Counting Successes: Effects and Transformations for Non-Deterministic Programs

Nick Benton[1], Andrew Kennedy[2], Martin Hofmann[3], and Vivek Nigam[4]

[1] Microsoft Research, Cambridge, UK
[2] Facebook, London, UK
[3] Ludwig-Maximilians-Universität, München, Germany
[4] UFPB, Joao Pessoa, Brazil

Abstract. We give a simple effect system for non-deterministic programs, tracking static approximations to the number of results that may be produced by each computation. A relational semantics for the effect system establishes the soundness of both the analysis and its use in effect-based program transformations.

Dedicated to Philip Wadler on the occasion of his 60^{th} *birthday.*

1 Introduction

Back in 1998, Wadler showed [30, 31] how type-and-effect systems, a form of static analysis for impure programs that was first introduced by Gifford and Lucassen [15, 21], may be re-presented, both syntactically and algorithmically, in terms of a variant of Moggi's computational metalanguage [22] in which the computation type constructor is refined by annotations that delimit the possible effects of computations. The same year, Tolmach described the use of a hierarchy of monadic types in optimizing compilation [26] and, in the same conference as Wadler's paper, two of us presented an optimizing compiler for Standard ML that used the same kind of refined monadic metalanguage as its intermediate language, inferring state, exception and divergence effects to enable optimizing transformations [9].

"That's all very well in practice," we thought, "but how does it work out in theory?" But devising a satisfactory semantics for effect-refined types that both interprets them as properties of the original, un-refined terms, and validates program transformations predicated on effect information proved surprisingly tricky, until we adopted another Wadleresque idea [29]: the relational interpretation of types. Interpreting static analyses in terms of binary relations, rather than unary predicates, deals naturally with independence properties (such as secure information flow or not reading parts of the store), is naturally extensional (by contrast with, say, instrumenting the semantics with a trace of side-effecting operations), and accounts for the soundness of program transformations at the same time as soundness of the analysis [2]. We have studied a series of effect systems, of ever-increasing sophistication, using relations, concentrating mainly on tracking uses of mutable state [8, 7, 4].

Here we consider a different effect: non-determinism. Wadler studied non-determinism in a famous thirty-year-old paper on how lazy lists can be used to program exception handling, backtracking and pattern matching in pure functional languages [28], and returned to it in his work on query languages [23]. That initial paper draws a distinction between two cases. The first is the use of lists to encode errors, or exceptions, where computations either fail, represented by the empty list, or succeed, returning a singleton list. The second is more general backtracking, encoding the kind of search found in logic programming languages, where computations can return many results. This paper is in the spirit of formalizing that distinction. We refine a non-determinism monad with effect annotations that approximate how many (different) results may be returned by each computation, and give a semantics that validates transformations that depend on that information. To keep everything as simple as possible, we work with a total language and a semantics that uses powersets, rather than lists or multisets, so we do not observe the order or multiplicity of results. The basic ideas could, however, easily be adapted to a language with recursion or a semantics with lists instead of sets.

2 Effects for Non-Determinism

2.1 Base Language

We consider a monadically-typed, normalizing, call-by-value lambda calculus with operations for failure and non-deterministic choice. A more conventionally-typed impure calculus may be translated into the monadic one via the usual 'call-by-value translation' [6], and this extends to the usual style of presenting effect systems in which every judgement has an effect, and function arrows are annotated with 'latent effects' [31].

We define value types A, computation types TA and contexts Γ as follows:

$$A, B := \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid A \times B \mid A \to TB$$
$$\Gamma := x_1 : A_1, \ldots, x_n : A_n$$

Value judgements, $\Gamma \vdash V : A$, and computation judgements, $\Gamma \vdash M : TA$, are defined by the rules in Figure 1. The presence of types on lambda-bound variables makes typing derivations unique, and addition and comparison should be considered just representative primitive operations.

Our simple language has an elementary denotational semantics in the category of sets and functions. The semantics of types is as follows:

$$[\![\texttt{unit}]\!] = 1 \qquad [\![\texttt{int}]\!] = \mathbb{Z} \qquad [\![\texttt{bool}]\!] = \mathbb{B} \qquad [\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$$

$$[\![A \to TB]\!] = [\![A]\!] \to [\![TB]\!] \qquad [\![TA]\!] = \mathbb{P}_{\mathit{fin}}([\![A]\!])$$

$$\frac{}{\Gamma \vdash n : \mathtt{int}} \qquad \frac{}{\Gamma \vdash b : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash () : \mathtt{unit}} \qquad \frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma \vdash V_1 : \mathtt{int} \quad \Gamma \vdash V_2 : \mathtt{int}}{\Gamma \vdash V_1 + V_2 : \mathtt{int}} \qquad \frac{\Gamma \vdash V_1 : \mathtt{int} \quad \Gamma \vdash V_2 : \mathtt{int}}{\Gamma \vdash V_1 > V_2 : \mathtt{bool}}$$

$$\frac{\Gamma \vdash V_1 : A \quad \Gamma \vdash V_2 : B}{\Gamma \vdash (V_1, V_2) : A \times B} \qquad \frac{\Gamma \vdash V : A_1 \times A_2}{\Gamma \vdash \pi_i\, V : A_i}$$

$$\frac{\Gamma, x : A \vdash M : TB}{\Gamma \vdash \lambda x : A.M : A \to TB} \qquad \frac{\Gamma \vdash V_1 : A \to TB \quad \Gamma \vdash V_2 : A}{\Gamma \vdash V_1\, V_2 : TB} \qquad \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathtt{val}\, V : TA}$$

$$\frac{\Gamma \vdash M : TA \quad \Gamma, x : A \vdash N : TB}{\Gamma \vdash \mathtt{let}\, x \Leftarrow M\, \mathtt{in}\, N : TB} \qquad \frac{\Gamma \vdash V : \mathtt{bool} \quad \Gamma \vdash M : TA \quad \Gamma \vdash N : TA}{\Gamma \vdash \mathtt{if}\ V\ \mathtt{then}\ M\ \mathtt{else}\ N : TA}$$

$$\frac{}{\Gamma \vdash \mathtt{fail} : TA} \qquad \frac{\Gamma \vdash M_1 : TA \quad \Gamma \vdash M_2 : TA}{\Gamma \vdash M_1\, \mathtt{or}\, M_2 : TA}$$

Fig. 1. Simple computation type system

The interpretation of the computation type constructor is the finite powerset monad. The meaning of contexts is given by $[\![x_1 : A_1, \ldots, x_n : A_n]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$, and we can then give the semantics of judgements

$$[\![\Gamma \vdash V : A]\!] : [\![\Gamma]\!] \to [\![A]\!] \qquad \text{and} \qquad [\![\Gamma \vdash M : TA]\!] : [\![\Gamma]\!] \to [\![TA]\!]$$

inductively in a standard way. The interesting cases are

$$\begin{aligned}
[\![\Gamma \vdash \mathtt{val}\, V : TA]\!]\, \rho &= \{[\![\Gamma \vdash V : A]\!]\, \rho\} \\
[\![\Gamma \vdash \mathtt{let}\, x \Leftarrow M\, \mathtt{in}\, N]\!]\, \rho &= \textstyle\bigcup_{v \in [\![\Gamma \vdash M : A]\!]\, \rho} [\![\Gamma, x : A \vdash N : TB]\!]\, (\rho, v) \\
[\![\Gamma \vdash \mathtt{fail} : TA]\!]\, \rho &= \emptyset \\
[\![\Gamma \vdash M_1\, \mathtt{or}\, M_2 : TA]\!]\, \rho &= ([\![\Gamma \vdash M_1 : TA]\!]\, \rho) \cup ([\![\Gamma \vdash M_1 : TA]\!]\, \rho)
\end{aligned}$$

