Kent Academic Repository

Mycroft, Alan, Orchard, Dominic A. and Petricek, Tomas (2016) *Effect Systems Revisited - Control-Flow Algebra and Semantics.* Lecture Notes in Computer Science . pp. 1-32. ISSN 0302-9743.

Downloaded from

https://kar.kent.ac.uk/61623/ The University of Kent's Academic Repository KAR

The version of record is available from

https://doi.org/10.1007/978-3-319-27810-0_1

This document version

Author's Accepted Manuscript

DOI for this version

Licence for this version

UNSPECIFIED

Additional information

Versions of research works

Versions of Record

If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in *Title of Journal* , Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries

If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our Take Down policy (available from https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies).

Effect systems revisited—control-flow algebra and semantics

Alan Mycroft[1], Dominic Orchard[2], and Tomas Petricek[1]

[1] University of Cambridge, UK {firstname.lastname}@cl.cam.ac.uk
[2] Imperial College London, UK d.orchard@imperial.ac.uk

Abstract. Effect systems were originally conceived as an inference-based program analysis to capture program behaviour—as a set of (representations of) effects. Two orthogonal developments have since happened. First, motivated by static analysis, effects were generalised to values in an algebra, to better model control flow (*e.g.* for may/must analyses and concurrency). Second, motivated by semantic questions, the syntactic notion of set- (or semilattice-) based effect system was linked to the semantic notion of monads and more recently to graded monads which give a more precise semantic account of effects.
We give a lightweight tutorial explanation of the concepts involved in these two threads and then unify them via the notion of an *effect-directed* semantics for a *control-flow algebra* of effects. For the case of effectful programming with sequencing, alternation and parallelism—illustrated with music—we identify a form of *graded joinads* as the appropriate structure for unifying effect analysis and semantics.

1 Introduction and musical homily

Instead of the usual introduction explaining effect systems and exemplifying their various forms, we start with a musical example. This motivates a particular algebraic approach to *describing* effects, including concurrency, based around the development of Nielson and Nielson along with Amtoft [26, 2].

Section 2 again starts as tutorial, first relating set-based effect systems with *syntactic* labelled monads (due to Wadler and Thiemann [39]) and later with a *semantic* relationship to graded monads [16, 28]. This relationship is parallel to that between types as syntax and types as semantic objects such as sets and domains—or that of algebra as symbol-pushing versus algebraic models.

Section 3 is more novel and argues that (graded) monads alone are insufficient to model effects representing parallelism, and even certain forms of conditional. We identify the notion of control-flow effect operators, as opposed to ordinary effect operators, to characterise the situation.

Section 4 continues by showing how the *joinad* structure [29, 32], which refines monads, can be "graded" (indexed) to unite the above two orthogonal developments of effect systems—giving a particular control-flow algebra which provides an algebraic-and-semantic model of effect systems including parallelism.

Anyway, enough of this chatter—the show must go on!

[2] Author's copy. Appears in Springer, LNCS Volume 9560

```
let happyBirthdayMelody() =
  for line = 1 to 4 do
    play(G, 0.75); play(G, 0.25);
    if line = 3 then play(G2, 1); play(E2, 1); play(C2, 1); play(B, 1); play(A, 1);
    else   play(A, 1); play(G, 1);
          if line = 2 then play(D2, 1); play(C2, 2); else play(C2, 1); play(B, 2);
```

Fig. 1. *Happy Birthday to Hanne and Flemming*

Motivation: richer effect systems. When writing about effect systems, many authors still consider only set-based systems. However, as shown by Nielson and Nielson [26], richer effect systems are useful. In addition to sequential composition, such systems also capture recursion (looping), choice (to model conditionals), and spawning threads. We demonstrate the importance of such rich effect structures in this section, albeit using parallel composition rather than spawning.

1.1 Effect systems for music

In our first example, we honour the celebratory nature of this paper and consider an effect system for music. More specifically, we look at a program (Figure 1) that plays the melody of the "Happy Birthday to You" song.[3] We use a simple imperative language[4] with a primitive $\mathsf{play}(N, l)$ which plays the note N (drawn from the usual CDEFGAB range, with suffix '2' meaning an octave higher, along with the silent note 'rest') for duration l (a rational number) and blocks until l time has elapsed. The function iterates over four phrases of the song. Each phrase starts with notes G G, so these are played always. The third phrase (*Happy birthday dear Hanne and Flemming*) has a different melody, which is handled using the first if. The first two phrases also differ (the second if).

Set-based effect system. The most basic effect system that we could add to the language is shown in Figure 2. It annotates programs with effects Φ—here the *set* of notes that are played. (We generally use F, G, H to range over effects, but while discussing music—perhaps containing the note F—we use Φ instead. Similarly the singleton type is written void rather than unit to avoid conflict with the monad unit operator later.) The (PLAY) rule annotates play with a singleton set containing the played note, ignoring its duration. Sequential composition (SEQ) and (IF) simply union the sets of sub-expressions and (FOR) ignores the repetition and just uses the annotation of the body. Thus, for the above program, the simple effect systems reports the effect $\{\mathsf{G}, \mathsf{A}, \mathsf{B}, \mathsf{C2}, \mathsf{D2}, \mathsf{E2}, \mathsf{G2}\}$.

This is a good start (we now know the range of notes that our piano needs to have!), but it does not tell us very much about the structure of the song.

[3] To slightly shorten the example, the melody of the last phrase repeats that of the first. Musicians are invited to use the correct notes: F2;F2;E2;C2;D2;C2.

[4] This easily maps to the λ-calculus-with-constants formulation used in later sections by replacing the for statement with if and a tail-recursive call, and by treating $e; e'$ as shorthand for let $x = e$ in e' for some fresh variable x.

$$(\textsc{play})\ \frac{}{\Gamma \vdash \mathsf{play}(N,l) : \mathsf{void},\ \{N\}} \quad (\textsc{if})\ \frac{\Gamma \vdash e_0 : \mathsf{bool}, \Phi_0 \quad \Gamma \vdash e_1 : \tau, \Phi_1 \quad \Gamma \vdash e_2 : \tau, \Phi_2}{\Gamma \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau,\ \Phi_0 \cup \Phi_1 \cup \Phi_2}$$

$$(\textsc{seq})\ \frac{\Gamma \vdash e_1 : \tau_1,\ \Phi_1 \quad \Gamma \vdash e_2 : \tau_2,\ \Phi_2}{\Gamma \vdash e_1; e_2 : \tau_2,\ \Phi_1 \cup \Phi_2} \quad (\textsc{for})\ \frac{\Gamma \vdash e : \mathsf{void},\ \Phi}{\Gamma \vdash \mathsf{for}\ i = n_1\ \mathsf{to}\ n_2\ \mathsf{do}\ e : \mathsf{void},\ \Phi}$$

Fig. 2. Simple set-based effect system for music

$$(\textsc{play})\ \frac{}{\Gamma \vdash \mathsf{play}(N,l) : \mathsf{void},\ N} \quad (\textsc{if})\ \frac{\Gamma \vdash e_0 : \mathsf{bool},\ \Phi_0 \quad \Gamma \vdash e_1 : \tau,\ \Phi_1 \quad \Gamma \vdash e_2 : \tau,\ \Phi_2}{\Gamma \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau,\ \Phi_0 \bullet (\Phi_1 + \Phi_2)}$$

$$(\textsc{seq})\ \frac{\Gamma \vdash e_1 : \tau_1,\ \Phi_1 \quad \Gamma \vdash e_2 : \tau_2,\ \Phi_2}{\Gamma \vdash e_1; e_2 : \tau_2,\ \Phi_1 \bullet \Phi_2} \quad (\textsc{for})\ \frac{\Gamma \vdash e : \mathsf{void},\ \Phi}{\Gamma \vdash \mathsf{for}\ i = n_1\ \mathsf{to}\ n_2\ \mathsf{do}\ e : \mathsf{void},\ \Phi^*}$$

Fig. 3. A richer effect system for music with Kleene star and choice

Adding Kleene star and choice. If we want to track the effects of our song more precisely, we can follow Nielson and Nielson and use an effect system with a richer structure [26]. For music, we might use annotations of the following structure:

$$\begin{array}{ll} \Phi = \mathsf{C}, \mathsf{D}, \mathsf{E}, \ldots, \mathsf{rest} & (\textit{primitives: notes, including } \mathsf{rest}) \\ \quad \mid\ \Phi_1 + \Phi_2 & (\textit{choice}) \\ \quad \mid\ \Phi_1 \bullet \Phi_2 & (\textit{sequencing}) \\ \quad \mid\ \Phi^* & (\textit{looping}) \end{array}$$

(where $(^*)$ binds more tightly than $(\bullet)$ which binds more tightly than $(+)$)

The effect system shown in Figure 3 uses the new structure of effect annotations. Sequential composition (SEQ) annotates the expression $e_1; e_2$ with $\Phi_1 \bullet \Phi_2$. The for loop is annotated with Φ^* meaning that the body is executed zero or more times. The conditional if-then-else is annotated with $\Phi_0 \bullet (\Phi_1 + \Phi_2)$ meaning that it evaluates the guard expression first, followed by one of the branches.

Using the revised effect system, the effect annotation of "Happy Birthday to You" becomes: $(\mathsf{G} \bullet \mathsf{G} \bullet (\mathsf{G2} \bullet \mathsf{E2} \bullet \mathsf{C2} \bullet \mathsf{B} \bullet \mathsf{A} + \mathsf{A} \bullet \mathsf{G} \bullet (\mathsf{D2} \bullet \mathsf{C2} + \mathsf{C2} \bullet \mathsf{B})))^*$. This contains a lot more information about the song! It is still an approximation—we only know there are zero or more repetitions, and not how the choices are made or note durations. However we do know in which order the notes are played and what variations there might be. Such an effect system could be used in a music programming language as an interface to communicate higher-order function (combinator) behaviour (see *e.g.*, effect systems for music live coding [1]).

Why richer effects matter. Effect systems can inform optimisations, aid program understanding, and help reject buggy programs. Richer effect systems therefore let us specify valid transformations/programs more precisely.

The above effect system has two interesting properties. Firstly, in the revised Nielson-Nielson-style system, each syntactic element is annotated with a distinct operation in an abstract effect structure. This means that we are, in some sense, describing the most powerful and general non-dependently-typed effect system

for the language. For example, separating sequencing and alternation allows both 'may' and 'must' properties to be analysed. For the effects of if with $\Phi_0 \bullet (\Phi_1 + \Phi_2)$, the usual set-based approach where $\bullet = + = \cup$ gives a 'may' analysis. A 'must' analysis is obtained by instead taking $+ = \cap$, *i.e.* each branch of a conditional must satisfy the minimal requirements specified by the effect.

Secondly, the laws (equational theory) of the programming language imply equations on the effect structure, and vice versa. For example, consider the following program law (inspired by the introduction of Benton *et al.* [4]):

$$\{\mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c'\};\ c'' \ \equiv\ \mathsf{if}\ b\ \mathsf{then}\ \{c; c''\}\ \mathsf{else}\ \{c'; c''\}$$

corresponding to effect algebra axiom $(\Phi + \Phi') \bullet \Phi'' = (\Phi \bullet \Phi'') + (\Phi' \bullet \Phi'')$, where sequential composition is right-distributive over alternation. Thus, axioms of the effect algebra make language properties more explicit and easier to understand.

Adding parallelism. These $\bullet / + / (^*)$ operators often suffice to summarise program behaviour—indeed they capture a range of classical dataflow analyses when appropriately interpreted. However, languages for expressing music need an additional construct, namely parallel composition, to play multiple phrases at the same time. For example, to accompany "Happy Birthday to You" with chords C, G7 and F, we need a parallel-composition construct e_1 par e_2 and define:

$$\begin{aligned}
&\mathsf{let}\ \mathsf{chord_C}(l) &&= \mathsf{play}(\mathsf{C}, l)\ \mathsf{par}\ \mathsf{play}(\mathsf{E}, l)\ \mathsf{par}\ \mathsf{play}(\mathsf{G}, l)\\
&\mathsf{let}\ \mathsf{chord_G7}(l) &&= \mathsf{play}(\mathsf{G}, l)\ \mathsf{par}\ \mathsf{play}(\mathsf{B}, l)\ \mathsf{par}\ \mathsf{play}(\mathsf{D}, l)\ \mathsf{par}\ \mathsf{play}(\mathsf{F}, l)\\
&\mathsf{let}\ \mathsf{chord_F}(l) &&= \mathsf{play}(\mathsf{F}, l)\ \mathsf{par}\ \mathsf{play}(\mathsf{A}, l)\ \mathsf{par}\ \mathsf{play}(\mathsf{C}, l)
\end{aligned}$$

Following the methodology of the previous section, we now need a corresponding extension of our effect system. We add a new operation for parallel composition to the effect structure, written $\Phi_1 \& \Phi_2$, and supply the following typing rule:

$$(\textsc{par})\ \frac{\Gamma \vdash e_1 : \tau_1,\ \Phi_1 \quad \Gamma \vdash e_2 : \tau_2,\ \Phi_2}{\Gamma \vdash e_1\ \mathsf{par}\ e_2 : \tau_1 \times \tau_2,\ \Phi_1 \& \Phi_2}$$

We can now write a program that accompanies the melody with the above chords as harmony. For the last two lines, we play C and F chords for line 3 and then C, G7, C for line 4, giving an effect that is the parallel composition & of the melody program above and the effect of the chords $(\mathsf{C}\&\mathsf{E}\&\mathsf{G}) \bullet ((\mathsf{F}\&\mathsf{A}\&\mathsf{C}) + ((\mathsf{G}\&\mathsf{B}\&\mathsf{D}\&\mathsf{F}) \bullet (\mathsf{C}\&\mathsf{E}\&\mathsf{G})))$. Note that the effect structure above allows us to capture the fact that one chord finishes before the next chord starts. However, we would need to enrich effects with durations to be able to use effect-based reasoning to argue that the melody and harmony synchronise as our ears expect.

