊛ ()
```

We now demonstrate less-trivial, realistic programs using the Coin effects, and the combination of semantic and equational reasoning about these programs. The properties of the Coin code we prove here hold for every interpretation of the Coin effect.

We adapt the example from Mu (2019b) (following up Chen et al. (2017)). The task was to model and reason about Spark, a popular platform for distributed data-parallel computation (Zaharia *et al.*, 2010). In Spark, Resilient Distributed Datasets (RDD) are partitioned and stored on distributed servers. When an aggregate operation is performed, results from the servers may arrive in different order. Thus a Spark computation is non-deterministic in nature — possibly with other side effects (e.g. state). While Spark programmers may generally wish that their programs compute the same function regardless of how the data is distributed, unpleasant surprises are not uncommon in practice. Consider the integral of x^{73} on $[-2,2]$. Since x^{73} is an odd function, the result ought to be 0. In our experiments, *AreaUnderCurve.of*, a function in the Spark machine learning library that computes numerical integration distributively, returns different results ranging from -8192.0 to 12288.0 on the same input, due to different orders of floating-point computation.

It is therefore desirable to figure out conditions under which a Spark aggregation is deterministic, that is, when we model it as a monadic program, it in fact equals **val** x for some x. When we model Spark in our language, the fact that sub-computations can be performed in arbitrary order is modeled by transforming a list of sub-results into one of its permutations. The following function non-deterministically computes a permutation of its input list:

```
perm : t list ⇒ t list ≔
  rec perm xs. match xs with
    | []    → []
    | x::xs → insert x ⊛ (perm ⊛ xs)
```

This recursive code relies on the following auxiliary function to non-deterministically insert a value x at some position in the given list xs:

```
insert : t → t list ⇒ t list ≔
 ↿x. rec loop xs.
   match xs with
   | []    → x::[]
   | y::ys → if coin then x::y::ys else (y::) ⊙ (loop ⊛ ys)
```

The code non-deterministically inserts x either at the head of xs, or somewhere inside xs (if it is non-empty).

B.1 Free theorem of non-deterministic list permutation

Among the many properties needed, we would like to make sure that perm commutes with map f for pure f:

Lemma 15 (*'Free' theorem of perm*)

$$\mathsf{perm} \circledast \mathsf{map\ f\ xs} \equiv \mathsf{map\ f} \odot (\mathsf{perm} \circledast \mathsf{xs})$$

Since perm is defined in terms of insert, we need the corresponding property about insert:

Lemma 16 (*'Free' theorem of insert*)
For all x:t, xs:t list, f:t→t', and an arbitrary expression e:t list of arbitrary effects,

1. $\mathsf{insert\ (f\ x)} \circledast \mathsf{map\ f\ xs} \equiv \mathsf{map\ f} \odot (\mathsf{insert\ x} \circledast \mathsf{xs})$
2. $\mathsf{insert\ (f\ x)} \circledast (\mathsf{map\ f} \odot \mathsf{e}) \equiv \mathsf{map\ f} \odot (\mathsf{insert\ x} \circledast \mathsf{e})$

If we just look at the types of perm : t list ⇒ t list and insert: t→t list⇒t list and treat the effectful arrows as regular function type, the properties reminds us of free theorems. We should remember, however, that insert is an effectful function, and the free theorem no longer comes for free. Incidentally, the proof shows that the theorem may be called free in a different sense: it holds for the 'free' non-determinism, regardless of the specific meaning of the Coin effect. No matter how Coin effects may be handled, we are *always* justified, in all possible contexts including under lambda, to replace the left-hand-side term of Lemma 16 with the right-hand-side, or vice-versa.

We demonstrate the semantic proof of Lemma 16: both sides of ≡ have the same denotations. First we need the denotation for insert, to be called *insert*, which we easily compute compositionally as below (it is clearly parametric in the type t of list elements).

$$\begin{array}{rll} [\![\vdash_v \mathsf{insert{:}t{\Rightarrow}t\ list{\Rightarrow}t\ list}]\!]() & = \mathit{insert} & \text{where} \\ \mathit{insert}\ x & \coloneqq \mathsf{fix}(\mathit{insF}\ x) & \\ \mathit{insF}\ x\ u & \coloneqq \lambda l.\begin{cases} \mathtt{V}(x :: []) & \text{if } l = [] \\ \mathtt{E}(\mathsf{Coin},\star,g\,u\,x\,y\,l') & \text{if } l = y :: l' \end{cases} & \\ g\,u\,x\,y\,l' & \coloneqq \lambda b.\mathit{if}(b, \mathtt{V}(x :: y :: l'), (y ::)^{\ddagger}(u\,l')) & \end{array}$$

Proof of Lemma 16
For part 1, we aim to show that

$$\mathit{insert}\ (f\,x)\ (\mathit{map}f\ l) = (\mathit{map}f)^{\ddagger}(\mathit{insert}\ x\ l)$$

where x is the denotation of x, f is of f, l of ys. The proof proceeds very similarly to the proof of Lemma 2. It uses induction, but *not* on the length of the list; rather, it is an induction on the approximants to the fixpoint.

We define the following sequences of functions:

$$\begin{array}{ll} G^0 \coloneqq \bot & G^{n+1} \coloneqq \mathit{insF}\ (f\,x)\ G^n \\ H^0 \coloneqq \bot & H^{n+1} \coloneqq \mathit{insF}\ x\ H^n \end{array}$$

It is easy to see that each function in the sequence is more defined than the earlier ones: $G^n \sqsubseteq G^{n+1}$ and $H^n \sqsubseteq H^{n+1}$, so that the sequences form an ascending chain – and possess least-upper bounds, by the definition of domains. From the continuity of all involved functions,

$$\begin{aligned} \mathit{insert}\ (f\,x)\ (\mathit{map}f\ l) &= (\mathsf{lub}G^n)\ (\mathit{map}f\ l) = \mathsf{lub}(G^n\ (\mathit{map}f\ l)) \\ (\mathit{map}f)^{\ddagger}(\mathit{insert}\ x\ l) &= (\mathit{map}f)^{\ddagger}((\mathsf{lub}H^n)\ l) = \mathsf{lub}((\mathit{map}f)^{\ddagger}(H^n\ l)) \end{aligned}$$

The lubs in the two lines are the same; in fact, we now demonstrate that $G^n\ (\mathit{map}f\ l) = (\mathit{map}f)^{\ddagger}(H^n\ l)$ for all n, by simple induction. The base case $n = 0$ is trivial. For the inductive case $n+1$, we rely on the property of $\mathit{map}f$ that if l is empty then so is $\mathit{map}f\ l$ and if l is $y :: l'$ for some y and l' then $\mathit{map}f\ l$ is likewise representable as $f\,y :: \mathit{map}f\ l'$. Thus it remains to show that

$$\begin{aligned} &\lambda b.\mathit{if}(b, \mathtt{V}(f\,x :: f\,y :: \mathit{map}f\ l'), (f\,y ::)^{\ddagger}(G^n\,(\mathit{map}f\ l'))) \\ =\ &\lambda b.\mathit{if}(b, \mathtt{V}(f\,x :: f\,y :: \mathit{map}f\ l'), (\mathit{map}f)^{\ddagger}((y ::)^{\ddagger}(H^n\,l'))) \end{aligned}$$

which immediately follows from Lemma 2, the properties of list map, and the inductive hypothesis.

Part 2 of the lemma is a simple consequence of Lemma 2. □

For Lemma 15, we show a syntactic proof for a change, using the already established equational laws – by structural induction on the list xs. (We skip the trivial base case of the empty list.) The proof is quite short thanks to Lemma 16, part 2, which deals with applications of insert x to arbitrary effectful expressions.

Proof of Lemma 15
Now, assuming the lemma holds for xs, we have for x::xs:

```
    perm ⊛ map f (x::xs)
≡   perm ⊛ (f x::map f xs)                {map laws}
≡   insert (f x) ⊛ (perm ⊛ map f xs)      {substitution laws}
≡   insert (f x) ⊛ (map f ⊙ (perm ⊛ xs))  {induction hypothesis}
≡   map f ⊙ (insert x ⊛ (perm ⊛ xs))      {insert law, part 2}
≡   map f ⊙ (perm ⊛ (x::xs))              {substitution law, opposite direction}
```

which completes the proof. □

B.2 Don't care non-determinism

Recall that our eventual goal is to clarify when the results of a Spark run do not depend on the partitioning scheme for Resilient Distributed Datasets. We shall model Spark as a map-reduce–like processing with the familiar foldl (often called 'reduce'): if $\triangleleft$ is a left-associative binary infix operation, $\mathit{foldl}\ (\triangleleft)\ z\ [x_1, \ldots, x_n]$ is $z \triangleleft x_1 \triangleleft \ldots \triangleleft x_n$. It can be also defined equationally as