So, for example

$$\begin{aligned}
&[\![\vdash \mathtt{let}\, f \Leftarrow \mathtt{val}\, (\lambda x : \mathtt{int}.\mathtt{if}\ x < 6\ \mathtt{then}\ \mathtt{val}\, x\ \mathtt{else}\ \mathtt{fail})\ \mathtt{in} \\
&\quad \mathtt{let}\, x \Leftarrow \mathtt{val}\, 1\, \mathtt{or}\, \mathtt{val}\, 2\, \mathtt{in}\, \mathtt{let}\, y \Leftarrow \mathtt{val}\, 3\, \mathtt{or}\, \mathtt{val}\, 4\, \mathtt{in}\, f(x + y) : T\mathtt{int}]\!] = \{4, 5\}
\end{aligned}$$

The semantics is adequate for the obvious operational semantics and a contextual equivalence observing, say, the set of unit values produced by a closed program.

2.2 Effect System

We now present an effect analysis that refines the simple type system by annotating the computation type constructor with information about *how many*

$$\frac{}{X \leq X} \qquad \frac{X \leq Y \quad Y \leq Z}{X \leq Z} \qquad \frac{X \leq X' \quad Y \leq Y'}{X \times Y \leq X' \times Y'}$$

$$\frac{X' \leq X \quad T_\varepsilon Y \leq T_{\varepsilon'} Y'}{(X \to T_\varepsilon Y) \leq (X' \to T_{\varepsilon'} Y')} \qquad \frac{\varepsilon \leq \varepsilon' \quad X \leq X'}{T_\varepsilon X \leq T_{\varepsilon'} X'}$$

Fig. 2. Subtyping refined types

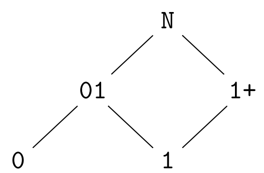
results a computation may produce. Formally, define *refined* value types X, computation types $T_\varepsilon X$ and contexts Θ by

$$\begin{aligned} X, Y &:= \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid X \times Y \mid X \to T_\varepsilon Y \\ \varepsilon &\in \{\texttt{0}, \texttt{1}, \texttt{01}, \texttt{1+}, \texttt{N}\} \\ \Theta &:= x_1 : X_1, \ldots, x_n : X_n \end{aligned}$$

A computation of type $T_0 X$ will always fail, i.e. produce zero results. One of type $T_1 X$ is deterministic, i.e. produces exactly one result. More generally, writing $|S|$ for the cardinality of a finite set S, a computation of type $T_\varepsilon X$ can only produce sets of results S such that $|S| \in [\![\varepsilon]\!]$, where $[\![\varepsilon]\!] \subseteq \mathbb{N}$:

$$\begin{aligned} [\![\texttt{0}]\!] &= \{0\} & [\![\texttt{1+}]\!] &= \{n \mid n \geq 1\} \\ [\![\texttt{1}]\!] &= \{1\} & [\![\texttt{N}]\!] &= \mathbb{N} \\ [\![\texttt{01}]\!] &= \{0, 1\} \end{aligned}$$

There is an obvious order on effect annotations, given by $\varepsilon \leq \varepsilon' \iff [\![\varepsilon]\!] \subseteq [\![\varepsilon']\!]$:



This order induces a subtyping relation on refined types, which is axiomatised in Figure 2. The refined type assignment system is shown in Figure 3. The erasure map, $U(\cdot)$, takes refined types to simple ones by forgetting the effect annotations:

$$\begin{gathered} U(\texttt{int}) = \texttt{int} \qquad U(\texttt{bool}) = \texttt{bool} \qquad U(\texttt{unit}) = \texttt{unit} \\ U(X \times Y) = U(X) \times U(Y) \\ U(X \to T_\varepsilon Y) = U(X) \to U(T_\varepsilon Y) \\ U(T_\varepsilon X) = T(U(X)) \end{gathered}$$

$$U(x_1 : X_1, \ldots, x_n : X_n) = x_1 : U(X_1), \ldots, x_n : U(X_n)$$

Lemma 1. *If $X \leq Y$ then $U(X) = U(Y)$, and similarly for computations.* □

$$\frac{}{\Theta \vdash n : \texttt{int}} \qquad \frac{}{\Theta \vdash b : \texttt{bool}} \qquad \frac{}{\Theta \vdash () : \texttt{unit}} \qquad \frac{}{\Theta, x : X \vdash x : X}$$

$$\frac{\Theta \vdash V_1 : \texttt{int} \quad \Theta \vdash V_2 : \texttt{int}}{\Theta \vdash V_1 + V_2 : \texttt{int}} \qquad \frac{\Theta \vdash V_1 : \texttt{int} \quad \Theta \vdash V_2 : \texttt{int}}{\Theta \vdash V_1 > V_2 : \texttt{bool}}$$

$$\frac{\Theta \vdash V_1 : X \quad \Theta \vdash V_2 : Y}{\Theta \vdash (V_1, V_2) : X \times Y} \qquad \frac{\Theta \vdash V : X_1 \times X_2}{\Theta \vdash \pi_i \, V : X_i} \qquad \frac{\Theta, x : X \vdash M : T_\varepsilon Y}{\Theta \vdash \lambda x : U(X).M : X \to T_\varepsilon Y}$$

$$\frac{\Theta \vdash V_1 : X \to T_\varepsilon Y \quad \Theta \vdash V_2 : X}{\Theta \vdash V_1 \, V_2 : T_\varepsilon Y} \qquad \frac{\Theta \vdash V : X}{\Theta \vdash \texttt{val}\, V : T_1 X}$$

$$\frac{\Theta \vdash M : T_\varepsilon X \quad \Theta, x : X \vdash N : T_{\varepsilon'} Y}{\Theta \vdash \texttt{let}\, x \Leftarrow M \,\texttt{in}\, N : T_{\varepsilon \cdot \varepsilon'} Y} \qquad \frac{\Theta \vdash V : \texttt{bool} \quad \Theta \vdash M : T_\varepsilon X \quad \Theta \vdash N : T_\varepsilon X}{\Theta \vdash \texttt{if}\, V \,\texttt{then}\, M \,\texttt{else}\, N : T_\varepsilon X}$$

$$\frac{}{\Theta \vdash \texttt{fail} : T_0 X} \qquad \frac{\Theta \vdash M_1 : T_{\varepsilon_1} X \quad \Theta \vdash M_2 : T_{\varepsilon_2} X}{\Theta \vdash M_1 \,\texttt{or}\, M_2 : T_{\varepsilon_1 + \varepsilon_2} X}$$

$$\frac{\Theta \vdash V : X \quad X \leq X'}{\Theta \vdash V : X'} \qquad \frac{\Theta \vdash M : T_\varepsilon X \quad T_\varepsilon X \leq T_{\varepsilon'} X'}{\Theta \vdash M : T_{\varepsilon'} X'}$$

Fig. 3. Refined type system

The use of erasure on bound variables means that the subject terms of the refined type system are the same as those of the unrefined one.