Our modelling of music is inspired by Nielson and Nielson who considered spawning processes in CML, in an effect system over what they termed *behaviours*. This included basic effects of channel allocation, sending and receiving (similar to *session types* [11]), a τ action for internal communication, binary effect operators for sequencing and alternation, and unary effect operator SPAWN to denote task spawning. We prefer to use a binary parallel-composition operator instead of SPAWN as commutativity of parallelism is more easily expressed.

Parallelism and other operators. How does the & operator relate to other effect operators? In CCS and the π-calculus, Milner's *interleaving-semantic* view identifies our $(\Phi_1 \,\&\, \Phi_2)$ with $(\Phi_1 \bullet \Phi_2) + (\Phi_2 \bullet \Phi_1)$; two concurrent *events* are equivalent to the same two events in some non-deterministic order [22]. In Milner's work, this rests on his assumption that events are atomic and, indeed, the model is sufficient for many purposes (*e.g.* when the primitive operations are serialised into one-at-a-time interactions).

However, doing so is against our aim of providing a fully *general* system and it excludes many effect systems we wish to consider. In music, events have *duration* and so we need to distinguish three basic effect operators: assuming Φ_1 and Φ_2 are notes (or pieces of music) then $\Phi_1 \bullet \Phi_2$ means play Φ_1 then Φ_2, and $\Phi_1 + \Phi_2$ means play Φ_1 or Φ_2, and $\Phi_1 \& \Phi_2$ means play Φ_1 at the same time as Φ_2. This justifies the idea of having three separate effect operators which can, if desired, be interpreted so as to satisfy interleaving, but not to build in interleaving.

1.2 Section conclusion and placement

> "The methodology of annotated type and effect systems consists of: (*i*) expressing a program analysis by means of an annotated type or effect system, (*ii*) showing the semantic correctness of the analysis, (*iii*) developing an inference algorithm and proving it syntactically sound and complete." (Nielson *et al.* [25])

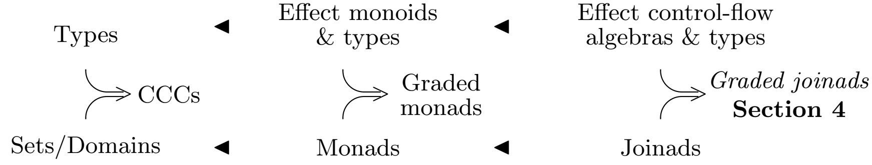
This paper addresses a particular subset of effect systems (*i*), ones expressible in terms of primitive effects composed with operations for sequential and parallel composition along with alternation (and iteration, considered briefly in Section 5). In Section 2, we overview some of the literature connecting effect systems to monads, first syntactically and then semantically, and we explain how graded monads [16] provide an *effect-directed semantics*. This refines a usual monadic semantics and aids correctness (*ii*) by unifying analysis and semantics.

Section 3 explores how (graded) monads have limited ability to express non-sequential control flow (choice/parallelism); monads per se capture only *sequential composition* (corresponding to $\bullet$ in effect annotations). For the musical example, we must also capture control flow corresponding to the + and & effect operations. We propose that *joinads* [32] fill this role. Section 4 recalls joinads, introducing a variant that we call *conditional joinads*; these are graded analogously to monads. The resulting *graded conditional joinads* provide an effect-directed denotational semantics for rich effect systems. Relevant work from the literature is scattered throughout, but more is given along with discussion in Section 5.

This paper does not consider inference *algorithms* (*iii*), and indeed we do not discuss polymorphism over type or effects, which are precursors to principality, or near principality, of type and effect *inference systems*.

Whilst much of the material here is grounded in category theory, we aim at a more accessible presentation, mostly in terms of set-theoretic or programming concepts. Throughout we assume an underlying Cartesian-closed category for our semantics and so use the λ-calculus as its internal language to aid readability.

Paper outline. The following diagram summarises the big picture of the paper, where ◀ indicates a richer structure to the right:



The top line gives syntax of program analyses (types and effects); the bottom gives the related semantic interpretations. The left-hand denotes the use of Cartesian-closed categories to refine a set- or domain-based semantics with types to give a *type-directed semantics*. In a similar way (in the middle part), graded monads unify (refine) a monadic semantics with the information from a traditional monoidal effect system, giving *effect-directed semantics*. The right-hand part expresses our analogous construction, relating the generalisation of monads to joinads with the generalisation of monoidal effects to richer (semiring-like) structures for control flow, by *graded joinads*.

2 Monads and effect systems

Effect systems are a class of static analyses for program side-effects [9, 37, 18]. They are typically inductively defined over the syntax (and types) of a program, presented as augmented typing rules (rather than a flow-based analysis such as dataflow), hence their full title: *type-and-effect systems*. (Another view is that the effect annotations are types of a different *kind*). Effect systems have been used for a variety of applications, including analysing memory access [9], message passing [12], control side-effects and unstructured control primitives (goto/comefrom) encoded as continuations [13], and atomicity in concurrency [7].

We overview effect systems briefly (Section 2.1) and relate these to semantics in a gradual way: first syntactically to monadic typing (Section 2.2), then semantically in a type-directed way (Section 2.3) and finally in an effect-directed way using *graded monads* (Section 2.4) (from [16, 28]).

Our base language is the call-by-value simply-typed λ-calculus with constants, conditionals, and parallel composition, with syntax:

$$e ::= x \mid k_\tau \mid \lambda x.e \mid e_1\, e_2 \mid \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \mid \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \mid e_1\, \mathsf{par}\, e_2$$

where k_τ are constants of type τ (k_τ of function type can produce effects). Throughout x, y range over variables. The par construct is considered only in Section 4; $\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$ is treated as a simple abbreviation for $(\lambda x.e_2)\, e_1$.

Semantics We mainly use a denotational-semantic framework: the meaning of an expression e is a value $[\![e]\!]$ in a mathematical domain, defined inductively over the structure of e. The meaning of an expression depends on both its free variables and its type derivation, accordingly $[\![\Gamma \vdash e : \tau]\!]$ is typically a value in the

space of denotations $\llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket$. Free-variable contexts $\Gamma = x_1 : \tau_1, ..., x_n : \tau_n$ are interpreted as environments ranged over by γ. Conventionally these are products $\llbracket \tau_1 \rrbracket \times ... \times \llbracket \tau_n \rrbracket$ indexed by positions but for our purposes we index by variable names written $\langle x_1 : \llbracket \tau_1 \rrbracket, ..., x_n : \llbracket \tau_n \rrbracket \rangle$. We write $\gamma[x \mapsto v]$ for the environment γ with variable x updated to value v. Finally, the space of denotations is often presented as a *category*. While this can be a unifying framework, it can impose additional overhead (*e.g.* the need for 'strength' of monads, Appendix A) so here we largely keep to a set-based special-case framework.

Correctness of denotational models is established here relative to an *axiomatic semantics* (equational theory)—a congruence relation $\equiv$ (on e), typically derived from an operational semantics (rewrite rules). A denotational semantics is *sound* when $e \equiv e' \Rightarrow \llbracket e \rrbracket = \llbracket e' \rrbracket$, and *complete* if the converse holds. This extends naturally to equalities $\Gamma \vdash e \equiv e' : \tau, F$ relative to a type (and effect) derivation, and their interpretations $\llbracket \Gamma \vdash e : \tau, F \rrbracket = \llbracket \Gamma \vdash e' : \tau, F \rrbracket$.

We assume the standard operational and axiomatic semantics of the call-by-value λ-calculus for our base language (with β-reduction on syntactic values $(\lambda x.e)\, v \to e[v/x]$ and its corresponding β-equality). The axiomatic semantics of if and par are given in Section 4.2 (p. 22). The denotational and axiomatic semantics are specialised for particular notions of effect and effectful primitives.

2.1 Traditional effects

Type and effect judgements take the form: $\Gamma \vdash e : \tau, F$ asserting that an expression e has a type τ and at most produces *immediate* effects F, in the context Γ. Effects F are taken from a set $\mathcal{F}$ and types have the form: $\tau ::= \tau_1 \xrightarrow{F} \tau_2 \mid \iota$ where ι ranges over primitive types (*e.g.* bool, int, void). Throughout τ ranges over types and F over effects. Function types give an anchor for the *latent* effects of a function, which arise when the function is applied. In Section 1, only immediate effects appeared explicitly, but we can see happyBirthdayMelody as having the latent effect of playing the music when called, but no immediate effects.

Early definitions of effect systems described a *lattice* structure on effects but rely only on effects forming a join *semi*-lattice, where the least-upper-bound (join) operation combines effect annotations and the least element annotates pure computations [9, 18]. This was demonstrated for sets with union: $(\mathcal{F} = \mathcal{P}(S), \cup, \emptyset)$. Originally, effects were considered for the polymorphic λ-calculus with type and effect polymorphism. The discussion in this paper is monomorphic. We use meta-variables for effects and types which may be instantiated, giving meta-level polymorphism.

Figure 4 shows a type and effect system for our base language, following Gifford and Lucassen's approach. We generalise from sets to an arbitrary bounded join semilattice $(\mathcal{F}, \sqcup, \bot)$. Figure 5 shows additional rules for an instance of the calculus which tracks memory accesses to memory regions ρ. This instance has additional types ref ρ and effectful primitives read and write with their respective effects rd_ρ and wr_ρ (note, these are derived rules, from (CONST) and (APP)). Figure 6 gives the usual (SUB) rule for *sub-effecting*, over-approximating effects with respect to an ordering [37] and an alternative syntax-directed (COERCE) rule.

$$(\textsc{var})\ \frac{}{\Gamma, x{:}\tau \vdash x : \tau,\ \bot} \qquad (\textsc{let})\ \frac{\Gamma \vdash e_1 : \tau,\ F \quad \Gamma, x{:}\tau \vdash e_2 : \tau',\ G}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau',\ F \sqcup G}$$

$$(\textsc{const})\ \frac{}{\Gamma \vdash k_\tau : \tau,\ \bot} \qquad (\textsc{if})\ \frac{\Gamma \vdash e_0 : \mathsf{bool},\ F \quad \Gamma \vdash e_1 : \tau,\ G \quad \Gamma \vdash e_2 : \tau,\ H}{\Gamma \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau,\ F \sqcup G \sqcup H}$$

$$(\textsc{abs})\ \frac{\Gamma, x{:}\tau \vdash e : \tau',\ F}{\Gamma \vdash \lambda x.e : \tau \xrightarrow{F} \tau',\ \bot} \qquad (\textsc{app})\ \frac{\Gamma \vdash e_1 : \tau \xrightarrow{H} \tau',\ F \quad \Gamma \vdash e_2 : \tau,\ G}{\Gamma \vdash e_1\, e_2 : \tau',\ F \sqcup G \sqcup H}$$

Fig. 4. Gifford-Lucassen effect system for an impure λ-calculus, using semilattice notation (concretely they used sets of effects).

$$(\textsc{write})\ \frac{\Gamma \vdash e_1 : \mathsf{ref}_\rho\, \tau,\ F \quad \Gamma \vdash e_2 : \tau,\ G}{\Gamma \vdash \mathsf{write}\ e_1\, e_2 : \mathsf{void},\ F \sqcup G \sqcup \mathsf{wr}_\rho} \qquad (\textsc{read})\ \frac{\Gamma \vdash e : \mathsf{ref}_\rho\, \tau,\ F}{\Gamma \vdash \mathsf{read}\ e : \tau,\ F \sqcup \mathsf{rd}_\rho}$$

Fig. 5. Effect-specific (derived) rules instantiating k_τ for memory access.

$$(\textsc{sub})\ \frac{\Gamma \vdash e : \tau,\ F}{\Gamma \vdash e : \tau,\ G}\ \text{if } F \sqsubseteq G \qquad (\textsc{coerce})\ \frac{\Gamma \vdash e : \tau,\ F}{\Gamma \vdash \mathsf{coerce}^{F,G} e : \tau,\ G}\ \text{if } F \sqsubseteq G$$

Fig. 6. Implicit and explicit sub-effecting rules.

2.2 Effects and monads—syntactically

We summarise Wadler and Thiemann's work which goes halfway towards a semantic unification of monads and effects [39]. We distinguish the notion of syntactic monads (basically a type constructor, written M here) from semantic monads (written T following tradition) which model effectful computation.

Monads are a class of algebraic structure from category theory that have been found to provide a useful model for the sequential composition of computations that have various kinds of side effect, such as state, non-determinism, exceptions, and continuations [23, 24]. The idea is that impure, call-by-value computations can be modelled semantically by functions $A \to \mathsf{T}B$ where A and B model input and output types respectively and $\mathsf{T}B$ is a structure[5] encoding the side effects which occur during the computation of a B output value. The operations and axioms of a monad provide an (associative) sequential composition $\hat{\circ}$ such that, given functions $f : A \to \mathsf{T}B$ and $g : B \to \mathsf{T}C$, then $g \,\hat{\circ}\, f : A \to \mathsf{T}C$ with an identity (modelling a trivially pure computation) $\hat{id} : A \to \mathsf{T}A$. Usual presentations of monads decompose the definition of $\hat{\circ}$. A more formal definition is delayed until Definition 1 (p. 10).

[5] Mathematically, T is an *endofunctor*, but languages such as Haskell (and the *Monad* language in this section) expose monads syntactically as *parametric type constructors* M; thus side-effecting functions have analogous types $\tau \to M\tau'$. We try not to labour either this distinction or that between types (often written A, B instead of τ, τ' above) and the categorical objects A, B which model them (more formally $[\![\tau]\!], [\![\tau']\!]$).

Wadler and Thiemann observed that (semilattice) effect systems and monads are *homomorphic*: they have the same shape and carry related information [39]. They show this via two languages: *Effect*, a λ-calculus with an effect system, recursion, and mutable references (for which Figures 4 and 5 give a similar definition), and *Monad*, a typed λ-calculus for monadic programming without an effect system. The standard monadic approach to programming (such as in Haskell) introduces a parameterised data type M to encapsulate and encode effects, where $M\tau$ represents a computation that may perform some effects to compute a value of type τ. In the *Monad* language, this type constructor M is additionally labelled with a set of effects F denoting the (maximum) set of side effects which may be performed by that computation, written M^F.

