Lifting Sequential Effects to Control Operators

Colin S. Gordon
Drexel University, Philadelphia, PA, USA
`https://cs.drexel.edu/~csgordon`
csgordon@drexel.edu

Abstract

Sequential effect systems are a class of effect system that exploits information about program order, rather than discarding it as traditional commutative effect systems do. This extra expressive power allows effect systems to reason about behavior over time, capturing properties such as atomicity, unstructured lock ownership, or even general safety properties. While we now understand the essential denotational (categorical) models fairly well, application of these ideas to real software is hampered by the variety of source level control flow constructs and control operators in real languages.

We address this new problem by appeal to a classic idea: macro-expression of commonly-used programming constructs in terms of control operators. We give an effect system for a subset of Racket's tagged delimited control operators, as a lifting of an effect system for a language without direct control operators. This gives the first account of sequential effects in the presence of general control operators. Using this system, we also re-derive the sequential effect system rules for control flow constructs previously shown sound directly, and derive sequential effect rules for new constructs not previously studied in the context of source-level sequential effect systems. This offers a way to directly extend source-level support for sequential effect systems to real programming languages.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning; Theory of computation → Type structures

Keywords and phrases Type systems, effect systems, quantales, control operators, delimited continuations

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.23

Related Version An extended version of this paper containing full proofs is available [32] on arXiv at `https://arxiv.org/abs/1811.12285`.

1 Introduction

Effect systems extend type systems to reason about not only the shape of data, and available operations – roughly, what a computation produces given certain inputs – but to also reason about *how* the computation produces its result. Examples include ensuring data race freedom by reasoning about what locks a computation assumes held during its execution [1, 23, 11, 10], restricting sensitive actions (like UI updates) to dedicated threads [33], ensuring deadlock freedom [24, 34, 1, 69], checking safe region-based memory management [72, 50], or most commonly checking that a computation handles (or at least indicates) all errors it may encounter – Java's checked exceptions [35] are the most widely used effect system.

Most effect systems discard information about program order: the same join operation on a join semilattice of effects is used to overapproximate different branches of a conditional or different subexpressions executed in sequence. Despite this simplicity, these traditional *commutative* effect systems (where the combination of effects is always a commutative operation) are powerful. Still, many program properties of interest are sensitive to evaluation order. For example, commutative effect systems handle scoped `synchronized` blocks as in Java with ease: the effect of (the set of locks required by) the `synchronized`'s body is permitted to contain the synchronized lock, in addition to the locks required by the overall construct.

But to support explicit lock acquisition and release operations that are not block-structured, an effect system must track whether a given expression acquires and/or releases locks, and must distinguish their ordering: releasing and then acquiring a given lock is not the same as acquiring before releasing. To this end, *sequential* effect systems (so named by Tate [70]) reason about effects with knowledge of the program's evaluation order.

Sequential effect systems are much more powerful than commutative effect systems, with examples extending through generic reasoning about program traces [68, 67] and even propagation of *liveness* properties from oracles [45] – well beyond what most type systems support. The literature includes sequential effect systems for deadlock freedom [34, 69, 1, 10], atomicity [25, 26], trace-based security properties [68, 67], safety of concurrent communication [4, 57], general linear temporal properties with a liveness oracle [45], and more. Yet for all the power of this approach, for years each of the many examples of sequential effect systems in the literature individually rederived much structure common to all sequential effect systems. Recent years have seen efforts to unify understanding of sequential effect systems with general frameworks, first denotationally [70, 42, 55, 6], and recently as an extension to the join semilattice model [30]. These frameworks can describe the structure of established sequential effect systems from the literature.

However, these generic frameworks stop short of what is necessary to apply sequential effect systems to real languages: they lack generic treatments of critical features of real languages that interact with evaluation order – control operators, including established features like exceptions and increasingly common features like generators [14].. And with the exception of the effect systems used to track correct return types with delimited continuations (answer type modification [16, 5, 44]), there are no sequential effect systems that consider the interaction of control and sequential effects. This means *promising sequential effect systems [68, 67, 45, 25, 34, 4, 57, 69, 10] cannot currently be applied directly to real languages like Java [35], Racket, C# [52], Python [71], or JavaScript [54].*

Control operators effectively reorder, drop, or duplicate portions of a program's execution at runtime, changing evaluation order. In order to reason precisely about flexible rearrangement of evaluation order, a sequential effect system must reason about control operators. The classic example is again Java's `try-catch`: if the body of a `try` block both acquires and releases a lock this is good, but if an exception is thrown mid-block the release may need to be handled in the corresponding catch. Clearly, applying sequential effect systems to real software requires support for exceptions in a sequential effect system. Working out just those rules is tempting, but exceptions interact with loops. The effect before a throw inside a loop – which a catch block may need to "complete" (e.g., by releasing a lock) – depends on whether the throw occurs on the first or nth iteration. Many languages include more than simply `try-catch`, for example with the *generators* (a form of coroutine) now found in C# [52], Python [71], and JavaScript [54]. These interact with exceptions *and* loops. Treating each new control operator individually seems inefficient.

An alternative to studying all possible combinations of individual control constructs in common languages is to study more general constructs, such as the very general delimited continuations [22, 19] present in Racket. These are useful in their own right (for Racket, or the project to add them to Java [36]), and can macro-express many control flow constructs and control operators of interest, including loops, exceptions [27], coroutines [39, 40], generators [14], and more [15]. Then general principles can be derived for the general constructs, which can then be applied to or specialized for the constructs of interest. This both solves the open question of how to treat general control operators with sequential effect systems, and leads to a basis for more compositional treatment of loops, exceptions, generators, and future additions to languages. This is the avenue we pursue in this paper.

Delimited continuations solve the generality problem, but introduce new challenges since sequential effect systems can track evaluation order [68, 67, 31, 45]. The effect of an expression that aborts out of a prompt depends on what was executed before the abort, but not after. The body of a continuation capture (`call/cc`) must be typed knowing the effect of the *enclosing context* – the code executing after, but not before (up to the enclosing prompt). We lay the groundwork for handling modern control operators in a sequential effect system:

- We give the first generic characterization of sequential effects for continuations, by giving a *generic* lifting of a control-unaware sequential effect system into one that can support tagged delimited continuations. The construction we describe provides a way to automatically extend existing systems with support for these constructs, and likewise will permit future sequential effect system designers to ignore control operations initially and add support later for free (by applying our construction). As a consequence, we can transfer prior sequential effect systems designed *without* control operators to a setting *with* control operators.
- We give sequential effect system rules for `while` loops, `try-catch`, and generators by deriving them from their macro-expression [20] in terms of more primitive operators. The loop characterization was previously known (and technically a control flow construct, not a general control operator), but was given as primitive. The others are new to our work, and necessary developments in order to apply sequential effect systems to most modern programming languages. The derivation approach we describe can be applied to other control operators that are not explicitly treated in the paper.
- We demonstrate how prior work's notion of an iteration operator [30, 31] derived from a closure operator on the underlying effect lattice is not specific to loops, but rather provides a general tool for solving recursive constraints in sequential effect systems.
- We prove syntactic type safety for a type system using our sequential control effect transformation with any underlying effect system.

2 Background

We briefly recall the details of sequential type-and-effect systems, and tagged delimited continuations. We emphasize the view of effect systems in terms of a *control flow algebra* [55] – an algebraic structure with operations corresponding to the ways an effect system might combine the effects from subexpressions in a program.

2.1 Sequential Effect Systems

Traditional type-and-effect systems extend the typing judgment $\Gamma \vdash e : \tau$ for an additional component. The extended judgment form $\Gamma \vdash e : \tau \mid \chi$ is read "under local variable assumptions Γ, the expression e evaluates to a value of type τ (or diverges), with effect χ during evaluation." The last clause of that reading is vague, but carries specific meanings for specific effect systems. For checked exceptions, it could be replaced by "possibly throwing exceptions χ during evaluation" where χ would be a set of checked exceptions. For a data race freedom type system reasoning about lock ownership, it could be replaced by "and is data race free if executed while locks χ are held."

The join semilattice structure of standard effect systems is well-known, as are the corresponding denotational analogues (e.g., indexed monads [74]). The limitation common to all of these systems, however, is that they discard program order, using the (commutative) join for any combination of effects. In contrast, there is growing work on *sequential* [70] effect systems, which capture a wide array of order-sensitive phenomena. This includes effect systems for atomicity [25, 26], deadlock freedom [69, 34, 10, 1], race freedom with explicit lock acquisition and release [69, 30], message passing concurrency safety [57, 4], security checks [68], and (with the aid of an oracle for liveness properties) general linear-time properties [45]. Tate labels these systems *sequential* effect systems [70], as their distinguishing feature is the use of an additional sequencing operator to join effects where one is known to be evaluated before another. Consider the sequential rules for functions, function application, conditionals, and while loops:

$$\textsc{T-App}\quad \frac{\Gamma \vdash e_1 : \tau \xrightarrow{\chi} \sigma \mid \chi_1 \qquad \Gamma \vdash e_2 : \tau \mid \chi_2}{\Gamma \vdash e_1\ e_2 : \sigma \mid \chi_1 \rhd \chi_2 \rhd \chi}
\qquad
\textsc{T-While}\quad \frac{\Gamma \vdash e_c : \mathsf{boolean} \mid \chi_c \qquad \Gamma \vdash e_b : \tau \mid \chi_b}{\Gamma \vdash \mathsf{while}\ e_c\ e_b : \mathsf{unit} \mid \chi_c \rhd (\chi_b \rhd \chi_c)^*}$$

$$\textsc{T-Lambda}\quad \frac{\Gamma, x : \tau \vdash e : \sigma \mid \chi}{\Gamma \vdash (\lambda x.\, e) : \tau \xrightarrow{\chi} \sigma \mid I}
\qquad
\textsc{T-If}\quad \frac{\Gamma \vdash e_c : \mathsf{bool} \mid \chi_c \qquad \Gamma \vdash e_t : \tau \mid \chi_t \qquad \Gamma \vdash e_f : \tau \mid \chi_f}{\Gamma \vdash \mathsf{if}\ e_c\ e_t\ e_f : \tau \mid \chi_c \rhd (\chi_t \sqcup \chi_f)}$$

The sequencing operator $\rhd$ is associative but *not* (necessarily) commutative. Thus the effect in the new T-App reflects left-to-right evaluation order: first the function position is reduced to a value, then the argument, and then the function body is executed. The conditional rule reflects the execution of the condition followed by *either* (via commutative join) the true or false branch. The while loop uses an iteration operator $(-)^*$ to represent 0 or more repetitions of its argument; we will return to its details later. The effect of T-While reflects the fact that the condition will always be executed, followed by 0 or more repetitions of the loop body and checking the loop condition again. The rule for typing lambda expressions switches from a bottom element, to a general unit effect: identity for sequential composition.

To formalize the intuition above, Gordon [30] proposed *effect quantales* as a model that captures prior effect systems' structure:

▶ **Definition 1** (Effect Quantale). *An* effect quantale *is a join-semilattice-ordered monoid with nilpotent top. That is, it is a structure $(E, \sqcup, \top, \rhd, I)$ where:*

- $(E, \sqcup, \top)$ *is an upper-bounded join semilattice*
- $(E, \rhd, I)$ *is a monoid*
- $\top$ *is nilpotent for sequencing* ($\forall x.\, x \rhd \top = \top = \top \rhd x$)
- $\rhd$ *distributes over* $\sqcup$ *on both sides:* $a \rhd (b \sqcup c) = (a \rhd b) \sqcup (a \rhd c)$ *and* $(a \sqcup b) \rhd c = (a \rhd c) \sqcup (b \rhd c)$

The structure extends a join semilattice with a sequencing operator, a designated error element to model possibly-undefined combinations, and laws specifying how the operators interact. Top ($\top$) is used as an indication of a type error, for modeling partial join or sequence operators: expressions with effect $\top$ are rejected. $\sqcup$ is used to model non-deterministic joins (e.g., for branches) as in the commutative systems, and $\rhd$ is used for sequencing. The default effect of "uninteresting" program expressions (including values) becomes the unit I rather than a bottom element (which need not exist). As a consequence of the distributivity laws, it follows that $\rhd$ is also monotone in both arguments, for the standard partial order derived from a join semilattice: $x \sqsubseteq y \equiv x \sqcup y = y$.

Gordon [30] also showed how to exploit *closure operators* [8, 9, 29] to impose a well-behaved notion of iteration (the $(-)^*$ operator from T-While) that coincides with manually-derived versions for the effect quantales modeling prior work for many effect quantales. Gordon [31] recently generalized the construction, and showed that large general classes of effect quantales meet the criteria to have such an iteration operator. The effect quantales for which the generalized iteration is defined are called *laxly iterable.* An effect quantale is *laxly iterable* if for every element x, the set of subidempotent elements ($\{s \mid s \rhd s \sqsubseteq s\}$) greater than both x and I has a least element. This is true of all known effect quantales corresponding to systems in the literature.

The iteration operator for an iterable effect quantale takes each effect x to the least subidempotent effect greater than or equal to $x \sqcup I$ (which exists, by the definition of laxly iterable). This iteration operator satisfies 5 essential properties for any notion of iteration [31], which we will find useful when deriving rules for loops. Iteration operators are *extensive* ($\forall e.\, e \sqsubseteq e^*$), *idempotent* ($\forall e.\, (e^*)^* = e^*$), *monotone* ($\forall e, f.\, e \sqsubseteq f \Rightarrow e^* \sqsubseteq f^*$), *foldable* ($\forall e.\, e \rhd e^* \sqsubseteq e^*$ and $e^* \rhd e \sqsubseteq e^*$), and *possibly-empty* ($\forall e.\, I \sqsubseteq e^*$). Another useful property of iteration that we will sometimes use is that $\forall x, y.\, x^* \sqcup y^* \sqsubseteq (x \sqcup y)^*$. Gordon [30, 31] gives more details on closure operators and the derivation of iteration. We merely require its existence and properties.

For our intended goal of giving a transformation of any arbitrary sequential effect system into one that can use tagged delimited continuations, we require *some* abstract characterization. We choose effect quantales as the abstraction for lifting for several reasons. First, they characterize the structure of a range of concrete systems from prior work [30, 31], while other proposals omit structure that is important to these concrete systems. Second, while effect quantales are not maximally general, they remain very general: the motivating example for Tate's work [70] (which *is* maximally general) can be modeled as an effect quantale. Third, we would like to check whether our derived rules are sensible; effect quantales are the only abstract characterization for which imperative loops have been investigated, offering appropriate points of comparison. Finally, the iteration construction on effect quantales offers a natural approach to solving recursive constraints on effects, which we will use in deriving closed-form derived rules for macro-expressed control flow constructs and control operators.