Lemma 2. *If $\Theta \vdash V : X$ then $U(\Theta) \vdash V : U(X)$, and similarly for computations.* □

It is also the case that the refined system does not rule out any terms from the original language. Let $G(\cdot)$ be the map from simple types to refined types that adds the 'top' effect `N` to all computation types, and then

Lemma 3. *If $\Gamma \vdash V : A$ then $G(\Gamma) \vdash V : G(A)$ and similarly for computations.* □

The interesting aspect of the refined type system is the use it makes of abstract multiplication (in the `let`-rule) and addition (in the `or` rule) operations on effects. The definitions are:

·	0	1	01	1+	N
0	0	0	0	0	0
1	0	1	01	1+	N
01	0	01	01	N	N
1+	0	1+	N	1+	N
N	0	N	N	N	N

+	0	1	01	1+	N
0	0	1	01	1+	N
1	1	1+	1+	1+	1+
01	01	1+	N	1+	N
1+	1+	1+	1+	1+	1+
N	N	1+	N	1+	N

The operations endow our chosen set of effect annotations with the structure of a commutative semiring with idempotent multiplication.

Lemma 4. *The + operation is associative and commutative, with* `0` *as a unit. The $\cdot$ operation is associative, commutative and idempotent, with* `1` *as unit and* `0` *as zero. We also have the distributive law* $(\varepsilon_1 + \varepsilon_2) \cdot \varepsilon_3 = \varepsilon_1 \cdot \varepsilon_3 + \varepsilon_2 \cdot \varepsilon_3$. □

The correctness statement concerning the abstract operations that we will need later is a consequence of a trivial fact about the cardinality of unions:

$$|A| \ \leq \ |A \cup B| \ \leq |A| + |B|$$

which leads to the following:

Lemma 5. *For any* ε_1,ε_2,

$$\bigcup_{a \in [\![\varepsilon_1]\!],\, b \in [\![\varepsilon_2]\!]} \{n \mid max(a,b) \leq n \ and \ n \leq a+b\} \ \subseteq [\![\varepsilon_1 + \varepsilon_2]\!]$$
$$\bigcup_{a \in [\![\varepsilon_1]\!]} \bigcup_{(b_1,\ldots,b_a) \in [\![\varepsilon_2]\!]^a} \{n \mid \forall i, b_i \leq n \ and \ n \leq \Sigma_i b_i\} \subseteq [\![\varepsilon_1 \cdot \varepsilon_2]\!].$$ □

The intuition behind the awkward-looking correctness condition for multiplication deserves some explanation. Consider how many results may be produced by `let` $x \Leftarrow M$ `in` N when M produces results $x_1, \ldots, x_a$ for some $a \in [\![\varepsilon_1]\!]$ and for each such result, x_i, $N(x_i)$ produces a set of results of size $b_i \in [\![\varepsilon_2]\!]$. Then, by the inequality above, the number n of results of the `let`-expression is bounded below by each of the b_is and above by the sum of the b_is. The set of all possible cardinalities for the `let`-expression is then obtained by unioning the cardinality sets for each possible a and each possible tuple $(b_1, \ldots, b_a)$.

The reader may also be surprised by the asymmetry of the condition for multiplication, given that we observed above that the abstract operation is commutative. But that commutativity is actually an accidental consequence of our particular choice of sets of cardinalities to track. Indeed, if M produces a single result and for each x, $N(x)$ produces exactly two results (a case we do not track here), then `let` $x \Leftarrow M$ `in` N produces two results. Conversely, however, if M produces two results and for each x, $N(x)$ produces a single result, then `let` $x \Leftarrow M$ `in` N can produce either one *or* two distinct results. This case also shows that, in general, `1` will only be left unit for multiplication. Idempotency also fails to hold in general.

We remark that the abstract operations are an example of the Cousots' $\alpha\gamma$ framework for abstract interpretation [12], and were in fact derived using a little list-of-successes ML program that computes with abstractions and concretions.

2.3 Semantics of Effects

The meanings of simple types are just sets, out of which we will carve the meanings of refined types as subsets, *together* with a coarser notion of equality.

We first recall some notation. If R is a (binary) relation on A and Q a relation on B, then we define relations on Cartesian products and function spaces by

$$R \times Q = \{((a,b),(a',b')) \in (A \times B) \times (A \times B) \ \mid (a,a') \in R,\ (b,b') \in Q\}$$
$$R \to Q = \{(f,f') \in (A \to B) \times (A \to B) \ \mid \forall (a,a') \in R.\, (f\, a,\ f'\, a') \in Q\}$$

A binary relation on a set is a *partial equivalence relation* (PER) if it is symmetric and transitive. If R and Q are PERs, so are $R \to Q$ and $R \times Q$. Write Δ_A for the diagonal relation $\{(a,a) \mid a \in A\}$, and $a : R$ for $(a,a) \in R$. If R is a PER on A and $a \in A$ then we define the 'equivalence class' $[a]_R$ to be $\{a' \in A | (a,a') \in R\}$, noting that this is empty unless $a : R$.

We can now define the semantics of each refined type as a partial equivalence relation on the semantics of its erasure as follows:

$$
\begin{aligned}
[\![X]\!] &\subseteq [\![U(X)]\!] \times [\![U(X)]\!] \\
[\![\texttt{int}]\!] = \Delta_{\mathbb{Z}} \qquad [\![\texttt{bool}]\!] &= \Delta_{\mathbb{B}} \qquad [\![\texttt{unit}]\!] = \Delta_1 \\
[\![X \times Y]\!] &= [\![X]\!] \times [\![Y]\!] \\
[\![X \to T_\varepsilon Y]\!] &= [\![X]\!] \to [\![T_\varepsilon Y]\!] \\
[\![T_\varepsilon X]\!] &= \{(S,S') \mid S \sim_X S' \text{ and } |S/[\![X]\!]| \in [\![\varepsilon]\!]\}
\end{aligned}
$$

The key clause is the last one, in which $S \sim_X S'$ means $\forall x \in S, \exists x' \in S', (x,x') \in [\![X]\!]$ and vice versa. The $\sim_X$ relation is a lifting of $[\![X]\!]$ to sets of values; this is a familar, canonical construction that appears, for example, in the definition of bisimulation or of powerdomains. The quotient $S/[\![X]\!]$ is defined to be $\{[x]_{[\![X]\!]} \mid x \in S\}$.[5] We observe that if $S \sim_X S'$ then $S/[\![X]\!] = S'/[\![X]\!]$ and $\emptyset \notin S/[\![X]\!]$.

The way one should understand the clause for computation types is that two sets S,S' are related when they have the same elements *up to* the notion of equivalence associated with the refined type X and, moreover, the cardinality of the sets (again, as sets of Xs, *not* as sets of the underlying type UX) is accurately reflected by ε.