The *Monad* language adds the following constructs for manipulating monadic computations, where (RETURN) constructs a pure monadic computation and (BIND) provides composition of monadic computations.

$$\text{(RETURN)}\ \frac{\Gamma \vdash_M e : \tau}{\Gamma \vdash_M \langle e \rangle : M^{\emptyset}\tau} \qquad \text{(BIND)}\ \frac{\Gamma \vdash_M e : M^F\tau \quad \Gamma, x : \tau \vdash e' : M^G\tau'}{\Gamma \vdash_M \mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, e' : M^{F \cup G}\tau'}$$

Effectful operations in the language are given monadic types, with state-using functions: $\mathsf{read} : \mathsf{ref}\,\tau \to M^{\{\mathsf{rd}\}}\tau$ and $\mathsf{write} : \mathsf{ref}\,\tau \to \tau \to M^{\{\mathsf{wr}\}}\tau$, which are composable with (BIND). For example, the following increments location r:

$$r : \mathsf{ref\ int} \vdash_M \mathsf{let}\ x \Leftarrow \mathsf{read}\ r\ \mathsf{in}\ \mathsf{write}\ r\ (x+1) : M^{\{\mathsf{rd},\mathsf{wr}\}}\mathsf{int}$$

Wadler and Thiemann show that all terms e in the *Effect* language can be translated[6] to terms $[e]$ in the *Monad* language, with type-and-effect judgements $\Gamma \vdash e : \tau, F$ of *Effect* mapped to $[\Gamma] \vdash_M [e] : M^F[\tau]$ of *Monad* where $[\tau]$ is defined:

$$[\tau \xrightarrow{F} \tau'] = [\tau] \to M^F[\tau']$$
$$[\mathsf{int}] = \mathsf{int} \quad \text{(and similarly for other base types)}$$

Contexts Γ are translated to $[\Gamma]$ by applying $[\tau]$ pointwise. For expressions e, we show (a simplified version of) Wadler and Thiemann's encoding for variables, abstraction, and application:

$$[x] = \langle x \rangle \qquad [\lambda x.e] = \langle \lambda x.[e] \rangle \qquad [e\, e'] = \mathsf{let}\ f \Leftarrow [e]\ \mathsf{in}\ \mathsf{let}\ x \Leftarrow [e']\ \mathsf{in}\ f\, x$$

Variables are translated by wrapping them in the "return" construct $\langle - \rangle$, lifting the computation to a trivially effectful monadic value of type $M^{\emptyset}[\tau]$ to keep in step with the type/effect translation. The translation of λ-abstraction translates the body, places it within a (pure) λ-term in *Monad*, and finally wraps this in an effectless $\langle - \rangle$ similarly to variables, thus having type $M^{\emptyset}([\tau] \to M^F[\tau'])$ for effects F in the body of the function. For application, the function term e (say with effect F and latent effect H) and argument term e' (with effect G) are translated and bound to f and x respectively using monadic binding, giving a left-to-right call-by-value evaluation order of the effects. Thus, $f : [\tau] \to M^H[\tau']$ and $x : [\tau]$ by the typing of let and the overall term has type $M^{F \cup G \cup H}[\tau']$.

[6] We use $[-]$ for *translation* into another language and reserve $[\![-]\!]$ for semantic *interpretation* (denotation) as in the next section.

2.3 Effects and monads—weakly semantically

So far, we discussed effect systems from a purely syntactic perspective. In this section, we interpret families of labelled syntactic monadic types M^F as a single semantic monad T, ignoring the effect label. This gives a semantics that is syntax- and type-directed. In Section 2.4, we show how *graded monads* provide a stronger *effect-directed* semantics, where effect labels are part of the semantics—*i.e.* each M^F is separately interpreted as a semantic object T_F.

The *Monad* language is based on a monadic denotational semantics (similar to Moggi's monadic meta language [24]). The translation from *Effect* to *Monad*, coupled with a monadic semantics for *Monad*, provides a monadic semantics for *Effect* (similar to a semantics for an impure λ-calculus [23]). The semantics of *Monad* is given by mapping the monadic let (BIND) and $\langle - \rangle$ (RETURN) constructs into the operations of a monad in the semantic domain. We first define monads, balancing both the categorical and programming language viewpoints:

Definition 1. *A* monad T *is an operator on spaces (here either categories,* e.g. *semantic domains, or programming language types) along with a family of constants* $\mathsf{unit}_A : A \to \mathsf{T}A$ *and a family of operations* $\mathsf{extend}_{A,B}$ *mapping from space* $A \to \mathsf{T}B$ *to space* $\mathsf{T}A \to \mathsf{T}B$ *satisfying the axioms:*

$$\mathsf{extend}\ \mathsf{unit}\ x = x \quad [\mathrm{M1}] \qquad\qquad \mathsf{extend}\ f\ (\mathsf{unit}\ x) = f\,x \quad [\mathrm{M2}]$$
$$\mathsf{extend}\ (\mathsf{extend}\ g\ (f\ x)) = \mathsf{extend}\ g\ (\mathsf{extend}\ f\ x) \quad [\mathrm{M3}]$$

Given functions $f : A \to \mathsf{T}B$ and $g : B \to \mathsf{T}C$, their composition, given by $g \,\hat{\circ}\, f = (\mathsf{extend}\ g) \circ f : A \to \mathsf{T}C$, is associative (by axiom [M3]) and has unit as identity (by [M1],[M2]). This definition is the *Kleisli triple* form of a monad.

Remark 1. Mathematically, monads are endofunctors on the category of spaces of semantic values. On the other hand, programming-language monads are unary type constructors with associated polymorphic unit and extend operations. In Haskell, unit is written `return`, and extend is written '`>>=`' (with its two arguments reversed), called *bind*. These are presumed to follow the above equations.

Example 1 (State monad). Let $\mathsf{State}\,A = S \to (A \times S)$ for some store type S, modelling a mapping from a store S to a result value A paired with a new store. State is a monad with the following operations:

$$\mathsf{extend}\,f\,x = \lambda s.\,\mathsf{let}\ (a, s') = x\ s\ \mathsf{in}\ (f\,a)\,s' \qquad\qquad \mathsf{unit}\,x = \lambda s.\,(x, s)$$

where $\mathsf{extend}\ f : A \to (S \to (B \times S))$ and $x : S \to (A \times S)$. The extend operation 'threads' state through a computation. An effectful function, of type $f : A \to \mathsf{State}\,B$, is a *state transformer* (by uncurrying $A \to (S \to (B \times S)) \cong A \times S \to B \times S$, *cf.* small-step operational semantics reductions $\langle e, s\rangle \to \langle e', s'\rangle$ mapping terms paired with stores). The unit operation lifts a value to a pure computation where the state is unchanged (a trivial state transformer).

Here we give a type-directed semantics for *Monad*, mapping type *derivations* to denotations as functions (more generally, morphisms) from the interpretation of the context Γ to that of the resulting type τ, *i.e.* $[\![\Gamma \vdash_M e : \tau]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$.

Monadic denotational semantics For the monadic semantics of *Monad*, the *syntactic* notion of a monad represented by the type constructors M^F is mapped to an abstract *semantic* monad T (note there is a single monad T capturing the meaning of the family of type constructors). Thus, the interpretation of types is:

$$[\![\tau \to \tau']\!] = [\![\tau]\!] \to [\![\tau']\!] \qquad [\![M^F \tau]\!] = \mathsf{T}[\![\tau]\!] \tag{1}$$

We assume some additional interpretation of base types ι into suitable sets in the domain of the semantics (*e.g.*, $[\![\mathsf{int}]\!] = \mathbb{Z}_\bot$).

The interpretation of type(-and-effect) derivations ending in (BIND) and (RETURN) rules are then (omitting the type subscripts on unit and extend):

$$\begin{aligned}&[\![\Gamma \vdash_M \mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, e' : M^{F \cup G}\tau']\!] \;:\; [\![\Gamma]\!] \to \mathsf{T}[\![\tau']\!] = \\ &\qquad \lambda\gamma\,.\; \mathsf{extend}\; (\lambda v.\, [\![\Gamma, x\!:\!\tau \vdash_M e' : M^G \tau']\!]\;\gamma[x \mapsto v])\;\; ([\![\Gamma \vdash_M e : M^F \tau]\!]\, \gamma) \\ &[\![\Gamma \vdash_M \langle e \rangle : M^\emptyset \tau]\!] : [\![\Gamma]\!] \to \mathsf{T}[\![\tau]\!] = \mathsf{unit} \circ [\![\Gamma \vdash_M e : \tau]\!]\end{aligned}$$

The semantics for *Monad* resembles the desugaring of Haskell's **do**-notation into methods of the `Monad` type class. Seen categorically, the semantics requires some additional structure: the monad T must be *strong*. This is implicit in the category of sets and Cartesian-closed categories and is elided here (see Appendix A).

This monadic semantics for *Monad* (and thus for *Effect* via translation $[-]$) can be shown sound and complete with respect to the axiomatic semantics (due to the strong monad axioms, see [23, 24] with semantics on similar calculi).

Wadler and Thiemann showed the syntactic correspondence between the types-and-effects of *Effect* and (annotated) monadic typing of *Monad*, and soundness results on their operational semantics. However they did not give a denotational semantics marrying effect annotations to monads—all labelled monadic type constructors M^F are interpreted within a single semantic monad T (Equation (1)) and hence lose the effect information F. While Wadler and Thiemann conjectured that a general 'coherent' denotational semantics can be given to unify effect systems with a monadic-style semantics, it was Katsumata who provided the missing piece—*graded monads* [16]. These allow each syntactic type $M^F \tau$ to be interpreted as a semantic object $\mathsf{T}_F[\![\tau]\!]$ in which only the effects represented by F are modelled. Thus Equation (1) is refined to $[\![M^F \tau]\!] = \mathsf{T}_F[\![\tau]\!]$.

2.4 Effects and monads—strongly semantically via gradedness

Graded monads provide a model of sequential composition for computational effects similar to monads, but which carry, and can be refined by, effect information. A graded monadic type T has two parameters, an effect (say F) and a type (say A as usual) written $\mathsf{T}_F A$. Note that we use superscripted effects on syntactic monads representing effects M^F but write the effects of graded monads as subscripts. Effects F are drawn from a (partially) ordered[7] monoid $(\mathcal{F}, \bullet, I, \sqsubseteq)$ generalising a semilattice $(\mathcal{F}, \sqcup, \bot, \sqsubseteq)$. Here $\bullet$ represents sequential

[7] Katsumata uses 'pre-ordered' but we simply consider $(\sqsubseteq)\cap(\sqsubseteq^{-1})$ equivalence classes.

composition of effects and I the trivial, pure effect; $(\bullet)$ must be $(\sqsubseteq)$-monotonic. The ordering $\sqsubseteq$ can capture both sub-effecting (treating a smaller effect as a larger one giving a 'may' analysis for if-then-else) and super-effecting (giving a 'must' analysis)—see Remark 2 (p. 17). For if-then-else it is often convenient that $\sqsubseteq$ has least upper bounds; we also return to this point later.

A graded monad structure on T provides (associative) sequential composition $\hat{\circ}$ for all $f : A \to \mathsf{T}_F B$ and $g : B \to \mathsf{T}_G C$ such that $g \,\hat{\circ}\, f : A \to \mathsf{T}_{F \bullet G} C$ with an identity $\hat{id} : A \to \mathsf{T}_I A$, constructing a pure computation.

Definition 2 (Graded monads [21, 16], without ordering). *Let $(\mathcal{F}, \bullet, I)$ be a monoid. An ($\mathcal{F}$-)*graded monad *T is a family of endofunctors $\mathsf{T}_F A$ (or, in the programming-language view, an effect-annotated unary type constructor) along with two families of operations (polymorphic functions): $\mathsf{unit}^I_A : A \to \mathsf{T}_I A$ and* extension *operations $\mathsf{extend}^{F,G}_{A,B}$ which map functions $g : A \to \mathsf{T}_G B$ to $\mathsf{extend}^{F,G}_{A,B}\, g : \mathsf{T}_F A \to \mathsf{T}_{F \bullet G} B$, satisfying the following axioms (omitting type subscripts) for all $F, G, H \in \mathcal{F}$:*

$$\mathsf{extend}^{F,I}\ \mathsf{unit}^I\ x = x \quad [\text{M1}] \qquad \mathsf{extend}^{I,F}\ f\ (\mathsf{unit}^I\ x) = f\ x \quad [\text{M2}]$$

$$\mathsf{extend}^{F,G \bullet H}\ (\mathsf{extend}^{G,H}\ h\ (g\ x)) = \mathsf{extend}^{F \bullet G,H}\ h\ (\mathsf{extend}^{F,G}\ g\ x) \quad [\text{M3}]$$

A graded monad is a homomorphism (*structure-preserving map*) between a monoidal algebra of effects $\mathcal{F}$ and a monoidal structure for effect semantics. The axioms [M1-3] rely on the monoid axioms on $\mathcal{F}$ *e.g.* [M1–2] as diagrams are:

$$\begin{array}{ccc} \mathsf{T}_F A & \xrightarrow{\mathsf{extend}^{F,I}\ \mathsf{unit}^I} & \mathsf{T}_{F\bullet I} A \\ id \downarrow & [\text{M1}] & \\ \mathsf{T}_F A & = & \mathsf{T}_{F\bullet I} A \end{array} \qquad \begin{array}{ccc} A & \xrightarrow{\mathsf{unit}^I} & \mathsf{T}_I A \\ f \downarrow & [\text{M2}] & \downarrow \mathsf{extend}^{I,F}\ f \\ \mathsf{T}_F B & = & \mathsf{T}_{I\bullet F} B \end{array} \tag{2}$$

The equality edges (double lines) explain the need for the monoid axioms of graded monads: for [M1] that $F = F \bullet I$ for all F and for [M2] that $F = I \bullet F$. The diagram for [M3], not shown for brevity, needs the associativity axiom.