```
foldl : (u → t → u) → u → t list → u
foldl f z []       ≡ z
foldl f z (x :: xs) ≡ foldl f (f z x) xs
```

To state the desired property we also need the auxiliary functions

```
units : t list ⇒ unit ≔ rec units xs. match xs with
  | []    → ()
  | y::ys → if coin then () else units ⊛ ys
unitsf : t list ⇒ unit ≔ rec unitsf xs. match xs with
  | []    → ()
  | y::ys → unitsf ⊛ ys; units ⊛ ys
noperm : t list ⇒ t list ≔ rec noperm xs. match xs with
  | []    → ()
  | y::ys → let l = noperm ⊛ ys in units ⊛ l; y::l
```

Here, units does not actually do anything always returning unit, but still making as many choices as the length of its argument list; unitsf is similarly useless but makes the factorial number of choices. Finally noperm ⊛ xs is unitsf ⊛ xs; xs (which is easy to prove equationally using the associative laws).

We begin with an easy lemma

Lemma 17
If the function f : u→t→u has the property f (f z x) y ≡ f (f z y) x for all x, y, and z, then

```
foldl f z ⊙ (insert x ⊛ xs) ≡ units xs; foldl f z (x::xs)
```

The assumed property of f is actually common among aggregation and ranking functions (e.g, 'count'). Our goal is to prove the corresponding property of perm, stating the 'don't care non-determinism' – when the result of foldl equals to that of a pure computation:

Theorem 2
If the function f : u→t→u has the property f (f z x) y ≡ f (f z y) x for all x, y, and z, then

```
foldl f z ⊙ perm xs ≡ unitsf xs; foldl f z xs
```

However, it is not easy. Mu (2019b) previously demonstrated an equational proof, which required postulating very strong equational laws on non-deterministic computations (such as commutativity and idempotence) – the laws that are not satisfied by many implementations of non-determinism, as we discussed in §1.1. To see why such strong laws seem to be indispensable, let's try to prove the theorem by induction. Here is the inductive step, using Lemma 17