We also extend the relational interpretation of refined types to refined contexts in the natural way:

$$
\begin{aligned}
[\![\Theta]\!] &\subseteq [\![U(\Theta)]\!] \times [\![U(\Theta)]\!] \\
[\![x_1 : X_1, \ldots, x_n : X_n]\!] &= [\![X_1]\!] \times \cdots \times [\![X_n]\!]
\end{aligned}
$$

Lemma 6. *For any Θ, X and ε, all of $[\![\Theta]\!]$, $[\![X]\!]$ and $[\![T_\varepsilon X]\!]$ are partial equivalence relations.* □

The interpretation of a refined type with the top effect annotation everywhere is just equality on the interpretation of its erasure:

Lemma 7. *For all A, $[\![G(A)]\!] = \Delta_{[\![A]\!]}$.* □

The following establishes semantic soundness for our subtyping relation:

Lemma 8. *If $X \leq Y$ then $[\![X]\!] \subseteq [\![Y]\!]$, and similarly for computation types.* □

And we can then show the 'fundamental theorem' that establishes the soundness of the effect system itself:

[5] It is tempting to replace $S \sim_X S'$ by $S/[\![X]\!] = S'/[\![X]\!]$, but $S/[\![X]\!]$ contains the empty set when there is an $x \in S$ with $(x,x) \notin [\![X]\!]$.

Theorem 1.

1. *If* $\Theta \vdash V : X$, $(\rho, \rho') \in [\![\Theta]\!]$ *then*

$$([\![U(\Theta) \vdash V : U(X)]\!]\,\rho,\ [\![U(\Theta) \vdash V : U(X)]\!]\,\rho') \in [\![X]\!]$$

2. *If* $\Theta \vdash M : T_\varepsilon X$, $(\rho, \rho') \in [\![\Theta]\!]$ *then*

$$([\![U(\Theta) \vdash M : T(U(X))]\!]\,\rho, [\![U(\Theta) \vdash M : T(U(X))]\!]\,\rho') \in [\![T_\varepsilon X]\!]$$

Proof. A largely standard induction; we just sketch the interesting cases.

Trivial computations. Let $\Gamma = U(\Theta)$ and $A = U(X)$. Given $(\rho, \rho') \in [\![\Theta]\!]$ we need to show

$$([\![\Gamma \vdash \mathtt{val}\, V : TA]\!]\,\rho,\ [\![\Gamma \vdash \mathtt{val}\, V : TA]\!]\,\rho') \in [\![T_1 X]\!]$$

which means

$$(\{[\![\Gamma \vdash V : A]\!]\rho\},\ \{[\![\Gamma \vdash V : A]\!]\rho'\}) \in \{(S, S') \mid S \sim_X S' \text{ and } |S/[\![X]\!]| = 1\}$$

Induction gives $([\![\Gamma \vdash V : A]\!]\rho, [\![\Gamma \vdash V : A]\!]\rho') \in [\![X]\!]$, which deals with the $\cdot \sim_X \cdot$ condition, and it is clear that $|\{[[\![\Gamma \vdash V : A]\!]]_{[\![X]\!]}\}| = 1$.

Choice. We want firstly that

$$[\![\Gamma \vdash M_1 \,\mathtt{or}\, M_2 : TA]\!]\rho \sim_X [\![\Gamma \vdash M_1 \,\mathtt{or}\, M_2 : TA]\!]\rho'$$

which is

$$[\![\Gamma \vdash M_1 : TA]\!]\rho \cup [\![\Gamma \vdash M_2 : TA]\!]\rho \sim_X [\![\Gamma \vdash M_1 : TA]\!]\rho' \cup [\![\Gamma \vdash M_2 : TA]\!]\rho'$$

Induction gives $[\![\Gamma \vdash M_1 : TA]\!]\rho \sim_X [\![\Gamma \vdash M_1 : TA]\!]\rho'$ and similarly for M_2, from which the result is immediate. Secondly, we want

$$|[\![\Gamma \vdash M_1 \,\mathtt{or}\, M_2 : TA]\!]\rho\ /\ [\![X]\!]| \in [\![\varepsilon_1 + \varepsilon_2]\!]$$

and because quotient distributes over union, this is

$$|[\![\Gamma \vdash M_1 : TA]\!]\rho\,/[\![X]\!]\ \cup\ [\![\Gamma \vdash M_2 : TA]\!]\rho\,/[\![X]\!]| \in [\![\varepsilon_1 + \varepsilon_2]\!]$$

By induction, $|[\![\Gamma \vdash M_1 : TA]\!]\rho\,/[\![X]\!]| \in [\![\varepsilon_1]\!]$, and similarly for M_2, so we are done by Lemma 5.

Sequencing. Pick $y \in [\![\Gamma \vdash \mathtt{let}\, x \Leftarrow M \,\mathtt{in}\, N : TB]\!]\rho$. By the semantics of `let`, there's an $x \in [\![\Gamma \vdash M : TA]\!]\rho$ such that $y \in [\![\Gamma, x : A \vdash N : TB]\!](\rho, x)$. By induction on M, there's an $x' \in [\![\Gamma \vdash M : TA]\!]\rho'$ such that $(x, x') \in [\![X]\!]$. So by induction on N, $[\![\Gamma, x : A \vdash N : TB]\!](\rho, x) \sim_Y [\![\Gamma, x : A \vdash N : TB]\!](\rho', x')$, and therefore $\exists y' \in [\![\Gamma, x : A \vdash N : TB]\!](\rho', x')$ with $(y, y') \in [\![Y]\!]$. Then as $y' \in [\![\Gamma \vdash \mathtt{let}\, x \Leftarrow M \,\mathtt{in}\, N : TB]\!]\rho'$, we are done.

For the cardinality part, note that

$$\begin{aligned}&\left(\bigcup_{x\in[\![\Gamma\vdash M:TA]\!]\rho}[\![\Gamma,x:A\vdash N:TB]\!](\rho,x)\right)/[\![Y]\!]\\ &=\bigcup_{x\in[\![\Gamma\vdash M:TA]\!]\rho}([\![\Gamma,x:A\vdash N:TB]\!](\rho,x)/[\![Y]\!])\\ &=\bigcup_{[x]\in[\![\Gamma\vdash M:TA]\!]\rho/[\![X]\!]}([\![\Gamma,x:A\vdash N:TB]\!](\rho,x)/[\![Y]\!])\end{aligned}$$

and then since, by induction, $|\Gamma \vdash M : TA]\!]\rho/[\![X]\!]| \in [\![\varepsilon_1]\!]$, and also for any $[x] \in [\![\Gamma \vdash M : TA]\!]\rho/[\![X]\!]$,

$$|[\![\Gamma, x : A \vdash N : TB]\!](\rho, x)/[\![Y]\!]| \in [\![\varepsilon_2]\!]$$

we are done by Lemma 5. □

2.4 Basic Equations

The semantics validates all the generic equations of the computational metalanguage: congruence laws, β and η laws for products, function spaces, booleans and computation types. We show some of these rules in Figure 4. The powerset monad also validates a number of more specific equations that hold without restrictions on the involved effects. These are shown in Figure 5: choice is associative, commutative and idempotent with `fail` as a unit, the monad is commutative, and choice and failure distribute over `let`.