Definition 3 (Graded monads, with ordering). *Katsumata's definition of graded monads includes the pre-ordering $\sqsubseteq$ and subsequently a family of morphisms, which we call $\mathsf{coerce}^{F,G}_A : \mathsf{T}_F A \to \mathsf{T}_G A$ for every $F \sqsubseteq G$ satisfying:*

$$\mathsf{coerce}^{F,F}_A = id_{\mathsf{T}_F A}\ \textit{(reflexivity)} \quad \mathsf{coerce}^{G,H}_A \circ \mathsf{coerce}^{F,G}_A = \mathsf{coerce}^{F,H}_A\ \textit{(transitivity)}$$

$$\mathsf{coerce}^{F\bullet X, G\bullet Y}_B \circ \mathsf{extend}^{F,X}_{A,B} f = \mathsf{extend}^{G,Y}_{A,B} (\mathsf{coerce}^{X,Y}_B \circ f) \circ \mathsf{coerce}^{F,G}_A\ \textit{(monotonicity)}$$

Example 2 (Graded state monad, appears in [27]). Example 1 showed the state monad $\mathsf{State}\, A = S \to (A \times S)$ in which all read and write operations are represented. We now refine this to a graded monad State_F in which only read and write operations expressed by F may be represented. Suppose S is the space of functions from a set of locations Loc, ranged over by ρ, to Val (abusively, we conflate the notions of 'region' and 'location' here.) Take effects $F \in \mathcal{F}$ to be sets of tokens $\mathsf{rd}\,\rho$ and $\mathsf{wr}\,\rho$, giving an ordered effect monoid $(\mathcal{F}, \cup, \emptyset, \subseteq)$. We refine

State to $\mathsf{State}_F\, A = (\mathsf{R}_F \rightarrow \mathsf{Val}) \rightarrow (A \times (\mathsf{W}_F \rightarrow \mathsf{Val}))$ where $\mathsf{R}_F = \{\rho \mid \mathsf{rd}\, \rho \in F\}$ and $\mathsf{W}_F = \{\rho \mid \mathsf{wr}\, \rho \in F\}$ are respectively the subsets of Loc where State_F might read and write. The effect-graded operations are then:

$$\mathsf{extend}^{F,G}\, f\, x = \lambda s.\, \mathsf{let}\, (a, s') = x\, (s|_{\mathsf{R}_F})\, \mathsf{in}\; \mathsf{let}\, (b, s'') = (f\, a)\, ((s \lhd s')|_{\mathsf{R}_G})\, \mathsf{in}\, (b, s' \lhd s'') \qquad \mathsf{unit}^{\emptyset}\, x = \lambda s.\, (x, s)$$

The incoming store s of extend is restricted into substore $s|_{\mathsf{R}_F}$ for the reads made by x. Operator $s \lhd s'$ merges stores preferring the right-hand-side mapping for locations $\rho \in \mathrm{dom}\,(s) \cup \mathrm{dom}\,(s')$, which is then restricted by $|_{\mathsf{R}_G}$ to the substore of locations read by $f\, a$. Locations read by $f\, a$ use values in s' in preference ($\lhd$) to those in s. Finally the resulting store prefers writes from s'' over those in s'.

Note that $\mathsf{unit}^{\emptyset}$ is isomorphic to the identity as $\mathsf{R}_\emptyset = \mathsf{W}_\emptyset = \emptyset$, and so s there is the empty mapping. Hence any denotation in $A \rightarrow \mathsf{State}_\emptyset B$ is necessarily a pure function—useful for enabling various optimisations relating to purity.

Graded monadic semantics Graded monads enable effect-directed semantics, where syntactic effect labels are incorporated as semantic objects; type-and-effect judgements are mapped to denotations of the form $[\![\Gamma \vdash e : \tau, F]\!] : [\![\Gamma]\!] \rightarrow \mathsf{T}_F[\![\tau]\!]$. This gives a monadic semantics for Wadler and Thiemann's *Monad* language, generalised to a monoidal effect system, where syntactic type constructors $M^F\tau$ are mapped to the (semantic) graded monad $\mathsf{T}_F[\![\tau]\!]$. The semantics of *Monad* is analogous to that of the previous section:

$$[\![\Gamma \vdash_M \mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, e' : M^{F \bullet G}\tau']\!] : [\![\Gamma]\!] \rightarrow \mathsf{T}_{F \bullet G}[\![\tau']\!] = \lambda\gamma\,.\, \mathsf{extend}^{F,G}\, (\lambda v.\, [\![\Gamma, x{:}\tau \vdash_M e' : M^G\tau']\!]\ \gamma[x \mapsto v])\ ([\![\Gamma \vdash_M e : M^F\tau]\!]\, \gamma)$$

$$[\![\Gamma \vdash_M \langle e \rangle : M^{\emptyset}\tau]\!] : [\![\Gamma]\!] \rightarrow \mathsf{T}_\emptyset[\![\tau]\!] = \mathsf{unit}^I \circ [\![\Gamma \vdash_M e : \tau]\!]$$

Via the translation from *Effect* to *Monad*, this also provides an effect-directed semantics of the effectful simply-typed λ-calculus via graded monads.

The axioms of a (strong) graded monad provide a sound semantics, with respect to a standard β-equational theory, as shown by Katsumata [16].

Grading for semantics-and-analysis co-design The correspondence between the effect annotations and the indices of a graded monad provides a kind of *co-design* principle for defining semantics and effect systems: start with a graded monad and follow the shape of a usual monadic semantics; an effect system for the term language emerges from the indices and the inductive definition of the semantics. Conversely, start with an effect system, say the one in the introduction for music. An effect-graded semantics then requires semantic operations annotated by each of the effect operations used,[8] with a structure that reflects that of the effect system. In this way, graded approaches aid a kind of co-design process between analysis and semantics.

[8] Section 3 addresses the subtlety here that if-then-else is reflected with *operation* $+$ in the music effect while Katsumata's graded monads use a *relation* $\sqsubseteq$.

This relationship extends to the equational theory of a language and the axioms of its underlying semantic structures. For example, consider the following equation (relative to a type derivation) of the *Monad* language:

$$\Gamma \quad \vdash_M \quad \mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, \langle x \rangle \;\equiv\; e \quad : \; M^F \, \tau$$

This syntactic equality relies on the monoid axiom $F \bullet I = F$ to ensure that the types of the left- and right-hand side ($M^{F \bullet I} \, \tau$ and $M^F \, \tau$ respectively) are equal. Soundness of the semantics, with respect to this equation, thus requires that $[\![\Gamma \vdash \mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, \langle x \rangle : M^{F \bullet I} \, \tau]\!] = [\![\Gamma \vdash e : M^F \, \tau]\!]$. The proof of this denotational equality uses the graded monad axiom [M1], which itself uses the monoidal axiom $F \bullet I = F$ (see diagram (2), p. 12). A proof search procedure (whether by-hand or automatic) can be guided by the link between the syntax of effect annotations and their corresponding indices (grades) in the semantics: the required semantic axioms are those which witness the syntactic axioms.

Terminology Graded monads have been previously called *parameterised effect monads* by Katsumata [16] (relating to the work of Mellies [20]) and *indexed monads* [28]. We opt for the name *graded monad* here to avoid confusion with the idea of indexed monads in topos theory and the parameterised monads of Atkey [3] or *parametricity.* The *graded* terminology has recently become a popular name for this concept [36, 21].

2.5 Type-directed and effect-directed analysis and semantics

As noted earlier, the semantics of an expression may depend on its associated type derivation. For example, in the presence of a derivation of $\vdash \lambda x.x : \mathsf{int} \to \mathsf{int}$, the term $\lambda x.x$ is interpreted as the identity function on $\mathbb{Z}$. We call these *type-directed* semantics or analyses (more precisely *type-derivation-directed*).

A type-directed semantics tends to simplify definitions and reasoning. For example, consider an untyped denotational semantics on a Scott domain satisfying $D \cong \mathbb{Z} + (D \to D)$ *vs.* a type-directed semantics where each type has a distinct domain, *e.g.*,$D_{\mathsf{int}} = \mathbb{Z}$ and $D_{\sigma \to \tau} = D_\sigma \to D_\tau$. The former has a more complicated semantics, with injections on sum types and deconstructors to identify semantically meaningful terms, which are unnecessary in the latter.

Our notion of *type-and-effect-directed* semantics (or just *effect-directed* for brevity) naturally extends this idea to effect annotations. In such a language the apply function $\lambda f.\lambda x.f\, x$ has many effects and types, in particular every instance of $(\tau \xrightarrow{F} \tau') \to \tau \xrightarrow{F} \tau'$ for types τ, τ' and effects F. Any expression e of this type can be interpreted monadically as belonging to semantic domain $([\![\tau]\!] \to \mathsf{T}[\![\tau']\!]) \to \mathsf{T}([\![\tau]\!] \to \mathsf{T}[\![\tau']\!])$ for some monad T. Via graded monads, an effect-directed semantic domain refines this to $([\![\tau]\!] \to \mathsf{T}_F[\![\tau']\!]) \to \mathsf{T}_I([\![\tau]\!] \to \mathsf{T}_F[\![\tau']\!])$. So if we knew, for example, that F is the trivial pure effect I and the graded monad is such that $\mathsf{T}_I A = A$ then an effect-directed semantics could simply interpret e as a value in $([\![\tau]\!] \to [\![\tau']\!]) \to [\![\tau]\!] \to [\![\tau']\!]$. This was seen with the State graded monad in Example 2 (p. 13).

3 Control-flow effects and monad limitations

Section 2 developed, in tutorial style, the theory for effects expressed monadically, including grading (precise denotational models of types and effects)—but limited to the situation where effect annotations form an ordered monoid $(\mathcal{F}, I, \bullet, \sqsubseteq)$, expressing sequential composition. This leaves the additional effect operators (+, &) for alternation (conditionals) and parallel composition (which Section 1 argued were essential for modelling music) and the question of how to incorporate them into an effect algebra and graded semantics. We defer treatment of (&) to Section 4, and here focus on the rather interesting issues centred around the question of how well can monads capture conditionals—both semantically and in terms of relating an (+)-enriched effect algebra to monad grading.

We explore three specific issues. Section 3.1 examines how well monads can give a general semantics to if-then-else and similar control-flow operations; Section 3.2 shows that while Wadler and Thiemann's work only handles a semilattice of effects, Katsumata's graded monads use an *ordered* monoid of effects and which can very nearly capture (+) as well as ($\bullet$). These two issues turn out to be two sides of the same coin. Finally, Section 3.3 argues that certain operations augmenting the usual monad operations of unit and extend should be characterised as *control-flow operators* and thus merit being operations on the augmented effect monoid too. This all sets the scene for Section 4 where *joinads* (a specific extension to monads), graded by a control-flow algebra dubbed *joinoids* (augmenting monoids), complete the development.

3.1 (Graded) monadic semantics for conditionals

Consider a denotational interpretation for (type derivations over) conditionals $[\![\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : \tau]\!]$ where e, e', e'' may be effectful expressions (in a simple system where effects do not form part of judgements). Following the denotational tradition, the denotation of a compound expression is some function of those of its sub-expressions, traditionally expressed for some $\underline{\mathsf{COND}}$ as:

$$[\![\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'']\!] = \underline{\mathsf{COND}}([\![e]\!], [\![e']\!], [\![e'']\!])$$

Formally, our semantics interprets type derivations, hence is more accurately:

$$[\![\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : \tau]\!] = \underline{\mathsf{COND}}([\![\Gamma \vdash e : \mathsf{bool}]\!], [\![\Gamma \vdash e' : \tau]\!], [\![\Gamma \vdash e'' : \tau]\!])$$

for some $\underline{\mathsf{COND}}_{X,A} : (X \to \mathsf{T}\,\mathbb{B}) \times (X \to \mathsf{T}A) \times (X \to \mathsf{T}A) \to (X \to \mathsf{T}A)$ instantiated at $X = [\![\Gamma]\!]$, the space of environments, and $A = [\![\tau]\!]$. Given that if-then-else does not bind variables, this can be written $\lambda(x, y, z).\lambda\gamma.\mathsf{COND}(x\,\gamma, y\,\gamma, z\,\gamma)$ where $\mathsf{COND}_A : \mathsf{T}\,\mathbb{B} \times \mathsf{T}A \times \mathsf{T}A \to \mathsf{T}A$. It is convenient to use the notation COND on computations and $\underline{\mathsf{COND}}$ in semantic rules to avoid the clutter of γ.

In categories which have coproducts (sum types), and hence booleans $\mathbb{B}$, the semantics of effectful if-then-else can be simply *derived* from existing monad

operations. Such categories (including our set-based framework) have a parametric operation[9] $\mathsf{cond}_A : \mathbb{B} \times A \times A \to A$ with axioms: $\mathsf{cond}(\mathsf{true}, x, y) = x$ and $\mathsf{cond}(\mathsf{false}, x, y) = y$. By instantiating cond at $\mathsf{T}A$ to get $\mathsf{cond}_{\mathsf{T}A} : \mathbb{B} \times \mathsf{T}A \times \mathsf{T}A \to \mathsf{T}A$, we obtain one possible definition for COND using sequential composition:

$$\mathsf{COND}_A(x, y, z) = \mathsf{extend}_{\mathbb{B},A}\ (\lambda b.\, \mathsf{cond}_{\mathsf{T}A}\ (b, y, z))\ x$$

This derived semantics is *dichotomous*—it encodes the laws that if-then-else returns one of its branches (*i.e.*, $\mathsf{if}\,\mathsf{true}\,\mathsf{then}\,e\,\mathsf{else}\,e' \equiv e$ and $\mathsf{if}\,\mathsf{false}\,\mathsf{then}\,e\,\mathsf{else}\,e' \equiv e'$)—and *sequential*—it encodes the axiom (assuming fresh variable x) that

$$\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' \quad \equiv \quad \mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ e'\ \mathsf{else}\ e''$$

There are other reasonable non-dichotomous or non-sequential semantics we may wish to model, thus the derived model above is quite limited. Prolog-style backtracking provides a good example of a non-dichotomous if-then-else as both branches are explored in some fixed order (leading to non-commutative + operator on effects); we could even imagine a variant of if-then-else where the boolean indicates whether the then branch is explored before or after the else branch. Similar non-commutativity arises for case expressions with overlapping patterns. Non-sequential if-then-else is exemplified in "*parallel if*" which satisfies the semantic property (for a system where non-termination is an effect):

$$[\![\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'']\!] = [\![e']\!] \quad \text{if} \quad [\![e']\!] = [\![e'']\!] \quad \text{(and even if } [\![e]\!] = \bot\text{)}.$$

Speculative behaviour (with software-transactional memory for rolling-back effects) is a non-sequential if-then-else. Non-dichotomous variants may also give a collecting semantics which captures computation *trees* instead of single traces—the derived semantics using cond_A can only capture a dynamic trace.