As a running example throughout the paper, we will use a simplification of various trace or history effect systems [68, 67, 45]. For a set (alphabet) of events Σ, consider the non-empty subsets of Σ^* – the set of possibly-empty strings of letters drawn from Σ (the strings, not the subsets, may be empty). This gives an effect quantale $\mathcal{T}(\Sigma)$ whose elements are these subsets or an additional top-most error element Err. Join is simply set union lifted to propagate Err. Sequencing is the double-lifting of concatenation, first to sets ($A \cdot B = \{xy \mid x \in A \wedge y \in B\}$), then again to propagate Err. The unit for sequencing is the singleton set of the empty string, $\{\epsilon\}$. If Σ is a set of events of interest – e.g., security events – then effects drawn from this effect quantale represent sets of possible finite event sequences executed by a program. Effects drawn from this effect quantale show the possible sequences of operations code may execute, which will allow us to show explicitly how fragments of program execution are rearranged when using control operators. For our examples, we will assume a family of language primitives $\mathsf{event}[\alpha]$ with effect $(\emptyset, \emptyset, \{\alpha\})$ (similar to Koskinen and Terauchi [45]), where α is drawn from a set Σ of possible events. The key challenge we face in this paper is, viewed through the lens of $\mathcal{T}(\Sigma)$, to ensure that when continuations are used, the effect system does not lose track of events of interest or falsely claim a critical event occurs where it may not.

2.2 Tagged Delimited Continuations

Control operators have a long and rich history, reaching far beyond what we discuss here. Many different control operators exist, and many are macro-expressible [20] in terms of each other (i.e., can be translated by direct syntactic transformation into another operator), though some of these translations require the assumption of mutable state, for example. But a priori there is no single most general construct to study which obviously yields insight on the source-level effect typing of other constructs. A suitable starting place, then, is to target a highly expressive set of operators that see use in a real language. If the operators

$$E ::= \bullet \mid (E\ e) \mid (v\ E) \mid (\%\ t\ E\ v) \mid (\mathsf{call/cc}\ t\ E) \mid (\mathsf{call/comp}\ t\ E) \mid (\mathsf{abort}\ t\ e)$$

$$\boxed{\sigma; e \xrightarrow{q} \sigma; e} \qquad \textsc{E-App}\ \frac{}{\sigma; ((\lambda x.\, e)\ v) \xrightarrow{I} \sigma; e[v/x]} \qquad \textsc{E-PromptVal}\ \frac{}{\sigma; (\%\ \ell\ v\ h) \overset{I}{\Rightarrow} \sigma; v}$$

$$\boxed{\sigma; e \overset{q}{\Rightarrow} \sigma; e} \qquad \textsc{E-Context}\ \frac{\sigma; e \xrightarrow{q} \sigma'; e'}{\sigma; E[e] \overset{q}{\Rightarrow} \sigma'; E[e']} \qquad \textsc{E-Abort}\ \frac{E' \text{ contains no prompts for } \ell}{\sigma; E[(\%\ \ell\ E'[(\mathsf{abort}\ \ell\ v)]\ h)] \overset{I}{\Rightarrow} \sigma; E[h\ v]}$$

$$\textsc{E-CallCC}\ \frac{E' \text{ contains no prompts for } \ell}{\sigma; E[(\%\ \ell\ E'[(\mathsf{call/cc}\ \ell\ k)]\ h)] \overset{I}{\Rightarrow} \sigma; E[(\%\ \ell\ E'[(k\ (\mathsf{cont}\ \ell\ E'))]\ h)]}$$

$$\textsc{E-InvokeCC}\ \frac{E' \text{ contains no prompts for } \ell}{\sigma; E[(\%\ \ell\ E'[((\mathsf{cont}\ \ell\ E'')\ v)]\ h)] \overset{I}{\Rightarrow} \sigma; E[(\%\ \ell\ E''[v]\ h)]}$$

Figure 1 Operational semantics.

are sufficiently expressive, this provides not only a sequential type system for an expressive source language directly, but also supports deriving type rules for *other* languages' control constructs, based on their macro-expression in terms of the studied control operators.

We study a subset of the tagged delimited control operators [22, 19, 64, 65, 66] present in Racket [27], shown in Figure 1. The semantics include both local ($\rightarrow$) and global ($\Rightarrow$) reductions on configurations consisting of a state σ and expression e. All continuations in Racket are delimited, and tagged. There is a form of *prompt* that limits the scope of any continuation capture: `(% tag e e2)` is a tagged prompt with tag `tag`, body `e`, and abort handler `e2`. Without tags, different uses of continuations – e.g., error handling or concurrency abstractions – can interfere with each other [64]; as a small example, if loops and exceptions were both implemented with *un*delimited continuations, throwing an exception from inside a loop inside a try-catch would jump to the loop boundary, not the catch. Thus prompts, the continuation-capturing primitives `call/cc` and `call/comp`, and the `abort` primitive all specify a tag, and only prompts with the specified tag are used to interpret continuation and abort boundaries. This permits jumping over unrelated prompts (e.g., so exceptions find the nearest *catch*, not merely the nearest control construct). In most presentations of delimited continuations, tags are ignored (equivalently, all tags are equal), while most implementations retain them for the reasons above. Here the tags are essential to the theory as well: an abort that "skips" a different prompt must be handled differently by our type-and-effect system.

`call/cc tag f` is the standard (delimited) call-with-current-continuation: `f` is invoked with a delimited continuation representing the current continuation up to the nearest prompt with tag `tag` (E-CallCC). Invoking that continuation (E-InvokeCC) replaces the context up to the nearest dynamically enclosing prompt with the same tag, leaving the delimiting prompt in place. Both capture and replacement are bounded by the nearest enclosing prompt for the specified tag. The surrounding captured or replaced context (E' in both rules) may contain prompts for other tags, but not the specified tag. Racket also includes `(abort t e)` (absent in many formalizations of continuations), which evaluates `e` to a value, then replaces the enclosing prompt (of the specified tag `t`) with an invocation of the handler applied to that value (E-Abort). Racket's rules differ from some uses of `abort` in the literature. Figure 1's rules are Flatt et al.'s rules [27] without continuation marks and `dynamic-wind`. Flatt et al. formalized Racket's control operators in Redex [21, 43], including showing they passed the Racket implementation tests for those features. We have verified the rules above continue to pass the relevant tests in Redex (see supplementary material [32]).

We chose this set of primitives, over related control operators [63] such as `shift/reset` or `shift0/reset0` which can simulate these primitives, for several reasons. First, they are general enough to use for deriving rules for higher-level constructs like generators from their macro-expansion. Second, the control operators we study are implemented as primitives in a real, mature language implementation (Racket), used in real software [46]. And finally, it is known [27] how these control operators interact with other useful control operators like `dynamic-wind` [38] (relevant to `finally` or `synchronized` blocks) and continuation marks [13]. Thus our Racket subset is a suitable basis for future extension, while we are unaware of established extensions of `shift/reset`, `shift0/reset0`, etc. with continuation marks or `dynamic-wind`.

The operators we study can express loops, exceptions, coroutines [39, 40], and generators [14]. Racket also includes *compositional* continuations, whose application extends the current context rather than discarding it, giving completeness with respect to some denotational models [66], and alleviating space problems when using `call/cc` to simulate other families of control operators (it is known to macro-express another popular form of delimited continuations, the combination of `shift` and `reset` [27]). Our technical report [32] extends our development to include compositional continuations as well.

One final point about the semantics worth noting is the presence of effect annotations on the reduction arrows. These semantics are further adapted from Flatt et al. [27] to "emit" the primitive effect of the reduction, which is typical of syntactic type safety proofs for effect systems, including ours (Section 6). They do not influence evaluation, but only mark a relationship between the reduction rules and static effects. Non-unit (non-I effects) arise from a choice of primitives that depends on the particular effect system studied.

3 Growing Sequential Effects: Control, Prophecies, and Blocking

To build intuition for our eventual technical solution, we motivate its components through a series of progressively more sophisticated use cases. We will use $\mathcal{T}(\Sigma)$ in all of our examples, because we find traces to be an effective way of explaining the difficulties the effect system must address related to program fragments (i.e., events) being repeated, skipped, or reordered. A reader may choose to impart security-specific meanings to these events (as Skalka et al. [68] do) or as any other protocol of personal interest (e.g., lock acquisition and release). However, our development in Section 4 is *not* specific to this effect quantale, but instead parameterized over an arbitrary effect quantale. Our goal is to develop a sequential effect system based on transforming an *underlying* base effect quantale Q into a structure we will call $\mathcal{C}(Q)$ with sequencing and join operations, and a unit effect. This ensures our transformation works for *any* valid effect quantale, which includes all sequential effect systems we are aware of [30, 31].

▶ Use Case 1 (Control-Free Programs). Since programs are not required to use control operators, our solution must include a restriction equivalent to the class of underlying effects to reason about. For example, if `event[`α`]` has effect $\{\alpha\} \in \mathcal{T}(\Sigma)$ and `event[`β`]` has effect $\{\beta\} \in \mathcal{T}(\Sigma)$, we should expect the effect of `event[`α`]; event[`β`]` to be somehow equivalent to sequencing those underlying effects – $\{\alpha\} \rhd \{\beta\} = \{\alpha\beta\}$. This suggests underlying effects should be at least a component of continuation-aware effects.

▶ Use Case 2 (Aborting Effects). The simplest control behavior we can use is to abort to a handler, and this interacts with both sequencing and conditionals. Consider:

```
        (% t ((if c (event[α]) (abort t 3)); event[β]) (λn. event[γ]) )
```

Assuming `c` is a variable (i.e., pure), there are two paths through this term:

- If `c` is true, the code will emit events α and β (in order), and not execute the handler
- If `c` is false, the code will abort to the handler, which will emit event γ.

So intuitively, the effect for this term should contain those traces; ideally the effect would be $\{\alpha\beta, \gamma\}$, containing only those traces. For an effect system to validate this effect for this term, it must not only track ordinary underlying effects, but also two aspects of the `abort` operation's behavior: it causes some code to be discarded (the `event[`β`]`), and it causes the handler to run. We can track this information by making effects pairs of two components: a set of behaviors up to an abort (which we will call the *control effect set*, since it tracks effects due to non-local control transfer), and an underlying effect for when no abort occurs. We could then give the body of the prompt the effect $(\{\mathsf{abort}(\{\epsilon\})\}, \{\alpha\beta\})$ to indicate that it either executes normally producing a trace $\alpha\beta$, or it aborts to the nearest handler after doing nothing (more precisely, after performing actions with the unit effect for $\mathcal{T}(\Sigma)$). The type rule for prompts can then recognize the body may abort, and for each possible prefix of an aborting execution, add into the overall (underlying) effect of the prompt the result of sequencing that prefix with the *handler*'s effect: here, $(\{\epsilon\} \rhd \{\gamma\}) \sqcup \{\alpha\beta\} = \{\alpha\beta, \gamma\}$.

Above the body effect was given as a whole from intuition, but in general this must be built compositionally from subexpression effects, motivating further questions. First, what is the effect of the subterm `abort t 3`? In particular, what is its underlying effect? One sound choice would be I ($\{\epsilon\}$ for $\mathcal{T}(\Sigma)$), but this would introduce imprecision: it would produce effects suggesting it was possible to execute only event β (since the conditional's underlying effect would include both α and the empty trace ϵ, sequenced with β). Instead, we will make the underlying component *optional*, writing $\bot$ when it is absent. We will continue to use metavariables Q to indicate a definitely present element of the underlying effect quantale, but will use the convention of underlining metavariables (e.g., $\underline{Q}$) when they may be $\bot$. This permits the conditional's underlying effect to only contain the trace α, because joining the branches' effects can simply ignore the missing underlying effect from the aborting branch.

Second, we should consider how the sequencing and join operations interact with abort effects. While component-wise union/join is a natural (and working) starting point, sequencing is less obvious. Prefixing the last example's body with an extra event is instructive:

```
(% t (event[δ]; (if c (event[α]) (abort t 3)); event[β]) (λn. event[γ]) )
```

Execution could generate two traces: $\delta\alpha\beta$ and $\delta\gamma$. So *both* traces in the effect for the last body should gain this δ prefix: not only the underlying effect component, but also the portion related to the `abort`. This suggests the following definition of sequencing:

$$(C_1, \underline{Q_1}) \rhd (C_2, \underline{Q_2}) = (C_1 \cup (\underline{Q_1} \rhd C_2), \underline{Q_1} \rhd \underline{Q_2})$$

Assuming there is a lifting (given later) of the underlying sequencing to possibly-absent underlying effects ($\underline{Q_1} \rhd \underline{Q_2}$), this reflects the natural ways of combining paths through these terms: the *control effect set* collecting abort behaviors should include both the abort behaviors from C_1 (an execution corresponding to one of those behaviors means nothing from the second effect will execute), as well as the result of first executing *normal* behaviors of the first effect (e.g., $\{\delta\}$) before the aborting behaviors of the second – $\underline{Q_1} \rhd C_2$. So the effect of this new example's prompt body should be the join of the two branches $((\{\mathsf{abort}(\{\epsilon\})\}, \bot) \sqcup (\emptyset, \{\alpha\}) = (\{\mathsf{abort}(\{\epsilon\})\}, \{\alpha\}))$, sequenced between the effects of event δ and event β: $(\emptyset, \{\delta\}) \rhd (\{\mathsf{abort}(\{\epsilon\})\}, \{\alpha\}) \rhd (\emptyset, \{\beta\}) = (\emptyset, \{\delta\}) \rhd (\{\mathsf{abort}(\{\epsilon\})\}, \{\alpha\beta\}) = (\{\mathsf{abort}(\{\delta\})\}, \{\delta\alpha\beta\})$. Repeating our informal prompt handling above gives us the new expected underlying effect $\{\delta\alpha\beta, \delta\gamma\}$. We refer to the way the control effect set accumulates prefixes on the left as *left-accumulation* of effects. This definition is associative, and distributes over the component-wise union/join. For now we continue with the simplifying assumption that all tags are `t` (equivalent to *untagged* continuations), and consider issues with multiple tags in Use Case 6. A final detail deferred to Section 4 is that the value thrown by an `abort` must be of the type expected by the corresponding handler.

This is already part-way to our goal of deriving type rules for common control operators from delimited continuations: the work so far supports basic checked exceptions:

$$[\![\mathsf{try}\ e\ \mathsf{catch}\ C \Rightarrow e_c]\!] = (\%\ C\ [\![e]\!]\ [\![e_c]\!]) \qquad [\![\mathsf{throw}_C e]\!] = (\mathsf{abort}\ t_C\ [\![e]\!])$$

The formal rules for prompts and aborts will directly dictate the rules for these macro-expressions of checked exceptions, just as the informal discussion here transfers directly to these macro-expressed checked exceptions.