The correctness of the basic congruence laws subsumes Theorem 1. Note that, slightly subtly, the reflexivity PER rule is invertible. This is sound because our effect annotations are purely descriptive (*Curry-style*, or *extrinsic* in Reynolds's terminology [24]) whereas the simple types are more conventionally prescriptive (*Church-style*, which Reynolds calls *intrinsic*). We actually regard the rules of Figure 3 as abbreviations for a subset of the equational judgements of Figure 4; thus we can allow the refined type of the conclusion of interesting equational rules to be different from (in particular, have a smaller effect than) the rules in Figure 3 would assign to one side. This shows up already: most of the rules in Figure 5 are type correct in simple syntactic sense as a consequence of Lemma 4. But the idempotency rule for choice is not, because the abstract addition is, rightly, not idempotent. The idempotency law effectively extends the refined type system with a rule saying that if M has type $T_\varepsilon X$, so does M `or` M.

In practical terms, having equivalences also improve typing allows inferred effects to be improved locally as transformations are performed, rather than requiring periodic reanalysis of the whole program to obtain the best results.

3 Using Effect Information

More interesting equivalences are predicated on the effect information. We present these in Figure 6.

The **Fail** transformation allows any computation with the `0` effect, i.e. that produces no results, to be replaced with `fail`.

PER rules (+ similar for computations):

$$\frac{\Theta \vdash V : X}{\Theta \vdash V = V : X} \qquad \frac{\Theta \vdash V = V' : X}{\Theta \vdash V' = V : X} \qquad \frac{\Theta \vdash V = V' : X \quad \Theta \vdash V' = V'' : X}{\Theta \vdash V = V'' : X}$$

$$\frac{\Theta \vdash V = V' : X \quad X \leq X'}{\Theta \vdash V = V' : X'}$$

Congruence rules (extract):

$$\frac{\Theta \vdash V_1 = V_1' : \mathtt{int} \quad \Theta \vdash V_2 = V_2' : \mathtt{int}}{\Theta \vdash (V_1 + V_2) = (V_1' + V_2') : \mathtt{int}} \qquad \frac{\Theta \vdash V = V' : X_1 \times X_2}{\Theta \vdash \pi_i\, V \;=\; \pi_i\, V' : X_i}$$

$$\frac{\Theta, x : X \vdash M = M' : T_\varepsilon Y}{\Theta \vdash (\lambda x : U(X).M) = (\lambda x : U(X).M') : X \to T_\varepsilon Y}$$

β rules (extract):

$$\frac{\Theta, x : X \vdash M : T_\varepsilon Y \quad \Theta \vdash V : X}{\Theta \vdash (\lambda x : U(X).M)\, V \;=\; M[V/x] : T_\varepsilon Y} \qquad \frac{\Theta \vdash V : X \quad \Theta, x : X \vdash M : T_\varepsilon Y}{\Theta \vdash \mathtt{let}\, x \Leftarrow \mathtt{val}\, V\, \mathtt{in}\, M = M[V/x] : T_\varepsilon Y}$$

η rules (extract):

$$\frac{\Theta \vdash V : X \to T_\varepsilon Y}{\Theta \vdash V = (\lambda x : U(X).V\, x) : X \to T_\varepsilon Y} \qquad \frac{\Theta \vdash M : T_\varepsilon X}{\Theta \vdash (\mathtt{let}\, x \Leftarrow M\, \mathtt{in}\, \mathtt{val}\, x) = M : T_\varepsilon X}$$

Commuting conversions:

$$\frac{\Theta \vdash M : T_{\varepsilon_1} Y \quad \Theta, y : Y \vdash N : T_{\varepsilon_2} X \quad \Theta, x : X \vdash P : T_{\varepsilon_3} Z}{\Theta \vdash \mathtt{let}\, x \Leftarrow (\mathtt{let}\, y \Leftarrow M\, \mathtt{in}\, N)\, \mathtt{in}\, P = \mathtt{let}\, y \Leftarrow M\, \mathtt{in}\, \mathtt{let}\, x \Leftarrow N\, \mathtt{in}\, P : T_{\varepsilon_1 \cdot \varepsilon_2 \cdot \varepsilon_3} Z}$$

Fig. 4. Monad-independent equivalences

The **Dead Computation** transformation allows the removal of a computation, M, whose value is unused, provided the effect of M indicates that it always produces at least one result. If M can fail then its removal is generally unsound, as that could transform a failing computation into one that succeeds.

The **Duplicated Computation** transformation allows two evaluations of a computation M to be replaced by one, provided that M produces at most one result. This is, of course, generally unsound, as, for example,

$$\begin{array}{l} \mathtt{let}\, x \Leftarrow \mathtt{val}\, 1\, \mathtt{or}\, \mathtt{val}\, 2\, \mathtt{in}\, \mathtt{let}\, y \Leftarrow \mathtt{val}\, 1\, \mathtt{or}\, \mathtt{val}\, 2\, \mathtt{in}\, \mathtt{val}\, (x+y) \\ \neq \mathtt{let}\, x \Leftarrow \mathtt{val}\, 1\, \mathtt{or}\, \mathtt{val}\, 2\, \mathtt{in}\, \mathtt{val}\, (x+x). \end{array}$$

The **Pure Lambda Hoist** transformation allows a computation to be hoisted out of a lambda abstraction, so it is performed once, rather than every time the

Choice:

$$\frac{\Theta \vdash M_1 : T_{\varepsilon_1} X \quad \Theta \vdash M_2 : T_{\varepsilon_2} X}{\Theta \vdash M_1 \,\texttt{or}\, M_2 = M_2 \,\texttt{or}\, M_1 : T_{\varepsilon_1+\varepsilon_2} X}$$

$$\frac{\Theta \vdash M : T_\varepsilon X}{\Theta \vdash M \,\texttt{or}\, M = M : T_\varepsilon X} \qquad \frac{\Theta \vdash M : T_\varepsilon X}{\Theta \vdash M \,\texttt{or}\, \texttt{fail} = M : T_\varepsilon X}$$

$$\frac{\Theta \vdash M_1 : T_{\varepsilon_1} X \quad \Theta \vdash M_2 : T_{\varepsilon_2} X \quad \Theta \vdash M_3 : T_{\varepsilon_3} X}{\Theta \vdash M_1 \,\texttt{or}\, (M_2 \,\texttt{or}\, M_3) = (M_1 \,\texttt{or}\, M_2) \,\texttt{or}\, M_3 : T_{\varepsilon_1+\varepsilon_2+\varepsilon_3} X}$$

Commutativity:

$$\frac{\Theta \vdash M : T_{\varepsilon_1} Y \quad \Theta \vdash N : T_{\varepsilon_2} X \quad \Theta, x : X, y : Y \vdash P : T_{\varepsilon_3} Z}{\Theta \vdash \texttt{let}\, x \Leftarrow M \,\texttt{in}\, \texttt{let}\, y \Leftarrow N \,\texttt{in}\, P = \texttt{let}\, y \Leftarrow N \,\texttt{in}\, \texttt{let}\, x \Leftarrow M \,\texttt{in}\, P : T_{\varepsilon_1 \cdot \varepsilon_2 \cdot \varepsilon_3} Z}$$