```
foldl f z ⊙ perm (x:xs)
≡ let l = perm ⊛ xs in foldl f z ⊙ (insert x ⊛ l)
≡ let l = perm ⊛ xs in units l; (foldl f z (x::l))
```

To apply the inductive hypothesis, we should somehow move units l 'out of the way'. If we postulate idempotence, it collapses to just one choice and disappears. However, without imposing any assumptions on non-deterministic computations, we are stuck, it seems.

The semantic approach turned out more helpful, leading us to think of equivalences other than equality. Reasoning modulo permutation proved particularly useful, as we now demonstrate. Specifically, we state, and then prove, the key lemmas about the mathematical properties of the denotations of perm and insert:

Lemma 18
Let *sort* be a stable list sorting function, in some fixed but otherwise arbitrary order, and f satisfies the premise of Thm. 2.

1. $foldl f\ z\ l = foldl f\ z\ (sort\, l)$
2. $sort^{\ddagger}(insertx\ l) = (\lambda\_.\, sort\,(x :: l))^{\ddagger}(units\, l)$
3. $sort^{\ddagger}(perm\ \ l) = sort^{\ddagger}(noperm\, l)$

The lemmas describe what the denotation *insert* and *perm* are. We have to describe not just the result of the computation but also its effects (captured in *units*, the denotation of units).

Proof
Part 1 is the property of *foldl* given f that satisfies the premise of Thm. 2. We prove the other parts by induction on the length of list l. We elide the trivial base cases of the empty list. For part 2, we calculate

$sort^{\ddagger}(insertx\ (y :: l))$

Definition of *insert* and the fixpoint

$= sort^{\ddagger}(insF\ x\ (insertx)\ (y :: l))$

Unrolling definitions

$= \mathtt{E}(\mathsf{Coin}, \star,\ \lambda b.\, if\,(b, \mathtt{V}(sort\,(x :: y :: l)), sort^{\ddagger}((y ::)^{\ddagger}(insertx\, l))))$

Property of sort: $sort\,(y :: l) = sort\,(y :: (sort\, l))$

$= \mathtt{E}(\mathsf{Coin}, \star,\ \lambda b.\, if\,(b, \mathtt{V}(sort\,(x :: y :: l)), sort^{\ddagger}((y ::)^{\ddagger}(sort^{\ddagger}(insertx\, l)))))$

Applying the induction hypothesis

$= \mathtt{E}(\mathsf{Coin}, \star,\ \lambda b.\, if\,(b, \mathtt{V}(sort\,(x :: y :: l)), sort^{\ddagger}((y ::)^{\ddagger}((\lambda\_.\, sort\,(x :: l))^{\ddagger}(units\, l)))))$

Property of lifting

$= \mathtt{E}(\mathsf{Coin}, \star,\ \lambda b.\, if\,(b, \mathtt{V}(sort\,(x :: y :: l)), ((\lambda\_.\, (sort \circ (y ::) \circ sort)(x :: l))^{\ddagger}(units\, l))))$

Again the property of sort

$= \mathtt{E}(\mathsf{Coin}, \star,\ \lambda b.\, if\,(b, \mathtt{V}(sort\,(x :: y :: l)), ((\lambda\_.\, sort\,(x :: y :: l))^{\ddagger}(units\, l))))$

$= (\lambda\_.\, sort\,(x :: y :: l))^{\ddagger}\mathtt{E}(\mathsf{Coin}, \star, \lambda b.\, if\,(b, \mathtt{V}(\star), units\, l))$

Definition of *units*, derived from units

$= (\lambda\_.\, sort\,(x :: y :: l))^{\ddagger}(units\,(y :: l))$

Part 3 follows from Part 2 and is proven similarly. The crucial identity

$$\begin{aligned} &(\lambda l'.\, (\lambda\_.\, sort\,(x :: l'))^{\ddagger}(units\, l'))^{\dagger}(perm\, l) \\ = &(\lambda l'.\, (\lambda\_.\, sort\,(x :: l'))^{\ddagger}(units\, l'))^{\dagger}(sort^{\ddagger}(perm\, l)) \end{aligned}$$

follows from the property of sort and the easily proven $units\, l = units\,(sort\, l)$ □

Theorem 2 is then the simple consequence of Lemma 18(1,3). We stress that we have proven the theorem without even considering any handler for non-determinism. That is, the theorem is valid for all possible implementations. This is in stark contrast with the previously available equational proof, which required very strong assumptions (which many implementations of non-determinism do not satisfy).

C Proof of the GetPut Law of State