In the music example, we used operators $\bullet$, $+$ and $\&$ to model the effects of the sequencing, conditional and parallel language constructs. However, for full generality, we should use a ternary operator $?{+}(F, G, H)$ to capture the effects of conditionals. Nonetheless, sequential semantic variants of if-then-else require that $?{+}(F, G, H) = F \bullet (G + H)$. The $+$ operator can be defined $G + H = ?{+}(I, G, H)$ where I is the identity of $\bullet$. Most interpretations of if-then-else in the rest of the paper are sequential, but not all are dichotomous—Example 4 shows a non-standard 'synchronous' semantics for conditionals in music, where duration of if-then-else is the maximum of both branches. In short, we choose not to require conditionals to have the monadic derived semantics.

A language with effects and parallelism similarly requires a semantic model for how effectful computation are composed in parallel. While we noted *one* interpretation of conditionals can be derived from coproducts in the domain, there is no analogous derived structure to be found for parallelism since the 'obvious' operation $par_{A,B} : A \times B \to A \times B$ can only be the identity function (by parametricity). In contrast, an operation $par'_{A,B} : \mathsf{T}A \times \mathsf{T}B \to \mathsf{T}(A \times B)$ (called merge in the next section) for some monad T can perform an effect-specific implementation of parallelism, *e.g.*, arbitrary effect interleaving.

[9] Categorically, a natural transformation, derived from coproducts with $\mathbb{B} = 1 + 1$.

3.2 Effect operators for conditional

Until now, monads have been graded by an ordered effect *monoid*. But we now want to model richer effect operators such as that of if-then-else as above, with $F \bullet (G + H)$ or $?\!+(F, G, H)$ for non-sequential variants.

It turns out that a special case for + can nearly be derived from the ordering structure. Katsumata writes: "When giving an effect system, it is desirable to have the join operator on effects (... [augmenting the] monoid structure), because we can use it to unify the effects given to different branches of case expressions." [16] Suppose all joins (least upper bounds) of $\sqsubseteq$ exist (not an existing requirement for graded monads), then we can define + to be the join operator—thus obtaining an effect algebra $(\mathcal{F}, I, \bullet, +)$. Monotonicity of $\bullet$ (w.r.t. $\sqsubseteq$) becomes a distributive law $x \bullet (y + z) = (x \bullet y) + (x \bullet z)$.

A general effect-directed denotational semantics for if (with ternary ?+ on effects) is then captured by a graded version of COND, to wit $\mathsf{COND}_A^{F,G,H} : \mathsf{T}_F \mathbb{B} \times \mathsf{T}_G A \times \mathsf{T}_H A \to \mathsf{T}_{?+(F,G,H)} A$, lifted on environments $[\![\Gamma]\!]$ to $\underline{\mathsf{COND}}_{[\![\Gamma]\!],A}^{F,G,H}$

$$
\begin{aligned}
&[\![\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : \tau, ?\!+(F, G, H)]\!] \\
&\quad = \underline{\mathsf{COND}}_{[\![\Gamma]\!],[\![\tau]\!]}^{F,G,H} ([\![\Gamma \vdash e : \mathsf{bool}, F]\!],\ [\![\Gamma \vdash e' : \tau, G]\!],\ [\![\Gamma \vdash e'' : \tau, H]\!])
\end{aligned}
$$

We can then analogously construct a graded monadic version of the *derived* (non-general) semantics of if-then-else. Assuming the effect ordered monoid additionally has all least upper bounds (written + as above) we set $?\!+(F, G, H) = F \bullet (G + H)$ and define $\mathsf{COND}_A^{F,G,H}$ by:

$$
\begin{aligned}
\mathsf{COND}_A^{F,G,H}(x, y, z) = \mathsf{extend}^{F,G+H}(\lambda b.\ \mathsf{cond}_{\mathsf{T}_{G+H} A}\,(b, &\ \mathsf{coerce}^{G,G+H}\ y, \\
&\ \mathsf{coerce}^{H,G+H}\ z))\ \ x
\end{aligned}
$$

Remark 2. Note that if we instantiate $\mathcal{F}$ to be *sets* of effects with $(\bullet) = (\cup)$ and $I = \{\,\}$ then interpreting $(+) = (\cup)$ and $(\sqsubseteq) = (\subseteq)$ produces a traditional 'may' set-based effect system, while interpreting $(+) = (\cap)$ and $(\sqsubseteq) = (\supseteq)$ gives a 'must' form of effect system. This does not appear to be generally appreciated, and shows that Katsumata's graded-monad-with-an-ordering approach to effect systems captures both the $(\bullet)$ and $(+)$ operators introduced by the Nielsons.

So, amusingly, monads provide *one* (derived) semantics for conditionals, and semilattice-ordered-monoids provide *one* way of separating the effects for $\bullet$ and +, but neither is fully general. Just as we argued that not all semantics for if-then-else could be factored via parametric conditional, we also argue that not all (+) operations on augmented effect monoids can be expressed as the least upper bound of an ordering $(\sqsubseteq)$ originally envisaged as sub-effecting. The required property is merely that $F \sqsubseteq F + G$ and $G \sqsubseteq F + G$ (we argued in Section 3.1 that + may not be commutative, and we also do not require its idempotency). Multisets of effects provide an example: we may naturally define (+) to capture addition on multiplicities, while $(\sqcup)$ captures maximum on multiplicities.

3.3 Control-flow operators

The semantics of Section 2 is abstract, using a (graded) monad to sequentially compose effects. The semantics can then be specialised to a particular notion of side effect (*e.g.*, state, exceptions) by instantiating the monad and providing effect-specific constants, such as with the state monad (Example 2) and the read and write operations. The denotations of these additional operations are necessarily of the form $A \to \mathsf{T}B$ (a *Kleisli morphism*) so that they can be composed via the monadic structure of the semantics. In a graded setting, these denotations introduce members of $\mathcal{F}$, *e.g.* $[\![\mathsf{read}]\!]_{\tau,\rho} : [\![\mathsf{ref}\,\tau]\!] \to \mathsf{T}_{\{\mathsf{rd}\rho\}}\,[\![\tau]\!]$.

We observe that any function with negative occurrences of the computation type $\mathsf{T}A$ (*i.e.*, left of a function arrow) cannot be the denotation of an expression. This is because the semantics generates only Kleisli morphisms and the interpretation of types only introduces T on the right hand side of an arrow. Instead, such operators are *effect control-flow operators*, *e.g.*, $\mathsf{COND}_A : \mathsf{T}\,\mathbb{B} \times \mathsf{T}A \times \mathsf{T}A \to \mathsf{T}A$.

From a different perspective, that of control-flow graphs, computation values $\mathsf{T}A$ correspond to closed basic blocks and functions $A \to \mathsf{T}B$ to open blocks with incoming dataflow which are composed by extend; unit constructs an empty block. By contrast, operations whose type has $\mathsf{T}A$ appearing to the left of a function arrow (*e.g.* $\mathsf{T}A \to \ldots$) correspond to control-flow operations. For example, the type $\mathsf{T}A \times \mathsf{T}A \to \mathsf{T}A$ corresponds to an operator which merges basic blocks for branching, and indeed bind (which is extend with its arguments flipped) $\mathsf{bind} : \mathsf{T}A \to (A \to \mathsf{T}B) \to \mathsf{T}B$ is the primitive control-flow operator for sequential composition which appends a closed basic block to an open basic block, creating a new composite block.

It is clear from our definition that $\mathsf{COND}_A : \mathsf{T}\,\mathbb{B} \times \mathsf{T}A \times \mathsf{T}A \to \mathsf{T}A$ is a control-flow operator. Similarly, $\mathsf{cond}_A : \mathbb{B} \times A \times A \to A$ is a control operator when instantiated at $A = \mathsf{T}\,B$ but can also be a non-control operator: *e.g.*, when $A = \mathbb{Z}$ it is effectively a multiplexer. Due to parametricity, $\mathsf{cond}_{\mathsf{T}\,B}$ cannot however do any 'interesting control flow', it must either select one branch or another, while a function $\mathsf{cond}'_A : \mathbb{B} \times \mathsf{T}A \times \mathsf{T}A \to \mathsf{T}A$ of the same type but parametric only in A *could* combine effects from both branches. Similarly, parametricity means instantiations of $par_{A,B} : A \times B \to A \times B$ cannot exhibit observable parallelism—this needs a function typed $par_{A,B} : \mathsf{T}A \times \mathsf{T}B \to \mathsf{T}(A \times B)$.

Control-flow operators also provide a link to abstract interpretation [6]. Primitive effectful operations in the concrete semantics are abstracted to effects in an effect algebra. Control-flow operators compose these primitive effectful operations in the concrete semantics and are abstracted to operations of the effect algebra. This yields effect monoids (or effect *joinoids* later in the paper). Effect-graded semantics provide a form of concrete "correct by construction" semantic models, corresponding to abstract effect algebras.

4 Joinads and rich effect systems for control flow

As discussed in the previous section, for richer languages (with parallelism, music, or speculative evaluation) we need richer semantic structure to model effects,

one that provides control combinators additional to sequential composition—most importantly for branching and parallel composition. The structure of a *joinad* does just this: extending monads with operations for modelling alternation (conditionals) and parallelism [29, 32].

This paper introduces a variant of joinads (Section 4.1), based on the COND conditional operator of Section 3, instead of the classical formulation (Section 4.3) based on choose and fail operations. We name these *conditional joinads*—more fully "joinads with conditional instead of choose and fail". The choose-and-fail variant is convenient for capturing pattern matching (its original motivation) but the conditional formulation is more flexible and convenient here.

We repeat the development of Section 2 showing how conditional joinads provide a *type-directed* semantics for conditionals and parallel composition (Section 4.2). We compare this with classical joinads (Section 4.3) which model conditionals similarly to the derived model of Section 3. We then introduce *graded conditional joinads* to give a more precise *effect-directed* semantics (Section 4.4).

4.1 Joinads and conditional joinads

Many monads are equipped with additional *combinators* that provide different ways of composing computations compared to the standard sequential composition guaranteed by a monad. This is particularly so in source-level uses of monads on data types in Haskell. The original motivation for joinads was to capture common combinators for parallel, concurrent, and reactive programming (and then develop a new notation for programming with joinads) [29, 32]. This is similar to our aim of capturing additional common ways of composing effectful computations. To quote the original work:

> "We identify *joinads*, an abstract notion of computation that is stronger than monads and captures many [of their] ad-hoc extensions. In particular, joinads are monads with ... additional operations: one of type $M\, a \to M\, b \to M\, (a \times b)$ captures various forms of parallel composition [and] one of type $M\, a \to M\, a \to M\, a$ that is inspired by choice ... Algebraically, [these] operations form a near-semiring with commutative multiplication." (Petricek, Mycroft, Syme [29])

The meaning of the operations differs for various notions of computation. For concurrency effects (the obvious interpretation), parallel composition means running tasks in parallel and choice is non-determinism. However, the operations also make sense for *parsers*— parallel composition means that two parsers both recognise an input, and choice means at least one parser recognises it [29].

The joinad structure appears in many libraries, for example, Mirage, a Library Operating System written in OCaml [19]. Mirage is effectively a large parameterised module, which when applied to modules representing the underlying hardware abstraction, can execute equally well as an application under Linux or as an entire OS on a bare-metal virtual machine. Its core is based on the co-operative threading library Lwt [38], which exhibits the joinad structure.

In Lwt, processes are expressed monadically (using return and extend as usual) for their sequential parts; the `<&>` (called 'merge' here) and `<?>` ('choose' here) provide parallelism and first-to-arrive alternation respectively.

4.2 Type-directed semantics using conditional joinads

As discussed in Section 3, the derived (graded) monadic semantics for conditionals is restrictive. Instead, conditional joinads are more flexible, allowing the semantics of conditionals to be parameterised. As an intermediate between monads and conditional joinads, we first extend monads with a conditional operation.

For brevity, we lift operations to environment-passing style, where for some $\mathsf{OP} : A \times B \to C$ then $\underline{\mathsf{OP}}_X = \lambda\gamma.\mathsf{OP}(f\,\gamma, g\,\gamma) : (X \to A) \times (X \to B) \to (X \to C)$. Such X are implicitly instantiated to $[\![\Gamma]\!]$ to avoid clutter.