Distribution:

$$\frac{\Theta \vdash M_1 : T_{\varepsilon_1} X \quad \Theta \vdash M_2 : T_{\varepsilon_2} X \quad \Theta, x : X \vdash N : T_{\varepsilon_3} Y}{\Theta \vdash \texttt{let}\, x \Leftarrow (M_1 \,\texttt{or}\, M_2) \,\texttt{in}\, N = (\texttt{let}\, x \Leftarrow M_1 \,\texttt{in}\, N) \,\texttt{or}\, (\texttt{let}\, x \Leftarrow M_2 \,\texttt{in}\, N) : T_{(\varepsilon_1+\varepsilon_2)\cdot\varepsilon_3} Y}$$

$$\frac{\Theta, x : X \vdash N : T_\varepsilon Y}{\Theta \vdash \texttt{let}\, x \Leftarrow \texttt{fail} \,\texttt{in}\, N = \texttt{fail} : T_0 Y}$$

$$\frac{\Theta \vdash M : T_{\varepsilon_3} X \quad \Theta, x : X \vdash N_1 : T_{\varepsilon_1} Y \quad \Theta, x : X \vdash N_2 : T_{\varepsilon_2} Y}{\Theta \vdash \texttt{let}\, x \Leftarrow M \,\texttt{in}\, (N_1 \,\texttt{or}\, N_2) = (\texttt{let}\, x \Leftarrow M \,\texttt{in}\, N_1) \,\texttt{or}\, (\texttt{let}\, x \Leftarrow M \,\texttt{in}\, N_2) : T_{\varepsilon_3\cdot(\varepsilon_1+\varepsilon_2)} Y}$$

$$\frac{\Theta \vdash M : T_\varepsilon X}{\Theta \vdash \texttt{let}\, x \Leftarrow M \,\texttt{in}\, \texttt{fail} = \texttt{fail} : T_0 Y}$$

Fig. 5. Monad-specific, effect-independent equivalences

function is applied, provided that it returns exactly one result (and, of course, that it does not depend on the function argument).

Theorem 2. *All of the equations shown in Figures 4, 5, and 6 are soundly modelled in the semantics:*

- *If* $\Theta \vdash V = V' : X$ *then* $\Theta \models V = V' : X$.
- *If* $\Theta \vdash M = M' : T_\varepsilon X$ *then* $\Theta \models M = M' : T_\varepsilon X$.

Proof. We present proofs for the equivalences in Figure 6.

Dead computation. If we let $\Gamma = U(\Theta)$, $A = U(X)$ and $B = U(Y)$ and $(\rho, \rho') \in [\![\Theta]\!]$ then we have to show

$$([\![\Gamma \vdash \texttt{let}\, x \Leftarrow M \,\texttt{in}\, N : TB]\!]\, \rho,\ [\![\Gamma \vdash N : TB]\!]\, \rho') \in [\![T_\varepsilon Y]\!]$$

Fail:

$$\frac{\Theta \vdash M : T_0 X}{\Theta \vdash M = \texttt{fail} : T_0 X}$$

Dead Computation:

$$\frac{\Theta \vdash M : T_{1+} X \quad \Theta \vdash N : T_\varepsilon Y}{\Theta \vdash \texttt{let}\, x \Leftarrow M \,\texttt{in}\, N = N : T_\varepsilon Y}$$

Duplicated Computation:

$$\frac{\Theta \vdash M : T_{01} X \quad \Theta, x : X, y : X \vdash N : T_\varepsilon Y}{\Theta \vdash \begin{array}{l} \texttt{let}\, x \Leftarrow M \,\texttt{in let}\, y \Leftarrow M \,\texttt{in}\, N \\ = \texttt{let}\, x \Leftarrow M \,\texttt{in}\, N[x/y] \end{array} : T_{01\cdot\varepsilon} Y}$$

Pure Lambda Hoist:

$$\frac{\Theta \vdash M : T_1 Z \quad \Theta, x : X, y : Z \vdash N : T_\varepsilon Y}{\Theta \vdash \begin{array}{l} \texttt{val}\,(\lambda x : U(X).\texttt{let}\, y \Leftarrow M \,\texttt{in}\, N) \\ = \texttt{let}\, y \Leftarrow M \,\texttt{in val}\,(\lambda x : U(X).N) \end{array} : T_1(X \to T_\varepsilon Y)}$$

Fig. 6. Effect-dependent equivalences

which is

$$\begin{array}{l} \bigcup_{x \in [\![\Gamma \vdash M : TA]\!]\rho} [\![\Gamma, x : A \vdash N : TB]\!](\rho, x) \sim_Y [\![\Gamma \vdash N : TB]\!]\rho' \quad \text{and} \\ \left| \bigcup_{x \in [\![\Gamma \vdash M : TA]\!]\rho} [\![\Gamma, x : A \vdash N : TB]\!](\rho, x) \,/\, [\![Y]\!] \right| \in [\![\varepsilon]\!]. \end{array}$$

Since for any x, $[\![\Gamma, x : A \vdash N : TB]\!](\rho, x) = [\![\Gamma \vdash N : TB]\!]\rho$, and induction on M tells us that $|[\![\Gamma \vdash M : TA]\!]\rho/[\![X]\!]| > 0$, so $|[\![\Gamma \vdash M : TA]\!]\rho| > 0$, that's just

$$[\![\Gamma \vdash N : TB]\!]\rho \sim_Y [\![\Gamma \vdash N : TB]\!]\rho' \quad \text{and} \quad |[\![\Gamma \vdash N : TB]\!]\rho/[\![Y]\!]| \in [\![\varepsilon]\!]$$

which is immediate by induction on N.

Duplicated computation. Let $\Gamma = U(\Theta)$, $A = U(X)$, $B = U(Y)$ and $(\rho, \rho') \in [\![\Theta]\!]$. We want (eliding contexts and types in semantic brackets to reduce clutter)

$$\begin{array}{l} \bigcup_{x \in [\![M]\!]\rho} \bigcup_{y \in [\![M]\!]\rho} [\![N]\!](\rho, x, y) \sim_Y \bigcup_{x' \in [\![M]\!]\rho'} [\![N[x/y]]\!](\rho', x') \\ \text{and} \left| \bigcup_{x \in [\![M]\!]\rho} \bigcup_{y \in [\![M]\!]\rho} [\![N]\!](\rho, x, y) \,/\, [\![Y]\!] \right| \in [\![\texttt{01} \cdot \varepsilon]\!] \end{array}$$

Let $a = |[\![M]\!]\rho/[\![X]\!]|$. By induction, $a \in [\![\texttt{01}]\!]$. If $a = 0$ then we must have $[\![M]\!]\rho = \emptyset$ and (also by induction) $[\![M]\!]\rho' = \emptyset$, so the first clause above is satisfied. For the second, we just have to check that $0 \in [\![\texttt{01} \cdot \varepsilon]\!]$ for any ε, which is true.

