Handling the Selection Monad

GORDON PLOTKIN, Google DeepMind, United States
NINGNING XIE, Google DeepMind and University of Toronto, Canada

The selection monad on a set consists of selection functions. These select an element from the set, based on a loss (dually, reward) function giving the loss resulting from a choice of an element. Abadi and Plotkin used the monad to model a language with operations making choices of computations taking account of the loss that would arise from each choice. However, their choices were optimal, and they asked if they could instead be programmer provided.

In this work, we present a novel design enabling programmers to do so. We present a version of algebraic effect handlers enriched by computational ideas inspired by the selection monad. Specifically, as well as the usual delimited continuations, our new kind of handlers additionally have access to *choice continuations*, that give the possible future losses. In this way programmers can write operations implementing optimisation algorithms that are aware of the losses arising from their possible choices.

We give an operational semantics for a higher-order model language λC, and establish desirable properties including progress, type soundness, and termination for a subset with a mild hierarchical constraint on allowable operation types. We give this subset a selection monad denotational semantics, and prove soundness and adequacy results. We also present a Haskell implementation and give a variety of programming examples.

CCS Concepts: • **Software and its engineering** → **Control structures**; **Semantics**; • **Theory of computation** → **Denotational semantics**; **Operational semantics**.

Additional Key Words and Phrases: Effect handlers, Selection monad, Continuations, Machine Learning Programming

ACM Reference Format:
Gordon Plotkin and Ningning Xie. 2025. Handling the Selection Monad. *Proc. ACM Program. Lang.* 9, PLDI, Article 218 (June 2025), 25 pages. https://doi.org/10.1145/3729321

1 Introduction

The *selection monad* [Escardó and Oliva 2010a, 2011, 2015, 2010b,c,d] has been used to explain fundamental phenomena in various areas of logic, including game theory, proof theory, and computational interpretations; it has also been used in connection with CPS transformations and with algorithm design [Hartmann and Gibbons 2022; Hedges 2015]. The monad has the form $S(X) = (X \longrightarrow R) \longrightarrow X$, where R, typically an ordered set such as the real numbers, can be thought of as a set of *losses*. A computation, meaning an element of $S(X)$, is a *selection function* that, given a *loss function* from X to R, picks an element of X. For example, the well-known selection function argmin takes a loss function and returns an element that minimizes its value.[1] The selection monad can be combined with other *auxiliary* monads T to produce *augmented* selection monads $S_T(X) = (X \longrightarrow R) \longrightarrow T(X)$ [Abadi and Plotkin 2019; Escardó and Oliva 2015]. This generalization proves useful when combining the selection monad with additional effects.

[1]Dually, we can think of R as a set of *rewards*, and recall the argmax function that picks a maximising element; the two viewpoints are equivalent in case R has a negation function, as with the reals. Below we talk only of losses.

Authors' Contact Information: Gordon Plotkin, Google DeepMind, Mountain View, United States, plotkin@google.edu; Ningning Xie, Google DeepMind and University of Toronto, Toronto, Canada, ningningxie@cs.toronto.edu.

ACM 2475-1421/2025/6-ART218
https://doi.org/10.1145/3729321

The connection between programming languages with decision-making operations and the selection monad was investigated by Abadi and Plotkin [2021, 2023]. They considered a language for a selection monad augmented by the writer monad $W(X) = R \times X$. It had a binary *choice operation* which could choose between two computations based on the losses the two choices entailed. Losses were reported by a loss operation. (They also considered a probabilistic extension.)

A key question they left open was how to empower programmers with the ability to define their own choice operations, with the choice of computations (i.e., the selection strategy) again based on the losses the possible choices of computation entail. Specifically, Abadi and Plotkin used argmax to model binary choice so that selections were optimal (or optimal-in-expectation). Implementing optimal selection implies perfect knowledge of all available choices and the ability to consistently make the best decision. As they acknowledge, this is not at all a reasonable assumption for applications such as, for example, learning algorithms [Carbune et al. 2019; Goodfellow et al. 2016; Ruder 2017] or game-playing, since programmers generally do not have efficient, or perhaps any, access to optimal choices. Considerations of this sort motivated their question asking for a mechanism allowing programmers to employ their own selection strategies.

We give one such way, answering their key open question. To do so, we develop a suitable version of *algebraic effect handlers* [Pretnar and Plotkin 2013]. Algebraic effect handlers provide a flexible mechanism for modular programming with user-defined effects. They have been explored in various languages including OCaml [Sivaramakrishnan et al. 2021], C/C++ [Alvarez-Picallo et al. 2024; Ghica et al. 2022], and WebAssembly [Phipps-Costin et al. 2023]. Algebraic effect handlers achieve modularity by decoupling syntax from semantics: the syntax is defined by user-specified effect operations, while their semantics are determined by handlers. A handler's operation receives the operation's *argument* and a *delimited continuation* which captures the evaluation context, i.e., from where the operation is performed to where its *result* can be used. Handlers can manipulate the continuation in various ways. For instance, a handler may choose to not resume the continuation, effectively implementing an exception mechanism, or it could resume the the continuation multiple times enabling backtracking and non-deterministic behaviors. However, standard algebraic effect handlers cannot handle operations based on a program's loss information, unless this information is provided explicitly as an argument to the operation or returned as part of the final result of the continuation — but both options are restrictive. Specifically, when performing an operation, complete loss information may not be immediately available. Similarly, returning the loss as part of the final result of the continuation would require all continuations (and functions) to return loss-result pairs, which is impractical.

We propose a novel language design that empowers programmers to write choice operations that can choose computations based on the losses the possible choices of computation entail. In this way they can employ their own selection strategies. Our fundamental insight is to combine algebraic effect handlers with computational ideas inspired by the selection monad. We achieve this by providing effect handlers with *choice continuations* (as well as the usual delimited continuations). These are a kind of loss continuation that yields the loss arising from an operation's possible result. Handlers can then define choice operations which compute selections from these choice continuations and use the result for their choice of computation, e.g., as an argument to a delimited continuation. Notably, choice continuations are not delimited, but have scopes that can be controlled by a localising construct that determines how much loss information is accessible. Such scopes can vary from inside the handler to beyond. As in Abadi and Plotkin [2021], losses are prescribed using a *loss* effect.

We make several contributions:

- We connect up the selection monad with effect handlers via λC, a higher-order model language incorporating the new kind of handlers with their choice continuations. We present a novel loss-continuation-based operational semantics for the language.

- We give λC a selection-monad-based denotational semantics and establish soundness and adequacy theorems for both big-step and giant-step operational semantics (Theorems 5.4, 5.5, and 5.6). We thereby both show that our computational ideas are in accord with the monadic ones which inspired them and establish a theoretical foundation for our extended effect handlers.
- We provide a library implementation in Haskell, following the operational semantics, and present programming examples, demonstrating expressiveness.
- Our work presents a novel combination of delimited and choice continuations. Our techniques may extend to other combinations of different kinds of continuations.

The rest of the paper is structured as follows. In §2, we review the selection monad and algebraic effect handlers, and illustrate our new language design. In §3 we present λC, our higher-order model language, extending effect handlers with loss primitives and choice continuations. We give λC a deterministic, progressive, and type-safe small-step operational semantics (Theorem 3.2). We prove that termination holds for a subset of the language with a hierarchical constraint on handler interfaces (Theorem 3.5); this is needed for the results in §5. In §4 we give a Haskell library, and give programming examples illustrating potential applications of the new language design. We hope our examples offer fresh insights in the effect handlers application space. In §5, we connect up the selection monad with our computational contributions. We give the hierarchical part of λC a selection-monad-based denotational semantics and establish our soundness and adequacy theorems. For space reasons, some rules and all proofs are omitted. A full version of proofs can be found in Plotkin and Xie [2025]. Finally, we discuss related and future work in §6.

2 Overview

We first introduce the selection monad, and algebraic effect handlers. We then present our language design, specifically how computational ideas inspired by the selection monad are combined with algebraic effect handlers.

2.1 The Selection Monad

While the selection monad $S(X) = (X \to R) \to X$ is available in any cartesian closed category, we focus on the category of sets. We assume R is a commutative monoid $(R, +, 0)$ (for example, the reals, or a finite product of the reals). This will be needed for the loss operation. As we have said, a computation $F \in S(X)$ acts as a *selection function* taking a *loss function* $\gamma \in (X \to R)$ and picking an element $F(\gamma) \in X$. The loss associated to $F \in S(X)$, given a loss function $\gamma : X \to R$, is defined to be $\mathbf{R}(F|\gamma) =_{\text{def}} \gamma(F(\gamma))$. (We remark that the selection monad is closely connected to the more familiar continuation monad $C(X) = (X \to R) \to R$. For, given F in $S(X)$, $\lambda\gamma.\, \mathbf{R}(F|\gamma)$ is in $C(X)$.)

So, for example, taking R to be the real numbers (with their usual addition), for finite sets X, $\mathsf{argmin}_X : (X \to R) \to X$ is an example selection function. Given a loss function $\gamma : X \to R$, $\mathsf{argmin}_X(\gamma)$ is an element x of X minimising $\gamma(x)$ (we assume available some way to choose when there is more than one such element). Then $\mathbf{R}(\mathsf{argmin}_X|\gamma)$ is just the minimum value that γ obtains.

To fully specify the selection monad we give its Kleisli triple structure, viz the units $(\eta_S)_X : X \to S(X)$ and the Kleisli extensions $f^{\dagger_S} : S(X) \to S(Y)$ of maps $f : X \to S(Y)$. (As explained in [Benton et al. 2000] these correspond, modulo currying, to Haskell's monadic **return** and bind (>>=) operations.) The units are given by: $\eta_S(x) = \lambda\gamma.\, x$ (Here, and below, we omit the objects X, when writing units.) For the Kleisli extension, first associate a *loss continuation transformer* $\tilde{f} : (Y \to R) \to (X \to R)$ to f by

$$\tilde{f}(\gamma) = \lambda x \in X.\mathbf{R}(f(x)|\gamma)$$

where we use $f(x)$ as the Y selection function. (Connecting again to the continuation monad, note that, modulo currying, $\tilde{f}$ has type $X \to C(Y)$.) Then the Kleisli extension is given by:

$$f^{\dagger_S}(F) = \lambda \gamma \in Y \to R.\ f\ (F\ (\tilde{f}\ \gamma))\ \gamma$$

Here the loss function γ on Y is transformed into a loss function $\tilde{f}(\gamma)$ on X, which is then used by F to select an element $x = F(\tilde{f}(\gamma))$ of X. Finally, f uses x and the original loss function γ to select an element of Y. As always, the Kleisli structure determines the monad's functorial action by the formula $S(f) = (\eta_S \circ f)^{\dagger_S}$, which latter, in this case, is $\lambda \gamma \in Y \to R.\ f(\gamma \circ f)$.

Continuing the example, Kleisli extension allows us to solve one-move games with evaluation function $\mathsf{eval} : X \times Y \to R$. Suppose $f : X \to S(X \times Y)$ is defined by:

$$f(x)(\gamma) = (x, \mathsf{argmin}(\lambda y.\ \gamma(x, y))$$

Then $f^{\dagger_S}(\mathsf{argmax})(\mathsf{eval})$ is a minimax pair (x_0, y_0) for eval, with $x_0 \in X$ maximising all possible $\mathsf{eval}(x, y)$, and $y_0 \in Y$ minimising all possible $\mathsf{eval}(x_0, y)$.

Turning to augmented monads $S_T(X) = (X \to R) \to T(X)$, as an example, take T to be the writer monad $W(X) = R \times X$, with R the reals. An example (augmented) selection function is then the "loss-recording" version of argmin that sends γ to $(\gamma(\mathsf{argmin}(x)), \mathsf{argmin}(\gamma))$. The unit of S_W is $\eta_{S_W}(x) = \lambda \gamma.\ (0, x)$. The loss associated to $F \in S_W(X)$ and $\gamma : X \to R$ is the sum of the loss incurred by F and the loss incurred by the loss function: $\mathbf{R}_W(F|\gamma) = \pi_0(F(\gamma)) + \gamma(\pi_1(F(\gamma)))$. The Kleisli extension $f^{\dagger_{S_W}} : S_W(X) \to S_W(Y)$ for $f : X \to S_W(Y)$ is then defined as below, where the losses incurred by F and f are added up (and where $\tilde{f}$ is defined similarly to the above):

$$\begin{aligned} f^{\dagger_{S_W}}(F) = \lambda \gamma : Y \to R.\ & \mathit{let}\ \langle r_1, x \rangle = (F\ (\tilde{f}\ \gamma))\ \mathit{in} \\ & \mathit{let}\ \langle r_2, y \rangle = (f\ x\ \gamma)\ \mathit{in}\ \langle r_1 + r_2, y \rangle \end{aligned}$$

The functorial action in this case is: $S_W(f) = \lambda \gamma.\ W(f)(f \circ \gamma)$. In general (see, e.g., Abadi and Plotkin [2023]) augmented selection monads S_T are available when R forms a T-algebra $\alpha : T(R) \to R$. In the case of the writer monad $W(X)$ just considered, $\alpha : W(R) \to R = +$.

For the denotational semantics of our model language λC we use families $F_\epsilon(W(X))$ of auxiliary monads and loss sets $F_\epsilon(R)$, with R the reals, parameterized by multisets ϵ of certain effects ℓ, where $F_\epsilon(X)$ is a free algebra monad with signature specified by ϵ. The resulting augmented selection monads are used to give the semantics of λC programs with effect multisets ϵ.