Definition 4 (Conditional monad). *Given booleans $\mathbb{B}$ in the base category $\mathcal{C}$, a* conditional monad *extends a monad T on $\mathcal{C}$ with the parametric operation (natural transformation) $\mathsf{mcond}_A : \mathsf{T}\mathbb{B} \times \mathsf{T}A \times \mathsf{T}A \to \mathsf{T}A$ satisfying axioms of associativity* (3, 4)*, commutativity* (5)*, units* (6, 7)*, and right-distributivity* (8)*:*

$$\mathsf{mcond}_A(\mathsf{unit}_{\mathbb{B}}\ b, x, \mathsf{mcond}_A(\mathsf{unit}_{\mathbb{B}}\ b', y, z)) \equiv \mathsf{mcond}_A(\mathsf{unit}_{\mathbb{B}}\ (b \vee b'), \mathsf{mcond}_A(\mathsf{unit}_{\mathbb{B}}\ b, x, y), z) \tag{3}$$

$$\mathsf{mcond}_A(\mathsf{unit}_{\mathbb{B}}\ b, \mathsf{mcond}_A(\mathsf{unit}_{\mathbb{B}}\ b', x, y), z) \equiv \mathsf{mcond}_A(\mathsf{unit}_{\mathbb{B}}\ (b \wedge b'), x, \mathsf{mcond}_A(\mathsf{unit}_{\mathbb{B}}\ b, y, z)) \tag{4}$$

$$\mathsf{mcond}_A(\mathsf{unit}_{\mathbb{B}}\ b, x, y) \equiv \mathsf{mcond}_A(\mathsf{unit}_{\mathbb{B}}\ \neg b, y, x) \tag{5}$$

$$\mathsf{mcond}_A(\mathsf{unit}_{\mathbb{B}}\ \mathsf{true}, x, \mathsf{unit}_A\ y) \equiv x \tag{6}$$

$$\mathsf{mcond}_A(\mathsf{unit}_{\mathbb{B}}\ \mathsf{false}, \mathsf{unit}_A x, y) \equiv y \tag{7}$$

$$\mathsf{extend}_{A,B}\ f\ \mathsf{mcond}_A(b, x, y) \equiv \mathsf{mcond}_B(b, \mathsf{extend}_{A,B}\ f\ x, \mathsf{extend}_{A,B}\ f\ y) \tag{8}$$

The idea behind mcond_A is that it generalises the standard conditional $\mathsf{cond}_A : \mathbb{B} \times A \times A \to A$ to a true control-flow operator with respect to effects. The two unit axioms (6, 7) are a restricted form of the standard (if-β) dichotomous behaviour of cond (that is, $\mathsf{cond}_A(\mathsf{true}, x, y) = x$ and $\mathsf{cond}_A(\mathsf{false}, x, y) = y$) when the guard and the unselected branch are both pure (*i.e.*, factor through unit).

The mcond operation provides a general operation for modelling the syntactic if construct from the source language. Given a typing-derivation for the term $\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e''$, a type-directed semantics is obtained by directly passing the semantics of sub-expressions to the effect control-flow operator mcond_A:

$$\begin{aligned} &[\![\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : \tau, F \bullet (G + H)]\!] \\ &\quad = \underline{\mathsf{mcond}}_{[\![\tau]\!]}\ ([\![\Gamma \vdash e : \mathsf{bool}, F]\!], [\![\Gamma \vdash e' : \tau, G]\!], [\![\Gamma \vdash e'' : \tau, H]\!]) \end{aligned} \tag{9}$$

Proposition 1. *A monad T on a category $\mathcal{C}$ with coproducts (providing booleans $\mathbb{B}$ and the cond_A operation) is a conditional monad, where mcond_A is defined:*

$$\mathsf{mcond}_A(x, y, z) = \mathsf{extend}_{\mathbb{B},A}\ (\lambda b.\, \mathsf{cond}_{\mathsf{T}A}\,(b, y, z))\ x$$

The proof follows straightforwardly from the monad and cond_A axioms. This gives the standard derived semantics for conditionals.

Definition 5 (Conditional joinads). *A conditional joinad extends a conditional monad* T *with a parametric operation* $\mathsf{merge}_{A,B} : \mathsf{T}A \times \mathsf{T}B \rightarrow \mathsf{T}(A \times B)$ *satisfying associativity* (10)*, commutativity* (11)*, unit* (12)*, and distributivity* (13)*:*

$$\mathsf{merge}_{A\times B,C}(\mathsf{merge}_{A,B}(x,y),z) \equiv \mathsf{map}\ \mathsf{assoc}\ \mathsf{merge}_{A,B\times C}(x, \mathsf{merge}_{B,C}(y,z)) \quad (10)$$

$$\mathsf{merge}_{A,B}(x,y) \equiv \mathsf{map}\ \mathsf{swap}\ (\mathsf{merge}_{B,A}(y,x)) \quad (11)$$

$$\mathsf{merge}_{A,B}(\mathsf{unit}_A\ x, y) \equiv \mathsf{map}\,(\lambda y'.(x,y'))\, y \quad (12)$$

$$\mathsf{merge}_{A,B}(\mathsf{mcond}_A(b,x,y),z) \equiv \mathsf{mcond}_A(b, \mathsf{merge}_{A,B}(x,z), \mathsf{merge}_{A,B}(y,z)) \quad (13)$$

where map *is the morphism-mapping of the functor* T*,* i.e.*, given* $f : A \rightarrow B$ *then* $\mathsf{map}\ f : \mathsf{T}A \rightarrow \mathsf{T}B$*, and* $\mathsf{assoc}\ (a,(b,c)) = ((a,b),c)$ *and* $\mathsf{swap}\ (a,b) = (b,a)$*.*

Categorically, merge therefore witnesses that T is a symmetric monoidal functor with additional right-distributivity with mcond.

We use the merge operation directly for the semantics of the par construct:

$$[\![\Gamma \vdash e\ \mathsf{par}\ e' : \tau \times \tau', F\&G]\!] = \underline{\mathsf{merge}}_{[\![\tau]\!],[\![\tau']\!]}([\![\Gamma \vdash e : \tau, F]\!], [\![\Gamma \vdash e' : \tau', G]\!]) \quad (14)$$

We now have a fully parameterised semantics for if-then-else and par via conditional joinads. For music, notes can be played in parallel; for concurrency, two tasks can be run in parallel (multiple threads) or using interleaved concurrency.

The axioms of a conditional joinad include the commutativity of merge (as in the original joinad formulation); this has pros and cons. On the one hand, commutativity provides a natural intuition for parallel execution (both true parallelism and non-deterministic interleaving). On the other hand, commutativity forbids various kinds of static scheduling by sequencing, *e.g.*, left-first or right-first scheduling, since sequential composition is typically not commutative.

Theorem 1 (Soundness). *Given a monadic semantics for the simply-effect-and-typed* λ*-calculus with a conditional joinad semantics for* if *(Definition 4) and* par *(Definition 5) then, for all* e, e', Γ, τ, F*:*

$$\Gamma \vdash e \equiv e' : \tau, F \quad \Rightarrow \quad [\![\Gamma \vdash e : \tau, F]\!] = [\![\Gamma \vdash e' : \tau, F]\!]$$

with respect to the following equational theory defined by $\equiv$ *(we omit the typing), augmenting CBV* β*-equality:*

$$
\begin{array}{rl}
(\text{IF}\beta 1') & \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e\ \mathsf{else}\ x \equiv e \\
(\text{IF}\beta 2') & \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ x\ \mathsf{else}\ e' \equiv e' \\
(\text{IF-DIST-PAR}) & (\mathsf{if}\ b\ \mathsf{then}\ e\ \mathsf{else}\ e')\ \mathsf{par}\ e'' \equiv \mathsf{if}\ b\ \mathsf{then}\ (e\ \mathsf{par}\ e'')\ \mathsf{else}\ (e'\ \mathsf{par}\ e'') \\
(\text{IF-DIST-SEQ}) & \mathsf{let}\ x = (\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'')\ \mathsf{in}\ e''' \\
 & \qquad \equiv \mathsf{if}\ e\ \mathsf{then}\ (\mathsf{let}\ x = e'\ \mathsf{in}\ e''')\ \mathsf{else}\ (\mathsf{let}\ x = e''\ \mathsf{in}\ e''') \\
(\text{PAR-PURE}) & x\ \mathsf{par}\ e \equiv (x, e) \\
(\text{PAR-SYM}) & e\ \mathsf{par}\ e' \equiv \mathsf{swap}\ (e'\ \mathsf{par}\ e) \\
(\text{PAR-ASSOC}) & e\ \mathsf{par}\ (e'\ \mathsf{par}\ e'') \equiv \mathsf{assoc}\ ((e\ \mathsf{par}\ e')\ \mathsf{par}\ e'')
\end{array}
$$

where in (IF-DIST-PAR) b *is pure* i.e. $\Gamma \vdash b : \mathsf{bool}, I$. *In* (PAR-PURE) *the left-hand side is a pure computation represented with a variable* x *thus* $\Gamma \vdash x : \tau, I$. *Further,* (IF-$\beta 1'$)(IF-$\beta 2'$) *have pure terms (variables) in the unselected branches.*

Proof. By induction on $\equiv$ and following from the conditional joinad axioms.

4.3 Classical joinads

We introduced the conditional variant of joinads. The original joinad structure [29, 32] has instead of $\mathsf{mcond}_A : \mathsf{T}\,\mathbb{B} \times \mathsf{T}A \times \mathsf{T}A \to \mathsf{T}A$ two operations called choose and fail (also known as the MonadPlus type class in Haskell [10]) of type:

$$\mathsf{choose}_A : \mathsf{T}A \times \mathsf{T}A \to \mathsf{T}A \qquad \mathsf{fail}_A : \mathsf{void} \to \mathsf{T}A$$

The choose operation models a choice between two computations. The fail operation creates a failing computation that is the unit element with respect to choose and choose must be associative, *i.e.* these two operations form a monoid on $\mathsf{T}A$. Furthermore, merge must be right-distributive with choose_A and fail_A absorbing with respect to extend and merge, that is, applying extend to a computation that fails produces a computation equivalent to fail, and failure of one parallel branch makes both fail. Algebraically, this means that operations form a *near-semiring* with choose as addition and merge as multiplication [29]. Similar structure is shown in the work of Rivas *et al.* [35]. This guarantees various desirable syntactic equivalences when used for a language semantics.

These operations (together with their axioms) let us encode conditionals as:

$$\begin{aligned}\mathsf{mcond}_A(x, y, z) = \mathsf{choose}_A\ &(\mathsf{extend}_{\mathbb{B},A}\ (\lambda b.\ \mathsf{cond}_A(b, y, \mathsf{fail}_A)\\ &(\mathsf{extend}_{\mathbb{B},A}\ (\lambda b.\ \mathsf{cond}_A(b, z, \mathsf{fail}_A))\ x)\end{aligned}$$

Here, both branches are turned into computations that fail if they should *not* be executed (and succeed otherwise). This definition of joinads was inspired by ML-style pattern matching and so the fail operation represents a *commit point*. Given suitable definitions for choose and fail, the above definition of mcond *can* capture the derived (from extend and cond) monadic semantics, but is also rather more general in that it can also use a free joinad (and hence a non-dichotomous conditional semantics)—leading to a trace containing a (free version of) choice at each conditional branch, but where one of the branches is trivially fail.

Remark 3. For atomic/independent computations, parallelism can be modelled as a choice between the two ways of sequencing the computations (see Section 1, also Milner [22]) *e.g.* merge (for par) can be defined in terms of choose and extend:

$$\begin{aligned}\mathsf{merge}_{A,B}(x, y) = \mathsf{choose}_{A\times B}\ &(\mathsf{extend}\ (\lambda a.\, \mathsf{extend}\ (\lambda b.\ \mathsf{unit}\,(a, b))\ y)\ x)\\ &(\mathsf{extend}\ (\lambda b.\, \mathsf{extend}\ (\lambda a.\ \mathsf{unit}\,(a, b))\ x)\ y)\end{aligned}$$

We might consider this as a candidate for modelling parallel composition, thus requiring fewer semantic primitives for a language with conditionals and parallelism. However, the above does not capture the semantics for our music language (where playing notes in parallel produces a different sound than that of

any sequencing) or for languages with parallelism based on multiple threads. More flexibility is therefore provided by making parallelism a separate semantic notion via the joinad (or conditional joinad) merge operation.

The next section generalises conditional joinads to a *graded* form to allow effect systems to refine the semantics of conditionals and parallelism.

4.4 Control-flow algebras and graded joinads

Traditional set-based 'may' effect systems use a semilattice of effects, which we see as a special case of effect *monoids*. Effect monoids are a simple *control-flow* algebra, capturing just sequential control flow. Monads can be seen as an instance of effect monoids, but over endofunctors (type constructors) encoding effects. These syntactic and semantic descriptions of effects are unified via *graded monads* to give an effect-directed semantics (Section 2).

Section 1 defined effect systems capable of capturing choice and parallelism via a rich control-flow algebra of effects, as suggested by Nielson and Nielson. We formalise this class of control-flow algebra below, calling it a *joinoid*. Both the effect systems of Section 1 and conditional joinads (Section 4.1) are instances of this control-flow algebra: at the level of syntax (analysis/types) and semantics respectively. However, the link between a joinoid-based effect system and its semantics (via conditional joinads) has only been loosely coupled and intuitive so far. This section introduces the *graded conditional joinad* structure (the joinad analogue of graded monads) to make this correspondence concrete, providing an effect-directed semantics for effect systems over a *joinoid* control-flow algebra.