The GetPut law to be proven here is as follows, where the equivalence it to be taken modulo the handler for State $\hbar$ presented in §4 and C_s is an evaluation context with no handlers for either Get or Put.

$$\textbf{let } \mathsf{x} = \mathsf{get} \textbf{ in } \mathsf{C}_s[\mathsf{put}\ \mathsf{x}] \equiv_h \textbf{let } \mathsf{x} = \mathsf{get} \textbf{ in } \mathsf{C}_s[\textbf{val } ()]$$

Proof
Let $\mathsf{C'}_s$ be another evaluation context with no handlers for State and let g'_s be $[\![\mathsf{C'}_s]\!]$. We write k for $[\![\mathsf{C}_s[\textbf{val } -]]\!]$, which is $\lambda\rho.\lambda x.[\![\mathsf{C}_s]\!]\rho\mathtt{V}(x)$. We have to verify that $\hbar \circ (g'_s\rho)$ pre-composed with the denotations of $\textbf{let } \mathsf{x} = \mathsf{get} \textbf{ in } \mathsf{C}_s[\mathsf{put}\ \mathsf{x}]$ and $\textbf{let } \mathsf{x} = \mathsf{get} \textbf{ in } \mathsf{C}_s[\textbf{val } ()]$ gives identical results. We compute the denotations as follows

$[\![\mathsf{get}]\!]\rho$	$\mathtt{E}(\mathsf{Get}, \star, \lambda s.\mathtt{V}(s))$
$[\![\mathsf{C}_s[\mathsf{put}\ \mathsf{x}]]\!]\rho$	$\mathtt{E}(\mathsf{Put}, \rho.\mathsf{x}, k\rho)$
$[\![\textbf{let } \mathsf{x} = \mathsf{get} \textbf{ in } \mathsf{C}_s[\mathsf{put}\ \mathsf{x}]]\!]\rho$	$\mathtt{E}(\mathsf{Get}, \star, \lambda s.\mathtt{E}(\mathsf{Put}, s, k\rho'))$
	where ρ' is $\rho \times \mathsf{x} : s$
$[\![\textbf{let } \mathsf{x} = \mathsf{get} \textbf{ in } \mathsf{C}_s[\textbf{val } ()]]\!]\rho$	$\mathtt{E}(\mathsf{Get}, \star, \lambda s.k\rho'\star)$
$\lambda\rho.\hbar(g'_s\rho\ [\![\textbf{let } \mathsf{x} = \mathsf{get} \textbf{ in } \mathsf{C}_s[\mathsf{put}\ \mathsf{x}]]\!]\rho)$	$\lambda\rho.\mathtt{V}(\lambda s.(\hbar\mathtt{E}(\mathsf{Put}, s, g'_s\rho \circ k\rho'))^{¶}s)$
	$= \lambda\rho.\mathtt{V}(\lambda s.(\mathtt{V}(\lambda s'.(\hbar(g'_s\rho(k\rho'\star)))^{¶}s))^{¶}s)$
	$= \lambda\rho.\mathtt{V}(\lambda s.(\hbar(g'_s\rho(k\rho'\star)))^{¶}s)$
$\lambda\rho.\hbar(g'_s\rho\ [\![\textbf{let } \mathsf{x} = \mathsf{get} \textbf{ in } \mathsf{C}_s[\textbf{val } ()]]\!]\rho)$	$\lambda\rho.\mathtt{V}(\lambda s.(\hbar(g'_s\rho(k\rho'\star)))^{¶}s)$

□

D Reasoning with State

As an illustration of using the (generalized) equational laws modulo handleS, we take an example adapted from Mu (2019a) on deriving monadic programs from specifications: deriving an efficient *n*-queens solver from the obvious specification of producing all permutations of queen arrangement in columns and filtering out those where queens beat each other diagonally. The test of safety of a queen arrangement involves the conversion of the list of queen column positions to the list of up- and down-diagonal positions, which can be written in terms of scanl. The common list processing function $\mathsf{scanl} : (\mathsf{b}{\to}\mathsf{a}{\to}\mathsf{b}) \to \mathsf{b} \to \mathsf{a\ list} \to \mathsf{b\ list}$ is inductively defined as

$$\begin{aligned}&\mathsf{scanl\ f\ s\ []} &&\equiv [\mathsf{s}]\\ &\mathsf{scanl\ f\ s\ (x :: xs)} &&\equiv \mathsf{s} :: \mathsf{scanl\ f\ (f\ s\ x)\ xs}\end{aligned}$$

such that $scanl\ f\ s\ [x_1, x_2, \ldots, x_n] = [s, f\,s\,x_1, f\,(f\,s\,x_1)\,x_2, \ldots, f\,(f\ldots)\,x_n]$. It is the generalization of the earlier foldl: the last element of scanl f s xs is foldl f s xs. One may discern in the scanl processing the combination of building a list according to the current state s and updating the current state. From the point of view of the *n*-queens solver derivation, it is desirable to separate the two aspects: represent the list consumption and production into

a foldr (which could later be fused with other list processors) and move the state 'out of the way', that is, to update it as a side-effect.

In an upshot, we would like to perform the scanl processing using mutable state, as