2.2 Algebraic Effect Handlers

Algebraic effect handlers provide a structured approach to managing effects in computations. We give a brief introduction here; for a more in depth account see, e.g. Bauer and Pretnar [2015]; Pretnar [2015]. Consider a non-deterministic choice operation, *decide*, which takes a unit and returns a Bool as its result. A computation can invoke operations by providing its argument. For example, the following program performs *decide* twice, and returns the conjunction of the results:[2]

$$f \triangleq x \leftarrow \mathit{decide}();\ y \leftarrow \mathit{decide}();\ x \mathbin{\&\&} y$$

We can define a handler for *decide* to specify its semantics. The handler below handles *decide* by invoking the continuation k with *True* and *False* respectively, and collecting the results:

$$\mathbf{with}\ \{\ \mathit{decide} \mapsto \lambda x\ k.\ (k\ \mathit{True}) \mathbin{++} (k\ \mathit{False}),\ \mathbf{return} \mapsto \lambda x.\ [x]\ \}\ \mathbf{handle}\ f$$

Within the *decide* clause, x is the operation argument, in this case a unit, and k represents the captured delimited continuation that takes the operation's result and resumes the computation from the original call site. The handler explores both branches of the non-deterministic computation

[2]For clarity, we write $x \leftarrow e_1; e_2$ as syntactic sugar for $(\lambda x.\ e_2)\ e_1$; and $e_1; e_2$ for when $x \notin \mathsf{fv}(e_2)$.

by calling k twice and concatenates the result lists. The **return** clause applies when a value x is returned from the computation. Here, it simply wraps x in a singleton list. The return clause is optional, as a handler that has no special return behavior can have **return** $\mapsto \lambda x.\, x$. By applying this handler to f, we effectively explore all possible results of the *decide* operation, resulting in $[\mathit{True}, \mathit{False}, \mathit{False}, \mathit{False}]$.

As can be seen, effect handlers offer modularity by separating syntax and semantics of effect operations. However, an effect handler's implementation can only depend on the operation argument and the continuation result, and it cannot use a program's loss information to make decisions. In practice, a program's loss may actually depend on the operation result, and programs don't always return a loss value as their final output.

2.3 This Work: Handling the Selection Monad

This paper introduces a novel language design that integrates algebraic effect handlers with computational ideas derived from the selection monad. Programmers can write handlers that make use of loss continuations to make selections. Our design allows programmers to define custom selection functions. More broadly, handlers, while maintaining modularity by only handling operations within their scope, can now additionally leverage future loss information.

To see how our handler design works, consider the following example program where we write **loss** to record a loss value:

$$\mathit{pgm} \triangleq b \leftarrow \mathit{decide}();\;\; i \leftarrow \mathbf{if}\ b\ \mathbf{then}\ 1\ \mathbf{else}\ 2;\;\; \mathbf{loss}(2 * i);\;\; \mathbf{if}\ b\ \mathbf{then}\ 'a'\ \mathbf{else}\ 'b'$$

The **loss** operation is a dedicated writer effect operation that records a loss value. As it is a writer effect, multiple **loss** operations within such programs will be aggregated. This allows for a flexible and modular approach to incorporating loss computations.

We can handle the choice operation *decide* using the loss information; for example,

$$\begin{aligned}\mathbf{with}\ \{\ \mathit{decide} \mapsto \lambda x\ k\ l.\ y \leftarrow l\ \mathit{True}; z \leftarrow l\ \mathit{False};\\ \mathbf{if}\ y <= z\ \mathbf{then}\ k\ \mathit{True}\ \mathbf{else}\ k\ \mathit{False}\ \}\ \mathbf{handle}\ \mathit{pgm}\end{aligned}$$

Importantly, as we see in this example, losses are made accessible to handlers through special ***choice continuations***. These are loss continuations which associate a loss to each possible result of an operation. Concretely, handler operation definitions receive the choice continuation as an additional argument. The example handler given above compares the losses associated with *True* and *False*, and resumes computation with the choice (boolean selection) minimizing the loss. Using this handler to handle the previous program, b will be assigned *True*, resulting a loss of 2 and result $'a'$. In this case, the handler implements argmin, corresponding to Abadi and Plotkin [2021, 2023]. Importantly, however, with our design the selection is implemented as a separate handler. With the delimited continuations and choice continuations available, the handler can implement a variety of selections beyond argmax, as we will see in §4.3.

Notably, while the continuation k is delimited, the choice continuation l has a useful different scope discipline, which is delimited by a local construct, and otherwise global. This allows the handler to make decisions based on fine-grained control over choice continuations and losses. We make use of it to, e.g. restrict choice continuation scopes within while loops. Further, we do not lose generality: restricting the loss value to be the loss accumulated from the continuation can be implemented as a special case by using local. From the semantical perspective, as shown by the Adequacy Theorem (Theorem 5.5), the design corresponds to a programmable selection monad.

semantics	
loss function	γ, k, l
syntax	
loss continuation	g
choice continuation	l, f_l
delimited continuation	k, f_k

Fig. 1. Terminology

Fig. 1 presents our terminology and corresponding symbols (ambiguities are resolved by context). The loss continuation $\mathtt{g}$ is the programming manifestations of the loss function γ of the selection monad, which takes the result of the program and returns a loss. On the other hand, when an operation is handled, the choice continuation l, takes an operation result and returns a loss.

3 A Model Calculus

In this section we present λC, our higher-order model language incorporating the new kind of handlers with both choice and delimited continuations. We give it an operational semantics, show the standard progress and type safety theorems, and prove termination for a subset of it subject to a mild restriction permitting no "effect loops" in operations.

3.1 Syntax

The syntax of types and effects is given in Fig. 2. Types are ranged over by σ and τ. We also write *par*, *in*, *out* for types when talking about parameter or operation types.

We assume available a set of basic types b (including **loss**), and a set of effect labels ℓ. We take *effects* ϵ to be multisets of effect labels; we use juxtaposition $\epsilon\epsilon'$ for multiset union and write $\epsilon \subseteq \epsilon'$ for sub-multiset. As well as basic types, types include product types $(\sigma_1, \ldots, \sigma_n)$, sum types $(\sigma + \tau)$, natural numbers **nat**, and lists $\mathbf{list}(\sigma)$ for iterations and folds (two examples of simple inductive types), and function types $(\sigma \to \tau \,!\, \epsilon)$, with argument type σ, and result type τ and effect ϵ.

We further assume available a *signature* Σ of effect label typings $\ell : Op(\ell)$ associating effect labels ℓ to finite non-empty sets of ℓ-*operations* *op* (with disjoint sets associated to different effect labels). Our language is patterned after the Koka language [Leijen 2014] with its grouping of operations *op* into effects ℓ. Each $op \in Op(\ell)$ is typed $op : out \to in$; we often write $op : out \xrightarrow{\ell} in$ for $op \in Op(\ell)$ [3].

Expressions e and handlers h are given in Fig. 3, where $x, y, p, k, l \ldots$ range over variables. Note that expressions include loss continuation expressions $\mathtt{g}$. We make use of standard λ-calculus abbreviations, for example $\lambda^\epsilon(x, y) : (\sigma, \tau).\, e$ for functions of pairs.

Expressions include constants c and applications of basic functions f. Abstractions $\lambda^\epsilon x : \sigma.\, e$ are explicitly typed, and annotated with their result effect ϵ. We support *parameterized handlers* [Plotkin and Pretnar 2009], which generalize effect handlers by keeping a local handler parameter that can be updated during resumption. Having parameterized handlers is not necessary, but is convenient when implementing stateful effects. The parameterized handler expression **with** h **from** e_1 **handle** e_2 handles the computation e_2 using handler h, whose parameter has initial value e_1. A program can perform an operation $op(e)$ by passing the operation *op* an argument e. The expression $\mathbf{loss}(e)$ invokes the writer effect operation **loss**, adding a loss e. Note that, unlike other operations, it is a built-in effect not associated to any effect label and so cannot be handled; it can however be used in handlers, e.g., to define variant loss operations.

The expression $e_1 \blacktriangleright (\lambda^\epsilon x : \sigma.\, e_2)$ is used to build ***loss continuations*** $\mathtt{g}$; these form a subset of expressions. Loss continuations $\mathtt{g}$ begin with the zero loss continuation $0_{\sigma,\epsilon} =_{\text{def}} \lambda^\epsilon x : \sigma.\, 0$, and get extended by $\lambda^\epsilon x : \sigma.\, e \blacktriangleright \mathtt{g}$ (we assume $\blacktriangleright$ binds more tightly than λ). Intuitively, $(\blacktriangleright)$ (pronounced as "then") accumulates losses: it first evaluates e_1, collects the loss, and passes the evaluation result as x to e_2. The expression $\langle e \rangle^\epsilon_{\mathtt{g}}$ localises loss continuations to expressions e by executing them with loss continuation $\mathtt{g}$; in contrast, the expression $\mathbf{reset}\, e$ localises losses to e, preventing them from passing outside $\mathbf{reset}\, e$. To impose both forms of localisation the two constructs can be combined, and we write $\langle\!\langle e \rangle\!\rangle^\epsilon_{\mathtt{g}}$ for $\mathbf{reset}\, \langle e \rangle^\epsilon_{\mathtt{g}}$. While the language supports the most general local construct $\langle e \rangle^\epsilon_{\mathtt{g}}$,

[3] One might rather have expected $op : in \to out$. The idea here is that an operation is an effect: an element of *out* is sent to start the effect, then the operation returns an element of *in* to continue the computation. It may help to think of, e.g., I/O effects, where, with the present convention, output operations have type $out \to ()$ and input operations have type $() \to in$.

$$\begin{array}{llll} \text{type} & \sigma, \tau & ::= & b \mid (\sigma_1, \ldots, \sigma_n)\ (n \geqslant 0) \mid \sigma + \tau \mid \mathbf{nat} \mid \mathbf{list}(\sigma) \mid (\sigma \to \tau \,!\, \epsilon) \\ \text{effect} & \epsilon & ::= & \{\} \mid \epsilon\ell \qquad \Sigma \ ::= \ \{\overline{\ell_i : Op(\ell_i)}\} \qquad Op(\ell) \ ::= \ \{\overline{op_i : out_i \longrightarrow in_i}\} \end{array}$$

Fig. 2. Syntax of types and effects

$$\begin{array}{llll} \text{expr} & e & ::= & c \mid f(e) \mid x \mid \lambda^{\epsilon} x\!:\!\sigma.\, e \mid e_1\, e_2 \\ & & & \mid (e_1, \ldots, e_n) \mid e.i \\ & & & \mid \mathbf{inl}_{\sigma,\tau}(e) \mid \mathbf{inr}_{\sigma,\tau}(e) \mid \mathbf{cases}\ e\ \mathbf{of}\ x_1\!:\!\sigma_1.\, e_1 \,[\!]\, x_2\!:\!\sigma_2.\, e_2 \\ & & & \mid \mathbf{zero} \mid \mathbf{succ}(e) \mid \mathbf{iter}(e_1, e_2, e_3) \\ & & & \mid \mathbf{nil}_{\sigma} \mid \mathbf{cons}(e_1, e_2) \mid \mathbf{fold}(e_1, e_2, e_3) \\ & & & \mid op(e) \mid \mathbf{loss}(e) \mid \mathbf{with}\ h\ \mathbf{from}\ e_1\ \mathbf{handle}\ e_2 \\ & & & \mid e_1 \blacktriangleright \lambda^{\epsilon} x\!:\!\sigma.\, e_2 \mid \langle e \rangle^{\epsilon}_{\mathrm{g}} \mid \mathbf{reset}\ e \\ \text{loss cont exp} & \mathrm{g} & ::= & \lambda^{\epsilon} x\!:\!\sigma.\, 0 \mid \lambda^{\epsilon} x\!:\!\sigma.\, e \blacktriangleright \mathrm{g} \\ \text{handler} & h & ::= & \left\{ \begin{array}{l} op_1 \mapsto \lambda^{\epsilon} z\!:\!(par, out_1, (par, in_1) \longrightarrow \mathbf{loss}\,!\,\epsilon, (par, in_1) \longrightarrow \sigma'\,!\,\epsilon).\, e_1, \\ \ldots, \\ op_n \mapsto \lambda^{\epsilon} z\!:\!(par, out_n, (par, in_n) \longrightarrow \mathbf{loss}\,!\,\epsilon, (par, in_n) \longrightarrow \sigma'\,!\,\epsilon).\, e_n, \\ \mathbf{return} \mapsto \lambda^{\epsilon} z\!:\!(par, \sigma).\, e \end{array} \right\} \\ & & & \qquad (op_1, \ldots, op_n \text{ enumerates some } Op(\ell)) \end{array}$$

Fig. 3. Syntax of expressions and handlers

with any g, we find that $\langle e \rangle^{\epsilon}_{0_{\sigma,\epsilon}}$, localizing the loss with respect to the zero continuation, suffices for our examples (§4.3). Thus, ($\blacktriangleright$) and loss continuations need not be part of the user-facing syntax.

A handler h includes a list of operation definitions and a return definition; it *handles* ℓ if this list enumerates $Op(\ell)$ and has *result effect* ϵ if the abstractions in the definitions have result effect ϵ. Operations *op* takes a parameter, an operation argument, a choice continuation l (following our design), and a delimited continuation k. The choice continuation is the key innovation of this calculus. Both continuations take a potentially updated parameter and the operation result. Note that l returns **loss**, while k returns σ', and the two continuations are decorated by the effects ϵ they may cause. Finally, the return clause takes the final parameter and the computation result of type σ.

Remark. For space reasons, in the rest of the paper we focus on a subset of the language excluding sum types, natural numbers, and lists. The full language is detailed in the appendix.

3.2 Typing Rules

Fig. 4 presents the typing rules. As usual, environments Γ are finite sets of bindings $x\!:\!\sigma$ of types to variables, with no variable bound twice (equivalently, functions from a finite set $\mathrm{Dom}(\Gamma)$ of variables to types). The judgment $\Gamma \vdash e\!:\!\sigma\,!\,\epsilon$ is that under the context Γ, the expression e has type σ and may produce effects in ϵ. The type σ is determined, due to the type and effect annotations in the syntax. When Γ is empty, we may write $e\!:\!\sigma\,!\,\epsilon$; we may also write $\Gamma \vdash e\!:\!\sigma$ or $e\!:\!\sigma$ to show the judgments hold for some ϵ. The judgment $\Gamma \vdash h\!:\!par, \sigma\,!\,\epsilon\ell \Rightarrow \sigma'\,!\,\epsilon$ is that h takes a parameter of type *par* and a computation of σ and returns a result of type σ', producing one less effect ℓ; all of *par*, σ, ϵ, ℓ and σ' are determined. True judgments have unique derivations.