▶ Use Case 3 (Invoking Simple Continuations). Operationally, invoking an existing continuation is similar to using `abort` in that it discards the surrounding context up to the nearest prompt. But unlike uses of `abort`, invoking a continuation does not cause handler execution – instead (by E-InvokeCC) the prompt and handler remain in place. For the type rule for prompts to treat this additional mechanism, effects must indicate that invocation may occur. We can do this by extending effects slightly: instead of C in the effect $(C, \underline{Q})$ only containing prefixes of aborting computations, it should also include prefixes of continuation invocations – this is why elements of C were labeled $\mathsf{abort}(-)$ in the previous example, and we can now include effects tagged by $\mathsf{replace}(-)$ to indicate control behaviors that *replace* the current continuation with a new one. Considering a term invoking a continuation `k`:

```
(% t (event[δ]; (if c (event[α]) (k ())); event[β]) (λn. event[γ]) )
```

As in the previous example, one possible trace of this program is $\delta\alpha\beta$ when `c` is true. When `c` is false, however, this program will emit event δ, then invoke `k`, replacing the rest of the body with `k`'s captured continuation (with unit in its hole). Thus typing this requires also knowing additional information about `k`, not required in the `abort` case. We require continuations to carry a latent effect similar to functions: while the latent effect of a function $(\lambda x.e)$ describes the effect of the term obtained by substituting an appropriately-typed value v into e – the effect of $e[v/x]$ – the latent effect of a continuation $(\mathsf{cont}^{\tau}\ \ell\ E)$ describes the effect of plugging an appropriately-typed value v into E's hole – the effect of $E[v]$.

Assuming `k` has only underlying effects – e.g., if $k = (\mathsf{cont}^{\mathsf{unit}}\ t\ (\bullet; \mathsf{event}[\eta]))$, then we should expect the type rule for invocation to take that underlying effect from `k`'s latent effect and move it to the control effect set, under $\mathsf{replace}(-)$. So if `k`'s latent effect is $(\emptyset, \{\eta\})$, the effect of `(k ())` should be $(\{\mathsf{replace}(\{\eta\})\}, \bot)$. Sequential composition should be extended to treat $\mathsf{replace}$ control effects similar to abort control effects, by accumulating on the left. With that adjustment, we can conclude the body above has effect $(\{\mathsf{replace}(\{\delta\eta\})\}, \{\delta\alpha\beta\})$. The type rule for prompts can treat these similarly to abort control effects, but without sequencing them with the handler (which doesn't execute in this case), producing an overall effect $(\emptyset, \{\delta\alpha\beta, \delta\eta\})$. Notice that this is slightly different from $\mathsf{abort}(-)$ control effects: those track *only* the prefix effect before the abort, but the approach just outlined would have $\mathsf{replace}(-)$ control effects include prefix effects before invoking the continuation *and* the underlying effect of behaviors *after* the control operation. This corresponds to the location of the behavior that executes after the control operation – remote for aborts (the handler is non-local) and local for continuation invocation (the continuation is at the call site).

▶ Use Case 4 (Invoking Continuations that Abort or Invoke Other Continuations). The example above assumed `k` had only underlying effects, but in general `k`'s body might use `abort` or invoke other continuations. In such cases, `k`'s latent effect would be some pair $(C, \underline{Q})$ for non-empty control effect set C. It turns out simply treating those naïvely – including them into the control effect set for invoking `k` – is adequate for now (we revisit this when considering multiple tags). If $k = (\mathsf{cont}^{\mathsf{unit}}\ t\ (\bullet; \mathsf{event}[\eta]; \mathsf{abort}\ t\ 3))$ in the previous example, then the latent effect will be $(\{\mathsf{abort}(\{\eta\})\}, \bot)$, and simply making the effect of `(k ())` be the same (dropping the absent underlying latent effect since the continuation can only return via control operators, but making the application's underlying effect $\bot$ because by definition it does not return directly) gets the expected result at the prompt, including the trace $\delta\eta\gamma$ from emitting δ, invoking `k`, emitting η (from `k`'s restored body) and aborting to the handler. Because the latent control effects from `k`'s body already includes the prefixes from the start of `k`'s body to the `abort`, the existing left accumulation in our definition of $\rhd$ correctly accumulates prefixes from the site of continuation invocation, into the continuation, to its uses of control operations. In general (for a single tag), invoking a continuation with latent effect (C, Q) has effect $(C \cup \{\mathsf{replace}(Q)\}, \bot)$, though this assumes a non-empty underlying effect for the continuation – an assumption the final type rules will need to relax, along with extension for multiple tags, and issues with the continuation's argument and result type.

▶ Use Case 5 (Capturing Continuations). Typing uses of `call/cc` is the most complex problem this effect system must address. For a term `(call/cc t (λk. e))`, the rule must type the body function, which means choosing a type for the variable `k` that will be bound to the continuation. We defer continuation argument and result types for now. The latent effect of the continuation parameter depends on the effect of the code in the captured context – code that is (at runtime) "between" the `call/cc` and the (dynamically) nearest prompt. Consider applying a purely local type rule for `call/cc` to this simple example:

```
        (% t ((call/cc t (λk. e)); (foo 3)) ...)
```

Here the context captured will clearly be `(•; (foo 3))`. This is awkward when typing the subterm `(call/cc t (λk. e))`, because that context is not a subterm of what is being typed.

We will take a "guess and check" approach[1] to typing `call/cc`, assuming a certain latent effect and ensuring it can be checked elsewhere when additional information is available (in our case, in the type rule for prompts), essentially a form of *prophecy* [2, 3]. We track prophecies in a third (final) component, the *prophecy set*, resulting in three-component effects $(P, C, \underline{Q})$. We call these three-part effects *continuation effects*, using metavariable χ. An individual prophecy records the assumed latent effect of a continuation. Since that continuation may capture further continuations, the prophecy must predict a full continuation effect, making prophecies and continuation effects mutually recursive.

Prophecies alone are only local guesses about non-local phenomena; the effect system requires a way to validate them. Intuitively, a prophecy that a `call/cc` captures a continuation with latent effect (P, C, Q) is valid if in any dynamic context the `call/cc` is evaluated, the effect of the program text "between" the capture and the nearest prompt has an effect less than (P, C, Q) in the partial order (to be defined). Checking this statically is non-trivial. While the full context captured is visible in `(% t ((call/cc t (λk. e)); event[α]) (λn. ...))`, enclosing the `call/cc` in a function makes it non-local, and that function may be invoked in multiple contexts. Minor changes to the context may also (need to) break typability: if this `call/cc`'s body requires that the context has only underlying effect $\{\alpha\}$, then adding an additional `event` to the end of the prompt's body must make the expression untypable.

[1] In the sense that a human constructing a typing derivation involving `call/cc` would need to informally guess a prophecy, then use the type rules to check that it was a sound guess.

We can again turn to the idea of accumulation, but *to the right.* Thus we write individual prophecies (again, ignoring tags and argument and result types of continuations) as prophecy χ obs χ' – indicating that for a certain `call/cc`, an effect of χ was prophecized, meaning that the term whose effect contains this prophecy assumed a latent effect χ for the continuation, and the effect of the term fragment between the `call/cc` and the boundary of the term has effect χ', which we call the *observation.* We will extend sequential composition to accumulate effects to the right into the observation. When type checking a prompt, there is no more to accumulate because the prompt is the boundary of the continuation capture: at that point, the observation reflects the *actual* effect of the code *statically* on control flow paths between `call/cc` and the prompt, so the prophecy was sound if the observed effect is less than the prophecy effect.

With this prophecy-and-observation approach, let us consider:

```
        (% t (((call/cc t (λk. e)); event[α]); event[β]) (λn. ...))
```

The type rule for `call/cc` can assume a surrounding context effect of $(\emptyset, \emptyset, \{\alpha\beta\})$. Let us assume for simplicity the inner body of the `call/cc` is pure (unit effect). Then the effect of the `call/cc` term can be $(\{\text{prophecy } (\emptyset, \emptyset, \{\alpha\beta\}) \text{ obs } (\emptyset, \emptyset, I)\}, \emptyset, I)$ (writing the unit effect as simply I for readability). Then the effect of sequencing that with the `event[α]` (i.e., $(\emptyset, \emptyset, \{\alpha\})$) should both update the underlying effect *and the observation* of the prophecy: $(\{\text{prophecy } (\emptyset, \emptyset, \{\alpha\beta\}) \text{ obs } (\emptyset, \emptyset, \{\alpha\})\}, \emptyset, \{\alpha\})$. Repeating this for the next event, we would get $(\{\text{prophecy } (\emptyset, \emptyset, \{\alpha\beta\}) \text{ obs } (\emptyset, \emptyset, \{\alpha\beta\})\}, \emptyset, \{\alpha\beta\})$. At that point, this has typed the entire body of the prompt, and the type rule for the prompt can check that the observed effect is no greater than the prophecized effect – in this case trivially since they are equal.

▶ Use Case 6 (Trouble with Tags). Work with delimited continuations typically formalizes work without tags, but then adds them to the implementation as a “straightforward” extension. There are, however, several ways in which handling multiple tags is *non-trivial* in this context, warranting an explicit treatment. First, nested continuations result in more subtlety when handling control effect sets in the prompt rule. An obvious change is for a prompt tagged `t` to ignore aborts and continuation invocations (elements of the control effect set) that target a different tag (which requires tracking the target tag for each replace or abort effect). However, this is insufficient. A continuation `k` that restores up to tag t may include a latent abort to tag $t2$ – but if the nearest prompt is tagged $t2$, this is subtle. Consider the continuation $k = (\text{cont}^{\text{unit}}\ t\ (\bullet; \text{abort}\ t2\ 3))$ in the context of:

```
        (% t2 (% t (% t2 ((k ()); event[α]) ...) ...) (λn. event[β]))
```

`k` is invoked nested inside multiple prompts of different tags: when `k` is restored, it will discard and replace the inner `t2` prompt (white background) entirely. This is important: after context restoration, the `abort` inside `k` will be executed, passing 3 to the *outermost* handler and emitting event β: the innermost `t2` prompt contains an abort to `t2`, yet the innermost handler will not execute; the *outermost* handler will.

```
(% t2 (% t ((); abort t2 3) ...) (λn. event[β])) ⇒* (λn. event[β]) 3
```

Our initial effect for invoking continuations with further control effects (Use Case 4) would have the inner prompt’s body effect contain abort $t2$ $\{\epsilon\}$, which our suggested prompt handling would then join into the *innermost* prompt due to the matching tag. This has two problems: it spuriously suggests the innermost prompt’s handler could execute, introducing a kind of imprecision; and because that abort effect would no longer be present in the effect of the outermost prompt’s body, it would be missed that the outer handler *could* run, making the approach unsound. We must refine our approach for these “jumps.”

$c \in \text{ControlEffect}$	$::=$	$\mathsf{replace}\ \ell : Q \rightsquigarrow \tau$	Invoked continuation up to ℓ, with prefix & underlying effect Q, continuation result type τ
	$\mid$	$\mathsf{abort}\ \ell\ Q \rightsquigarrow \tau$	Abort up to ℓ after effects Q, throwing value of type τ
	$\mid$	$\boxed{c}_\ell$	Blocked control effect, frozen until nearest prompt for ℓ (triggered inside restored continuation targeting ℓ)
$p \in \text{Prophecy}$	$::=$	$\mathsf{prophecy}\ \ell\ \chi\ \rightsquigarrow \tau\ \mathsf{obs}\ \chi'$	Prophecy of latent continuation effect χ, effect χ' observed since point of prophecy (`call/cc`)
	$\mid$	$\boxed{p}_\ell$	Prophecy blocked until ℓ
$\chi \in \text{ContinuationEffect}$	$=$	$\mathsf{set\ Prophecy} \times \mathsf{set\ ControlEffect} \times \mathsf{option\ UnderlyingEffect}$	

Figure 2 Grammar of continuation effects over an existing effect quantale $Q \in \text{UnderlyingEffect}$.

We resolve this by adding one final nuance to control effect sets. We allow control effects to be either basic (the replace or abort effects we have already seen, with target tags) or *blocked* $\boxed{c}_t$ for a control effect c. A blocked control effect $\boxed{c}_t$ is ignored by prompts (left in the control effect set) until a prompt tagged t is reached – that prompt will *unblock* the effect, leaving c in the control effect set of the prompt. When invoking a continuation, instead of simply including the latent control effects in the effect of the invocation, the type rule will include the latent control effects *blocked* until the target tag of the continuation. So in the example above, the inner prompt's body effect would instead include $\boxed{\mathsf{abort}\ t2\ \{\epsilon\}}_t$. This would be ignored (propagated) by the inner prompt's typing (so the inner handler would not be spuriously considered), unblocked by the middle prompt's typing, and finally resolved in the outer prompt's typing (which would trigger consideration of the outer handler). Because the overall effect of any execution path that triggers such blocked control effects must still execute code along the way to the continuation invocation, blocked control effects still accumulate on the left.

Prophecies raise similar issues that lead to similar introduction of blocked prophecies: if `k` above instead captured a continuation up to a prompt tagged *t2*, the white-background portion of the term discarded when `k` was invoked would not be part of the captured continuation, so it should not be incorporated into the observation part of a prophecy. Because of this, blocked prophecies *must not accumulate while blocked.* The difference is this: blocked control effects continue to accumulate on the left because control operations do not discard code that occurs on the way to a control effect, while blocked prophecies must "skip over" the code discarded by control operations, which always appears to the right (later in source program order).

4 Continuation Effects

This section makes the outlines of the previous section precise, and fills in missing details (such as coordinating the type of a value thrown with the argument type of a handler). Figure 2 defines the effects of $\mathcal{C}(Q)$ derived from the examples above, for underlying effect quantale Q. Continuation-aware effects of an underlying effect quantale Q are effects χ of three components: a prophecy set P, a control effect set C, and an optional underlying effect $\underline{Q}$. Basic control effects include effects representing aborts to a tagged prompt ($\mathsf{abort}\ \ell\ Q \rightsquigarrow \tau$) or invoking continuations that replace the context up the nearest tagged prompt ($\mathsf{replace}\ \ell : Q \rightsquigarrow \tau$), as suggested by Use Cases 2 and 3. These versions are additionally tagged with the specific prompt tag they target (ℓ), and each carries a type τ – for replacement this is the result type of the restored continuation, and for aborts this is the type of the value thrown (each intuitively a kind of result type for each control behavior). Both control effects and prophecies may also be blocked until a prompt with a certain tag if they originate inside a continuation that was invoked (per Use Case 6). Note that blocking constructors may nest arbitrarily deeply, because one restored continuation may restore another continuation which may restore another... and so on. For a term with effect $(P, C, \underline{Q})$:

- The prophecy set P contains prophecies for all uses of `call/cc` within the term (some possibly blocked if introduced by restoring a continuation).
- The control effect set C describes all possible exits of the term via control operations (abort or continuation invocation). For aborts of continuation invocation that occur directly, C will contain basic control effects. For aborts or continuation invocation that may occur in the body of a restored continuation, there will be blocked control effects.
- The underlying effect $\underline{Q}$ describes (an upper bound on) the underlying effect of any execution of the term that does not exit via control operator.