$$\begin{array}{l}\mathsf{scanM} : (\mathsf{b} \to \mathsf{a} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{a\ list} \Rightarrow \mathsf{b\ list}\\ \mathsf{scanM} \coloneqq \diagup\!\!\diagup \mathsf{f}. \diagup\!\!\diagup \mathsf{s}_0.\mathbf{fn}\,\mathsf{xs}.\ \mathsf{put}\ \mathsf{s}_0;\ \mathsf{foldr}\ \mathsf{g}\ (\mathbf{fn}\,().[])\ \mathsf{xs} \circledast ()\end{array}$$

The foldr's step function g is defined by:

$$\mathsf{g} \coloneqq \diagup\!\!\diagup \mathsf{x}. \diagup\!\!\diagup \mathsf{u}.\mathbf{fn}\,().\ \mathbf{let}\ \mathsf{s} = \mathsf{get}\ \mathbf{in}\ (\mathsf{f\ s\ x}::) \odot (\mathsf{put}\ (\mathsf{f\ s\ x});\ \mathsf{u} \circledast ())$$

We would like to relate scanM to scanl and hence verify that it does what we want from it. To this end, we derive the inductive characterization of scanM, using the equational laws of State, among others:

$$\begin{array}{l}\mathsf{scanM\ f\ s}_0\ [] \equiv_h \mathsf{put\ s}_0;\ []\\ \mathsf{scanM\ f\ s}_0\ (\mathsf{x}::\mathsf{xs})\\ \quad \equiv_h \mathsf{put\ s}_0;\ \mathsf{g\ x}\ (\mathsf{foldr\ g}\ (\mathbf{fn}\,().\ [])\ \mathsf{xs}) \circledast ()\\ \quad \equiv_h \mathsf{put\ s}_0;\ (\mathbf{fn}\,().\ \mathbf{let}\ \mathsf{s} = \mathsf{get}\ \mathbf{in}\\ \qquad\qquad (\mathsf{f\ s\ x}::) \odot (\mathsf{put}\ (\mathsf{f\ s\ x});\ (\mathsf{foldr\ g}\ (\mathbf{fn}\,().\ [])\ \mathsf{xs}) \circledast ())) \circledast ()\\ \quad \equiv_h \mathsf{put\ s}_0;\ \mathbf{let}\ \mathsf{s} = \mathsf{get}\ \mathbf{in}\\ \qquad\qquad (\mathsf{f\ s\ x}::) \odot (\mathsf{put}\ (\mathsf{f\ s\ x});\ (\mathsf{foldr\ g}\ (\mathbf{fn}\,().\ [])\ \mathsf{xs}) \circledast ())\\ \quad \{\ \mathsf{PutGet\ law}\ \}\\ \quad \equiv_h \mathsf{put\ s}_0;\ (\mathsf{f\ s}_0\ \mathsf{x}::) \odot (\mathsf{put}\ (\mathsf{f\ s}_0\ \mathsf{x});\ (\mathsf{foldr\ g}\ (\mathbf{fn}\,().\ [])\ \mathsf{xs}) \circledast ())\\ \quad \{\mathsf{General\ PutPut\ law}\ \}\\ \quad \equiv_h (\mathsf{f\ s}_0\ \mathsf{x}::) \odot (\mathsf{put}\ (\mathsf{f\ s}_0\ \mathsf{x});\ (\mathsf{foldr\ g}\ (\mathbf{fn}\,().\ [])\ \mathsf{xs}) \circledast ())\\ \quad \equiv_h (\mathsf{f\ s}_0\ \mathsf{x}::) \odot \mathsf{scanM\ f}\ (\mathsf{f\ s}_0\ \mathsf{x})\ \mathsf{xs}\end{array}$$

We have taken advantage of the general PutPut law to get rid of $\mathsf{put\ s}_0$ when another put is present in the evaluation context, but not immediately adjacent. One can now see by straightforward induction:

$$\mathsf{scanM\ f\ s\ xs} \quad \equiv_h \quad \mathbf{let}\ \mathsf{ys} = (\mathsf{scanl\ f\ s\ xs})\ \mathbf{in}\ \mathsf{put}\ (\mathsf{last\ ys});\ (\mathsf{tail\ ys})$$

(where last gives the last element of the list). Thus scanM indeed does the scanl processing.

E Reasoning with Non-determinism and Local State

This section illustrates the reasoning with non-determinism and local state, relying on the $\equiv_{NS}$ equivalence introduced in §7.1. We continue the scanM example from §D. Recall, the goal was to derive an efficient *n*-queens solver from an inefficient but obviously correct specification. The specification is to non-deterministically generate a list xs of queen column positions, and filter out 'bad' arrangements. The filtering can be expressed as follows (see the pattern discussed in §6.4)

$$\mathsf{filtert}_1\ \mathsf{ok\ f\ s}_0\ \mathsf{xs} \coloneqq \mathbf{if}\ \mathsf{all\ ok} \odot (\mathsf{tail} \circledast \mathsf{scanl\ f\ s}_0\ \mathsf{xs})\ \mathbf{then}\ \mathsf{xs}\ \mathbf{else}\ \mathsf{fail}$$

where $\mathsf{scanl\ f\ s}_0$ (with appropriately chosen f and s_0, which we elide) computes diagonals for each queen, and all ok checks that they are 'safe' (that is, no queen is at the diagonal of another). Here ok:t→bool and