The typing rules are mostly standard for effect handler calculi. For rules CONST and FUN, we assume available the types of constants $c\!:\!b$, including $r\!:\!\mathbf{loss}$, for all $r \in R$, and primitive functions $f: \sigma \longrightarrow \tau$ (with σ and τ first order), including $+ : (\mathbf{loss}, \mathbf{loss}) \longrightarrow \mathbf{loss}$ (which we will write infix). Values can have any effect (rules CONST, VAR, PRD, and ABS). In particular, in rule ABS, the function body has the annotated type ϵ but the abstraction can have any effect ϵ'. In rule APP, we check that

$\boxed{\Gamma \vdash e : \sigma \,!\, \epsilon}$ *(Typing Expressions)*

$$\textsc{var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma \,!\, \epsilon} \qquad \textsc{const}\ \frac{(c : b)}{\Gamma \vdash c : b \,!\, \epsilon} \qquad \textsc{fun}\ \frac{\Gamma \vdash e : \sigma \,!\, \epsilon \qquad (f : \sigma \to \tau)}{\Gamma \vdash f(e) : \tau \,!\, \epsilon} \qquad \textsc{prd}\ \frac{\Gamma \vdash e_i : \sigma_i \,!\, \epsilon \qquad (i = 1, \ldots, n)}{\Gamma \vdash (e_1, \ldots, e_n) : (\sigma_1, \ldots, \sigma_n) \,!\, \epsilon}$$

$$\textsc{prj}\ \frac{\Gamma \vdash e : (\sigma_1, \ldots, \sigma_n) \,!\, \epsilon \qquad (i = 1, \ldots, n)}{\Gamma \vdash e.i : \sigma_i \,!\, \epsilon} \qquad \textsc{abs}\ \frac{\Gamma, x : \sigma \vdash e : \tau \,!\, \epsilon}{\Gamma \vdash \lambda^{\epsilon} x : \sigma .\, e : (\sigma \to \tau \,!\, \epsilon) \,!\, \epsilon'}$$

$$\textsc{app}\ \frac{\Gamma \vdash e_1 : (\sigma \to \tau \,!\, \epsilon) \,!\, \epsilon \qquad \Gamma \vdash e : \sigma \,!\, \epsilon}{\Gamma \vdash e_1\ e_2 : \tau \,!\, \epsilon} \qquad \textsc{op}\ \frac{op : out \xrightarrow{\ell} in \in \Sigma \qquad \Gamma \vdash e : out \,!\, \epsilon \qquad \ell \in \epsilon}{\Gamma \vdash op(e) : in \,!\, \epsilon}$$

$$\textsc{loss}\ \frac{\Gamma \vdash e : \mathbf{loss} \,!\, \epsilon}{\Gamma \vdash \mathbf{loss}(e) : () \,!\, \epsilon} \qquad \textsc{handle}\ \frac{\Gamma \vdash h : par, \sigma \,!\, \epsilon\ell \Rightarrow \sigma' \,!\, \epsilon \qquad \Gamma \vdash e_1 : par \,!\, \epsilon \qquad \Gamma \vdash e_2 : \sigma \,!\, \epsilon\ell}{\Gamma \vdash \mathbf{with}\ h\ \mathbf{from}\ e_1\ \mathbf{handle}\ e_2 : \sigma' \,!\, \epsilon}$$

$$\textsc{then}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \sigma \,!\, \epsilon_1 \\ \Gamma, x : \sigma \vdash e_2 : \mathbf{loss} \,!\, \epsilon_2 \qquad (\epsilon_2 \subseteq \epsilon_1)\end{array}}{\Gamma \vdash e_1 \blacktriangleright (\lambda^{\epsilon_2} x : \sigma .\, e_2) : \mathbf{loss} \,!\, \epsilon_1} \qquad \textsc{glocal}\ \frac{\begin{array}{c}\Gamma \vdash e : \sigma \,!\, \epsilon_1 \\ \Gamma \vdash g : \sigma \to \mathbf{loss} \,!\, \epsilon_2 \qquad (\epsilon_2 \subseteq \epsilon_1 \subseteq \epsilon)\end{array}}{\Gamma \vdash \langle e \rangle_{g}^{\epsilon_1} : \sigma \,!\, \epsilon} \qquad \textsc{reset}\ \frac{\Gamma \vdash e : \sigma \,!\, \epsilon}{\Gamma \vdash \mathbf{reset}\, e : \sigma \,!\, \epsilon}$$

$\boxed{\Gamma \vdash h : par, \sigma \,!\, \epsilon\ell \Rightarrow \sigma' \,!\, \epsilon}$ *(Typing Handlers)*

$$\textsc{handler}\ \frac{\begin{array}{c} Op(\ell) = \{op_1, \ldots, op_n\} \qquad op_i : out_i \to in_i \quad (i = 1, \ldots, n) \\ \Gamma \vdash e_i : ((par, out_i, (par, in_i) \to \mathbf{loss} \,!\, \epsilon, (par, in_i) \to \sigma' \,!\, \epsilon) \to \sigma' \,!\, \epsilon) \,!\, \epsilon \qquad \Gamma \vdash e : ((par, \sigma) \to \sigma' \,!\, \epsilon) \,!\, \epsilon \end{array}}{\Gamma \vdash \{op_1 \mapsto e_1, \ldots, op_n \mapsto e_n, \mathbf{return} \mapsto e\} : par, \sigma \,!\, \epsilon\ell \Rightarrow \sigma' \,!\, \epsilon}$$

Fig. 4. Typing rules for λC

the argument has the expected type, and that the effects of the function, the function body, and the argument match.[4] Rules PRD and PRJ are self-explanatory.

When performing an operation (rule OP), the global context tells us that the operation takes a type *out* and has return type *in*. Then we check that the operation argument has type *out*, and the return type is *in*. Moreover, we need to make sure that the result effect ϵ includes the effect label ℓ. For the loss operation (rule LOSS), the operation takes a loss and has unit return type.

Rule HANDLE takes care of handling. The rule checks that e_1 has the correct parameter type, and the computation being handled has type σ, while the handler h takes a computation in σ and has return type σ', and thus the final result has type σ'. Rule THEN checks the loss calculation expression. Note that $\lambda^{\epsilon_2} x : \sigma .\, e_2$ may produce fewer effects than than the expression e. Rule GLOCAL checks loss-continuation-localized computations. Note again that the loss continuation may produce fewer effects than the localized computation; further there is an "effect conversion" from the effects of that computation to the effects of the whole localized computation. These variations in effects are needed to account for the loss continuations built up by the operational semantics (see the discussion of rule (F) below); they also provide programming flexibility.

Finally, the rule HANDLER type-checks handler definitions. A handler for ℓ handles all operations in $Op(\ell)$. For each operation $op_i : out_i \to in_i$, the corresponding clause e_i takes a parameter of type *par*, an operation argument of type out_i, a choice continuation of type $((par, in_i) \to \mathbf{loss} \,!\, \epsilon)$ and a delimited continuation of type $((par, in_i) \to \sigma' \,!\, \epsilon)$, and has return type σ'. The return clause e

[4] Our type-theoretical formalism does not enjoy sub-effecting, similar to row-based effect style [Hillerström and Lindley 2016; Leijen 2017]. Semantically too there is an obstacle. For a sub-effecting rule APP, the outer ϵ should be any super-effects of the inner one. However, a semantics using a monad family T_ϵ, then needs covariance in the ϵ and our selection monad family is not. On the other hand, rules rule THEN and rule GLOCAL employ sub-effecting—needed for the operational semantics, and holding in our denotational semantics. (See the discussions in Sections 3.3 and 5.)

takes a parameter of type *par* and the computation result of type σ, and has return type σ'. Because of the return clause, the handler takes a σ-computation and returns a σ'-computation.

3.3 Operational Semantics

In this section, we present the small-step operational semantics of λC. Our rules axiomatize a judgment $g \vdash_\epsilon e \xrightarrow{r} e'$ that under the loss continuation g, e makes a transition to e', producing loss $r \in R$. The decorative ϵ provides auxiliary information needed to construct loss continuations. Here g produces the loss caused by the rest of the program, given the result of executing e.

Before giving the rules we need some syntax to cover values, contexts, stuck expressions, and redexes. Fig. 5 presents the syntactical classes needed for the operational semantics. *Values* v include variables, constants, value tuples, and lambda expressions. (Note that values can be typed with any effect, and we generally just write $\Gamma \vdash v : \sigma$ or $\vdash v : \sigma$.) *Continuation contexts* K are either a hole $\square$, or a regular frame F or special frame S followed by a continuation context. (We distinguish between regular frames and special frames,since, as we will see, they extend loss continuations differently.) We write $K[e]$ for the expression obtained by filling the hole in K with e.

Stuck expressions $u = K[op(v)]$ are operation invocations $op(v)$ that cannot be handled by handlers in continuation contexts K; We write $\mathsf{h}_{\text{eff}}(K)$ for the multiset of effect labels that K handles; it is defined inductively with main clause $\mathsf{h}_{\text{eff}}(\textbf{with } h \textbf{ from } v \textbf{ handle} K') = \mathsf{h}_{\text{eff}}(K')\ell$, where h handles ℓ; we further set $\mathsf{h}_{\text{op}}(K) = \{op \in Op(\ell) | \ell \in \mathsf{h}_{\text{eff}}(K)\}$, the set of operations handled by K. *Terminal expressions* w are values or stuck expressions; they cannot reduce; in contrast, (closed) *redexes* R are expressions that do.

Expressions can be analysed uniquely:

Lemma 3.1 (Expression analysis). *Every expression has exactly one of the following five forms: (1) a value v (for a unique v), (2) a stuck expression $K[op(v)]$ (for unique K, op, and v), (3) a redex R (for a unique R), (4) $F[e]$ (for unique F and e, with e not a value or stuck), or (5) $S[e]$ (for unique S and e, with e not a value or stuck).*

Small-step operational semantics. Fig. 6 presents our small-step operation semantics rules for the judgment $g \vdash_\epsilon e \xrightarrow{r} e'$. Program execution starts with the zero loss continuation. Further loss continuations are progressively built up during execution, in order for subprograms to pass their results to their enclosing contexts and so on to the program's loss continuation. (All this is quite analogous to how one computes with ordinary continuations.)

Redexes. Many redex rules do not use the loss continuation, and produce a zero loss. For (R1), we assume available deterministic total reductions for primitive functions. In (R4), $\textbf{loss}(r)$ produces a loss r returning $()$. Rules (R8) and (R9) are natural, expressing that the computation terminates once a value is reached.

Operations get handled by rule (R5). The handler operation clause is applied to parameter v_1, operation argument v_2, **delimited continuation** f_k, and ***choice continuation*** f_l. The continuation f_k takes the handler parameter and the operation result to be used when resuming, localised to the current loss continuation g when called, since the continuation is captured under the loss continuation g. The choice continuation f_l is built from the standard handler continuation and the current loss continuation using the $\blacktriangleright$ construct; it therefore has access to all the losses resulting from executing the operation, and so the handler can make decisions based on that information. In rule (R6), when a value returns from a handler, the return clause from the handler applies, taking as arguments the current handler parameter and the value. In rule (R7), we evaluate $v \blacktriangleright \lambda^{\epsilon_2} x : \sigma.\ e$ by substituting v for x in e and localising the resulting expression to the zero loss continuation, as the purpose of $\blacktriangleright$ is to calculate a loss independently of the current loss continuation.

$$
\begin{array}{llcl}
\text{value} & v & ::= & x \mid c \mid (v_1, \ldots, v_n) \mid \lambda^{\epsilon} x\,{:}\,\sigma.\, e \\
\text{regular frame} & F & ::= & f(\square) \mid (v_1, \ldots, v_k, \square, e_{k+2}, \ldots, e_n) \mid \square.i \mid \square\ e \mid v\ \square \\
 & & \mid & op(\square) \mid \mathbf{loss}(\square) \mid \mathbf{with}\ h\ \mathbf{from}\ \square\ \mathbf{handle}\ e \\
\text{special frame} & S & ::= & \mathbf{with}\ h\ \mathbf{from}\ v\ \mathbf{handle}\ \square \mid \square \blacktriangleright (\lambda^{\epsilon} x\,{:}\,\sigma.e) \mid \langle \square \rangle^{\epsilon}_{\mathsf{g}} \mid \mathbf{reset}\ \square \\
\text{cont context} & K & ::= & \square \mid F[K] \mid S[K] \\
\text{stuck expr} & u & ::= & K[op(v)] \quad (op \notin \mathsf{h}_{\mathrm{op}}(K)) \\
\text{terminal expr} & w & ::= & v \mid u \\
\text{redex} & R & ::= & f(v) \mid v.i \mid v_1\ v_2 \mid \mathbf{loss}(v) \\
 & & \mid & \mathbf{with}\ h\ \mathbf{from}\ v_1\ \mathbf{handle}\ K[op(v_2)] \quad (op \notin \mathsf{h}_{\mathrm{op}}(K), op \in h) \\
 & & \mid & \mathbf{with}\ h\ \mathbf{from}\ v_1\ \mathbf{handle}\ v_2 \\
 & & \mid & v \blacktriangleright \lambda^{\epsilon} x\,{:}\,\sigma.e_1 \mid \langle v \rangle^{\epsilon}_{\mathsf{g}} \mid \mathbf{reset}\ v
\end{array}
$$