To define sequencing and join formally, we must first lift the underlying effect quantale's operators to deal with missing effects:

$$\underline{Q_1} \rhd \underline{Q_2} = \begin{cases} \top & \text{if } \underline{Q_1} = \top \vee \underline{Q_2} = \top \\ \bot & \text{if } \underline{Q_1} = \bot \vee \underline{Q_2} = \bot \\ Q_1 \rhd Q_2 & \text{otherwise} \end{cases} \quad \underline{Q_1} \sqcup \underline{Q_2} = \begin{cases} \top & \text{if } \underline{Q_1} = \top \vee \underline{Q_2} = \top \\ Q_1 & \text{if } \underline{Q_2} = \bot \\ Q_2 & \text{if } \underline{Q_1} = \bot \\ Q_1 \sqcup Q_2 & \text{otherwise} \end{cases}$$

We also require a way to prefix control effects with an underlying effect, to implement left accumulation (recursively), extending the ideas of Use Cases 2, 3, and 6 to the final definition:

$$\begin{aligned} Q \rhd \boxed{c}_\ell &= \boxed{Q \rhd c}_\ell \\ Q \rhd \mathsf{replace}\ \ell : Q' \rightsquigarrow \tau &= \mathsf{replace}\ \ell : (Q \rhd Q') \rightsquigarrow \tau \\ Q \rhd \mathsf{abort}\ \ell\ Q' \rightsquigarrow \tau &= \mathsf{abort}\ \ell\ (Q \rhd Q') \rightsquigarrow \tau \end{aligned}$$

and we will lift this to operate on control effect *sets* and possibly-absent underlying effects:

$$\underline{Q} \rhd C = \mathsf{if}\ (\underline{Q} = \bot)\ \mathsf{then}\ \emptyset\ \mathsf{else}\ (\mathsf{map}\ (Q \rhd \_)\ C)$$

Likewise, we must define a means of right-accumulating in prophecies:

$$\begin{aligned} &\boxed{\mathsf{prophecy}\ \ell\ (P, C, \underline{Q}) \rightsquigarrow \tau\ \mathsf{obs}\ (P', C', \underline{Q'})}_{\ell'} \blacktriangleright \chi'' = \boxed{\mathsf{prophecy}\ \ell\ (P, C, \underline{Q}) \rightsquigarrow \tau\ \mathsf{obs}\ (P', C', \underline{Q'})}_{\ell'} \\ &\mathsf{prophecy}\ \ell\ (P, C, \underline{Q}) \rightsquigarrow \tau\ \mathsf{obs}\ (P', C', \underline{Q'}) \blacktriangleright (P'', C'', \underline{Q''}) \\ &\quad = \mathsf{prophecy}\ \ell\ (P, C, \underline{Q}) \rightsquigarrow \tau\ \mathsf{obs}\ ((P' \blacktriangleright (P'', C'', \underline{Q''})) \cup P'', C' \cup (\underline{Q'} \rhd C''), \underline{Q'} \rhd \underline{Q''}) \end{aligned}$$

which we also lift to operate on prophecy *sets* (not shown, but analogous to the lifting of left-accumulation). Finally, this is enough to define sequencing and join:

$$\begin{aligned} (P_1, C_1, \underline{Q_1}) \sqcup (P_2, C_2, \underline{Q_2}) &= (P_1 \cup P_2, C_1 \cup C_2, \underline{Q_1} \sqcup \underline{Q_2}) \\ (P_1, C_1, \underline{Q_1}) \rhd (P_2, C_2, \underline{Q_2}) &= ((P_1 \blacktriangleright (P_2, C_2, \underline{Q_2})) \cup P_2, C_1 \cup (\underline{Q_1} \rhd C_2), \underline{Q_1} \rhd \underline{Q_2}) \end{aligned}$$

Joins are implemented component-wise, using set union on prophecy or control effect sets, and the (option-lifted) join from the underlying effect quantale. The sequencing operator, and its relation to the accumulation on prophecies, is a bit complex and warrants some further explanation. The sequence operator $\rhd$ is defined according to the ideas driven by Use Cases 2, 3, and 5. Underlying effects are sequenced by reusing the underlying effect quantale. Control effects are handled by left-accumulating. In fact, in the case where there are no prophecies (`call/cc`s) involved, the handling of the control effect sets and underlying effects is exactly as in Use Case 2, just extended for the additional control effects. Deferring the right-accumulation $\blacktriangleright$ for one further moment, the full sequencing operator produces a resulting prophecy set as the union of prophecies from the second effect (which are unaffected by the first) with the result of the first effect's prophecies accumulating effects from the right, since that effect will (in the type rules) correspond to behavior that will be part of the captured continuation. The right accumulation for prophecies implemented by $\blacktriangleright$ essentially just implements sequential composition of the observation with the accumulated effect – the recursive use of $\blacktriangleright$ could equivalently just be $\rhd$, but direct recursion is easier to prove things about than mutual recursion. As suggested by Use Case 6, blocked prophecies do not accumulate. It is easy to confirm that the unit element for $\rhd$ is $(\emptyset, \emptyset, I)$, where I is the identity of the underlying effect quantale.

▶ Remark 2 (Sets of Underlying Effects). We have described most of the structure of lifting an effect quantale to support delimited control: sequencing and join operators that distribute over each other, along with a unit for sequencing. We have not yet stated whether the result $\mathcal{C}(Q)$ of the lifting is an effect quantale. As described so far, it is not quite an effect quantale: there is no single distinguished top in the partial order induced by $\sqcup$: for any effect $(P, C, \underline{Q})$, a larger effect can be obtained by adding new control effects or prophecies. And because the underlying $\top$ can appear in multiple ways, conceptually many different incomparable effects should be considered erroneous. Introducing a distinguished top element Err, and wrapping the sequencing and join definitions above with an additional operation producing Err any time the operations above produce effects containing underlying top. This would produce an effect quantale, but we take an alternative approach that also enhances flexibility, without adding special cases for $\top$ throughout the system.

Using *sets* of structures containing underlying effects can lead to the extra set structure being "too picky" in distinguishing effects, in the sense of distinguishing intuitively equivalent effects. Consider the following join:

$$(\emptyset, \{\mathsf{abort}\ \ell\ \{\alpha\}\}, \bot) \sqcup (\emptyset, \{\mathsf{abort}\ \ell\ \{\beta\}\}, \bot) = (\emptyset, \{\mathsf{abort}\ \ell\ \{\alpha\}, \mathsf{abort}\ \ell\ \{\beta\}\}, \bot)$$

which could be the effect of `if c (event[`α`]; abort `ℓ` 3) (event[`β`]; abort `ℓ` 3)`. Their join is *not* the very similar $(\emptyset, \{\mathsf{abort}\ \ell\ \{\alpha, \beta\}\})$, which also indicates an abort after one of the two same underlying effects, and is the effect of a minor rewrite of the last expression: `(if c (event[`α`]) (event[`β`])); abort `ℓ` 3`. Both effects indicate executing α or β before aborting, and replacing one with the other inside a prompt body will not change the *prompt*'s effect (per Use Cases 2 and 3): the prompt will sequence each with the handler effect, and join them together). Yet because $\sqcup$ is defined using set union, they are incomparable in the induced partial order $x \sqsubseteq y \leftrightarrow x \sqcup y = y$, because joining them yields a *third* effect: $(\emptyset, \{\mathsf{abort}\ \ell\ \{\alpha\}, \mathsf{abort}\ \ell\ \{\beta\}, \mathsf{abort}\ \ell\ \{\alpha, \beta\}\}, \bot)$. This is not a valuable distinction. We would like at least $(\emptyset, \{\mathsf{abort}\ \ell\ \{\alpha\}, \mathsf{abort}\ \ell\ \{\beta\}\}, \bot) \sqsubseteq (\emptyset, \{\mathsf{abort}\ \ell\ \{\alpha, \beta\}\})$ because every abort prefix on the left is over-approximated by an abort prefix on the right.[2] One way to achieve this would be to replace the set union of control effect sets or prophecy sets in the sequencing and join operators with operators that also recursively joined the underlying effects of all aborts to the same tag, joined the underlying effects of all replacements to the same tag, and joined the observed effects of all prophecies to the same tag with the same prediction (plus joining types). The partial order induced by this modification would establish this desirable order.

Directly extending $\sqcup$ and $\rhd$ as given to *both* recursively join when combining sets and lift underlying $\top$ to a special Err would yield a proper effect quantale, but add significant complexity that is orthogonal to the key ideas of our approach. *Instead*, we make two

[2] While in this $\mathcal{T}(\Sigma)$ example the effects are equivalent, examples in other effect quantales only justify $\sqsubseteq$.

$$C_1 \sqsubseteq C_2 = \bigwedge \begin{cases} \forall \ell, Q, \tau.\, \mathsf{replace}\ \ell : Q \rightsquigarrow \tau \in C_1 \Rightarrow \exists Q'.\, \mathsf{replace}\ \ell : Q' \rightsquigarrow \tau \in C_2 \wedge Q \sqsubseteq Q' \\ \forall \ell, Q, \tau.\, \mathsf{abort}\ \ell\ Q \rightsquigarrow \tau \in C_1 \Rightarrow \exists Q'.\, \mathsf{abort}\ \ell\ Q' \rightsquigarrow \tau \in C_2 \wedge Q \sqsubseteq Q' \end{cases}$$

$$P_1 \sqsubseteq P_2 =$$

$$(\forall \ell, \chi, \chi', \tau.\, \mathsf{prophecy}\ \ell\ \chi\ \ \rightsquigarrow \tau\ \mathsf{obs}\ \chi' \in P_1 \Rightarrow \exists \chi''.\, \mathsf{prophecy}\ \ell\ \chi\ \ \rightsquigarrow \tau\ \mathsf{obs}\ \chi'' \in P_2 \wedge \chi' \sqsubseteq \chi'') \wedge$$
$$(\forall \ell, \ell', \chi, \chi', \tau.\, \boxed{\mathsf{prophecy}\ \ell\ \chi\ \ \rightsquigarrow \tau\ \mathsf{obs}\ \chi'}_{\ell'} \in P_1 \Rightarrow \exists \chi''.\, \boxed{\mathsf{prophecy}\ \ell\ \chi\ \ \rightsquigarrow \tau\ \mathsf{obs}\ \chi''}_{\ell'} \in P_2 \wedge \chi' \sqsubseteq \chi'')$$

$$(P, C, \underline{Q}) \sqsubseteq (P', C', \underline{Q'}) \leftrightarrow P \sqsubseteq P' \wedge C \sqsubseteq C' \wedge \underline{Q} \sqsubseteq \underline{Q'}$$

$$\chi \approx \chi' = (\mathsf{NoUnderlyingTop}(\chi, \chi') \wedge \chi \sqsubseteq \chi' \wedge \chi' \sqsubseteq \chi) \vee (\mathsf{UnderlyingTop}(\chi) \wedge \mathsf{UnderlyingTop}(\chi'))$$

Figure 3 Direct partial order and equivalence on continuation effects.

adjustments. First, we define a direct partial order on continuation effects in Figure 3, where an effect χ is less than another effect χ' if every prophecy observation, abort effect, replacement effect, or underlying effect in χ is over-approximated by *some* corresponding component in χ'. This captures the intuition behind the desirable ordering outlined above. We also define a corresponding equivalence relation $\approx$ on continuation effects, which equates all continuation effects containing an underlying $\top$ (anywhere in the effect) and otherwise uses the partial order to induce equivalence. In the type rules considered in Section 4, this is the notion of subeffecting used (rather than the traditional partial order derived from a join), and all effects are considered modulo the equivalence relation. Quotienting $\mathcal{C}(Q)$ by the equivalence $\approx$ yields a proper effect quantale, equivalent to the direct but verbose version outlined above. See our technical report [32]) for further details.

We consider a type-and-effect system for a language with the constructs from Figure 1. Our extended technical report [32] extends these results for composable continuations. Our expressions and types are:

$$\begin{array}{rl} \text{Expressions } e ::= & p \mid \lambda x.\, e \mid (e\ e) \mid (\%\ \ell\ E\ v) \mid (\mathsf{call/cc}\ \ell\ e) \mid (\mathsf{abort}\ \ell\ e) \mid (\mathsf{cont}\ \ell\ E) \mid \mathsf{if}\ e\ e\ e \\ \text{Values } v ::= & (\lambda x.\, e) \mid \mathsf{cont}\ \ell\ E \mid v_p \\ \text{Types } \tau, \gamma ::= & \mathsf{unit} \mid \mathsf{bool} \mid \tau \xrightarrow{\chi} \tau \mid (\mathsf{cont}\ \ell\ \tau\ \chi\ \tau) \mid \mu X.\, \tau \end{array}$$

p and v_p are parameters of the system following Gordon's work [30, 31]: primitives (which can include operations such as locking primitives or `event[`α`]`) and primitive values (e.g., for encoding locations). Gordon's soundness framework also parameterizes operational semantics by an abstract notion of state, and semantics for primitives manipulating state; we assume (and later use) a similar framework, which admits a range of concrete examples.

Types include common primitive types, function types with latent effects, equirecursive types (needed for typing loops), as well as a type for continuation values that we discuss with the type rule for invoking continuations. The type rules for lambda abstraction, function application, and conditionals are as in Section 2 (though using continuation effects), so we do not discuss them further. Typing uses of primitives requires additional rules and parameters to define the additional types (e.g., lock or location types) and their relationship to operational primitives, following Gordon [30, 31]. For intuition, readers may assume p is simply the `event[_]` primitive from our running example. We include subtyping (<:), including the standard type-and-effect subsumption rule, and function subtyping that is covariant in the body's latent effect. Figure 4 gives central type rules for this paper, for prompts, aborts, continuation capture, and continuation invocation.

We will discuss the type rules in relation to the Use Cases from Section 3.