If $a = 1$ we can pick any $x \in [\![M]\!]\rho$ and $x' \in [\![M]\!]\rho'$ and know $\forall y \in [\![M]\!]\rho, (x, y) \in [\![X]\!]$ as well as $\forall y' \in [\![M]\!]\rho', (x, y') \in [\![X]\!]$. Then by induction on N and the fact that $S \sim_Y S'$ implies $S \cup S' \sim_Y S$ we have

$$\begin{aligned}\textstyle\bigcup_{x\in[\![M]\!]\rho}\bigcup_{y\in[\![M]\!]\rho}[\![N]\!](\rho,x,y) &\sim_Y [\![N]\!](\rho,x,x)\\ &\sim_Y [\![N]\!](\rho',x',x')\\ &\sim_Y \textstyle\bigcup_{x'\in[\![M]\!]\rho'}[\![N]\!](\rho',x',x')\\ &= \textstyle\bigcup_{x'\in[\![M]\!]\rho'}[\![N[x/y]]\!](\rho',x')\end{aligned}$$

For the second part, we get $|[\![N]\!](\rho,x,x) \,/\, [\![Y]\!]| \in [\![\varepsilon]\!]$ by induction, and we then just need to know that $[\![\varepsilon]\!] \subseteq [\![\mathtt{01}\cdot\varepsilon]\!]$, which is easily checked.

Pure lambda hoist. Define $\Gamma = U(\Theta)$, $A = U(X)$, $B = U(Y)$, $C = U(Z)$ and pick $(\rho,\rho') \in [\![\Theta]\!]$. We need

$$\begin{aligned}&\Big(\Big\{\lambda x\in[\![A]\!].\textstyle\bigcup_{z\in[\![M]\!]\rho}[\![N]\!](\rho,x,z)\Big\}, \textstyle\bigcup_{z\in[\![M]\!]\rho'}\{\lambda x\in[\![A]\!].[\![N]\!](\rho',x,z)\}\Big)\\ &\quad\in [\![T_1(X\to T_\varepsilon Y)]\!]\end{aligned}$$

Since the first component of the pair above is a singleton, the cardinality constraint associated with the outer computation type is easily satisfied. For the $\sim$ part, we look at typical elements of the first and second components above. By induction on M, we can pick $z' \in [\![M]\!]\rho'$ and we claim that for any such z',

$$\Big(\lambda x\in[\![A]\!].\textstyle\bigcup_{z\in[\![M]\!]\rho}[\![N]\!](\rho,x,z),\ \lambda x\in[\![A]\!].[\![N]\!](\rho',x,z')\Big) \in [\![X\to T_\varepsilon Y]\!]$$

which will suffice. So assume $(x,x') \in [\![X]\!]$ and we want

$$\Big(\textstyle\bigcup_{z\in[\![M]\!]\rho}[\![N]\!](\rho,x,z),\ [\![N]\!](\rho',x',z')\Big) \in [\![T_\varepsilon Y]\!]$$

The cardinality part of the above is immediate by induction on N. If y is an element of the union, then $y \in [\![N]\!](\rho,x,z)$ for some $z \in [\![M]\!]\rho$. But then $(z,z') \in [\![Z]\!]$ because $|[\![M]\!]\rho/[\![Z]\!]| = 1$, so $\exists y' \in [\![N]\!](\rho',x',z')$ with $(y,y') \in [\![Y]\!]$. Conversely, if $y' \in [\![N]\!](\rho',x',z')$ then for any $z \in [\![M]\!]$ there's $y \in [\![N]\!](\rho,x,z)$ with $(y,y') \in [\![Z]\!]$, so the two expressions are in the $\sim_Z$ relation, as required. □

For example, if we define

$$\begin{aligned}f_1 &= \lambda g:\mathtt{unit}\to T\mathtt{int}.\mathtt{let}\ x \Leftarrow g\,()\ \mathtt{in}\ \mathtt{let}\ y \Leftarrow g\,()\ \mathtt{in}\ \mathtt{val}\ x+y\\ f_2 &= \lambda g:\mathtt{unit}\to T\mathtt{int}.\mathtt{let}\ x \Leftarrow g\,()\ \mathtt{in}\ \mathtt{val}\ x+x\end{aligned}$$

then we have $\vdash f_1 = f_2 : (\mathtt{unit}\to T_{01}\mathtt{int}) \to T_{01}\mathtt{int}$ and hence, for example,

$$\vdash (\mathtt{val}\ f_1)\ \mathtt{or}\ (\mathtt{val}\ f_2)\ =\ \mathtt{val}\ f_2 : T_1((\mathtt{unit}\to T_{01}\mathtt{int})\to T_{01}\mathtt{int}).$$

Note that the notion of equivalence really is type-specific. We have

$$\not\vdash f_1 = f_2 : (\mathtt{unit}\to T_{\mathtt{N}}\mathtt{int}) \to T_{\mathtt{N}}\mathtt{int}$$

and that equivalence indeed does not hold in the semantics, even though both f_1 and f_2 are related to themselves at (i.e. have) that type.

Extensions. The syntactic rules can be augmented with anything proved sound in the model. For example, one can add a subtyping rule $T_{1+}X \le T_1 X$ for any X such that $|[\![UX]\!]/[\![X]\!]| = 1$. Or one can manually add typing or equational judgements that have been proved by hand, without compromising the general equational theory. For example, Wadler [28] considers parsers that we could give types of the form $P_\varepsilon X = \texttt{string} \to T_\varepsilon(X \times \texttt{string})$. One type of the alternation combinator

$$alt(p_1, p_2) = \lambda s : \texttt{string}. \begin{array}{l} \texttt{let}\ (v, s') \Leftarrow p_1 s\ \texttt{in val}\ (\texttt{inl} v, s') \\ \texttt{or let}\ (v, s') \Leftarrow p_2 s\ \texttt{in val}\ (\texttt{inr} v, s') \end{array}$$

is $P_{01}X \times P_{01}Y \to P_{\texttt{N}}(X + Y)$. But if we know that $p_1 : P_{01}X$ and $p_2 : P_{01}Y$ cannot both succeed on the same string, then we can soundly ascribe $alt(p_1, p_2)$ the type $P_{01}(X + Y)$.

A further extension is to add pruning. One way is

$$\frac{\Gamma \vdash M_1 : TA \quad \Gamma \vdash M_2 : TA}{\Gamma \vdash M_1\ \texttt{orelse}\ M_2 : TA}$$

$$[\![\Gamma \vdash M_1\ \texttt{orelse}\ M_2 : TA]\!]\rho = \begin{cases} [\![\Gamma \vdash M_2 : TA]\!]\rho \text{ if } [\![\Gamma \vdash M_1 : TA]\!]\rho = \emptyset \\ [\![\Gamma \vdash M_1 : TA]\!]\rho \text{ otherwise} \end{cases}$$

with refined typing

$$\frac{\Theta \vdash M_1 : T_{\varepsilon_1}X \quad \Theta \vdash M_2 : T_{\varepsilon_2}X}{\Theta \vdash M_1\ \texttt{orelse}\ M_2 : T_{\varepsilon_1 \rhd \varepsilon_2}X}$$

where $\varepsilon_1 \rhd \varepsilon_2$ is defined to be $\varepsilon_1 + \varepsilon_2$ if $\texttt{0} \le \varepsilon_1$ and ε_1 otherwise. The `orelse` operation can be used to improve efficiency in search-style uses of non-determinism and is, of course, the natural combining operation to use in error-style uses.