Fig. 5. Syntactical classes used for operational semantics

$$
\begin{array}{lllll}
(R1) & \mathsf{g} \vdash_{\epsilon} f(v) & \xrightarrow{0} & v' & (f(v) \to v') \\
(R2) & \mathsf{g} \vdash_{\epsilon} (v_1, \ldots, v_n).i & \xrightarrow{0} & v_i & \\
(R3) & \mathsf{g} \vdash_{\epsilon} (\lambda^{\epsilon} x\,{:}\,\sigma.\, e)\ v & \xrightarrow{0} & e[v/x] & \\
(R4) & \mathsf{g} \vdash_{\epsilon} \mathbf{loss}(r) & \xrightarrow{r} & () &
\end{array}
$$

$$
(R5)\quad \frac{\begin{array}{c} op \notin \mathsf{h}_{\mathrm{op}}(K) \qquad op \mapsto v_o \in h \qquad v_1 : par \qquad op\,{:}\,out \xrightarrow{\ell} in \\ h \text{ has effect } \epsilon \qquad f_k = \lambda^{\epsilon}(p, y) : (par, in).\ \langle \mathbf{with}\ h\ \mathbf{from}\ p\ \mathbf{handle}\ K[y] \rangle^{\epsilon}_{\mathsf{g}} \\ f_l = \lambda^{\epsilon}(p, y) : (par, in).\ (\mathbf{with}\ h\ \mathbf{from}\ p\ \mathbf{handle}\ K[y]) \blacktriangleright \mathsf{g} \end{array}}{\mathsf{g} \vdash_{\epsilon} \mathbf{with}\ h\ \mathbf{from}\ v_1\ \mathbf{handle}\ K[op(v_2)] \xrightarrow{0} v_o(v_1, v_2, f_l, f_k)}
$$

$$
\begin{array}{lllll}
(R6) & \mathsf{g} \vdash_{\epsilon} \mathbf{with}\ h\ \mathbf{from}\ v_1\ \mathbf{handle}\ v_2 & \xrightarrow{0} & v_r(v_1, v_2) & (\mathbf{return} \mapsto v_r \in h) \\
(R7) & \mathsf{g} \vdash_{\epsilon} v \blacktriangleright \lambda^{\epsilon_1} x\,{:}\,\sigma.e & \xrightarrow{0} & \langle e[v/x] \rangle^{\epsilon_1}_{\lambda^{\epsilon_1} x:\sigma.\, 0} & \\
(R8) & \mathsf{g} \vdash_{\epsilon} \langle v \rangle^{\epsilon_1}_{\mathsf{g}_1} & \xrightarrow{0} & v & \\
(R9) & \mathsf{g} \vdash_{\epsilon} \mathbf{reset}\ v & \xrightarrow{0} & v &
\end{array}
$$

$$
(F)\quad \frac{\lambda^{\epsilon} x\,{:}\,\tau.\, F[x] \blacktriangleright \mathsf{g} \vdash_{\epsilon} e \xrightarrow{r} e'}{\mathsf{g} \vdash_{\epsilon} F[e] \xrightarrow{r} F[e']}
$$

$$
(S1)\quad \frac{\begin{array}{c} h \text{ has effect } \epsilon \qquad h \text{ handles } \ell \qquad \mathbf{return} \mapsto v_r \in h \\ v_r : (par, \sigma) \to \sigma' \,!\, \epsilon \qquad \lambda^{\epsilon} x\,{:}\,\sigma.\, (v_r(v, x) \blacktriangleright \mathsf{g}) \vdash_{\epsilon\ell} e \xrightarrow{r} e' \end{array}}{\mathsf{g} \vdash_{\epsilon} \mathbf{with}\ h\ \mathbf{from}\ v\ \mathbf{handle}\ e \xrightarrow{r} \mathbf{with}\ h\ \mathbf{from}\ v\ \mathbf{handle}\ e'}
$$

$$
(S2)\quad \frac{\mathsf{g}_1 \vdash_{\epsilon} e \xrightarrow{r} e'}{\mathsf{g} \vdash_{\epsilon} (e \blacktriangleright \mathsf{g}_1) \xrightarrow{0} r + (e' \blacktriangleright \mathsf{g}_1)}
$$

$$
(S3)\quad \frac{\mathsf{g}_1 \vdash_{\epsilon_1} e \xrightarrow{r} e'}{\mathsf{g} \vdash_{\epsilon} \langle e \rangle^{\epsilon_1}_{\mathsf{g}_1} \xrightarrow{r} \langle e' \rangle^{\epsilon_1}_{\mathsf{g}_1}}
$$

$$
(S4)\quad \frac{\mathsf{g} \vdash_{\epsilon} e \xrightarrow{r} e'}{\mathsf{g} \vdash_{\epsilon} \mathbf{reset}\ e \xrightarrow{0} \mathbf{reset}\ e'}
$$

Fig. 6. Small-step operational semantics rules

Regular frames. Rule (F) evaluates expressions e inside regular frames F.[5] Importantly, we adjust the loss continuation g to $\lambda^{\epsilon} x:\tau.\, F[x] \blacktriangleright g$ when evaluating e. This is because the loss continuation of e is to pass its result v to the context $F[v]$ whose value is then passed to its enclosing loss context, meanwhile accumulating incurred losses. Note that, to apply this rule, the decorative ϵ is used. This is the only rule that does so, and it is needed to make the rule deterministic.

Special frames. Lastly, (S1)-(S4) evaluate inside special frames. These adjust the loss continuations or losses differently from rule (F). The loss continuation in (S1) uses the return clause from the handler, since after e is evaluated with the aid of the handler the final result is passed to the return function. Also, rule (S1) is where the effect associated with the judgment changes. It changes from ϵ to $\epsilon\ell$, when evaluating e. Rule (R5) builds up a loss continuation by combining an expression with a loss continuation with fewer effects. As a result, sub-effecting is needed in the typing rule THEN to ensure type safety of the operational semantics. Similarly, sub-effecting is used in the typing rule GLOCAL, since rule (R7) further wraps the body of a loss continuation within a local construct. In rule (S2), the current loss continuation is not imported inside the "then" construct. Instead e is evaluated relative to the loss continuation g_1; moreover, the loss r produced during the evaluation is added to the final result. In rule (S3) the loss continuation also changes, as with the local construct; note that the loss created by e is exported. Finally, in rule (S4) the loss continuation does not change, but the loss created by e is not exported. The standard results hold for our operational semantics:

THEOREM 3.2.

(1) *(Terminal expressions) If e is terminal, then it can make no transition, i.e., $g \vdash_{\epsilon} e \xrightarrow{r} e'$ holds for no g, r, ϵ, e'.*

(2) *(Determinism) If $g \vdash_{\epsilon} e \xrightarrow{r} e'$ and $g \vdash_{\epsilon} e \xrightarrow{r'} e''$ then $r = r'$ and $e' = e''$.*

(3) *(Progress) If $e : \sigma \,!\, \epsilon_1$ is non-terminal, then $g \vdash_{\epsilon_1} e \xrightarrow{r} e'$ holds for some r and e' for any $g : \sigma \longrightarrow \mathbf{loss} \,!\, \epsilon_2$ with $\epsilon_2 \subseteq \epsilon_1$.*

(4) *(Type safety) If $g : \sigma \longrightarrow \mathbf{loss} \,!\, \epsilon_2$, $g \vdash_{\epsilon_1} e \xrightarrow{r} e'$, with $\epsilon_2 \subseteq \epsilon_1$, and $e : \sigma \,!\, \epsilon_1$ then $e' : \sigma \,!\, \epsilon_1$.*

Example. We consider the example program from §2.3 to demonstrate the operational semantics:

$$pgm \triangleq b \leftarrow decide();\;\; i \leftarrow \mathbf{if}\ b\ \mathbf{then}\ 1\ \mathbf{else}\ 2;\;\; \mathbf{loss}(2 * i);\;\; \mathbf{if}\ b\ \mathbf{then}\ 'a'\ \mathbf{else}\ 'b'$$
$$h \triangleq \{\ decide \mapsto \lambda x\ k\ l.\ y \leftarrow l\ True; z \leftarrow l\ False; \mathbf{if}\ y <= z\ \mathbf{then}\ k\ True\ \mathbf{else}\ k\ False\ \}$$

We evaluate the program under the zero continuation, and omit handler parameters. We write C for the character type, and B for the boolean type. First, the operation is handled (rule (R5)):

$$0_{C,\{\}} \vdash_{\{\}} \mathbf{with}\ h\ \mathbf{handle}\ pgm \xrightarrow{0} (y \leftarrow f_l\ True; z \leftarrow f_l\ False; \mathbf{if}\ y <= z\ \mathbf{then}\ f_k\ True\ \mathbf{else}\ f_k\ False) \quad (1)$$

where $f_k = \lambda b : B.\ \langle \mathbf{with}\ h\ \mathbf{handle}\ (i \leftarrow \mathbf{if}\ b\ \mathbf{then}\ 1\ \mathbf{else}\ 2;\ \ \mathbf{loss}(2 * i);\ \ \mathbf{if}\ b\ \mathbf{then}\ 'a'\ \mathbf{else}\ 'b')\rangle^{\{\}}_{0_{C,\{\}}}$

$f_l = \lambda b : B.\ (\mathbf{with}\ h\ \mathbf{handle}\ (i \leftarrow \mathbf{if}\ b\ \mathbf{then}\ 1\ \mathbf{else}\ 2;\ \ \mathbf{loss}(2 * i);\ \ \mathbf{if}\ b\ \mathbf{then}\ 'a'\ \mathbf{else}\ 'b')) \blacktriangleright 0_{C,\{\}}$

We then evaluate $(f_l\ True)$. Rule (F) changes the loss continuation to $g \triangleq \lambda^{\epsilon} y:\tau.\,(z \leftarrow f_l\ False;$ $\mathbf{if}\ y <= z\ \mathbf{then}\ f_k\ True\ \mathbf{else}\ f_k\ False) \blacktriangleright 0_{C,\{\}}$. Rule (R3) reduces the application and produces a 0 loss. Now we evaluate the following expression under g:

$$g \vdash_{\{\}} (\mathbf{with}\ h\ \mathbf{handle}\ (i \leftarrow \mathbf{if}\ True\ \mathbf{then}\ 1\ \mathbf{else}\ 2;\ \ \mathbf{loss}(2 * i);\ \ \mathbf{if}\ True\ \mathbf{then}\ 'a'\ \mathbf{else}\ 'b')) \blacktriangleright 0_{C,\{\}} \quad (2)$$

Importantly, the $\blacktriangleright$ operator disregards g, and evaluates the expression under $0_{C,\{\}}$ (rule (S2)). This behavior ensures that continuations are consistently evaluated under the loss continuation they are captured at. Without it, evaluating continuations would yield different results based on how continuations are used within the handler, which is undesirable. Then, evaluating the program

$$0_{C,\{\}} \vdash_{\{\}} (\mathbf{with}\ h\ \mathbf{handle}\ (i \leftarrow \mathbf{if}\ True\ \mathbf{then}\ 1\ \mathbf{else}\ 2;\ \ \mathbf{loss}(2 * i);\ \ \mathbf{if}\ True\ \mathbf{then}\ 'a'\ \mathbf{else}\ 'b')) \quad (3)$$

[5]The use of frames is a way to present the administrative rules of small-step semantics via a single rule; the idea seems to be folklore. We could as well have used evaluation contexts [Felleisen and Hieb 1992].

produces a loss 2 and a value $'a'$. According to rule (S2), the loss is added to the result of $('a' \blacktriangleright 0_{C,\{\}})$, producing the result $2 + 0 = 2$. Substituting 2 for y in expression (1), we get

$$(z \leftarrow f_l\ \mathit{False}; \mathbf{if}\ 2 <= z\ \mathbf{then}\ f_k\ \mathit{True}\ \mathbf{else}\ f_k\ \mathit{False}) \tag{4}$$

Similarly, $(f_l\ \mathit{False})$ evaluates to 4, and thus the computation reduces to $(f_k\ \mathit{True})$. Continuing the evaluation will produce the final result $'a'$ and the loss 2.

Big-step operational semantics. Finally, we define a big-step operational semantics judgment $\mathrm{g} \vdash e \overset{r}{\Longrightarrow} w$, that under loss continuation g, expression e evaluates to terminal expression w.

$$\frac{}{\mathrm{g} \vdash_\epsilon w \overset{0}{\Longrightarrow} w} \qquad \frac{\mathrm{g} \vdash_\epsilon e_1 \overset{r}{\longrightarrow} e_2 \qquad \mathrm{g} \vdash_\epsilon e_2 \overset{s}{\Longrightarrow} w}{\mathrm{g} \vdash_\epsilon e \overset{r+s}{\Longrightarrow} w}$$

Fig. 7. Big-step operational semantics rules

It follows immediately from Theorem 3.2 that the big-step semantics is deterministic and type safe:

Corollary 3.3. *Given $e : \sigma\,!\,\epsilon$, and $\mathrm{g} : \sigma \to \mathbf{bool}\,!\,\epsilon'$ with $\epsilon' \subseteq \epsilon$, there is at most one $r \in R$ and terminal expression w such that $\mathrm{g} \vdash e \overset{r}{\Longrightarrow} w$ and then $w : \sigma\,!\,\epsilon$.*

3.4 Termination

We establish termination with a suitable well-foundedness assumption on the effects allowed in the input and output types of operations. We use the termination result to establish adequacy in §5. A result of this type for a standard handler calculus appears in Forster et al. [2017]. However they did not have loss continuations which, as we will see, leads to complex computability definitions.

Well-foundness of effects. Unfortunately, not all effect handler programs terminate. Adapting from Bauer and Pretnar [2013], consider an effect cow with the corresponding handler h:

$$cow : \{\ moo : \mathbf{unit} \to (\mathbf{unit} \to \mathbf{unit}\,!\,cow)\ \} \qquad h = \{\ moo \mapsto \lambda(p, x, \ell, k).\ k\ (\lambda^{cow} y.\ moo(())\ ())\ \}$$

Then the program $e \triangleq \mathbf{with}\ h\ \mathbf{from}\ v\ \mathbf{handle}\ moo(())\ ()$ diverges:

$$e \longrightarrow \mathbf{with}\ h\ \mathbf{from}\ v\ \mathbf{handle}\ (\lambda^{cow} y.\ moo(())\ ())\ () \longrightarrow e \longrightarrow \ldots$$

To rule out such programs where effect labels occur inside the input or output types of their operations, for this subsection and §5 we make use of a well-foundedness assumption on effects. Specifically, we write $e(\epsilon)$ and $e(\sigma)$ for the set of effect labels appearing in ϵ or σ. So, for example $e(\sigma \to \tau\,!\,\epsilon) = e(\sigma) \cup e(\tau) \cup \{\ell | \ell \in \epsilon\}$. Our well-foundedness assumption is that there is an ordering $\ell_1, \ldots, \ell_n$ of the labels such that:

$$op : out \overset{\ell_j}{\longrightarrow} in\ \wedge\ \ell_i \in e(out) \cup e(in) \implies i < j$$

We then define the effect levels of ϵ and σ by: $l(\epsilon) = \max_i\{i | \ell_i \in e(\epsilon)\}$ and $l(\sigma) = \max_i\{i | \ell_i \in e(\sigma)\}$. The size $|\sigma|$ of types is defined standardly (e.g., $|\sigma \to \tau\,!\,\epsilon| = 1 + |\sigma| + |\tau| + |\epsilon|$).