T-Abort handles Use Case 2. As suggested in that discussion, the effect of an abort is to introduce a control effect signifying an abort to the targeted label, with an absent underlying effect. The initial prefix of the abort is "empty" – the underlying unit effect I – and the

$$\textsc{V-Effects}\ \dfrac{\begin{array}{c}\forall \tau', Q.\,(\mathsf{abort}\ \ell\ Q \rightsquigarrow \tau') \in \ulcorner C \lrcorner_\ell \Rightarrow \tau' <: \sigma \quad \forall Q, \tau'.\,(\mathsf{replace}\ \ell : Q \rightsquigarrow \tau') \in \ulcorner C \lrcorner_\ell \Rightarrow \tau' <: \tau \\ \left(\begin{array}{c}\forall \chi_{proph}, \tau', P_p, C_p, \underline{Q_p}.\ \mathsf{prophecy}\ \ell\ \chi_{proph} \rightsquigarrow \tau'\ \mathsf{obs}\ (P_p, C_p, \underline{Q_p}) \in \ulcorner P \lrcorner_\ell \Rightarrow \\ \ulcorner P_p \lrcorner_\ell \sqsubseteq \ulcorner P_{proph} \lrcorner_\ell \wedge \ulcorner C_p \lrcorner_\ell \sqsubseteq \ulcorner C_{proph} \lrcorner_\ell \wedge \underline{Q_p} \sqsubseteq \underline{Q_{proph}} \wedge \tau <: \tau'\end{array}\right)\end{array}}{\mathsf{validEffects}(P, C, \underline{Q}, \ell, \tau, \sigma)}$$

$$\textsc{T-Prompt}\ \dfrac{\Gamma \vdash e : \tau \mid (P, C, \underline{Q}) \qquad \Gamma \vdash h : \sigma \xrightarrow{(\emptyset,\emptyset,Q_h)} \tau \mid I \qquad \mathsf{validEffects}(P, C, \underline{Q}, \ell, \tau, \sigma)}{\Gamma \vdash (\%\ \ell\ e\ h) : \tau \mid (\ulcorner P \lrcorner_\ell \setminus^{Q_h} \ell, \ulcorner C \lrcorner_\ell \setminus \ell, \underline{Q} \sqcup \left(\bigsqcup \ulcorner C \lrcorner_\ell|_\ell^{Q_h}\right))}$$

$$\textsc{T-CallCont}\ \dfrac{\mathsf{NonTrivial}(\chi_k) \qquad \Gamma \vdash e : (\mathsf{cont}\ \ell\ \tau\ \chi_k\ \gamma) \xrightarrow{\chi} \tau \mid \chi_e}{\Gamma \vdash (\mathsf{call/cc}\ \ell\ e) : \tau \mid (\chi_e \rhd \chi) \rhd (\{\mathsf{prophecy}\ \ell\ \chi_k \rightsquigarrow \gamma\ \mathsf{obs}\ (\emptyset, \emptyset, I)\}, \emptyset, I)}$$

$$\textsc{T-AppCont}\ \dfrac{\Gamma \vdash k : \mathsf{cont}\ \ell\ \tau'\ (P, C, \underline{Q})\ \tau'' \mid \chi_k \qquad \Gamma \vdash e : \tau' \mid \chi_e}{\Gamma \vdash (k\ e) : \tau \mid \chi_k \rhd \chi_e \rhd (\boxed{P}_\ell, \boxed{C}_\ell \cup (\underline{Q} \rhd \{\mathsf{replace}\ \ell : I \rightsquigarrow \tau''\}), \bot)}$$

$$\textsc{T-Abort}\ \dfrac{\Gamma \vdash e : \tau \mid \chi_e}{\Gamma \vdash \mathsf{abort}\ \ell\ e : \sigma \mid \chi_e \rhd (\emptyset, \{\mathsf{abort}\ \ell\ I \rightsquigarrow \tau\}, \bot)}$$

Figure 4 Typing control operators with continuation effects.

control effect tracks the type of the thrown value. The overall effect of an `abort` expression sequences this after the effect of reducing e to a value, since this occurs earlier in evaluation order than the abort operation itself. A subtlety worth noting, is that this also ensures a `call/cc` inside e will correctly accumulate the pending abort in the captured context.

T-AppCont handles Use Cases 3 and 4. First consider the basic case where the invoked continuation has only simple underlying effects (i.e., P and C in the continuation's latent effect are both $\emptyset$). As with T-Abort, subterms are reduced to values before the control behavior occurs, so those effects are sequenced before the control behavior itself. With that taken care of, we may temporarily assume both k and e are already syntactic values to simplify discussion of effect. In this case the rule simplifies to exactly what the example in Use Case 3 suggested: the underlying effect is absent (because the term does not return normally, but via a control behavior), and a control effect is introduced reflecting that a continuation with underlying effect $\underline{Q}$ is invoked. A subtlety here is that we cannot simply write $\mathsf{replace}\ \ell : \underline{Q} \rightsquigarrow \tau''$, because $\underline{Q}$ may be $\bot$, which would make that control effect invalid. Instead we (ab)use the left-accumulation operator on control effects: prefixing the *unit-effect* replacement effect with $\underline{Q}$ will give the expected result when $\underline{Q}$ is present, and otherwise result in the empty set. The replacement effect also records the result type of the invoked continuation, and the rule also ensures the argument provided to the continuation is the expected type.

T-AppCont also handles Use Case 4, adjusted per Use Case 6: the latent control effects of the continuation are included, but *blocked until* ℓ, to ensure no prompt rule (discussed shortly) resolves those behaviors before the behaviors escape a prompt tagged ℓ. Unlike the discussion in Section 3, this finished rule also supports the case where the restored continuation contains (possibly-nested) uses of `call/cc`, blocking the latent prophecies as well.

The conclusion of T-AppCont critically overloads the syntax for constructing blocked prophecies and control effects, to block prophecy *sets* and control effect *sets*. This overload *almost* maps the appropriate blocking constructor over each set – however, it first checks that this will not result in a control effect of the form $\boxed{\boxed{C}_\ell}_\ell$ for some tag ℓ (similarly for prophecies). Such a control effect would represent a control effect that should propagate directly through *two* prompts tagged ℓ, but this is dynamically impossible: any number of nested restorations of continuations to the same tag remains within the same prompt, e.g.:

```
(% ℓ ((cont ℓ (•; ((cont ℓ (•; abort ℓ 5)) 4))) 3) h)
⇒ (% ℓ (3; ((cont ℓ (•; abort ℓ 5)) 4)) h) ⇒ (% ℓ ((cont ℓ (•; abort ℓ 5)) 4) h)
⇒ (% ℓ (4; abort ℓ 5) h) ⇒ (% ℓ (abort ℓ 5) h) ⇒ (h 5)
```

Naïvely mapping the blocking constructor would yield the control effect set $\{\boxed{\boxed{\mathsf{abort}\ \ell\ I \leadsto \mathsf{nat}}_{\ell}}_{\ell}\}$ for the body; our discussion with Use Case 6 about each prompt removing one layer of blocking (which we will see does inform T-Prompt) would then not pair the abort with the local handler that is invoked. With the modified mapping, the simplification of the control effect set is instead $\{\boxed{\mathsf{abort}\ \ell\ I \leadsto \mathsf{nat}}_{\ell}\}$ (only one layer of blocking), which will match the `abort` to the correct handler.

T-CallCont is a bit more subtle. Standard for any type rule for `call/cc`, the rule ensures the result type of the expression itself (τ) is also the return type of the body function (for executions that return normally) and the argument type assumed for the continuation (since the location of the `call/cc` becomes the hole the argument replaces at invocation). As suggested in the discussion of Use Case 5, the effect arising from the use of `call/cc` itself is a prophecy effect, recording the assumed latent effect of the captured continuation and the assumed result type of the captured continuation – both of which make their way into the assumed type of the argument to the `call/cc` body. The initial observation is empty, because this effect corresponds intuitively to the point in the execution from which the prediction begins – but this point is the heart of another key subtlety. As with other rules, the subterm e must be reduced before anything else, so its effect is sequenced before others. But naïvely ordering the body's (latent) effect would place it after the prophecy, as dynamically the continuation is captured before the body is evaluated (since the continuation becomes the argument in E-CallCC). However, this would result in the prophecy effect observing the body of the `call/cc` – which is incorrect, as that behavior will not be part of the captured continuation. Thus we place the prophecy *after* the body's latent effect.

For a small variation on Use Case 5's first example, this gives us:

$$(\%\ t\ \underbrace{(\ \overbrace{(\mathsf{call/cc}\ t\ (\lambda k.e))}^{\chi_e \rhd (\{\mathsf{prophecy}\ t\ (\emptyset,\emptyset,\{\alpha\}\ \mathsf{obs}\ (\emptyset,\emptyset,I))\},\emptyset,I)}\ ;\ \overbrace{\mathsf{event}[\alpha]}^{(\emptyset,\emptyset,\{\alpha\})}\ )}_{\chi_e \rhd (\{\mathsf{prophecy}\ t\ (\emptyset,\emptyset,\{\alpha\}\ \mathsf{obs}\ (\emptyset,\emptyset,\{\alpha\}))\},\emptyset,\{\alpha\})}\ ...)$$

The individual effects of the `call/cc` and `event` expressions (given above the term) simplify to the effect below the term. The prophecy from the `call/cc` observes the event that would be captured in its context. Because e's effect χ_e is to the *left* of the resulting prophecy, e's non-captured behavior is *not* observed, and the prophecy under the term will appear in the prophecy set of the overall prompt's body.

T-CallCont has one extra antecedent, constraining the prophecy to predict a *non-trivial* effect, to avoid degeneracy. The simplest problematic prediction would be to predict an effect of $(\emptyset, \emptyset, \bot)$. Consider the effect of the code `event[α]; (k ())`. The effect would be $(\emptyset, \emptyset, \{\alpha\}) \rhd (\emptyset, \emptyset, \bot) = (\emptyset, \emptyset, \bot)$. This is problematic: the term *does* have a behavior, but the effect reflects *no* behavior. This could be introduced by a circularity:

```
        (% t (let k = (call/cc t (λk. k)) in (event[α]; (k ()))) ...)
```

This term is in fact the macro-expansion for an infinite loop that executes the event forever. Assuming the degenerate latent effect in the `call/cc` gives `k` the degenerate effect, which gives the body of the the let-expression – the context captured by `call/cc` – a degenerate effect, allowing the observed effect to match the prophecy (both degenerate). Requiring a non-empty control effect set or underlying effect avoids this collapse. (This is not a termination-sensitivity issue; a terminating while loop has the same challenge, but a larger term.)

$$P \setminus^Q \ell = \{p \setminus^Q \ell \mid p \in P \land \mathsf{OuterTag}(p) \neq \ell\} \quad C \setminus \ell = \{c \in C \mid \mathsf{OuterTag}(c) \neq \ell\}$$

$$C|^Q_\ell = \{Q' \rhd Q \mid \mathsf{abort}\ \ell\ Q' \rightsquigarrow \_ \in C\} \cup \{Q' \mid \mathsf{replace}\ \ell : Q' \rightsquigarrow \_ \in C\}$$

$$\ulcorner \mathsf{prophecy}\ \ell'\ (P, C, Q) \rightsquigarrow \tau\ \mathsf{obs}\ (P', C', Q') \lrcorner_\ell = \mathsf{prophecy}\ \ell'\ (P, C, Q) \rightsquigarrow \tau\ \mathsf{obs}\ (\ulcorner P' \lrcorner_\ell, \ulcorner C' \lrcorner_\ell, Q')$$

$$\ulcorner \boxed{\mathsf{prophecy}\ \ell'\ (P, C, Q) \rightsquigarrow \tau\ \mathsf{obs}\ (P', C', Q')}_\ell \lrcorner_\ell = \mathsf{prophecy}\ \ell'\ (P, C, Q) \rightsquigarrow \tau\ \mathsf{obs}\ (\ulcorner P' \lrcorner_\ell, \ulcorner C' \lrcorner_\ell, Q')$$

$$\ulcorner \boxed{\mathsf{prophecy}\ \ell'\ (P, C, Q) \rightsquigarrow \tau\ \mathsf{obs}\ (P', C', Q')}_{\ell''} \lrcorner_\ell = \boxed{\mathsf{prophecy}\ \ell'\ (P, C, Q) \rightsquigarrow \tau\ \mathsf{obs}\ (P', C', Q')}_{\ell''} \quad (\text{if } \ell \neq \ell'')$$

$$(\mathsf{prophecy}\ \ell'\ \chi \rightsquigarrow \tau\ \mathsf{obs}\ (P', C', Q')) \setminus^Q \ell = \mathsf{prophecy}\ \ell'\ \chi \rightsquigarrow \tau\ \mathsf{obs}\ (P' \setminus^Q \ell, C' \setminus \ell, Q' \sqcup (C'|^Q_\ell))$$

$$(\boxed{\mathsf{prophecy}\ \ell'\ \chi \rightsquigarrow \tau\ \mathsf{obs}\ (P', C', Q')}_\ell) \setminus^Q \ell = \mathsf{prophecy}\ \ell'\ \chi \rightsquigarrow \tau\ \mathsf{obs}\ (P' \setminus^Q \ell, C' \setminus \ell, Q' \sqcup (C'|^Q_\ell))$$

$$(\mathsf{prophecy}\ \ell'\ \chi \rightsquigarrow \tau\ \mathsf{obs}\ (P', C', Q')\ \mathsf{until}\ \ell'') \setminus^Q \ell = \mathsf{prophecy}\ \ell'\ \chi \rightsquigarrow \tau\ \mathsf{obs}\ (P', C', Q'))\ \mathsf{until}\ \ell'' \quad (\text{if } \ell \neq \ell'')$$

Figure 5 Auxilliary definitions used by type rules.

T-Prompt is the most complex type rule in the system, because it serves many roles. In addition to giving an overall type and effect to the prompt, it must check that:

- Any `abort` to this handler throws values whose type is a subtype of the handler's argument.
- Any continuation invoked in the body that will restore up to this prompt has a result type that is a subtype of the prompt's own result type.
- Any `call/cc` that captures a continuation delimited by this prompt was typed assuming the result was a *supertype* of the prompt's actual result type.
- Any `call/cc` that captures a continuation delimited by this prompt was typed assuming a valid latent effect for that continuation (i.e., that prophecies targeting that tag are upper-bounds on the observations).