4 Discussion

We have given an elementary relational semantics to a simple effect system for non-deterministic programs, and shown how it may be used to establish effect-dependent program equivalences. Extending or adapting the constructions to richer languages or slightly different monads should be straightforward. One can also enrich the effect language itself, for example by adding conjunctive refinements and effect polymorphism, as we have done previously [3]. The simple style of effect system presented here seems appropriate for fairly generic compilation of a source language with a pervasively non-deterministic semantics, but for which much code could actually be expected to be deterministic. For serious optimization of non-trivially non-deterministic code, one would need to combine effects with refinements on values, to formalize the kind of reasoning used in the parser example above.

Non-determinism monads are widely used to program search, queries, and pattern matching in functional languages. In Haskell, the basic constructs we use here are abstracted as the `MonadPlus` class, though different instances satisfy different laws, and there has been much debate about which laws one should expect to hold in general [32, 25, 27].[6] Several researchers have studied efficient implementations of functional non-determinism and their various equational properties [17, 14].

Static analysis of functional non-determinism is not so common, though Kammar and Plotkin have developed a general theory of effects and effect-based transformations, based on the theory of algebraic effects [18]. Non-determinism is just one example of that theory, and Kammar and Plotkin establish some equational laws that are very similar to the ones presented here. One interesting difference between their work and that we describe here is that our refinements of the computation type are not necessarily monads in their own right. The interpretation of $T_0 X$ is $\{\emptyset, \emptyset)\}$, which is not preserved by the underlying monadic unit $a \mapsto \{a\}$. If we were to track slightly more refined cardinalities (e.g. sets of size two) then, as we have already observed, the abstract multiplication would no longer be idempotent (or commutative), which also implies that the $T_\varepsilon(\cdot)$s would no longer be themselves monads.

Katsumata has presented an elegant general theory of effect systems, using monoidal functors from a preordered monoid (the effect annotations) to endofunctors on the category of values [19]. The effect system given here is an instance of Katsumata's theory. Our very concrete approach to specific effects is by comparison, perhaps rather unsophisticated. On the other hand, the elementary approach seems to scale more easily to richer effect systems, for example for concurrency [5]. (Indeed, it would be natural to augment concurrent state effects with non-determinism information.) Ahman and Plotkin are developing a still more general framework for refining algebraic effects, which can express temporal properties and of which our analysis should be a special case [1].

There is considerable literature on determinism and cardinality analyses in the context of logic programming (e.g. [10, 13]) with applications including introducing cuts and improving the efficiency of parallel search. Many of these analyses can also detect mutual exclusion between tests [20]. Mercury allows programmers to specify determinism using (we were pleased to discover) the same cardinalities as we do here (`0` = `failure`, `1` = `det`, `01` = `semidet`, `1+` = `multidet`, `N` = `nondet`) and similar abstract operations in the checking algorithm [16].

Acknowledgements. We thank the referees for their thorough and helpful comments.

References

1. D. Ahman and G. D. Plotkin. Refinement types for algebraic effects. In *Abstracts of the 21st Meeting 'Types for Proofs and Programs' (TYPES)*, pages 10–11. Institute of Cybernetics, Tallinn University of Technology, 2015.

[6] Phil was involved in this debate at least as far back as 1997 [11].

2. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *31st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 14–25. ACM, 2004.
3. N. Benton and P. Buchlovsky. Semantics of an effect analysis for exceptions. In *3rd ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, pages 15–26. ACM, 2007.
4. N. Benton, M. Hofmann, and V. Nigam. Abstract effects and proof-relevant logical relations. In *41st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 619–632. ACM, 2014.
5. N. Benton, M. Hofmann, and V. Nigam. Effect-dependent transformations for concurrent programs. arXiv:1510.02419, 2015.
6. N. Benton, J. Hughes, and E. Moggi. Monads and effects. In *Applied Semantics, Advanced Lectures*, volume 2395 of *LNCS*, pages 42–122. Springer-Verlag, 2002.
7. N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations with dynamic allocation. In *9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 87–96. ACM, 2007.
8. N. Benton, A. Kennedy, M. Hofmann, and L. Beringer. Reading, writing and relations: Towards extensional semantics for effect analyses. In *4th Asian Symposium on Programming Languages and Systems (APLAS)*, volume 4279 of *LNCS*, pages 114–130. Springer-Verlag, 2006.
9. N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Third ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 129–140. ACM, 1998.
10. C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of Prolog. In *International Symposium on Logic Programming*, pages 457–471. MIT Press, 1994.
11. K. Claessen, P. Wadler, et al. Laws for monads with zero and plus. Haskell mailing list. `http://www.mail-archive.com/haskell%40haskell.org/msg01583.html`, May 1997. Accessed January 2016.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.
13. S.K. Debray and D.S. Warren. Functional computations in logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(3):451–481, 1989.
14. S. Fischer, O. Kiselyov, and C.-C. Shan. Purely functional lazy nondeterministic programming. *J. Functional Programming*, 21(4/5):413–465, 2011.
15. D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, pages 28–38. ACM, 1986.
16. F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the Australian Computer Science Conference*, pages 337–34, 1996.
17. R. Hinze. Deriving backtracking monad transformers. In *Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 186–197. ACM, 2000.
18. O. Kammar and G. D. Plotkin. Algebraic foundations for effect-dependent optimizations. In *39th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 349–360. ACM, 2012.

19. S. Katsumata. Parametric effect monads and semantics of effect systems. In *41st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 633–646. ACM, 2014.
20. P. López-Garcia, F. Bueno, and M. Hermenegildo. Determinacy analysis for logic programs using mode and type information. In *14th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR)*, volume 3573 of *LNCS*, pages 19–35. Springer-Verlag, 2004.
21. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *15th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 47–57. ACM, 1988.
22. E. Moggi. Computational lambda-calculus and monads. In *4th Annual Symposium on Logic in Computer Science (LICS)*, pages 14–23. IEEE Computer Society, 1989.
23. S. Peyton Jones and P. Wadler. Comprehensive comprehensions. In *ACM SIGPLAN Workshop on Haskell*, pages 61–72. ACM, 2007.
24. J. C. Reynolds. The meaning of types – from intrinsic to extrinsic semantics. Technical Report BRICS RS-00-32, BRICS, University of Aarhus, December 2000.
25. E. Rivas, M. Jaskelioff, and T. Schrijvers. From monoids to near-semirings: The essence of MonadPlus and Alternative. In *17th International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 196–207. ACM, 2015.
26. A. Tolmach. Optimizing ML using a hierarchy of monadic types. In *Second International Workshop on Types in Compilation (TIC)*, volume 1473 of *LNCS*, pages 97–115. Springer-Verlag, 1998.
27. T. Uustalu. A divertimento on MonadPlus and nondeterminism. *Journal of Logical and Algebraic Methods in Programming*, 2016. To appear.
28. P. Wadler. How to replace failure by a list of successes: A method for excpetion handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture (FPCA)*, volume 201 of *LNCS*, pages 113–128. Springer-Verlag, 1985.
29. P. Wadler. Theorems for free! In *Fourth International Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359. ACM, 1989.
30. P. Wadler. The marriage of effects and monads. In *Third ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 63–74. ACM, 1998.
31. P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic*, 4(1):1–32, 2003.
32. A. Yakeley et al. MonadPlus reform proposal. `https://wiki.haskell.org/MonadPlus_reform_proposal`, 2006. Accessed January 2016.