Our denotational semantics is defined for programs satisfying the assumption. We remark that the assumption holds for all our programming examples. In the (also terminating) EFF language [Forster et al. 2017], the assumption is baked into the language design: operation types are only well-defined if they can be shown so using only previously well-typed operations.

Computability. Our proof uses suitable mutually recursively-defined notions of computability [Tait 1967]. We define the following main notions:

- *computability* of closed values $v : \sigma$,
- *loss computability* of closed loss continuations $g : \sigma \to \textbf{loss} \,!\, \epsilon$, and
- *computability* of closed expressions $e : \sigma \,!\, \epsilon$.

We define these notions by the following mutually-recursive clauses. They employ two auxiliary notions. One is an inductively defined notion of *G-computability* of expressions, where G is a set of loss continuations; the other is a notion of *R-computability* of real-valued expressions. A proof of termination involving a similar inductively defined computability predicate was given in Kuchta [2022] for a handler language without loss continuations.

(1) (a) Every constant $c : b$ of ground type is computable.
  (b) A closed value $(v_1, \dots, v_n) : (\sigma_1, \dots, \sigma_n)$ is computable if every $v_i : \sigma_i$ is computable.
  (c) A closed value $\lambda^{\epsilon} x : \sigma.\, e : \sigma \to \tau \,!\, \epsilon$ is computable if, for every computable value $v : \sigma$, the expression $e[v/x] : \tau \,!\, \epsilon$ is computable.

(2) The property of G-computability of closed expressions $e : \sigma \,!\, \epsilon$, for a set G of closed loss continuations of type $g : \sigma \to \textbf{loss} \,!\, \epsilon'$ for some $\epsilon' \subseteq \epsilon$, is the least such property $P_{\sigma,\epsilon}$ of these expressions such that one of the following three possibilities holds:
  (a) e is a computable closed value.
  (b) e is a stuck expression $K[op(v)]$, with $op : out \xrightarrow{\ell} in$, where $v : out$ is a computable closed value, and where, for every computable closed value $v_1 : in$, $P_{\sigma,\epsilon}(K[v_1])$ holds.
  (c) For every $g \in G$, if $g \vdash_{\epsilon} e \xrightarrow{r} e'$ then $P_{\sigma,\epsilon}(e')$ holds.

(3) (a) An expression $e : \textbf{loss} \,!\, \epsilon$ is R-computable iff it is $\{0_{\textbf{loss},\epsilon}\}$-computable.
  (b) A closed loss continuation $\lambda^{\epsilon} x : \sigma.\, e : \sigma \to \textbf{loss} \,!\, \epsilon$ is loss computable if $e[v/x] : \textbf{loss} \,!\, \epsilon$ is R-computable for every computable closed value $v : \sigma$.

(4) A closed expression $e : \sigma \,!\, \epsilon$ is computable iff it is L-computable, where L is the set of closed loss-computable loss continuations $g : \sigma \to \textbf{loss} \,!\, \epsilon'$, for some $\epsilon' \subseteq \epsilon$.

To see these definitions are proper, note that each is parameterized by types σ and effects ϵ associated to the relevant syntactic entities, namely $(\sigma, \{\})$ for values $v : \sigma$, and (σ, ϵ) for loss continuations $g : \sigma \to \textbf{loss} \,!\, \epsilon$ and expressions $e : \sigma \,!\, \epsilon$. We associate pairs of natural numbers, lexicographically ordered, to such pairs by: $m(\sigma, \epsilon) = (l(\sigma) \max l(\epsilon), |\sigma|)$. These measures do not increase in passing from the definition of one notion to another, so every link in the multigraph of definitional dependencies is non-increasing. The measures also decrease when passing from the value-computability to itself or to computability. So every loop in the graph contains a decreasing link, and we see that the various notions are well-defined. Computability extends to open expressions and loss continuations in the usual way, via substitution by computable closed values.

Lemma 3.4 (Fundamental Lemma).

(1) Every loss continuation $\Gamma \vdash g : \sigma \to \textbf{loss} \,!\, \epsilon$ is loss computable.
(2) Every expression $\Gamma \vdash e : \sigma \,!\, \epsilon$ is computable.

We can deduce termination from computability.

Theorem 3.5 (Termination). *For $e_1 : \sigma \,!\, \epsilon$ and $g : \sigma \to \textbf{bool} \,!\, \epsilon'$ with $\epsilon' \subseteq \epsilon$, there are no infinite sequences: $g \vdash e_1 \xrightarrow{r_1} e_2 \xrightarrow{r_2} \dots \xrightarrow{r_{n-2}} e_{n-1} \xrightarrow{r_{n-1}} e_n \dots$*

Combining this with Theorem 3.3 we obtain the following theorem. It covers any effect multiset ϵ; when ϵ is empty, the terminal is a value by the well-typing.

Theorem 3.6. *For $e : \sigma ! \epsilon$ and $g : \sigma \to \textbf{bool} ! \epsilon'$ with $\epsilon' \subseteq \epsilon$ we have $g \vdash e \overset{r}{\Rightarrow} w$ for a unique $r \in R$ and terminal expression w (and then $w : \sigma ! \epsilon$).*

4 Programming with the Selection Monad

Having established the operational semantics of our design, we implemented it as an effect handler library in Haskell. In this section we first present the programming interface, then briefly explain the embedding, and, lastly, present programming examples.

4.1 Effect Handler Interface

We begin with a simple example to demonstrate the programming interface. An effect is declared as a datatype with its fields being operations. For example, the following datatype:

```
[effect| data NDet = NDet {decide :: Op () Bool}]
```

declares a *NDet* effect (§2.2) with an operation *decide* from () to *Bool*. This embedding uses a Template Haskell interface (similar to Kammar et al. [2013]) to reduce burdensome syntax.

We can perform an operation and handle an effectful program as follows. A handler (*NDet* {*decide* = *operation* (...)}) is simply an instance of the data type with field *decide*. The function *operation* takes a lambda expression ($\lambda x\ l\ k \to e$) and returns type *Op*, whose arguments are, respectively, the operation argument, the choice continuation, and the delimited continuation. The function *handlerRet* takes a return clause, a handler definition, and the computation to be handled. We can also use *handler* without a return clause. The implementation also supports parameterized handlers.

```
pgm =
  handlerRet (λx → return [x])
             (NDet {decide = operation (λx l k →
                (++) ⟨$⟩ k True ⟨∗⟩ k False)}) $
    do y ← perform decide (); return (not y)
```

Finally, a computation is written in a **do** block. This can invoke operations using *perform*, by providing an operation and its argument (in this case *decide* and ()). We can run the program by calling *runSel*. For example, *runSel pgm* returns [*False*, *True*].

4.2 The Selection Monad

We define the key datatype *Sel r e a* implementing the programming interface: the loss type *r* is any *Monoid* (not just a specific numerical type), *e* is the program's effect, and *a* is it's type:

```
newtype Sel r e a = Sel {unSel :: Monoid r ⇒ (a → Eff r e r) → Eff r e (r, a))}
```

It takes a loss continuation ($a \to$ *Eff* $r\ e\ r$) and returns a loss-value tuple (r, a). (Note that the definition corresponds to the semantic model $S_\epsilon(X) = (X \to F_\epsilon(R)) \to F_\epsilon(W(X))$ (§2.1).) We use the *Eff* datatype to represent effectful programs. Our design is independent of the concrete strategy used for implementing effect handlers. We implemented them using *multi-prompt delimited continuations* [Dyvbig et al. 2007; Xie and Leijen 2021]. This implementation closely follows the operational semantics of effect handlers. For example, we can define *loss* as follows:

```
loss r = Sel $ λ_ → return (r, ())
```

The definition corresponds to rule $(R4)$ in Fig. 6, which ignores the loss continuation, produces a loss r, and returns a unit value.

We present the monad instance declaration for *Sel* on the right. The definition requires some explanations. The *return* definition is straightforward: we ignore the loss continuation and the handler context, and return a pure tuple with a zero loss. This corresponds to the evaluation of terminal expressions (Fig. 7). The bind definition corresponds to rule (F) in Fig. 6. First, given the loss continuation g, we would first like to evaluate e. However, we first need to extend the loss continuation. Here, g is a loss continuation for $(e \gg= f)$, not for e. Therefore, we transform the loss continuation where ▹ implements the *then* operator, and evaluate e to the extended loss continuation. The result of evaluating e is then passed to f, where f takes a and the loss continuation g, yielding a loss $r2$ and a value b. Lastly, the two losses are combined ($r1$<>$r2$), and the final result is b.

```
instance Monad (Sel r e) where
  return x = Sel (λg → return (mempty, x))
  e >>= f = Sel $ λg → do
    (r1, a) ← unSel e (λa → (f a) ▹ g)
    (r2, b) ← unSel (f a) g
    return (r1<>r2, b)
```

4.3 Examples

Example: Greedy algorithms. A greedy selection strategy always picks the choice that maximizes (or minimizes) losses. We define the *Max* effect with a *max* operation; a corresponding handler can selects the element maximizing the loss, where *maxWith* implements argmax:

```
[effect| data Max = Max {max :: Op [a] a}]
hmax = handler Max {max = operation (λx l k → do b ← maxWith l x; k b)}
```

As an example, we can define criteria for selecting a *String* based on its length and the number of distinct characters, with greater losses (really, rewards) for better strings:

```
len x = loss (fromIntegral (length x))
distinct x = let i = fromIntegral (length (group (sort x))) in loss (i ∗ i)
```

where *sort* sorts the string, and the *group* function collects consecutive identical characters into separate lists. Thus, the number of groups is the number of distinct characters in the string.

We can then define a program *password* that picks a password from a list based on these criteria, where ++ concatenates two lists. Using *hmax* to handle *max*, *runSel* \$ *hmax password* returns `"password is abc"`, since `"abc"` has the greatest reward.

```
password = do
  s ← perform max ["aaa", "aabb", "abc"]
  len s
  distinct s
  return $ "password is " ++ s
```

Example: Optimizations. Greedy algorithms always pick the optimal option. However, it is not always possible to enumerate all possible choices and identify the best one.

We consider optimization algorithms, specifically *stochastic gradient descent* (SGD), a widely used method for iterative optimization. Starting with initial parameters, SGD minimizes a cost function by repeatedly updating the parameters in the direction opposite to their gradients, calculated after processing each randomly selected data point.

We implement gradient descent as a handler that chooses new parameters as follows, where the function *autodiff f x* calculates the gradient of a differentiable function f at point x. The *optimize* clause first calculates the gradient *ds* by differentiating the choice continuation l with respect to the parameters p. It then updates the parameters using *zipWith*, where 0.01 is the *learning rate*. Lastly, it resumes the continuation with updated parameters *p'*.

```
[effect| data Opt = Opt {optimize :: Op [Float] [Float]}]
hOpt = handler (Opt {optimize = operation (λp l k →
  do ds ← autodiff l p
     let p' = zipWith (λw d → (w − 0.01 ∗ d)) p ds
     k p')})
```

As a concrete example, we use the simplest form of *linear regression* [Legendre 1806] with only one variable, a standard example when explaining SGD. Given a dataset of $(x_i, y_i)_{i=1,..,n}$, where x_i and y_i are real numbers, the goal is to find a weight w and a bias b that minimize the cost function $\sum_{i=1,..n,}(f(x_i) - y_i)^2$ for the linear model $f(x) = wx + b$.

The program *linearReg* on the right defines a linear regression model. It takes the current parameters w and b (represented as a list) and a data point x and *target*, and returns new parameters w' and b'.

```
linearReg [w, b] x target =
  do [w', b'] ← perform optimize [w, b]
     let y = w' ∗ x + b'
     loss $ (target − y) ∗ (target − y)
     return [w', b']
```

The program first calls an operation *optimize* with the current parameters $[w, b]$ and receives updated parameters $[w', b']$. It then calculates the predicted value y using these new parameters and calls *loss* with the corresponding squared error. Finally, the program returns updated parameters.

We combine the *hOpt* handler and the *linearReg* program as follows.

```
foldM (λp (x, y) → lreset $ hOpt $ linearReg p x y) random_params training_data
```

The program traverses the training dataset, applying gradient descent to each data point. Note that we apply *lreset* that combines local and reset within the loop body, so each iteration makes decisions based on its own loss. Moreover, we can introduce a random effect to shuffle the training data, introducing stochasticity into the process.

Example: Hyperparameters. In the gradient descent handler, we used the learning rate 0.01. For training programs, variables such as the learning rate that govern the training process are called *hyperparameters*. The process of finding their optimal configuration is known as *hyperparameter optimization* [Feurer and Hutter 2019].

We can abstract the learning rate as a separate effect operation as shown on the right. A handler that always returns a pre-defined learning rate can be defined as:

```
[effect| data LR = LR {lrate :: Op () Float}]
gd = handler (Opt {optimize = operation (λp l k →
  do ds ← autodiff l p
     α ← perform lrate ()
     let p' = zipWith (λw d → (w − α ∗ d)) p ds
     k p')})
```

```
readLR α = handler (LR {lrate = operation (λx _ k → k α)})
```

More interestingly, a handler for hyperparameter tuning can compare losses from different configurations. As an example, the handler below implements a simple *grid search* that exhaustively explores a subset of the hyperparameter space, in this case, for simplicity, two options:

```
tuneLR (α₁, α₂) = handlerRet (λ_ → return α₁)
LR {lrate = operation (λ_ l _ → do err1 ← l α₁; err2 ← l α₂
                                   if err1 < err2 then return α₁ else return α₂)}
```

The handler calculates the losses for the two learning rates respectively, and returns the one with a lower loss, without resuming the computation.