The first three checks are handled by the subtyping constraints in the auxiliary judgment V-Effects. Notice that the antecedents of V-Effects quantify over elements of *unblocked* control effect sets and prophecy sets. Unblocking, written $\ulcorner - \lrcorner_\ell$ is defined for prophecies in Figure 5: it maps a corresponding per-element unblocking operation, which is a no-op on elements that were not blocked, a no-op on elements blocked until a *different* tag, and strips off a block constructor for ℓ if that is the outermost constructor. Unblocking for control effect sets is defined analogously. Intuitively, this corresponds to the fact that any control effect or prophecy blocked until a prompt for ℓ has now *reached* a prompt for ℓ.

The last of the checks above is handled by the prophecy-related antecedent of V-Effects, which *nearly* checks that the observations for a given prophecy are a subeffect of what was predicted – that would be $(P_p, C_p, Q_p) \sqsubseteq \chi_{proph}$, which would be sound but overly conservative. Instead, the comparison of the prophecy and observation, compares the *unblocked* versions of prophecy and control effect sets: $\ulcorner P_p \lrcorner_\ell \sqsubseteq \ulcorner P_{proph} \lrcorner_\ell$ and $\ulcorner C_p \lrcorner_\ell \sqsubseteq \ulcorner P_{proph} \lrcorner_\ell$. This is useful because if a context captured by `call/cc` contains an invocation of itself, the prophecy arising from T-CallCont will observe a *blocked* version of *its own prophecy* (from T-AppCont), making the naive subeffect check too conservative: no prophecy can predict a blocked version of itself. Rather than being an esoteric concern, this is actually quite practical: encodings of loops using delimited continuations do exactly this. Unblocking for the prompt's tag in V-Effects resolves this: just like the quantifications unblock because those sets have now reached the corresponding prompt, the prophecy validation must reflect that the observation has now "reached" the corresponding prompt.

Let us revisit the example of Use Case 5 discussed above with T-CallCont. The prophecy in that derivation contains no blocked prophecies, so T-Prompt will effectively check that the observation is less than the prophecy – that $(\emptyset, \emptyset, \{\alpha\}) \sqsubseteq (\emptyset, \emptyset, \{\alpha\})$, which trivially succeeds because the prophecy predicted its context's behavior exactly.

We can see the role of unblocking more clearly by revisiting the motivating example for requiring prophecized effects to be non-trivial (eliding types for brevity).

$$(\%\ t\ (\mathsf{let}\ k = \overbrace{(\mathsf{call/cc}\ t\ (\lambda k.\, k))}^{(\{\mathsf{prophecy}\ t\ (\emptyset,\{\mathsf{replace}\ t\ \alpha^*\},\bot)\ \mathsf{obs}\ (\emptyset,\emptyset,I))\},\emptyset,I)}\ \mathsf{in}\ (\overbrace{\mathsf{event}[\alpha]}^{(\emptyset,\emptyset,\{\alpha\})};\ (\overbrace{k\ ()}^{(\emptyset,\{\boxed{\mathsf{replace}\ t\ \alpha^*}_t\},\bot)})))\ldots)$$

The effect of the prompt's body is the sequencing (with $\rhd$) of the three individual subterm effects written above the term fragments, writing α^* for the set of all finite traces consisting of only α. Simplifying the sequencing of the two right-most effects first will result in a control effect set $\{\boxed{\mathsf{replace}\ t\ (\{\alpha\} \rhd \alpha^*)}_t\}$ (invoke the continuation after an α event, with aggregate behavior of some non-zero finite number of α events). Simplifying again, the prophecy will observe this, and T-Prompt will check (via V-Effects) that $\boxed{\{\boxed{\mathsf{replace}\ t\ (\{\alpha\} \rhd \alpha^*)}_t\}}_t \sqsubseteq \boxed{\{\mathsf{replace}\ t\ \alpha^*\}}_t$, which is equivalent to checking $\{\mathsf{replace}\ t\ (\{\alpha\} \rhd \alpha^*)\} \sqsubseteq \{\mathsf{replace}\ t\ \alpha^*\}$, which is true because $\{\alpha\} \rhd \alpha^* \sqsubseteq \alpha^*$ in $\mathcal{T}(\Sigma)$ (the set of non-empty finite traces containing only α is a subset of the set of possibly-empty finite traces containing only α).

Finally, T-Prompt must give a type and effect to the prompt expression itself. The underlying effect must include (1) the body's underlying effect, (2) the effect of any possible paths through the body that abort to the local handler and execute it (including those resulting from invoking continuations up to this prompt prior to aborting), and (3) the effect of any possible paths through the body that invoke one or more continuations up to this prompt before returning normally. (1) is simply $\underline{Q}$. (2) is the result of sequencing any abort prefix for ℓ in the (unblocked) control effect set (which will incorporate aborts resulting from continuation invocation as well). (3) is simply the set of replacement effects in the (unblocked) C. (2) and (3) are computed via a projection operator $-|_{\ell}^{Q_h}$ applied to the unblocked control effect set, defined in Figure 5. The superscript on the operator is the underlying effect of the handler (constrained to have no control effects), and the subscript is the choice of relevant prompt tag. The resulting set of underlying effects is joined together with the body's underlying effect.

The prophecy and control effect sets of the overall prompt should propagate prophecies and control effects targeting other prompts, and remove those targeting the prompt at hand. To this end, the conclusion of T-Prompt unblocks both sets and then removes (1) prophecies related to this prompt (which have now been validated) and (2) control effects related to this prompt (which have been incorporated into the prompt's underlying effect, because they address control behaviors scoped to this prompt). The (unblocked) control effects are simply filtered with $- \setminus \ell$ (Figure 5), which retains only basic control effects targeting other tags and control effects still blocked until other tags. (Thus, the nested control effect from Use Case 6 appears blocked in the effect of the innermost prompt.) The prophecy set is filtered similarly, but the filtered results are also transformed – the remaining observations must be adapted to model the changes to effects from going "through" this prompt. Thus the filtering operation $- \setminus^{Q} \ell$ on prophecy sets (Figure 5) selects those prophecies related to other prompts, then recursively transforms their observations – recursively filtering (and transforming) the observed prophecy and control effect sets, and joining transformations of their content into the observations' underlying effect, just as in the conclusion of T-Prompt itself. This is important due to interactions between tags: a prophecy to an outer prompt may observe in its context aborts to an inner prompt: $- \setminus^{Q} \ell$ joins such abort prefixes with the handler that would run, and joins that into the underlying effect of the prophecy.

5 Iterating Continuation Effects

Prior work on effect quantales [30, 31] introduced the notion of lax iterability to introduce a loop iteration operator, as outlined in Section 2. We would like to reuse this operator construction for two reasons. First, we would like to check that if we macro-express loop constructs and derive rules for them as we proposed earlier, that they are consistent with manually-derived rules from prior work, which use the iteration operator. Second, the iteration operator has properties that make it useful for solving recursive constraints over effects, such as those that arise in building derived rules for control flow constructs and control operators later in the paper. Of course, lax iterability and the construction are defined on standard effect quantales,[3] so do not apply directly to $\mathcal{C}(Q)$. Fortunately, one can apply the construction to $\mathcal{C}(Q)$ quotiented by $\approx$ (a standard effect quantale), and since those elements are equivalence classes, simply use the behavior on elements to iterate in $\mathcal{C}(Q)$:

▶ **Theorem 3** (Lax Iterability with Continuations). *For a laxly iterable underlying effect quantale Q, the effect quantale $\mathcal{C}(Q)/\approx$ is also laxly iterable, with the closure operator given by lifting the following operator from elements of $\mathcal{C}(Q)$ to the corresponding equivalence class.*

$$(P, C, \underline{Q})^* = (\bigcup_{i\in\mathbb{N}} P \blacktriangleright (P, C, \underline{Q})^i, \underline{Q}^* \rhd C, \underline{Q}^*)$$

Notice that when $P = \emptyset$ and $C = \emptyset$, this specializes to the intuitive embedding of the ($\bot$-extended) underlying iteration: $(\emptyset, \emptyset, \underline{Q}^*)$. When only $P = \emptyset$ this specializes to $(\emptyset, \underline{Q}^* \rhd C, \underline{Q}^*)$, intuitively reflecting that any control exits occur after repeating $\underline{Q}$ some number of times first. While these examples "merely" drop certain components of Theorem 3, it helps to work from the simplest case up to the more complex versions, since the examples above correspond intuitively to various execution paths. The infinite union in the prophecy set is the most subtle part of the operation to explain. Consider an expression with the structure while c (... (call/cc t...) ...): Assume the tag t for the continuation captured inside the loop does not occur elsewhere inside the loop – in particular, that the captured continuation would extend *outside* the loop. Considering the runtime execution, in some sense the prophecy captured by the *first* loop iteration must predict not only the regular execution and exceptional executions of future iterations, but even the need for more prophecies to be generated by the `call/cc`s in future iterations as well! This is why the set of prophecies must still be sequenced with some form of themselves, rather than just some subset. During static typechecking, we must therefore conservatively overapproximate the number of iterations following a prophecy. It may be 0, 1, 2, ... or any number. So the approximation must consider all of those possibilities, hence the infinite union of finite repetitions following the prophecies. This requires prophecy sets to be possibly-infinite, but only countably so.

6 Type Safety

We have proven syntactic type soundness for the type system presented in Section 4. We summarize the proof here; see the technical report for full details [32]. We continue to reuse Gordon's parameterization for soundness [30, 31], making the proof generic over a choice of abstract states (ranged over by σ) and related parameters subject to some restrictions.

[3] Lax iterability is defined for a version of effect quantales using partial operators [31] instead of the distinguished $\top$ used here to simulate partiality. But the definitions simplify to those used here when given total operations and using side-conditions to reject effects containing $\top$.

Progress is uninteresting (if primitives satisfy progress), in the sense that effects play no essential role (they are merely "pushed around" and the proof looks otherwise like standard progress proofs). Preservation is similar to the common formulation for syntactic type soundness results of sequential effect systems [34, 30, 68, 67]. It follows from single-step preservation: informally, for a well-typed runtime state σ and term e, if $\sigma; e \xrightarrow{q} \sigma'; e'$, then σ' and e' are also well-typed, and moreover if the static effects of e and e' are χ and χ' respectively, then $(\emptyset, \emptyset, q) \rhd \chi' \sqsubseteq \chi$ – that is, sequencing the actual effect of the reduction with the residual effect of the reduced expression is soundly bounded by the effect of the original expression. This is the typical form of syntactic type safety proofs for sequential effect systems [34, 25].

Explaining the formal statement requires explaining the full details of parameterization. For brevity and lack of space, we offer the formal statement of single-step preservation specialized for our running example of $\mathcal{T}(\Sigma)$ and $\mathsf{event}[-]$ with trivial (i.e., unit) state[4] for which many of the conditions simplify to True:

▶ **Corollary 4** (Single-Step Preservation for $\mathcal{T}(\Sigma)$)**.** *If $\Gamma \vdash e : \tau \mid \chi$ and $(); e \overset{q}{\Rightarrow} (); e'$, then there exists a $\tau' <: \tau$ and a χ' such that $\Gamma \vdash e' : \tau' \mid \chi'$ and $q \rhd \chi' \sqsubseteq \chi$.*

A key lemma for soundness of the rule for `call/cc` precisely relates prophecy observations to typing continuations. Informally, we prove that for a term $E[e]$, if the effect of e contains a prophecy $\mathsf{prophecy}\ \ell\ \chi \rightsquigarrow \tau; \mathsf{obs}\ (\emptyset, \emptyset, I)$ and E contains no prompts for tag ℓ, then (1) the effect of $E[e]$ contains a prophecy $\mathsf{prophecy}\ \ell\ \chi \rightsquigarrow \tau\ \mathsf{obs}\ \chi'$ (accumulating some observation), and (2) plugging an appropriately-typed value v into E produces a term $E[v]$ with static effect χ' (exactly matching the observation). The assumptions of the lemma are exactly the conditions when considering E-CallCC in the preservation proof: $E[e]$ is the body of the delimiting prompt and e is a use of `call/cc`, so by T-CallCont an appropriate prophecy exists in e's effect ensuring the prophecy is gives sound latent effect for typing the captured continuation.

▶ Remark 5 (Syntactic vs. Semantic Soundness). Our proof imparts no semantic meaning to effects beyond syntactically relating the dynamic and static effects – it does not check that a certain effect enforces what it is meant to (e.g., deadlock freedom), unless like some finite trace effects [68, 45] the relation between static and dynamic effects is already the intent. This is common to any syntactic type safety approach for generic effects [51]. Gordon [31] has extended his proof approach for semantic pre- and post-condition type properties, such as ensuring locking effects accurately describe lock acquisition and release, but limited to safety properties; in principle this should be adaptable to our setting. Denotational approaches to abstract effect systems [42, 55, 70, 6] inherently give actual semantics, and therefore can ensure liveness properties.

7 Deriving Sequential Effect Rules

Section 4 developed the core type rules which give sequential effects to programs making direct use of tagged delimited control. As we have discussed, most programs do not use the full power of delimited control, and instead use only control flow constructs or weaker control operators. This section uses the new type type-and-effect rules to *derive* sequential effect rules for a range of control flow constructs and weaker control operators macro-expressed in terms of prompts.

[4] Meaningful loops obviously require more meaningful state.

Our examples fall into two groups. First, we consider checking consistency of derived rules for typical control flow constructs with those hand-designed in prior work – for infinite loops and while loops. Second, we consider derived rules for constructs *never before addressed in sequential effect systems*: exceptions and generators.

In each case, we give a derived type rule for the construct of interest. While we are most explicit in Section 7.1, in each case our process for deriving the rule is the following:

1. Assume closed typing derivations for subexpressions (e.g., loop bodies)
2. Apply the type rules from Section 4 to give a closed-form rule for the macro's expansion to be well-typed under the assumed subexpression types. Typically these have several undetermined choices for metavariables representing effects, with non-trivial constraints to close the typing derivation.
3. Simplify the type rule by giving solutions to the constrained-but-undetermined metavariables in terms of the subexpressions' effects. This gives type rules that are simpler, and possibly less general, but given entirely in terms of the subexpressions' effects. The simplifications are typically involve rewriting by the laws satisfied by $\mathcal{C}(Q)$, and using the iteration operator from prior work [30, 31] to solve recursive constraints on undetermined effects.

Due to space constraints, we show detailed derivations only for simple infinite loops (Section 7.1), giving only final results for others. The calculations for the remaining examples are done in the same way, and are available in the accompanying technical report [32].