Definition 6 (Joinoid). *Let $\mathcal{F}$ be a set with $I \in \mathcal{F}$, binary operations $\bullet$ and $\&$, a ternary operator $?\!+$ and a binary relation $\sqsubseteq$. Then $(\mathcal{F}, \bullet, I, \&, ?\!+, \sqsubseteq)$ is a* joinoid control-flow algebra *(*joinoid *for short) if, letting $F + G = ?\!+(I, F, G)$:*

- *$(\mathcal{F}, \bullet, I)$ is a monoid, representing sequential composition and purity;*
- *$(\mathcal{F}, \&, I)$ is a commutative monoid, representing parallel composition;*
- *$(\mathcal{F}, +)$ is a semigroup, representing choice between two conditional; branches*
- *with right-distributivity axioms:*

$$(F + G) \bullet H = (F \bullet H) + (G \bullet H) \qquad (F + G) \,\&\, H = (F \,\&\, H) + (G \,\&\, H)$$

- *all operations are monotonic with respect to $\sqsubseteq$.*

Definition 7 (Graded conditional joinads). *Given a joinoid on $\mathcal{F}$, a* graded conditional joinad *is a graded monad T for the ordered monoid $(\mathcal{F}, \bullet, I, \sqsubseteq)$ together with the following two parametric operations:*

$$\mathsf{merge}^{F,G}_{A,B} : \mathsf{T}_F A \times \mathsf{T}_G B \to \mathsf{T}_{F\&G}(A \times B)$$
$$\mathsf{mcond}^{F,G,H}_{A} : \mathsf{T}_F \mathbb{B} \times \mathsf{T}_G A \times \mathsf{T}_H A \to \mathsf{T}_{?+(F,G,H)} A$$

which satisfy analogous equations to a conditional joinad (Definition 4, p. 20, and Definition 5, p. 21) but with the presence of the grades and where $\mathsf{coerce}^{F,G}_{A}$ commutes with merge *and* mcond *to witness monotonicity.*

Remark 4. Graded monads are a lax[10] homomorphism between a monoids of effects $(\mathcal{F}, \bullet, I)$ and monoidal structure over $\mathcal{C}$ (of composing type constructors on $\mathcal{C}$). Similarly, graded joinads are a lax homomorphism, given by T and witnessed by the graded conditional joinad operations, between a joinoid of effects $(\mathcal{F}, \bullet, I, \&, ?{+}, \sqsubseteq)$ and a joinoid structure over $\mathcal{C}$. Thus, the joinoid axioms are preserved by T. For example, &-commutativity $F\&G = G\&F$ is preserved by T as witnessed by the axiom: (where $x : \mathsf{T}_F A$ and $y : \mathsf{T}_G B$)

$$\mathsf{merge}^{F,G}_{A,B}(x, y) : \mathsf{T}_{F\&G}(A \times B) \;\equiv\; \mathsf{map\ swap\ merge}^{G,F}_{B,A}(y, x) : \mathsf{T}_{G\&F}(A \times B)$$

Example 3 (Graded non-determinism joinad). Non-deterministic computations can be modelled as computations that return a list of possible results. The standard monadic model is to use $\mathsf{List}\ A = \mathsf{void} + (A \times \mathsf{List}\ A)$. Using a graded joinad, we can be more precise—and add an annotation that captures an *upper bound* on the length of the list resulting from a computation. Thus our graded type is $\mathsf{List}_n\ A = \sum_{m \leq n} A^m$ which represents a list that has at most n elements.

The associated joinoid control-flow algebra is $(\mathbb{N}, *, 1, *, \lambda(x, y, z).x{*}(y \max z))$. Sequential composition multiplies the degrees of non-determinism of the two computations as does parallel composition. For conditionals (?+) multiplies the degree of the guard with the maximum of the two branches. These annotations are consistent (sound) with the following graded joinad operations. We write $[v_1, ..., v_n]$ for lists of length n, with :: for *cons* and @ for concatenation:

$$\begin{array}{ll}
\mathsf{mcond}^{n,m,p}([], x, y) = [] & \mathsf{mcond}^{n,m,p}(\mathsf{true} :: g, x, y) = x \mathbin{@} \mathsf{mcond}^{n-1,m,p}(g, x, y) \\
 & \mathsf{mcond}^{n,m,p}(\mathsf{false} :: g, x, y) = y \mathbin{@} \mathsf{mcond}^{n-1,m,p}(g, x, y) \\
\mathsf{unit}\ x = [x] & \\
\multicolumn{2}{l}{\mathsf{merge}^{n,m}([u_1, ..., u_n], [v_1, ..., v_m]) = [(u_1, v_1), ..., (u_1, v_m), (u_2, v_1), ...(u_n, v_m)]} \\
\multicolumn{2}{l}{\mathsf{extend}\ f\ [u_1, ..., u_n] = f(u_1) @ ... @ f(u_n)}
\end{array}$$

The unit operation returns a singleton list and extend concatenates lists produced by applying f to all possible inputs (indeed, $n * 1 = n$). The merge operation takes the cross product and mcond concatenates the results of either the left or right branch depending on each possible guard. Note that merge is commutative up-to isomorphism, or commutative where equality is order-agnostic.

The key point of this example is that the graded conditional joinad structure itself captures the essence of effect annotations. The definition is consistent with respect to the lengths specified in the effect grades. In some way, the semantics *already* entails the effect system for the language. This is made explicit next.

Effect-directed semantics using graded joinads. The previous graded monadic semantics connected the structure and axioms of an effect system to a semantics for sequential composition, but did not capture parallelism or all

[10] Laxity means that the homomorphic map $\mathsf{T} : \mathcal{F} \rightarrow [\mathcal{C}, \mathcal{C}]$ from effects to type-constructors (endofunctors) on $\mathcal{C}$ has functions witnessing the mapping between structure on $\mathcal{F}$ and on $[\mathcal{C}, \mathcal{C}]$, *e.g.*, $\mathsf{merge}^{F,G}_{A,B} : \mathsf{T}_F A \times \mathsf{T}_G B \rightarrow \mathsf{T}_{F\&G}(A \times B)$, rather than equalities, *e.g.*, $\mathsf{T}_F A \times \mathsf{T}_G B = \mathsf{T}_{F\&G}(A \times B)$

possible forms of alternation. Graded joinads provide the opportunity for more fine-grained semantics and reasoning above conditionals and parallel composition. The following gives the graded conditional joinad effect-directed semantics:

$$\llbracket \Gamma \vdash e \ \mathsf{par}\ e' : \tau \times \tau',\ F\&G \rrbracket = \ \underline{\mathsf{merge}}^{F,G}_{\llbracket\tau\rrbracket,\llbracket\tau'\rrbracket}\ (\llbracket \Gamma \vdash e : \tau, F\rrbracket, \llbracket \Gamma \vdash e' : \tau', G\rrbracket)$$
$$\llbracket \Gamma \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau,\ ?{+}(F,G,H) \rrbracket = \underline{\mathsf{mcond}}^{F,G,H}_{\llbracket\tau\rrbracket}\ (\llbracket \Gamma \vdash e_0 : \mathsf{bool}, F\rrbracket, \llbracket \Gamma \vdash e_1 : \tau, G\rrbracket, \llbracket \Gamma \vdash e_2 : \tau, H\rrbracket)$$

As before, the semantics is defined over derivations, hence the left-hand side of each interpretation ends in the (PAR) and (IF) type-and-effect rules respectively. Effect annotations in the judgements correspond to the grades on the operations.

Theorem 2 (Syntactic soundness). *For all judgements $\Gamma \vdash e : \tau, F$ then: $\llbracket \Gamma \vdash e : \tau, F\rrbracket : \llbracket\Gamma\rrbracket \to \mathsf{T}_F\llbracket\tau\rrbracket$ where $\llbracket \tau \xrightarrow{F} \tau' \rrbracket = \llbracket\tau\rrbracket \to \mathsf{T}_F\llbracket\tau'\rrbracket$ and there is an interpretation for all base types.*

Proof. A straightforward analysis of the definition of $\llbracket - \rrbracket$.

Theorem 3 (Soundness). *Given a graded conditional joinad semantics for the simply-effect-and-typed λ-calculus with* if *then, for all e, e', Γ, τ, F:*

$$\Gamma \vdash e \equiv e' : \tau, F \quad \Rightarrow \quad \llbracket \Gamma \vdash e : \tau, F\rrbracket = \llbracket \Gamma \vdash e' : \tau, F\rrbracket$$

with respect to the equational theory for our language, in Theorem 1 (p. 22).

The syntactic soundness theorem (essentially that $\llbracket - \rrbracket$ preserves the typing structure and the semantics has corresponding grades) closes the gap between richer type-and-effect systems and semantics based on graded conditional joinads. It demonstrates the usefulness of the general approach advocated in this paper—a language with semantics based on graded conditional joinads comes equipped with an effect systems based on a joinoid control-flow algebra. Conversely, if we start with a joinoid effect system, the annotations can be used to determine the right structure of our semantics.

Example 4 (Graded music joinad). We tie the graded joinad discussion back to our motivating musical example with a simple graded conditional joinad model. As in Section 1, we use a joinad control-flow algebra to capture possible notes but not to capture their timing. Recall that music effects were drawn from terms defined by $\Phi = \mathcal{N} \mid \Phi_1 + \Phi_2 \mid \Phi_1 \bullet \Phi_2 \mid \Phi_1 \& \Phi_2$ where $\mathcal{N}$ is the set of all possible notes and rests, *e.g.* $\mathcal{N} = \{\mathsf{C}, \mathsf{D}, \mathsf{E}, \ldots, \mathsf{rest}\}$. We adjoin an additional effect ϵ representing a zero-length rest to be the identity for $\bullet$. Equality on Φ terms is defined such that the joinoid axioms hold.

Musical computations are modelled by tuples of a value, a duration d drawn from $\mathbb{R}$, and a *soundtrack*—a function g mapping from time within interval $[0, d] \in \mathbb{R}$ to sets of notes to be played at that time, returning $\emptyset$ outside that interval. This is given by the data type $\mathsf{Music}_F\, A = A \times \mathbb{R} \times (\mathbb{R} \to \mathcal{P}(\mathcal{N})|_F)$ where sets of notes drawn from $\mathcal{P}(\mathcal{N})$ are restricted to the notes appearing in

effect annotation F, written $|_F$. Soundtracks g and g' are combined with a time offset d for g' using the operator $g + g'@d = \lambda t.\textsf{if}\ t \leq d\ \textsf{then}\ g(t)\ \textsf{else}\ g'(t-d))$.

We provide the following graded conditional joinad definition, with additional effect-specific operation play for modelling note-playing.

$$
\begin{aligned}
\mathsf{mcond}^{m,n,p}\ (\mathsf{true}, d, g)\ (v', d', g')\ (v'', d'', g'') &= (v',\ d + \mathsf{max}(d', d''),\ g + g'@d) \\
\mathsf{mcond}^{m,n,p}\ (\mathsf{false}, d, g)\ (v', d', g')\ (v'', d'', g'') &= (v'',\ d + \mathsf{max}(d', d''),\ g + g''@d) \\
\mathsf{merge}^{m,n}\ (v, d, g)\ (v', d', g') &= ((v, v'),\ \mathsf{max}(d, d'),\ \lambda t.\, g(t) \cup g'(t)) \\
\mathsf{unit}^{\epsilon}\ v &= (v,\ 0,\ \lambda t.\, \emptyset) \\
\mathsf{extend}^{m,n}\ f\ (v, d, g) &= \textsf{let}\ (v', d', g') = f\, v\ \textsf{in}\ (v',\ d + d',\ g + g'@d) \\
\mathsf{play}^{n}\ (n, d) &= ((),\ d,\ \lambda t.\ \textsf{if}\ 0 \leq t \leq d \wedge n \neq \mathsf{rest}\ \textsf{then}\ \{n\}\ \textsf{else}\ \emptyset)
\end{aligned}
$$

For example, $\mathsf{play}(\mathsf{C}, 0.75) : \mathsf{Music}_{\mathsf{C}}\,[\![\mathsf{void}]\!]$ which is modelled by the unit value $()$ of type void, the duration 0.75, and the constant function $\lambda t.\{\mathsf{C}\}$.

As discussed earlier, there is an important design decision regarding mcond. Consider the expression $\textsf{if}\ b\ \textsf{then}\ \mathsf{play}(\mathsf{D}, 0.5)\ \textsf{else}\ \mathsf{play}(\mathsf{E}, 1)$. In our semantics, the expression *always* takes time 1: if b is true, it plays D for 0.5 and then rests for 0.5. This is because our mcond operation *implicitly synchronises* the branches. This is only possible because we interpret conditionals using the joinad control-flow operator mcond that has access to computations of both of the branches.

Now consider the *derived* semantics for conditionals obtained via extend along with $\mathsf{cond}_A : \mathbb{B} \to A \to A \to A$ instantiated at computation types $A = \mathsf{Music}_F[\![\mathsf{void}]\!]$. This leads to quite a different semantics in that we have:

$$
\begin{aligned}
\mathsf{mcond}\ (\mathsf{true}, d, g)\ (v', d', g')\ (v'', d'', g'') &= (v',\ d + d',\ g + g'@d) \\
\mathsf{mcond}\ (\mathsf{false}, d, g)\ (v', d', g')\ (v'', d'', g'') &= (v'',\ d + d'',\ g + g''@d)
\end{aligned}
$$

Here, the total time of the if operation is the total time of the executed branch, meaning that the conditional, now being dichotomous, does not perform implicit synchronisation. By turning mcond into a to-be-specified control-flow operator instead of requiring the monad-derived semantics, we get additional flexibility and can choose between the two behaviours. (This example provides another practical use for non-dichotomous semantics for if-then-else.)

4.5 Classical joinads—grading and control-flow algebra

Classical joinads (with choose and fail instead of mcond) can similarly be formulated as a control-flow algebra [29]. We briefly give the definitions here.

Definition 8 (Joinad control-flow algebra). $(\mathcal{F}, \bullet, +, \&, 0, I)$ *is a* joinad control-flow algebra *if* $(\mathcal{F}, \bullet, I)$ *is a monoid,* $(\mathcal{F}, +, 0, \bullet)$ *is a near-semiring and* $(\mathcal{F}, +, 0, \&)$ *is a near-semiring with commutative* $\&$*. This can be extended with an ordering* $\sqsubseteq$ *on* $\mathcal{F}$ *w.r.t. which operations* $\bullet, +, \&$ *are required to be monotonic.*

This definition captures the structure and axioms of a joinad. The first near-semiring requirement means that $(\mathcal{F}, +, 0)$ is a monoid, that 0 is the $\bullet$-absorbing

element ($0 \bullet F = 0$), and $(F+G) \bullet H = (F \bullet H) + (G \bullet H)$ (sequencing distributes over alternation). The second near-semiring requirement implies that $(\mathcal{F}, \&)$ is a semigroup, $0\&F = 0$, and & distributes over alternation.