Example: Two-player games. A minimax game corresponds to the philosophy of minimizing potential loss in a worst-case scenario. It involves two players: maximizer, who seeks to maximize the loss, and minimizer, who aims to minimize it (§2.1). As an example, consider a game with four final states. The losses associated with both player's decisions are shown in the table on the right.

A \ B	Left	Right
Left	5	3
Right	2	9

A corresponding minimizer handler can be defined as:

```
[effect| data Min a = Min {min :: Op [a] a}]
hmin = handler Min {min = operation (λx l k →
  do b ← minWith l x; k b)}
```

```
data Strategy = Left | Right deriving (Enum)
minimax = do
  a ← perform max [Left, Right]
  b ← perform min [Left, Right]
  loss $ [[5, 3], [2, 9]] !! (fromEnum a) !! (fromEnum b)
  return (a, b)
```

We can then play this simple game, where the maximizer A chooses over the minimizer B, as given on the right, where we encode the loss table as a nested list, and (!!) is list index operator. The program (*runSel* \$ *hmax* \$ *hmin minimax*) returns (*Left*, *Right*) with loss 3. This is because A maximizes over the choices of B. Specifically, B, the minimizer, chooses 3 (between 3 and 5), and 2 (between 2 and 9). Then A, the maximizer, chooses 3 (between 3 and 2). Note how the loss is shared by two handlers.

In the training domain, *generative adversarial networks* [Goodfellow et al. 2014] is also a two-player game. The algorithm simultaneously trains two models that contest with each other: the generative model learns to generate samples, while the discriminative model learns to distinguish between real and generated samples. More explicitly, it corresponds to $\min_G \max_D \ (\mathbb{E}_{x \sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z \sim p_{\text{noise}}}[\log(1 - D(G(z)))])$, where the discriminator is a minimizer and the generator is a maximizer.

Example: Nash equilibrium. In game theory, a *Nash equilibrium* describes a situation where no player can improve their outcome by unilaterally changing their strategy, assuming all other players maintain their current strategies.

A classic example is the *prisoner's dilemma*, illustrated in the table below. Here, the loss is represented as a pair, indicating the respective prison sentences for prisoner A and prisoner B. If both prisoners cooperate (by staying silent), they each serve one year (loss of 1). However, if one prisoner cooperates while the other defects, the cooperating prisoner serves 5 years, while the defecting prisoner goes free. If both defect, they each serve 3 years. In this scenario, defection always yields a better individual outcome regardless of the other prisoner's choice.

A \ B	defects	cooperates
defects	(3, 3)	(0, 5)
cooperates	(5, 0)	(1, 1)

We define game steps as follows, using *Strategy* for defection (*Left*) and cooperation (*Right*).

```
data Step = Move Strategy | Stay Strategy deriving (Eq)
[effect| data Play = Play {play :: Op (Step, Step) (Step, Step)}]
```

Given both players' strategies, the handler on the right tries to reduce one player's loss by adjusting their strategy while holding the other player's strategy unchanged. The function *getStrtgy* extracts the strategy from the current step, and *move* modifies the strategy of the specified player. Each player compares their own loss and decides whether to adjust their strategy or stay unchanged.

```
hNash = handler Play {play = operation (λ(a, b) l k → do
  let (a1, b1) = (getStrtgy a, getStrtgy b)
  let (a2, b2) = (move a1, move b1)
  l1 ← l (Stay a1, Stay b1); l2 ← l (Stay a2, Stay b1)
  l3 ← l (Stay a1, Stay b2)
  if (fst l2 < fst l1) then k (Move a2, Stay b1)
  else if (snd l3 < snd l1) then k (Stay a1, Move b2)
  else k (Stay a1, Stay b1))}
```

The *game* program below iteratively adjusts the players' strategies until both choose to *Stay*. This signifies that a Nash equilibrium has been reached, and the program terminates. The program *runSel* \$ *game* (*Move Right*) (*Move Right*) returns the strategies (*Stay Left*, *Stay Left*) through 2 steps, indicating that both prisoners defect. This outcome represents a Nash equilibrium, as neither prisoner can improve their individual outcome by unilaterally changing their strategy. It is easy to imagine an alternative handler that minimizes the total loss for both players. In that case, the game would return (*Stay Right*, *Stay Right*), which minimizes the combined loss.

```
game a b = do
  (a', b') ← lreset $ hNash $ do
    (a1, b1) ← perform play (a, b)
    let (a2, b2) = (getStrtgy a1, getStrtgy b1)
    loss $ [[(3, 3), (0, 5)], [(5, 0), (1, 1)]]
        !! (fromEnum a2) !! (fromEnum b2))
    return (a1, b1)
  if isStay a' && isStay b' then return (a, b)
  else lreset $ game a' b'
```

5 Denotational semantics

We next give λC a denotational semantics using a suitably augmented selection monad. We give soundness and adequacy theorems, thereby both showing that our computational ideas inspired by the selection monad are in fact in accord with it, and also providing a theoretical foundation for our combination of algebraic effect handlers and the selection monad.

5.1 Semantics of Types

As discussed in Section 2.1, our semantics employs a family $S_\epsilon(X) = (X \to R_\epsilon) \to W_\epsilon(X)$ of augmented selection monads where $W_\epsilon(X) = F_\epsilon(R \times X)$ and $R_\epsilon = F_\epsilon(R)$, with R the reals. The F_ϵ are used to interpret unhandled effect operations, and R_ϵ is the free W_ϵ-algebra on the one-point set. The W_ϵ are the commutative combination [Hyland et al. 2006] of the F_ϵ and the writer monad $R \times -$. Algebraically this choice of monad combination corresponds to the loss operation commuting with the other operations. Semantically it results in loss effects commuting with operation calls; via the Soundness Theorem 5.4, this is congruent with the operational semantics. Also, recalling the discussion on subtyping in §3.2, note that the S_ϵ are not effect-covariant, as the R_ϵ appear contravariantly.

We define $F_\epsilon(X)$ to be the least set Y such that:

$$Y = \left(\sum_{\ell \in \epsilon, op: out \xrightarrow{\ell} in, 0 < i \leqslant \epsilon(\ell)} \mathcal{S}[\![out]\!] \times Y^{\mathcal{S}[\![in]\!]} \right) + X$$

There is an inclusion $F_{\epsilon_1}(X) \subseteq F_\epsilon(X)$ if $\epsilon_1 \subseteq \epsilon$; which we use without specific comment. The elements of $F_\epsilon(X)$ can be thought of as *effect values* or *interaction trees*, much as in [Forster et al. 2017; Plotkin and Power 2001; Xia et al. 2019]. They are trees whose internal nodes are decorated with four things: an effect $\ell \in \epsilon$, an ℓ-operation $op : out \to in$, a handler execution depth index, and an element of $\mathcal{S}[\![out]\!]$. Nodes have successor nodes for each element of $\mathcal{S}[\![in]\!]$; and the leaves of the tree are decorated with elements of X. The idea is that such trees indicate possible computations in which various operations occur before finally yielding a value in X. Trees of this kind were used to give a monadic denotational semantics to an algebraic effect language in [Forster et al. 2017].

Note the circularity in these definitions: the F_ϵ are defined from the S_ϵ,and vice versa. However the effect levels strictly decrease in the first case (because of the well-foundedness assumption) and do not increase in the second (recall §3.4), justifying the definitions.

Given the S_ϵ we can define $\mathcal{S}[\![\sigma]\!]$ the semantics of types, as in Figure 8, where we assume available a given semantics $[\![b]\!]$ of basic types, including $\mathcal{S}[\![\textbf{loss}]\!] = R$.

$$\begin{array}{lcl} \mathcal{S}[\![b]\!] & = & [\![b]\!] \\ \mathcal{S}[\![(\sigma_1, \dots, \sigma_n)]\!] & = & \mathcal{S}[\![\sigma_1]\!] \times \dots \times \mathcal{S}[\![\sigma_n]\!] \\ \mathcal{S}[\![\sigma \to \tau \,!\, \epsilon]\!] & = & \mathcal{S}[\![\sigma]\!] \to S_\epsilon(\mathcal{S}[\![\tau]\!]) \end{array}$$

Fig. 8. Semantics of types

5.2 Monads

For our semantics we need the monadic structure of the S_ϵ, which is available via the free algebra structures of the W_ϵ and the F_ϵ. For the F_ϵ, say that an ϵ*-algebra* is a set X equipped with functions

$$\varphi_{\ell,op,i} : \mathcal{S}[\![out]\!] \times X^{\mathcal{S}[\![in]\!]} \to X \qquad (\ell \in \epsilon, op : out \xrightarrow{\ell} in, 0 < i \leqslant \epsilon(\ell))$$

Then $F_\epsilon(X)$ is the free such algebra taking the functions to be: $\varphi^X_{\ell,op,i}(o, k) =_{\text{def}} ((\ell, op, i), (o, k))$, with the unit at X being given by $\eta_{F_\epsilon}(x) = x$, ignoring injections into sums. If $(Y, \psi_{\ell,op,i})$ is another such algebra, the unique homomorphic extension $f^{\dagger_{F_\epsilon}}$ of a function $f : X \longrightarrow Y$ is given by setting

$$f^{\dagger_{W_\epsilon}}(x) = f(x) \qquad\qquad f^{\dagger_{F_\epsilon}}((\ell, op, i), (o, k)) = \psi_{\ell,op,i}(o, f^{\dagger_{F_\epsilon}} \circ k)$$

Turning to $W_\epsilon(X) = F_\epsilon(R \times X)$, we can again see this as a free algebra monad. Say that an *action ϵ-algebra* is an ϵ-algebra $(Y, \psi_{\ell,op,i})$ together with an additive action $\cdot : R \times Y \longrightarrow Y$ commuting with the $\psi_{\ell,\gamma,op,i}$ (by an additive action we mean one such that $0 \cdot y = y$ and $r \cdot (s \cdot y) = (r + s) \cdot y$). Then $F_\epsilon(R \times X)$ is the free such algebra with operations $\varphi^{R\times X}_{\ell,op,i}$ and action given by:

$$r \cdot u =_{\mathrm{def}} \mathrm{let}_{F_\epsilon}\ s \in R, x \in X \text{ be } u \text{ in } (r + s, x)$$

The unit is $\eta_{W_\epsilon}(x) = (0, x)$, and if $(Y, \psi_{\ell,op,i}, \cdot)$ is another such algebra, the unique homomorphic extension $f^{\dagger_{W_\epsilon}}$ of a function $f : X \longrightarrow Y$ is given by $f^{\dagger_{W_\epsilon}}(r, x) = r \cdot f(x)$ and:

$$f^{\dagger_{W_\epsilon}}((\ell, op, i), (o, k)) = \psi_{\ell,op,i}(o, f^{\dagger_{W_\epsilon}} \circ k)$$

Turning finally to the augmented selection monad $S_\epsilon(X) = (X \to R_\epsilon) \longrightarrow W_\epsilon(X)$. The unit η_{S_ϵ} at X is given by $\eta_{S_\epsilon}(x) = \lambda\gamma \in X \longrightarrow R_\epsilon.\, \eta_{W_\epsilon}(x)$ (and recall that $\eta_{W_\epsilon}(x) = (0, x)$). For the Kleisli extension, rather than follow the definitions in, e.g., Abadi and Plotkin [2021] via a W_ϵ-algebra on R_ϵ we give definitions that are a little easier to read.

First, R_ϵ is an action ϵ-algebra, with $\psi_{\ell,op,i} : \mathcal{S}[\![out]\!] \times R_\epsilon^{\mathcal{S}[\![in]\!]} \longrightarrow R_\epsilon$ given by $\psi_{\ell,op,i}(o, k) = \varphi^R_{\ell,op,i}(o, k)$ and action $R \times R_\epsilon \longrightarrow R_\epsilon$ given by: $r \cdot u =_{\mathrm{def}} \mathrm{let}_{F_{1,\epsilon}}\ s \in R$ be u in $r + s$. Next (using that R_ϵ is an action ϵ-algebra) the loss $\mathbf{R}_\epsilon(F|\gamma) \in R_\epsilon$ associated to $F \in S_\epsilon(Y)$ and loss function $\gamma : Y \longrightarrow R_\epsilon$ is

$$\mathbf{R}_\epsilon(F|\gamma) =_{\mathrm{def}} \gamma^{\dagger_{W_\epsilon}}(F(\gamma))$$

Then the Kleisli extension $f^{\dagger_{S_\epsilon}} : S_\epsilon(X) \longrightarrow S_\epsilon(Y)$ of a function $f : X \longrightarrow S_\epsilon(Y)$ is defined by:

$$f^{\dagger_{S_\epsilon}}(F) = \lambda\gamma \in Y \longrightarrow R_\epsilon.\, \mathrm{let}_{W_\epsilon}\ x \in X \text{ be } F(\lambda x \in X.\mathbf{R}_\epsilon(fx|\gamma)) \text{ in } fx\gamma \tag{5}$$

Finally $S_\epsilon(X)$ is an ϵ-algebra, with functions $\overline{\varphi}_{\ell,op,i} : \mathcal{S}[\![out]\!] \times S_\epsilon(X)^{\mathcal{S}[\![in]\!]} \longrightarrow S_\epsilon(X)_\epsilon$ given by:

$$\overline{\varphi}^X_{\ell,op,i}(o, f)(\gamma) = \varphi^{R\times X}_{\ell,op,i}(o, \lambda x \in in.\, f(x)(\gamma))$$

5.3 Semantics of Expressions and Handlers

Given an environment $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ we take $\mathcal{S}[\![\Gamma]\!]$ to be the functions (called *environments*) ρ on $\mathrm{Dom}(\Gamma)$ such that $\rho(x_i) \in \mathcal{S}[\![\sigma_i]\!]$, for $i = 1, \ldots, n$. In Figure 9 we give semantics to typed expressions, and handlers according to the following schemes:

$$\frac{\Gamma \vdash e : \sigma \,!\, \epsilon}{\mathcal{S}[\![e]\!] : \mathcal{S}[\![\Gamma]\!] \longrightarrow S_\epsilon(\mathcal{S}[\![\sigma]\!])} \qquad \frac{\Gamma \vdash h : par, \sigma \,!\, \epsilon\ell \Rightarrow \sigma' \,!\, \epsilon}{\mathcal{S}[\![h]\!] : \mathcal{S}[\![\Gamma]\!] \longrightarrow (\mathcal{S}[\![par]\!] \times S_{\epsilon\ell}(\mathcal{S}[\![\sigma]\!])) \longrightarrow S_\epsilon(\mathcal{S}[\![\sigma']\!])}$$