```
all:(t→bool) → t list → bool ≔ ⩤ok. foldr (⩤x.⩤a. band (ok x) a) true
```

In §D we converted scanl f s_0 to the stateful scanM expression, so that the filtering can be written as

```
filtert₂ ok f s₀ xs ≔
  let ys = scanM f s₀ ⊛  xs in
   if all ok ys then xs else fail
```

where scanM was written as a foldr (of a bit complicated stateful function). The reason, we said in §D, is to later fuse it, with the foldr in all ok. We now demonstrate the fused result:

```
filtert₃ ok f s₀ xs ≔ put s₀; (foldr g (fn (). nil) xs ⊛ ())
where
 g ≔ ⩤x.⩤u.fn ().
   let s = get in
    if ok (f s  x) then (put (f s  x); ((x::) ⊙ (u ⊛ ()))) else fail
```

We replaced two traversals of xs (one in scanl/scanM and the other in all) with the single traversal. Most importantly, once we detect an unsafe diagonal (that is, once the ok test returns false), we fail right away, without further computations and tests.

Theorem 3
For all ok:t→bool, f:t→t'→t, s_0:t and xs:t' list, we have filtert₂ ok f s_0 xs $\equiv_{NS}$ filtert₃ ok f s_0 xs

The proof, given below, is rather simple. The key property is

```
if (⩤xs.false) ⊙ scanM f s xs then x::xs else fail ≡NS fail
```

which is the immediate corollary of Lemma 11.

Expanding all definitions, the statement of Thm. 3 reads as follows

```
put s₀; (let ys = foldr g (fn ().[]) xs ⊛ () in if all ok ys then xs else fail)
≡NS put s₀; (foldr g' (fn ().[]) xs ⊛ ())
where
 g ≔ ⩤x.⩤u.fn ().
   let s = get in (f s x::) ⊙ (put (f s x); u ⊛ ())
 g' ≔ ⩤x.⩤u.fn ().
   let s = get in
    if ok (f s  x) then (put (f s  x); ((x::) ⊙ (u ⊛ ()))) else fail
```

from which follows that it is enough to prove

```
let ys = foldr g (fn ().[]) xs ⊛ () in if all ok ys then xs else fail
≡NS foldr g' (fn ().[]) xs ⊛ ())
```

Proof
The proof is by induction on the list xs. We elide the trivial base case. For the inductive case x::xs we have:

```
let ys = foldr g (fn ().[]) (x::xs) ⊛ () in
  if all ok ys then (x::xs) else fail
```

```
≡NS { expanding foldr }
let ys = let s = get in
   (f s x::) ⊙ (put (f s x); foldr g (fn ().[]) xs ⊛ ()) in
   if all ok ys then (x::xs) else fail
≡NS { associativity, lifting compositions }
 let s = get in
 let ys' = (put (f s x); foldr g (fn ().[]) xs ⊛ ()) in
   if all ok (f s x::ys') then (x::xs) else fail
≡NS { definition of all }
 let s = get in
 let ys' = (put (f s x); foldr g (fn ().[]) xs ⊛ ()) in
   if ok (f s x) && all ok ys' then (x::xs) else fail
≡NS { case analysis }
 let s = get in
 if ok (f s x) then
    let ys' = (put (f s x); foldr g (fn ().[]) xs ⊛ ()) in
     if all ok ys' then (x::xs) else fail
  else
   let ys' = (put (f s x); foldr g (fn ().[]) xs ⊛ ()) in
   fail
≡NS { Lemma 11 }
 let s = get in
 if ok (f s x) then
    let ys' = (put (f s x); foldr g (fn ().[]) xs ⊛ ()) in
     if all ok ys' then (x::xs) else fail
  else fail
≡NS { associativity, lifting compositions }
 let s = get in
 if ok (f s x) then
    put (f s x);
    (x::) ⊙ (let ys' = foldr g (fn ().[]) xs ⊛ () in if all ok ys' then xs else fail)
  else fail
≡NS { inductive hypothesis }
 let s = get in
 if ok (f s x) then
    put (f s x); ((x::) ⊙ foldr g' (fn ().[]) xs ⊛ ())
  else fail
≡NS { definition of foldr and g' }
foldr g' (fn ().[]) (x::xs) ⊛ ())
```

□