7.1 Infinite Loops

Consider a simple definition of an infinite loop using the constructs we have derived here:

$$[\![\mathsf{loop}\ e]\!] = (\%\ \ell\ (\mathsf{let}\ cc = (\mathsf{call/cc}\ \ell\ (\lambda k.\, k))\ \mathsf{in}\ ([\![e]\!]; cc\ cc))\ (\lambda\_.\, \mathsf{tt}))$$

The term above executes e repeatedly, forever (assuming e does not abort). Thus, its effect *ought* to indicate that e's effect, which we take to be $(\emptyset, \emptyset, Q_e)$,[5] is repeated arbitrarily many times. We take this expansion as the body of a macro $[\![\mathsf{loop}\ e]\!]$. This program can be well-typed in our system, with an appropriate effect (assuming the underlying effect quantale is *laxly iterable* per Section 2). The body of the call/cc is pure, but for the expression to be well-typed, the call/cc's own effect must prophecize some effect $(\emptyset, C_p, \underline{Q_p})$ of the enclosing continuation up to the prompt for ℓ (because no `call/cc` occurs in the continuation of another, the prophecy set can be empty).

The right-accumulator of the prophecy effect, initially $(\emptyset, \emptyset, I)$, eventually accumulates a control set $(Q_e \rhd \boxed{C_p}_\ell) \cup ((Q_e \rhd \underline{Q_p}) \rhd \{\mathsf{replace}\ \ell : I \rightsquigarrow \mathsf{unit}\})$ and underlying effect $\bot$, because between capturing the continuation and the prompt, the program evaluates e (underlying effect Q_e) and then invokes the captured continuation (prophecized effect $(\emptyset, C_p, \underline{Q_p})$, underlying effect $\bot$). This is also the resulting control effect set for the body; we will refer to it as C. The type rule for the prompt itself removes all ℓ-related prophecies and control effects, leaving both empty (since we assume no control effects escape e, C_p should only contain ℓ-related effects, while the prophecy set contains the single prophecy from the `call/cc`). For the underlying effect, T-PROMPT joins the immediate underlying effect Q_e (from the overall judgment, not the prophecy) with all ℓ-related behaviors in $C - e$ has no escaping control effects, and the macro-expanded loop contains no aborts, so C_p ought to have only replace effects, meaning C contains only replace effects, and $Q_e \sqcup \bigsqcup \ulcorner C \lrcorner_\ell |^I_\ell$ will join

[5] Note the non-$\bot$ underlying effect; well-typed expressions do not have degenerate effects.

the underlying effect of all continuations invoked by the body. T-PROMPT also performs some checking of result types (which all hold trivially since all types involved are unit), and prophecy validity checks that yield constraints we can solve to derive a closed-form type rule for the loop.

Completing a typing derivation with final underlying effect $Q_\ell = Q_e \sqcup \bigsqcup \ulcorner C \lrcorner_\ell |^I_\ell$ is possible given the solutions to the effect-related constraints imposed by $\mathsf{validEffects}$: $\bot \sqsubseteq \underline{Q_p}$, and $(Q_e \rhd \boxed{C_p}_\ell) \cup ((Q_e \rhd \underline{Q_p}) \rhd \{\mathsf{replace}\ \ell : I \rightsquigarrow \mathsf{unit}\}) \sqsubseteq C_p$. These could be read off a hypothetical derivation (see appendices in the technical report [32]) yielding the derived rule

$$\frac{\Gamma \vdash e : \tau \mid (\emptyset, \emptyset, Q_e) \quad \bot \sqsubseteq \underline{Q_p} \quad (Q_e \rhd \boxed{C_p}_\ell) \cup ((Q_e \rhd \underline{Q_p}) \rhd \{\mathsf{replace}\ \ell : I \rightsquigarrow \mathsf{unit}\}) \sqsubseteq C_p}{\Gamma \vdash [\![\mathsf{loop}\ e]\!] : \mathsf{unit} \mid (\emptyset, \emptyset, Q_e \sqcup \bigsqcup \ulcorner C \lrcorner_\ell |^I_\ell)}$$

However, this rule is more complex than we would like for a simple infinite loop (note we have not expanded $C = (Q_e \rhd \boxed{C_p}_\ell) \cup ((Q_e \rhd \underline{Q_p}) \rhd \{\mathsf{replace}\ \ell : I \rightsquigarrow \mathsf{unit}\})$), and also exposes details of the continuation-aware effects – which is undesirable if the goal is to derive closed rules for using the loop by itself, without developer access to full continuations, and there is an additional requirement that the prophecy used in the derivation is non-trivial (from T-PROMPT). These constraints can be satisfied by $\underline{Q_p} = Q_e^*$ (thus not $\bot$, ensuring a non-trivial prophecy), with $C_p = \{\mathsf{replace}\ \ell : Q_e^* \rightsquigarrow \mathsf{unit}\}$ (so $\boxed{C_p}_\ell = C_p$). The choice for C_p ensures than any "unrolling" of the loop to include any number of Q_e prefixes (as generated by the left operand of the union in the last constraint) is in fact less than the replacement effect ($Q_e \rhd Q_e^* \sqsubseteq Q_e^*$). This then implies that $Q_e \sqcup \bigsqcup \ulcorner C \lrcorner_\ell |^I_\ell \sqsubseteq Q_e \sqcup (Q_e^*) \sqsubseteq Q_e^*$, by properties of Gordon's iteration operator [30, 31] (Section 2). Assuming $cc \notin \Gamma$ (or hygienic macros) and applying subsumption, this leads us to the pleasingly simple derived rule:

$$\text{D-INFLOOP}\ \frac{\Gamma \vdash e : \tau \mid (\emptyset, \emptyset, Q_e)}{\Gamma \vdash [\![\mathsf{loop}\ e]\!] : \mathsf{unit} \mid (\emptyset, \emptyset, Q_e^*)}$$

7.2 Other Derived Rules

We have similarly macro-expressed traditional while loops, exceptions (`try-catch` and `throw`), and generators [14] (a form of coroutine now common in C#, Python, and JavaScript), and derived rules for each. Figure 6 gives derived rules for these constructs under a variety of conditions, each derived following a similar process to D-INFLOOP.

Loops do not *require* the use of control operators to express, of course. However, they are an important consistency check. Notably, D-WHILE recovers, via the approach above, exactly the rule for while loops that was hand-crafted in prior work [25, 26] and only recently given general treatment [30, 31]. D-ABORTINGWHILE is a limited generalization of D-WHILE, for the case where the condition or body by use `abort` (e.g., throw exceptions). The underlying effect – for when the loop completes normally – is as in D-WHILE. The control effect reflects the various times an abort may be thrown, based on the assumptions about aborts in c and e: during the initial execution of the condition, the initial execution of the body, or by *subsequent* executions of the condition or body. The derived rules for `try-catch` and `throw` reflect their simple expression in terms of prompts and `abort`.

The generator rule is more complex, requiring a bit of careful thought about the well-formedness condition $\mathsf{GenProphs}$, but still following the general approach taken for D-LOOP. Recall that generators are a construct for producing some sequence of values asynchronously: a generator is an object that may be queried repeatedly, and each query either produces a new value or indicates there are no more values. This is similar to an iterator, except generators are written in direct style. A language or library typically exposes a `yield` construct to

$$\textsc{D-While}\quad \frac{\Gamma \vdash e : (\emptyset, \emptyset, Q_e) \qquad \Gamma \vdash c : (\emptyset, \emptyset, Q_c)}{\Gamma \vdash [\![\mathsf{while}\ c\ e]\!] : \mathsf{unit} \mid (\emptyset, \emptyset, Q_c \rhd (Q_e \rhd Q_c)^*)} \qquad \textsc{D-FullInfLoop}\quad \frac{\Gamma \vdash e : \tau_e \mid \chi_e}{\Gamma \vdash [\![\mathsf{loop}\ e]\!] : \mathsf{unit} \mid \chi_e^*}$$

$$\textsc{D-AbortingWhile}\quad \frac{l \notin C_c \qquad l \notin C_e \qquad \mathsf{AbortsOnly}(C_c \cup C_e) \qquad \Gamma \vdash c : (\emptyset, C_c, Q_c) \qquad \Gamma \vdash e : (\emptyset, C_e, Q_e)}{\Gamma \vdash [\![\mathsf{while}_\ell\ c\ e]\!] : \mathsf{unit} \mid (\emptyset, \bigcup \left\{ \begin{array}{c} C_c \cup (Q_c \rhd Q_e \rhd C_c) \cup (Q_c \rhd (Q_e \rhd Q_c)^* \rhd C_e), \\ (Q_c \rhd (Q_e \rhd Q_c)^* \rhd Q_e \rhd C_c) \end{array} \right\}, Q_c \rhd (Q_e \rhd Q_c)^*)}$$

$$\textsc{D-TryCatch}\quad \frac{\begin{array}{c}\Gamma \vdash e : \tau \mid (\emptyset, \{\mathsf{abort}\ \ell_C\ Q \rightsquigarrow C\}, Q_e) \\ \Gamma \vdash h : C \xrightarrow{(\emptyset,\emptyset,Q_h)} \tau \mid (\emptyset, \emptyset, I)\end{array}}{\Gamma \vdash [\![\mathsf{try}\ e\ \mathsf{catch}\ C \Rightarrow h]\!] : \tau \mid (\emptyset, \emptyset, Q_e \sqcup (Q \rhd Q_h))} \qquad \textsc{D-Throw}\quad \frac{\Gamma \vdash e : C \mid (\emptyset, \emptyset, Q_e)}{\Gamma \vdash [\![\mathsf{throw}_C\ e]\!] : \tau \mid (\emptyset, \{\mathsf{abort}\ \ell_C\ Q_e \rightsquigarrow C\}, I)}$$

$$\mathsf{GenProphs}(P, \ell, E) = \begin{array}{l} \forall \ell', P', C', Q', P'', C'', Q'', \tau. \\ \mathsf{prophecy}\ \ell'\ (P', C', Q') \rightsquigarrow \tau\ \mathsf{obs}\ (P'', C'', Q'') \in P \Rightarrow \\ \ell' = \ell \wedge \tau = \mathsf{bool} \wedge \mathsf{GenProphs}(P', \ell, E) \wedge C' \sqsubseteq \{\mathsf{abort}\ \ell\ E^* \rightsquigarrow (\mathsf{option}\ \tau)\} \\ \wedge Q' \sqsubseteq E^* \wedge \ulcorner P'' \lrcorner_\ell \sqsubseteq \ulcorner P' \lrcorner_\ell \wedge C'' \sqsubseteq C' \wedge Q'' \sqsubseteq Q' \end{array}$$

$$\textsc{D-Iterate}\quad \frac{\begin{array}{c}\Gamma \vdash f : \left(\begin{array}{c} (\tau \xrightarrow{(\{\mathsf{prophecy}\ gen\ (P_p, C_p, E^*) \rightsquigarrow (\mathsf{option}\ \tau)\ \mathsf{obs}\ (\emptyset,\emptyset,I)\}, \{\mathsf{abort}\ gen\ I \rightsquigarrow (\mathsf{option}\ \tau)\}, \bot)} \mathsf{unit}) \xrightarrow{I} \\ (\mathsf{unit} \xrightarrow{(\emptyset, \{\mathsf{abort}\ gen\ I \rightsquigarrow (\mathsf{option}\ \tau)\}, \bot)} \mathsf{unit}) \xrightarrow{(P,C,E^*)} \mathsf{unit} \end{array} \right) \mid (\emptyset, \emptyset, I) \\ C \sqsubseteq \{\mathsf{abort}\ gen\ (E^*) \rightsquigarrow (\mathsf{option}\ \tau)\} \qquad \mathsf{GenProphs}(P, gen, E)\end{array}}{\Gamma \vdash [\![\mathsf{iterate}\ init\ gen\ f]\!] : \mathsf{unit} \xrightarrow{(\emptyset,\emptyset,E^*)} \mathsf{option}\ \tau \mid (\emptyset, \emptyset, I)}$$

Figure 6 A collection of derived rules.

produce a value for the client. (This now describes Python, C#, F#, and JavaScript, among other languages.) After the client consumes each value and queries the generator again, control resumes immediately after the last-executed `yield`, continuing until another value is `yield`ed or generation is complete. Consider a simple client of Coyle and Crogono's implementation using Scheme macros and `call/cc` [14] (see our technical report for Racket code [32]):

```
(define next (iterate (λ (yield done) (yield "a") (yield "b") (done))))
```

the iterator body takes two arguments, `yield` and `done` for yielding a value, and indicating generation is complete respectively. `next` is the result of building a generator from this function; `iterate` (the generator implementation) supplies functions for `yield` and `done` (described below).

Our macro iterate *init gen f* (*init* and *gen* are tags) expands to a heap-storage-based generator encoding similar to Coyle and Crogono's [14] (and also given explicitly with type annotations in our technical report [32]). A prompt is used to delimit the scope of the generator body. Yielding a value captures the current continuation up to that prompt, stores it in a heap cell, then uses `abort` to throw the yielded value. The rule D-Iterate assumes f is a function of two arguments as above: the first is a function playing the role of `yield`, the second a function that marks the iterator as complete (`done`). The macro returns a function which, when invoked, returns an option containing either the next element produced by the generator, or a failure. When invoked, this function introduces a new prompt for tag *gen*, and invokes the stored continuation to produce the next element (via the `yield` parameter) or a completion. The rule assumes the activity "between" successive yields can be over-approximated by an underlying effect E, and stores the continuations with underlying effect E^*. Because uses of the first parameter, `yield`, capture continuations, a prophecy choice must be made (P_p) for that particular generator, for the yield to predict the appropriate remainder of the generator body. D-Iterate is specialized here to the assumption that all prophecies and aborts in the generator body are related to the tag *gen* only, which allows us to lift the requirements of prophecy validation into the GenProphs predicate. A complete rule permitting at least exceptions to escape a generator is possible, but would be very complex (so much so that C# strongly restricts their interactions [53]).

8 Related Work

Recent years have seen great progress on semantic models for sequential effect systems [70, 42, 55], centering on what are now known as *graded monads*: monads indexed by some kind of monoid (to model sequential composition), commonly a partially-ordered monoid following Katsumata [42]. Gordon [30] focused on capturing common structure for prior concrete effect systems, leading to the first abstract characterization of sequential effect systems with singleton effects, effect polymorphism, and iteration of sequential effects.

To the best of our knowledge we are the first to use the term "accumulator" as we do to identify this as a reusable technique. However accumulators have appeared before. Koskinen and Terauchi's effect system [45] uses left-accumulators for safety and liveness properties (requiring an oracle for liveness). Effects in their system are a pair of sets, one a set of finite traces (for terminating executions) and the other a set of infinite traces (for non-terminating executions). The infinite traces left-accumulate: code that comes after a non-terminating expression in program-order never runs. On the other hand, *finite* executions from code *before* an infinite execution extend the prefix of the infinite executions. Earlier, Neamtiu et al. [56] defined *contextual* effects to track what (otherwise order-unaware) effects occurred before or after an expression, to ensure key correctness properties for code using dynamic software updates.