We make use of an abbreviation, available for any monad M:

$$\mathrm{let}_M\ x \in X \text{ be } exp_1 \text{ in } exp_2 =_{\mathrm{def}} (\lambda x \in X.\, exp_2)^{\dagger_M}(exp_1)$$

for mathematical expressions exp_1 and exp_2. This abbreviation makes monadic binding available at the meta-level, and that makes for more transparent formulas. We assume given semantics $[\![c]\!] \in [\![b]\!]$, for constants $c : b$, and $[\![f]\!] : [\![\sigma]\!] \longrightarrow [\![\tau]\!]$ for basic function symbols $f : \sigma \longrightarrow \tau$ (with $[\![r]\!] = r$ and $+$ denoting the addition of R). We also use an auxiliary "loss function" semantics. For $\Gamma, x : \sigma \vdash e : \mathbf{loss} \,!\, \epsilon$ we define $\mathcal{L}[\![\lambda^\epsilon x : \sigma.\, e]\!] : \mathcal{S}[\![\Gamma]\!] \longrightarrow \mathcal{S}[\![\sigma]\!] \longrightarrow R_\epsilon$ by:

$$\mathcal{L}[\![\lambda^\epsilon x : \sigma.\, e]\!](\rho) = \lambda a \in \mathcal{S}[\![\sigma]\!].\, \mathrm{let}_{F_\epsilon}\ r_1, r_2 \in R \text{ be } \mathcal{S}[\![e]\!](\rho[a/x])(\lambda r \in R.\, 0) \text{ in } r_2$$

The denotational semantics of expressions, up to application, is generic to any monadic semantics of call-by-value effectful languages. The semantics of operation calls uses the ϵ-algebraic structure of the S_ϵ. Other semantics read as denotational versions of operational computations. For example, the semantics of $e_1 \blacktriangleright \lambda^\epsilon x : \sigma_1.\, e_2$ ignores the current loss continuation, passes the value a of e_1 (with loss continuation the loss denotation of $\lambda^\epsilon x : \sigma_1.\, e_2$) to e_2, evaluates that (with zero loss continuation),

$$\boxed{\dfrac{\Gamma \vdash e : \sigma \,!\, \epsilon}{\mathcal{S}\llbracket e \rrbracket : \mathcal{S}\llbracket \Gamma \rrbracket \longrightarrow S_\epsilon(\mathcal{S}\llbracket \sigma \rrbracket)}} \qquad \textit{(Expressions)}$$

$$\begin{array}{lcl}
\mathcal{S}\llbracket x \rrbracket(\rho) & = & \eta_{S_\epsilon}(\rho(x)) \\
\mathcal{S}\llbracket c \rrbracket(\rho) & = & \eta_{S_\epsilon}(\llbracket c \rrbracket) \\
\mathcal{S}\llbracket f(e) \rrbracket(\rho) & = & \mathrm{let}_{S_\epsilon}\ a \in \mathcal{S}\llbracket \sigma_1 \rrbracket \text{ be } \mathcal{S}\llbracket e \rrbracket(\rho) \text{ in } \eta_{S_\epsilon}(\llbracket f \rrbracket(a)) \quad (f : \sigma_1 \to \sigma) \\
\mathcal{S}\llbracket (e_1, \ldots, e_n) \rrbracket(\rho) & = & \mathrm{let}_{S_\epsilon}\ a_1 \in \mathcal{S}\llbracket \sigma_1 \rrbracket \text{ be } \mathcal{S}\llbracket e_1 \rrbracket(\rho) \text{ in} \\
 & & \ldots \\
 & & \mathrm{let}_{S_\epsilon}\ a_n \in \mathcal{S}\llbracket \sigma_n \rrbracket \text{ be } \mathcal{S}\llbracket e_n \rrbracket(\rho) \text{ in} \\
 & & \eta_{S_\epsilon}((a_1, \ldots, a_n)) \qquad\qquad (\sigma = (\sigma_1, \ldots, \sigma_n)) \\
\mathcal{S}\llbracket e.i \rrbracket(\rho) & = & S_\epsilon(\pi_i)(\mathcal{S}\llbracket e \rrbracket(\rho)) \\
\mathcal{S}\llbracket \lambda^{\epsilon_1} x : \sigma.\, e \rrbracket(\rho) & = & \eta_{S_\epsilon}(\lambda a \in \mathcal{S}\llbracket \sigma \rrbracket.\, \mathcal{S}\llbracket e \rrbracket(\rho[a/x])) \\
\mathcal{S}\llbracket e_1\ e_2 \rrbracket(\rho) & = & \mathrm{let}_{S_\epsilon}\ \varphi \in \mathcal{S}\llbracket \sigma_1 \rrbracket \to S_\epsilon(\mathcal{S}\llbracket \sigma \rrbracket) \text{ be } \mathcal{S}\llbracket e_1 \rrbracket(\rho) \text{ in} \\
 & & \mathrm{let}_{S_\epsilon}\ a \in \mathcal{S}\llbracket \sigma_1 \rrbracket \text{ be } \mathcal{S}\llbracket e_2 \rrbracket(\rho) \text{ in } \varphi(a) \\
 & & \qquad\qquad (\Gamma \vdash e_1 : \sigma_1 \to \sigma \,!\, \epsilon) \\
\mathcal{S}\llbracket op(e) \rrbracket(\rho) & = & \mathrm{let}_{S_\epsilon}\ a \in \mathcal{S}\llbracket out \rrbracket \text{ be } \mathcal{S}\llbracket e \rrbracket(\rho) \text{ in } \overline{\varphi}^{X}_{\ell, op, \epsilon(\ell)}(a, (\eta_{S_\epsilon})_{\mathcal{S}\llbracket in \rrbracket}) \\
 & & \qquad\qquad (op : out \xrightarrow{\ell} in, \sigma = in) \\
\mathcal{S}\llbracket \mathbf{with}\ h\ \mathbf{from}\ e_1\ \mathbf{handle}\ e_2 \rrbracket(\rho) & = & \mathrm{let}_{S_\epsilon}\ a \in \mathcal{S}\llbracket par \rrbracket \text{ be } \mathcal{S}\llbracket e_1 \rrbracket(\rho) \text{ in } \mathcal{S}\llbracket h \rrbracket(\rho)(a, \mathcal{S}\llbracket e_2 \rrbracket(\rho)) \\
 & & \qquad\qquad (\Gamma \vdash e_1 : par) \\
\mathcal{S}\llbracket \mathbf{loss}(e) \rrbracket(\rho) & = & \lambda \gamma \in R_\epsilon^{\mathcal{S}\llbracket \sigma \rrbracket}.\, \mathrm{let}_{F_\epsilon}\ r \in R, a \in R \text{ be } \mathcal{S}\llbracket e \rrbracket(\rho)(\gamma) \text{ in } (a + r, ()) \\
\mathcal{S}\llbracket e_1 \blacktriangleright \lambda^{\epsilon_1} x : \sigma_1.\, e_2 \rrbracket(\rho) & = & \lambda \gamma \in R_\epsilon^{\mathcal{S}\llbracket \sigma \rrbracket}. \\
 & & \quad \mathrm{let}_{F_\epsilon}\ r_1 \in R, a \in \mathcal{S}\llbracket \sigma_1 \rrbracket \text{ be } \mathcal{S}\llbracket e_1 \rrbracket(\rho)(\mathcal{L}\llbracket \lambda^\epsilon x : \sigma_1.\, e_2 \rrbracket(\rho)) \text{ in} \\
 & & \quad \mathrm{let}_{F_{\epsilon_1}}\ r_2, r_3 \in R \text{ be } \mathcal{S}\llbracket e_2 \rrbracket(\rho[a/x])(\lambda r \in R.\, 0) \text{ in } (r_2, r_1 + r_3) \\
\mathcal{S}\llbracket \langle e \rangle^{\epsilon_1}_{\mathrm{g}} \rrbracket(\rho) & = & \lambda \gamma \in R_\epsilon^{\mathcal{S}\llbracket \sigma \rrbracket}.\, \mathcal{S}\llbracket e \rrbracket(\rho)\mathcal{L}\llbracket \mathrm{g} \rrbracket(\rho) \\
\mathcal{S}\llbracket \mathbf{reset}\, e \rrbracket(\rho) & = & \lambda \gamma \in R_\epsilon^{\mathcal{S}\llbracket \sigma \rrbracket}.\, \mathrm{let}_{F_\epsilon}\ r_1 \in R, a \in \mathcal{S}\llbracket \sigma \rrbracket \text{ be } \mathcal{S}\llbracket e \rrbracket(\rho)(\gamma) \text{ in } \eta_{W_\epsilon}(a)
\end{array}$$

Fig. 9. Semantics of expressions

and adds the loss r_1 of e_1 to the result r_3, keeping the resulting loss r_2. The sub-effecting in the typing rule THEN is here reflected in the semantic inclusion $F_{\epsilon_1}(X) \subseteq F_\epsilon(X)$.

Semantics of Handlers. We build up the semantics of handlers in stages. Here is the high-level idea. Ignoring environments and parameters, we seek a semantics: $\mathcal{S}\llbracket h \rrbracket : S_{\epsilon\ell}(\mathcal{S}\llbracket \sigma \rrbracket) \to S_\epsilon(\mathcal{S}\llbracket \sigma' \rrbracket)$, equivalently $\mathcal{S}\llbracket h \rrbracket \gamma G \in W_\epsilon(\mathcal{S}\llbracket \sigma' \rrbracket)$ for $\gamma : \mathcal{S}\llbracket \sigma' \rrbracket \to R_\epsilon$, and $G \in S_{\epsilon\ell}(\mathcal{S}\llbracket \sigma \rrbracket)$. Following the standard approach to the semantics of handlers [Pretnar and Plotkin 2013] we exploit a free algebra property, here that of of $F_{\epsilon\ell}(R \times \mathcal{S}\llbracket \sigma \rrbracket)$, constructing an $\epsilon\ell$-algebra on $F_\epsilon(R \times \mathcal{S}\llbracket \sigma' \rrbracket)$ using h's operation definitions (it may not be an action one), then obtaining a homomorphism to it from $F_{\epsilon\ell}(R \times \mathcal{S}\llbracket \sigma \rrbracket)$, and finally applying that to $G\gamma'$, with γ' obtained from γ using h's return function.

So, consider a handler h:

$$\left\{\begin{array}{l}
op_1 \mapsto \lambda^\epsilon z : (par, out_1, (par, in_1) \to \mathbf{loss} \,!\, \epsilon, (par, in_1) \to \sigma' \,!\, \epsilon).\, e_1, \ldots, \\
op_n \mapsto \lambda^\epsilon z : (par, out_n, (par, in_n) \to \mathbf{loss} \,!\, \epsilon, (par, in_n) \to \sigma' \,!\, \epsilon).\, e_n, \\
\mathbf{return} \mapsto \lambda^\epsilon z : (par, \sigma).\, e_r
\end{array}\right\}$$

where $\Gamma \vdash h : par, \sigma \,!\, \epsilon\ell \Rightarrow \sigma' \,!\, \epsilon$, and fix $\rho \in \mathcal{S}\llbracket \Gamma \rrbracket$ and $\gamma \in R_\epsilon^{\mathcal{S}}\llbracket \sigma' \rrbracket$. We first construct an $\epsilon\ell$-algebra on $A = W_\epsilon(\mathcal{S}\llbracket \sigma' \rrbracket)^{\mathcal{S}\llbracket par \rrbracket}$. So for $\ell_1 \in \epsilon\ell$, $op : out \xrightarrow{\ell_1} in$, and $0 < i \leqslant (\epsilon\ell)\ell_1$ we need functions $\psi_{\ell, op, i} : \mathcal{S}\llbracket out \rrbracket \times A^{\mathcal{S}\llbracket in \rrbracket} \to A$. For $\ell_1 \in \epsilon$, $op : out \xrightarrow{\ell_1} in$, and $0 < i \leqslant \epsilon(\ell_1)$, we set

$$\psi_{\ell_1, op, i}(o, k) = \lambda p \in \mathcal{S}\llbracket par \rrbracket.\, ((\ell_1, op, i), (o, \lambda a \in \mathcal{S}\llbracket in \rrbracket.\, kap))$$

and, for op_j and $i = \epsilon(\ell) + 1$ we set

$$\psi_{\ell,op_j,i}(o,k) = \lambda p \in \mathcal{S}[\![par]\!].\,\mathcal{S}[\![e_j]\!](\rho[(p,o,l_1,k_1)/z])\gamma$$

where $k_1(p,a) = kap$ and $l_1(p,a) = \lambda\gamma_1 \in R_\epsilon^{\mathcal{S}[\![\sigma']\!]}.\,\delta_\epsilon(\gamma^{\dagger W_\epsilon}(kap))$. (in the definition of l_1 we use the fact that R_ϵ is an action ϵ-algebra, and $\delta_\epsilon : F_\epsilon(R) \to F_\epsilon(R \times R)$ is $F_\epsilon(\lambda r \in R.\,(0,r))$).