Definition 9 (Graded classical joinads). *Given* $(\mathcal{F}, \bullet, +, \&, 0, I, \sqsubseteq)$*—a* joinad control-flow algebra*—then a* graded joinad *is an ordered graded monad for the ordered monoid* $(\mathcal{F}, \bullet, I, \sqsubseteq)$ *together with the following three operations:*

$$\mathsf{choose}^{F,G}_{A} : \mathsf{T}_F A \times \mathsf{T}_G A \to \mathsf{T}_{F+G} A \qquad \mathsf{fail}_A : \mathsf{void} \to \mathsf{T}_0 A$$
$$\mathsf{merge}^{F,G}_{A,B} : \mathsf{T}_F A \times \mathsf{T}_G B \to \mathsf{T}_{F\&G}(A \times B)$$

The operations are required to satisfy the joinad control-flow algebra laws (Definition 8), which are syntactically the same as standard joinad laws, but annotated with corresponding effects. We omit these for brevity.

This structure provides a useful effected-directed model for effectful pattern matching, in contrast to standard if-then-else conditionals. All examples of joinads [32, 29] can be turned into graded joinads via the trivial (single element) joinoid control-flow algebra or by adding some suitable effect algebra which refines the existing semantics. For parsers, annotations may capture the degree of non-determinism (how many choices there are) and the length of the required input. In parallel programming, the annotations on graded joinads can estimate the maximal evaluation time (with $\bullet$ as addition; + and & taking the maximum) or the minimal evaluation time (same, with + as minimum).

5 Discussion

Kleene algebras and recursion In our musical introduction, we introduced iteration (for loops) and modelled this in the effect system by a Kleene-star-like unary operator Φ^*. Recursion, or iteration, is another useful control-flow operator that we may wish to distinguish in an effect system and its semantics.

Similarly to conditionals, we can give a derived semantics for effectful recursion in terms of underlying operations in the semantic domain. Given fix which maps every $f : A \to A$ to $\mathsf{fix}_A f : A$, we can derive an effectful fixed-point: $\mathsf{mfix}_A = \mathsf{fix}_{\mathsf{T}A}\ (\mathsf{extend}_{A,A}\, f)$ operator mapping $f : A \to \mathsf{T}A$ to $\mathsf{mfix}_A\, f : \mathsf{T}A$ (*i.e.*, a fixed point is taken over the monadic extension of f, *i.e.*, $\mathsf{mfix}_A\, f = (\mathsf{extend}_{A,A}\, f) \circ (\mathsf{extend}_{A,A}\, f) \circ ...$). This is similar to the approach of Kleene monads [10]. Interestingly, replacing extend with the graded monad version in the above fixed-point definition forces an additional requirement on the effect algebra, that $\bullet$ is idempotent; that is, $f : A \to \mathsf{T}_F A$ is mapped to $\mathsf{mfix}^F_A f : \mathsf{T}_F A$ where $\mathsf{mfix}^F_A f = \mathsf{fix}_{\mathsf{T}_F A}(\mathsf{extend}^{F,F}_{A,A} f)$ where $\mathsf{extend}^{F,F}_{A,A} : \mathsf{T}_F A \to \mathsf{T}_{F \bullet F} A$ and thus $F \bullet F = F$. For some effect systems this would suffice, but for others we may want to introduce an effect element ω ('repeat forever') or traces via regular languages. We see this as a maxim: *give a semantic operator for every effect operator; sometimes these can be derived from existing operators but we should avoid building in this as a requirement.*

Following the philosophy of this paper, an abstract effect-directed semantics for recursion is best served by a control-flow algebra for effect annotations with F^* and a graded operation mfix^F_A which maps $f : A \to \mathsf{T}_F A$ to $\mathsf{mfix}^F_A\, f : \mathsf{T}_{F^*} A$. This provides a more general model for static analysis and semantics.

Other related work Benton *et al.* previously defined an effect-directed semantics for state, similar in motivation to the work on graded monads but specialised [4]. This is used for precise semantic reasoning based on refinements from effect analysis. They show various effect-driven transformations, which are proven sound in their semantics. Their work gives a deep treatment to some of the themes we have touched on here more broadly. We focused more on the idea of generalising the treatment of control-flow operators in the presence of effects.

The work on *algebraic effects* and *handlers* provides an alternative approach, connecting effect systems and monads [15, 34]. This approach focuses on effectful operations (read/write/*etc.*) and equations between them. This is a change of perspective to monads, which consider first an encoding of effects rather than the effectful operations. The work of Power and Plotkin starts with the effectful operations and generates an encoding as the free structure arising from the operations quotiented by their equational theory [33]. Recent work by Kammar *et al.* has used these approaches to give effect-dependent optimisations with a sound semantics [14, 15]. That work can be similarly described as *effect-directed semantics*, but from a different perspective to that laid down by the line of work of Wadler-Thiemann, Katsumata, and this paper. The work on algebraic effects largely focuses on the *building blocks* of effects: the effectful operations and their algebraic theories. We have instead focused on the *scaffolding*: control-flow structures which compose effectful computations.

Coeffects, the dual of effects, which track how a program depends on its context or how it consumes resources have been similarly given a *coeffect*-directed categorical denotational semantics [31, 30, 5]. Coeffect structures tend to comprise some form of resource semiring with a semiring-graded *comonad* in the semantics [8].

Conclusions and further work The semantic understanding of effect systems and their use for static analysis has rather diverged. Nielson and Nielson developed richer effect algebras, but left the proof of correctness (with respect to semantics) to users of these algebras. By contrast, Wadler, Thiemann, Katsumata *et al.*, (and Atkey via parameterised monads [3]) have developed models which link (semilattice-based) effects directly to semantic models, so that a model of a computation only includes elements consistent with their effect annotations.

We showed that monads and graded monads do not capture richer control flow (in particularly parallelism and some forms of conditional). We argued that certain operators in extensions of monads are *control-flow operators* distinguishable by their type, and provided a link between such types and control-flow graphs. We showed that joinads (monads extended with operations for alternation and parallelism, on top of the existing monadic sequential composition) provide a practical example of these control-flow algebras, and that they express

concepts similar to those of Nielson and Nielson. Joinads can also be graded in a similar manner to Katsumata's graded monads, thus providing a framework where semantic models can only express values appropriate to syntactic effects.

Further work might explore other control-flow algebras that could be similarly "graded", beyond those discussed here, such as backtracking [17]. Another avenue is to establish the conditions under which the graded connection between syntax and semantics induces soundness, or even goes as far as completeness.

Acknowledgements We thank Matthew Danish, Ohad Kammar, Shinya Katsumata, Jeremy Yallop, and the anonymous referees for their helpful comments. Any remaining errors are our own. The second author is funded by EPSRC EP/K011715/1 and thanks Nobuko Yoshida for her support.

References

1. Samuel Aaron, Dominic Orchard, and Alan F. Blackwell. Temporal semantics for a live coding language. In *Proc. 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*, pages 37–47. ACM, 2014.
2. Torben Amtoft, Flemming Nielson, and Hanne Nielson. *Type and Effect Systems: Behaviours for Concurrency.* Imperial College Press, 1999.
3. Robert Atkey. Parameterised notions of computation. In *Proceedings of MSFP*. Cambridge University Press, 2006.
4. Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. Reading, writing and relations. In *Programming Languages and Systems*, pages 114–130. Springer, 2006.
5. Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In *Proc. of ESOP 2014*, pages 351–370. Springer, 2014.
6. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL*, pages 238–252. ACM, 1977.
7. Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of PLDI '03*. ACM, 2003.
8. Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In *Programming Languages and Systems*, pages 331–350. Springer, 2014.
9. David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of Conference on LISP and Functional Programming*, LFP '86, 1986.
10. Sergey Goncharov, Lutz Schröder, and Till Mossakowski. Kleene monads: handling iteration in a framework of generic effects. In *Algebra and Coalgebra in Computer Science*, pages 18–33. Springer, 2009.
11. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, pages 122–138. Springer, 1998.
12. Pierre Jouvelot and David K. Gifford. Communication Effects for Message-Based Concurrency. Technical report, Massachusetts Institute of Technology, 1989.
13. Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *Proceedings of PLDI '89*. ACM, 1989.

14. Ohad Kammar. Algebraic theory of type-and-effect systems, 2014. PhD dissertation, The University of Edinburgh.
15. Ohad Kammar and Gordon D. Plotkin. Algebraic foundations for effect-dependent optimisations. In *Proceedings of POPL 2012*, pages 349–360. ACM, 2012.
16. Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Proceedings of POPL 2014*, pages 633–645. ACM, 2014.
17. Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers (functional pearl). In *Proceedings of ICFP 2005*, pages 192–203. ACM, 2005.
18. John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57. ACM, 1988.
19. Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *SIGPLAN Not.*, 48(4):461–472, March 2013.
20. Paul-André Mellies. Parametric monads and enriched adjunctions. Available via `http://www.pps.univ-paris-diderot.fr/~mellies/tensorial-logic.html`, 2012.
21. Stefan Milius, Dirk Pattinson, and Lutz Schröder. Generic Trace Semantics and Graded Monads. In *Proceedings of 6th International Conference in Algebra and Coalgebra in Computer Science*, 2015.
22. Robin Milner. *Communication and concurrency*, volume 84. Prentice Hall, New York, 1989.
23. Eugenio Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science*, pages 14–23. IEEE, 1989.
24. Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.
25. Flemming Nielson, Patrick Cousot, Mads Dam, Pierpaolo Degano, Pierre Jouvelot, Alan Mycroft, and Bent Thomsen. Logical and operational methods in the analysis of programs and systems. In *Analysis and Verification of Multiple-Agent Languages*, pages 1–21. Springer, 1997.
26. Flemming Nielson and Hanne Nielson. Type and effect systems. *Correct System Design*, pages 114–136, 1999.
27. Dominic Orchard and Tomas Petricek. Embedding effect systems in Haskell. In *Proceedings of ACM SIGPLAN symposium on Haskell*, pages 13–24. ACM, 2014.
28. Dominic A. Orchard, Tomas Petricek, and Alan Mycroft. The semantic marriage of monads and effects. *CoRR*, abs/1401.5391, 2014.
29. Tomas Petricek, Alan Mycroft, and Don Syme. Extending Monads with Pattern Matching. In *Proceedings of Haskell Symposium*, Haskell 2011, 2011.
30. Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: Unified static analysis of context-dependence. In *ICALP (2)*, pages 385–397, 2013.
31. Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 123–135. ACM, 2014.
32. Tomas Petricek and Don Syme. Joinads: a retargetable control-flow construct for reactive, parallel and concurrent programming. In *Proceedings of PADL*, 2011.
33. Gordon Plotkin and John Power. Notions of computation determine monads. In *Foundations of Software Science and Computation Structures*, pages 342–356. Springer, 2002.

34. Gordon Plotkin and Matija Pretnar. A logic for algebraic effects. In *Logic in Computer Science, 2008. LICS'08*, pages 118–129. IEEE, 2008.
35. Exequiel Rivas, Mauro Jaskelioff, and Tom Schrijvers. From monoids to near-semirings: the essence of MonadPlus and Alternative. In *Proc. of Int. Symp. on Principles and Practice of Declarative Programming*, pages 196–207. ACM, 2015.
36. A. L. Smirnov. Graded monads and rings of polynomials. *Journal of Mathematical Sciences*, 151(3):3032–3051, 2008.
37. Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Logic in Computer Science, 1992. LICS'92*, pages 162–173. IEEE, 1992.
38. Jérôme Vouillon. Lwt: A Cooperative Thread Library. In *Proceedings of ACM SIGPLAN Workshop on ML*, pages 3–12, New York, NY, USA, 2008. ACM.
39. Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4:1–32, January 2003.

A Issues surrounding monadic strength

A monadic semantics for an effectful simply-typed λ-calculus requires that monads are *strong*. This captures the idea (implicit in the category of sets, but not in all categories) that a free variable may be captured by an outer λ-binding. Strong monads have an additional operation: $\mathsf{str}_{A,B} : A \times \mathsf{T}B \to \mathsf{T}(A \times B)$ satisfying various axioms (see [23, 24]) which amount to saying that the effects encoded in the result of str are the effects encoded by the second argument.

The monadic semantics for (BIND) in *Monad* (shown in Section 2.3, p. 11), omitting the type subscripts on extend and str, is then:

$$
\begin{aligned}
&[\![\Gamma \vdash_M \mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, e' : M^{F \bullet G}\tau']\!] \;:\; [\![\Gamma]\!] \to \mathsf{T}[\![\tau']\!] \\
&\qquad = \lambda\gamma\,.\; \mathsf{extend}\; [\![\Gamma, x\!:\!\tau \vdash_M e' : M^{G}\tau']\!] \;\; (\mathsf{str}\,(\gamma,\, [\![\Gamma \vdash_M e : M^{F}\tau]\!]\; \gamma))
\end{aligned}
$$

The str operation turns an environment $\gamma : [\![\Gamma]\!]$ and a result $\mathsf{T}[\![\tau]\!]$ into $\mathsf{T}([\![\Gamma]\!] \times [\![\tau]\!])$ for composition with $\mathsf{extend}\; [\![\Gamma, x\!:\!\tau \vdash_M \ldots]\!] : \mathsf{T}([\![\Gamma]\!] \times [\![\tau]\!]) \to \mathsf{T}[\![\tau']\!]$.

Graded monads can be similarly strong with operation $\mathsf{str}^{F}_{A,B} : A \times \mathsf{T}_F B \to \mathsf{T}_F(A \times B)$, satisfying analogous axioms to the usual strong monad axioms [16]. The graded semantics is then the analogous one to the above.

We might consider adding an effect operation $\mathsf{S} : \mathcal{F} \to \mathcal{F}$ corresponding to use of strength in the semantics *e.g.*, $\mathsf{str}^{F}_{A,B} : A \times \mathsf{T}_F B \to \mathsf{T}_{\mathsf{S}F}(A \times B)$. However, an axiom of (non-graded) strong monads is that for all $x \in A, y \in \mathsf{T}B$ then $\mathsf{map}\,\mathsf{fst}\,(\mathsf{str}_{A,B}(x, y)) = y$ which for graded strength would imply that $\mathsf{S}\,F = F$. We accordingly exclude S from the effect algebra since it is necessarily identity.