Effect systems treating continuations are nearly as old as effect systems themselves [41]. To the best of our knowledge, we are the first to integrate *sequential* effects with exceptions, generators, or general delimited continuations – or any control flow construct beyond while loops, including any form of continuation, tagged or otherwise. As mentioned in Section 2, the original motivation for tags was to prevent encodings of separate control operators from interfering with each other [64], which is critical for our goals, strictly more expressive than untagged continuations, and motivates important elements of the theory (blocking). The only other work we know of focusing on effects with tagged delimited continuations is Pretnar and Bauer's variant [62] of algebraic effects and handlers [62] where operations may be handled by outer handle constructs (not just the closest construct as in other algebraic effects work). Their commutative effects ensures all algebraic operations are handled by some enclosing handler.

Tov and Pucella [73] examined the interaction of *untagged* delimited continuations with substructural types (a coeffect [58]). Delbianco and Nanevski adapted Hoare Type Theory for untagged *algebraic* continuations [17], which include prompts and handlers, but place handlers at the site of an abort rather than at the prompt in order to satisfy some useful computational equalities (see below). As a consequence, encoding non-trivial control flow constructs in their system becomes significantly more complex; for example, simulating the standard semantics of throwing exceptions to the nearest enclosing catch block for the exception type would require catching, dispatching, and re-throwing at every prompt. This and lack of tagging would make compositional study of multiple control flow constructs / control operators difficult, and as our work shows the treatment of multiple tags is not a trivial extension of untagged semantics. Atkey [6] considered denotational semantics for (untagged) *composable* continuations in his parameterized monad framework for (denotational) sequential effect systems, essentially giving a denotation of a type-and-effect system for answer type modification [5, 16] – a kind of sequential effect which can be used to allow continuations to temporarily change the result type of a continuation, as long as it is known (via the effects) that it will be changed back. Thus Atkey considered answer type modification effects as an instance of a sequential effect specific to using control operators, rather than having an application-domain-focused effect (like exceptions or traces) work with continuations or giving an account of general sequential effects for control operators. Readers familiar with answer type modification may wonder about directly supporting it in our generic core language. We have not yet considered this deeply, but note that (i) directly ascribing answer type modifications to control operations would require assigning *specific* answer type modification effects to the control operations, not just effects derived from primitives, but (ii) Kobori et al. [44] showed that tagged shift/reset can express answer type modification in a type system that does not track answer types explicitly, so such an extension may provide no additional power (treating convenience as another matter).

Algebraic effects with handlers [60] are a means to describe the *semantics* of effects in terms of a set of operations (the effectful operations) along with handlers that interpret those operations as actions on some resource. The combination yields an algebra characterizing equality of different effectful program expressions, hence the term "algebraic". Languages with algebraic effects include an effect system to reason about which effects a computation uses, to ensure they are handled. Some implementations even use Lindley and Cheney's effect adaptation [49] of row polymorphism [75] to support effect inference [47]. Handlers for algebraic effects receive both the action to interpret and the continuation of the program following the effectful action. Thus they can implement many control operators, including generators and cooperative multithreading [48], as with the delimited continuations we study. In an untyped setting without tagging, algebraic handlers can simulate (via macro translation) `shift0`/`reset0` [28], which can simulate prompts and handlers [63] (with correct space complexity, not only extensionally-correct behavior); with those limitations, handlers are as powerful as the constructs we study. For the common commutative effect system for handlers that ensures all operations are handled, Forster et al. [28] prove that the translation from handlers to prompts (`shift0`) is not type-and-effect preserving, and conjecture the reverse translation also fails to preserve types. They conjectured that adding polymorphism to each system would enable a type-and-effect preserving translation (again, without tagging, for a commutative effect system), which was recently confirmed by Piróg et al. [59] for a class of commutative effect systems.

The effect systems considered for algebraic effects thus far have only limited support for reasoning about sequential effects. The types given for individual algebraic effects do support reasoning about the existence of a certain type of resource before and after the computation [12, 7]. However, the way this is done corresponds to a parameterized monad [6], which Tate showed [70] crisply do not include all meaningful sequential effect systems. His examples that cannot be modeled as parameterized monads include examples that *can* be modeled as effect quantales [31], such as the non-reentrant locking effect system Tate uses to motivate aspects of his approach.

General considerations of sequential effect systems have not yet been explored for algebraic effects. When it is considered, it seems likely ideas from our development (particularly prophecies) will be useful. For example, Dolan et al. [18] offer two reasons for dynamically enforcing linearity of continuations in their handlers: performance, but also avoiding the sorts of errors prevented by sequential effect systems, such as closing a file twice by reusing a continuation.

It also seems plausible that our approach could be adapted to algebraic effects and handlers. With an effectively-tagged version of handlers [62], a similar macro-expression of control flow constructs and control operators is likely feasible, in particular adapting our notion of prophecy and observation to handlers: in this case, the continuations themselves are seen by handlers that are direct subexpressions of the handling construct itself, so prophecies might observe “outside-in” rather than our system’s “inside-out” accumulation.

The approach we take to deriving type rules for control flow constructs and control operators is reminiscent of work done in parallel with ours by Pombrio and Krishnamurthi [61]. They address the problem of producing useful type rules when a language semantics and type rules are defined directly for a simpler core language, and a full source language is defined using syntactic sugar (i.e., macros) that expand into core language expressions with the intended semantics, such as the approach taken by λ_{JS} [37]. There the issue is that type errors given in terms of the elaborated core terms are difficult to understand for developers writing in the unelaborated source language. Pombrio and Krishnamurthi offer an approach to automatically lift core language type rules through the desugaring process to the source language, providing sensible source-level type errors. Their work focuses on type systems without effects, but including such notions as subtyping and existential types. They do not consider control operators (delimited continuations) or effects (neither commutative nor sequential). Extending their approach to support the language features and types (effects) we consider would make our approach more useful to effect system designers, though this is non-trivial due to the many ways to combine sequential effects.

9 Conclusions

We have given the first general approach to integrating arbitrary sequential effect systems with tagged delimited control operators, which allows lifting existing sequential effect systems without knowledge of control operators to automatically support tagged delimited control. We have used this characterization to derive sequential effect system rules for standard control flow structures macro-expressed via continuations, including deriving known forms (loops) and giving the first characterization of exceptions and generators in sequential effect systems.

References

1 Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2, //month = mar), 2006.
2 Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *LICS*, 1988.
3 Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
4 Torben Amtoft, Flemming Nielson, and Hanne Riis Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, London, UK, 1999.
5 Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In *APLAS*, pages 239–254, 2007.
6 Robert Atkey. Parameterised Notions of Computation. *Journal of Functional Programming*, 19:335–376, July 2009.
7 Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. In *International Conference on Algebra and Coalgebra in Computer Science*, pages 1–16. Springer, 2013.
8 Garrett Birkhoff. *Lattice theory*, volume 25 of *Colloquium Publications*. American Mathematical Soc., 1940. Third edition, eighth printing with corrections, 1995.

9 Thomas Scott Blyth. *Lattices and ordered algebraic structures*. Springer Science & Business Media, 2006.

10 Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002.

11 Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2001.

12 Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 133–144. ACM, 2013. doi:10.1145/2500365.2500581.

13 John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *European symposium on programming*, pages 320–334. Springer, 2001.

14 Christopher Coyle and Peter Crogono. Building abstract iterators using continuations. *SIGPLAN Not.*, 26(2):17–24, January 1991. doi:10.1145/122179.122181.

15 Olivier Danvy. An analytical approach to program as data objects, 2006. DSc thesis, Department of Computer Science, Aarhus University.

16 Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical report, DIKU — Computer Science Department, University of Copenhagen, July 1989.

17 Germán Andrés Delbianco and Aleksandar Nanevski. Hoare-style reasoning with (algebraic) continuations. In *ICFP*, 2013.

18 Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. Concurrent system programming with effect handlers. In Meng Wang and Scott Owens, editors, *Trends in Functional Programming*, pages 98–117, Cham, 2018. Springer International Publishing.

19 Matthias Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 180–190, 1988. doi:10.1145/73560.73576.

20 Matthias Felleisen. On the expressive power of programming languages. *Science of computer programming*, 17(1-3):35–75, 1991.

21 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.

22 Matthias Felleisen and Daniel P. Friedman. A reduction semantics for imperative higher-order languages. In *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 15-19, 1987, Proceedings*, pages 206–223, 1987. doi:10.1007/3-540-17945-3_12.

23 Cormac Flanagan and Martín Abadi. Object Types against Races. In *CONCUR*, 1999.

24 Cormac Flanagan and Martín Abadi. Types for Safe Locking. In *ESOP*, 1999.

25 Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.

26 Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *TLDI*, 2003.

27 Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In *ICFP*, 2007.

28 Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.*, 1(ICFP):13:1–13:29, August 2017. doi:10.1145/3110257.

29 Laszlo Fuchs. *Partially ordered algebraic systems*, volume 28 of *International Series of Monographs on Pure and Applied Mathematics*. Dover Publications, 2011. Reprint of 1963 Pergamon Press version.

30 Colin S. Gordon. A Generic Approach to Flow-Sensitive Polymorphic Effects. In *ECOOP*, 2017.

31 Colin S. Gordon. Polymorphic Iterable Sequential Effect Systems. Technical Report arXiv cs.PL/cs.LO 1808.02010, Computing Research Repository (Corr), August 2018. In Submission.. arXiv:1808.02010.

32 Colin S. Gordon. Sequential Effect Systems with Control Operators. Technical Report arXiv cs.PL 1811.12285, Computing Research Repository (CoRR), December 2018. `arXiv:1811.12285`.

33 Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. JavaUI: Effects for Controlling UI Object Access. In *ECOOP*, 2013.

34 Colin S. Gordon, Michael D. Ernst, and Dan Grossman. Static Lock Capabilities for Deadlock Freedom. In *TLDI*, 2012.

35 James Gosling, Bill Joy, Guy L Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification: Java SE 8 Edition*. Pearson Education, 2014.

36 OpenJDK HotSpot Group. OpenJDK Project Loom: Fibers and Continuations, 2019. URL: `https://wiki.openjdk.java.net/display/loom/Main`.

37 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *ECOOP*, 2010.

38 Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Trans. Program. Lang. Syst.*, 9(4):582–598, October 1987. `doi:10.1145/29873.30392`.

39 Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *LISP and Functional Programming*, pages 293–298, 1984.

40 Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining coroutines with continuations. *Comput. Lang.*, 11(3/4):143–153, 1986. `doi:10.1016/0096-0551(86)90007-X`.

41 P. Jouvelot and D. K. Gifford. Reasoning about continuations with control effects. In *PLDI*, 1989.

42 Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *POPL*, 2014.

43 Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: On the effectiveness of lightweight mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 285–296, New York, NY, USA, 2012. ACM. `doi:10.1145/2103656.2103691`.

44 Ikuo Kobori, Yukiyoshi Kameyama, and Oleg Kiselyov. Answer-type modification without tears: Prompt-passing style translation for typed delimited-control operators. In Olivier Danvy and Ugo de'Liguoro, editors, *Proceedings of the Workshop on Continuations, WoC 2016, London, UK, April 12th 2015*, volume 212 of *EPTCS*, pages 36–52, 2015. `doi:10.4204/EPTCS.212.3`.

45 Eric Koskinen and Tachio Terauchi. Local temporal reasoning. In *CSL/LICS*, 2014.

46 Shriram Krishnamurthi, Peter Walton Hopkins, Jay A. McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT scheme web server. *Higher-Order and Symbolic Computation*, 20(4):431–460, 2007. `doi:10.1007/s10990-007-9008-y`.

47 Daan Leijen. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming (MSFP)*, 2014.

48 Daan Leijen. Structured asynchrony with algebraic effects. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2017, pages 16–29, New York, NY, USA, 2017. ACM. `doi:10.1145/3122975.3122977`.

49 Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 91–102. ACM, 2012.

50 J. M. Lucassen and D. K. Gifford. Polymorphic Effect Systems. In *POPL*, 1988.

51 Daniel Marino and Todd Millstein. A Generic Type-and-Effect System. In *TLDI*, 2009. `doi:10.1145/1481861.1481868`.

52 Microsoft. C# Language Specification: Enumerable Objects, 2018. URL: `https://github.com/dotnet/csharplang/blob/master/spec/classes.md#enumerable-objects`.

53 Microsoft. C# Reference: yield Exception Handling, 2018. URL: `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/yield#exception-handling`.

54 Mozilla. Mozilla Developer Network Documentation: function*, 2018. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*`.

55 Alan Mycroft, Dominic Orchard, and Tomas Petricek. Effect systems revisited — control-flow algebra and semantics. In *Semantics, Logics, and Calculi*. Springer, 2016.

56 Iulian Neamtiu, Michael Hicks, Jeffrey S Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *POPL*, pages 37–49, 2008.

57 Flemming Nielson and Hanne Riis Nielson. From cml to process algebras. In *CONCUR*, 1993.

58 Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: A calculus of context-dependent computation. In *ICFP*, 2014.

59 Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Typed equivalence of effect handlers and delimited control. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

60 Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.

61 Justin Pombrio and Shriram Krishnamurthi. Inferring type rules for syntactic sugar. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 812–825, New York, NY, USA, 2018. ACM. `doi:10.1145/3192366.3192398`.

62 Matija Pretnar and Andrej Bauer. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10, 2014.

63 Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007.

64 Dorai Sitaram. Handling control. In *PLDI*, 1993.

65 Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.

66 Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: full abstraction for models of control. In *LISP and Functional Programming*, pages 161–175, 1990. `doi:10.1145/91556.91626`.

67 Christian Skalka. Types and trace effects for object orientation. *Higher-Order and Symbolic Computation*, 21(3):239–282, 2008.

68 Christian Skalka, Scott Smith, and David Van Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, 18(2), 2008.

69 Kohei Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *APLAS*, 2008.

70 Ross Tate. The sequential semantics of producer effect systems. In *POPL*, 2013.

71 Python Development Team. Python Enhancement Proposal 255: Simple Generators, 2001. URL: `https://www.python.org/dev/peps/pep-0255/`.

72 Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and computation*, 132(2):109–176, 1997.

73 Jesse A. Tov and Riccardo Pucella. A theory of substructural types and control. In *OOPSLA*, 2011.

74 Philip Wadler and Peter Thiemann. The Marriage of Effects and Monads. *Transactions on Computational Logic (TOCL)*, 4:1–32, 2003.

75 Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 92–97. IEEE, 1989.