We use this algebra to extend the map $s : R \times \mathcal{S}[\![\sigma]\!] \longrightarrow A$ defined by

$$s(r,a) = \lambda p \in \mathcal{S}[\![par]\!].\,r \cdot (\mathcal{S}[\![e_r]\!](\rho[(p,a)/z])\gamma)$$

(Recall that **return** $\mapsto \lambda^\epsilon z : (par, \sigma).\,e_r$ is in h.) The semantics of the handler h is then given by:

$$\mathcal{S}[\![h]\!](\rho)(p,G)(\gamma) = s^{\dagger F_{\epsilon\ell}}(G(\lambda a \in \mathcal{S}[\![\sigma]\!].\,\mathbf{R}_\epsilon(\mathcal{S}[\![e]\!](\rho[(p,a)/z])|\gamma)))(p)$$

5.4 Soundness and Adequacy of Operational Semantics

Below we may omit ρ in $\mathcal{S}[\![e]\!](\rho)$ (or $\mathcal{V}[\![v]\!](\rho)$) when e (respectively v) is closed. In Fig. 10 we define a "value semantics" $\mathcal{V}[\![v]\!] : \mathcal{S}[\![\Gamma]\!] \longrightarrow \mathcal{S}[\![\sigma]\!]$ for values $\Gamma \vdash v : \sigma$. It helps us to state our soundness and adequacy results.

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\mathcal{V}[\![x]\!](\rho) & = & \rho(x) & \mathcal{V}[\![(v_1,\ldots,v_n)]\!](\rho) & = & (\mathcal{V}[\![v_1]\!](\rho),\ldots,\mathcal{V}[\![v_n]\!](\rho)) \\
\mathcal{V}[\![c]\!](\rho) & = & [\![c]\!] & \mathcal{V}[\![\lambda^{\epsilon_1}x : \sigma_1.e]\!](\rho) & = & \lambda a \in \mathcal{S}[\![\sigma]\!].\,\mathcal{S}[\![e]\!](\rho[a/x])
\end{array}$$

Fig. 10. Semantics of values

Lemma 5.1. *For any value* $\Gamma \vdash v : \sigma\,!\,\epsilon$ *we have:* $\mathcal{S}[\![v]\!](\rho) = \eta_{S_\epsilon}(\mathcal{V}[\![v]\!](\rho))$.

As terminal expressions can be stuck we also need a lemma for their semantics:

Lemma 5.2. *For terminal* $\Gamma \vdash K[op(v)] : \sigma\,!\,\epsilon$ *with* $op : out \xrightarrow{\ell} in$ *we have:*

$$\mathcal{S}[\![K[op(v)]]\!](\rho) = \lambda\gamma.\,\varphi_{\ell,op,\epsilon(\ell)}^{R\times\mathcal{S}[\![\sigma]\!]}(\mathcal{V}[\![v]\!](\rho), \lambda a \in \mathcal{S}[\![in]\!].\,\mathcal{S}[\![K[x]]\!](\rho[a/x])(\gamma))$$

For soundness, we assume that the semantics of basic functions is sound w.r.t. the operational semantics, i.e. $f(v) \to v' \implies [\![f]\!]([\![v]\!]) = [\![v']\!]$. We have small-step soundness:

Theorem 5.3 (Small-step Soundness). *Suppose* $e : \sigma\,!\,\epsilon$ *and* $g : \sigma \to \mathbf{loss}\,!\,\epsilon_1$ *with* $\epsilon_1 \subseteq \epsilon$. *Then:*

$$g \vdash_\epsilon e \xrightarrow{r} e' \quad\implies\quad \mathcal{S}[\![e]\!]\mathcal{L}[\![g]\!] = r \cdot (\mathcal{S}[\![e']\!]\mathcal{L}[\![g]\!])$$

and that implies evaluation (big-step) soundness:

Theorem 5.4 (Evaluation soundness). *For all* $e : \sigma\,!\,\epsilon$ *and* $g : \sigma \to \mathbf{loss}\,!\,\epsilon'$ *with* $\epsilon' \subseteq \epsilon$ *we have:*

(1) *If* $g \vdash_\epsilon e \overset{r}{\Rightarrow} v$ *then* $\mathcal{S}[\![e]\!]\mathcal{L}[\![g]\!] = (r, \mathcal{V}[\![v]\!])$.

(2) *If* $g \vdash_\epsilon e \overset{r}{\Rightarrow} K[op(v)]$ *then* $\mathcal{S}[\![e]\!]\mathcal{L}[\![g]\!] = r \cdot \mathcal{S}[\![K[op(v)]]\!]\mathcal{L}[\![g]\!]$

Combining soundness and termination (Theorem 3.5) we obtain adequacy:

Theorem 5.5 (Adequacy). *For all* $e : \sigma\,!\,\epsilon$ *and* $g : \sigma \to \mathbf{loss}\,!\,\epsilon'$ *with* $\epsilon' \subseteq \epsilon$ *we have:*

(1) *If* $\mathcal{S}[\![e]\!]\mathcal{L}[\![g]\!] = (r,a)$ *then, for some* v, $g \vdash_\epsilon e \overset{r}{\Rightarrow} v$ *and* $\mathcal{V}[\![v]\!] = a$.

(2) *If* $\mathcal{S}[\![e]\!]\mathcal{L}[\![g]\!] = \varphi_{\ell,op,\epsilon(\ell)}^{R\times\mathcal{S}[\![\sigma]\!]}(a,f)$ *then, for some* $K[op(v)]$, $g \vdash_\epsilon e \overset{r}{\Rightarrow} K[op(v)]$, $a = \mathcal{V}[\![v]\!]$, *and* $f = \lambda b \in \mathcal{S}[\![in]\!].\,r \cdot \mathcal{S}[\![K[x]]\!](x \mapsto b)\mathcal{L}[\![g]\!]$.

As usual, if σ is first-order and the denotation map $[\![c]\!]$ of constants is 1-1, we have the corollary:

$$g \vdash_\epsilon e \overset{r}{\Rightarrow} v \quad\Longleftrightarrow\quad \mathcal{S}[\![e]\!]\mathcal{L}[\![g]\!] = (r, \mathcal{V}[\![v]\!])$$

Fixing σ and ϵ, set $E = \{e : \sigma \,!\, \epsilon\}$, and let EV, the set of *effect values*, be the least set such that:

$$\mathrm{EV} \quad = \quad \sum_{\ell \in \epsilon, op: out \xrightarrow{\ell} in} V_{out} \times \mathrm{EV}^{V_{in}} \;+\; (R \times V_\sigma)$$

where, for any τ, $V_\tau =_{\mathrm{def}} \{v | v : \tau\}$. Following [Plotkin 2009; Plotkin and Power 2001], we evaluate expressions as far as effect values. Fixing $g : \sigma \longrightarrow \mathbf{loss} \,!\, \epsilon'$ (with $\epsilon' \subseteq \epsilon$) define (using the evident R-action on EV) a *giant step* evaluation function Eval : E $\rightharpoonup$ EV (shown total via computability) by:

$$\mathrm{Eval}(e) = \begin{cases} (r, v) & (g \vdash_\epsilon e \overset{r}{\Rightarrow} v) \\ ((\ell, op), (v, \lambda w \in \mathrm{V}_{in}.\, r \cdot \mathrm{Eval}(K[w]))) & (g \vdash_\epsilon e \overset{r}{\Rightarrow} K[op(v)], op : out \xrightarrow{\ell} in) \end{cases}$$

Next let $\preceq$ be the least relation between EV and $W_\epsilon(\mathcal{S}[\![\sigma]\!])$ such that (1) $(r, v) \preceq (r, \mathcal{V}[\![v]\!])$ and (2)$((\ell, op), (v, f)) \preceq ((\ell, op, \epsilon(\ell)), (\mathcal{V}[\![v]\!], g))$ if $\forall w \in V_{in}. f(w) \preceq g(\mathcal{V}[\![w]\!])$.

Theorem 5.6 (Giant Step Adequacy). *For all $e : \sigma \,!\, \epsilon$ we have:* $\mathrm{Eval}(e) \preceq \mathcal{S}[\![e]\!]\mathcal{L}[\![g]\!]$.

6 Related Work and Conclusion

Our work may be the first advocating a language design based on effect handlers and the selection monad. It is closest to Abadi and Plotkin [2021, 2023] who used an argmax selection function to make their choices. As they themselves said, this is unreasonable when there is no access to optimal strategies. Further, neither handlers nor choice continuations were provided (though they did suggest trying monadic reflection and reification [Filinski 1994]). Lago et al. [2022] proposed using effect handlers for *reinforcement learning* (RL) [Sutton and Barto 2018], but did not support choice continuations. Basic RL (e.g., *multi-armed bandits* as in Lago et al. [2022]), does not need choice continuations as action losses are directly given, and can be transmitted to a user-defined loss effect (and ordinary state effects can be used to represent learner's states). More sophisticated RL algorithms benefit from choice continuations, e.g., *deep reinforcement learning* [Riedmiller 2005] where policies are approximated by neural networks, and so need gradient descent.

There are several directions for future work. First, we are interested in improving the performance of the selection monad. Specifically, the choice continuation shares expressions with the delimited continuation, (though this need not lead to recomputations). For instance, in the gradient descent handler *hOpt* in §4.3, l is differentiated with respect to the current parameter, and k is resumed with the updated parameter. In broader scenarios, we expect that further program transformations and advanced compiler optimizations (e.g., memoization) will mitigate recomputations. Moreover, a more efficient approach to jointly make nested choices is described in Hartmann et al. [2024], using a generalized form of selection monad. We would also like to integrate our design into existing languages and frameworks. e.g., JAX [Bradbury et al. 2018], a functional programming DSL popular for large-scale ML tasks (see [Piponi 2022]). Interesting too are frameworks providing choices for users, such as Carbune et al. [2019]. There are several interesting possibilities for advancing our framework: adding recursive functions or iteration; adding effect polymorphism as in Leijen [2017]; obtaining subeffecting using effect inclusions $\epsilon' \subseteq \epsilon$ to type expressions yielding ϵ results from ϵ'-continuations; and allowing users to locally vary the reward monoid (e.g., to a product with independent localising constructs, facilitating multi-objective optimization).

Acknowledgments

We thank Martín Abadi, Adam Paszke, Dimitrios Vytiniotis, and Dan Zheng for helpful discussion, and the reviewers for their helpful comments.

References

Martín Abadi and Gordon Plotkin. 2021. Smart choices and the selection monad. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science* (Rome, Italy) *(LICS '21)*. Association for Computing Machinery, New York, NY, USA, Article 81, 14 pages. doi:10.1109/LICS52264.2021.9470641

Martín Abadi and Gordon Plotkin. 2023. Smart choices and the selection monad. *Logical Methods in Computer Science* 19 (2023). doi:10.46298/lmcs-19(2:3)2023

Martín Abadi and Gordon D. Plotkin. 2019. A Simple Differentiable Programming Language. *Proc. ACM Program. Lang.* 4, POPL, Article 38 (dec 2019), 28 pages. doi:10.1145/3371106

Mario Alvarez-Picallo, Teodoro Freund, Dan R. Ghica, and Sam Lindley. 2024. Effect Handlers for C via Coroutines. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 358 (Oct. 2024), 28 pages. doi:10.1145/3689798

Andrej Bauer and Matija Pretnar. 2013. An effect system for algebraic effects and handlers. In *Algebra and Coalgebra in Computer Science: 5th International Conference, CALCO 2013, Warsaw, Poland, September 3-6, 2013. Proceedings 5*. Springer, 1–16. doi:10.1007/978-3-642-40206-7_1

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming* 84, 1 (2015), 108–123. doi:10.1016/j.jlamp.2014.02.001

Nick Benton, John Hughes, and Eugenio Moggi. 2000. Monads and effects. In *International Summer School on Applied Semantics*. Springer, 42–122. doi:10.1007/3-540-45699-6_2

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax

Victor Carbune, Thierry Coppey, Alexander Daryin, Thomas Deselaers, Nikhil Sarda, and Jay Yagnik. 2019. SmartChoices: Hybridizing Programming and Machine Learning. doi:10.48550/ARXIV.1810.00619

R. Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. 2007. A monadic framework for delimited continuations. *Journal of Functional Programming* 17, 6 (2007), 687–730. doi:10.1017/S0956796807006259

Martin Escardó and Paulo Oliva. 2010a. Selection functions, bar recursion and backward induction. *Mathematical Structures in Computer Science* 20, 2 (2010), 127–168. doi:10.1017/S0960129509990351

Martin Escardó and Paulo Oliva. 2011. Sequential games and optimal strategies. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 467, 2130 (2011), 1519–1545. doi:10.1098/rspa.2010.0471

Martin Escardó and Paulo Oliva. 2015. The Herbrand Functional Interpretation of the Double Negation Shift. arXiv:1410.4353 [cs.LO] https://arxiv.org/abs/1410.4353

Martín Hötzel Escardó and Paulo Oliva. 2010b. Computational Interpretations of Analysis via Products of Selection Functions.. In *CiE*. Springer, 141–150. doi:10.1007/978-3-642-13962-8_16

Martín Hötzel Escardó and Paulo Oliva. 2010c. The Peirce Translation and the Double Negation Shift. In *Programs, Proofs, Processes*, Fernando Ferreira, Benedikt Löwe, Elvira Mayordomo, and Luís Mendes Gomes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–161. doi:10.1007/978-3-642-13962-8_17

Martín Hötzel Escardó and Paulo Oliva. 2010d. What Sequential Games, the Tychonoff Theorem and the Double-Negation Shift Have in Common. In *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming* (Baltimore, Maryland, USA) *(MSFP '10)*. Association for Computing Machinery, New York, NY, USA, 21–32. doi:10.1145/1863597.1863605

Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical computer science* 103, 2 (1992), 235–271. doi:10.1016/0304-3975(92)90014-7

Matthias Feurer and Frank Hutter. 2019. Hyperparameter optimization. *Automated machine learning: Methods, systems, challenges* (2019), 3–33. doi:10.1007/978-3-030-05318-5_1

Andrzej Filinski. 1994. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 446–457. doi:10.1145/174675.178047

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

Dan Ghica, Sam Lindley, Marcos Maroñas Bravo, and Maciej Piróg. 2022. High-level effect handlers in C++. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 183 (Oct. 2022), 29 pages. doi:10.1145/3563445

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger (Eds.), Vol. 27. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2014/file/f033ed80deb0234979a61f95710dbe25-Paper.pdf

Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press, Cambridge, MA, USA. http://www.deeplearningbook.org.

Johannes Hartmann and Jeremy Gibbons. 2022. Algorithm design with the selection monad. In *International Symposium on Trends in Functional Programming*. Springer, 126–143. doi:10.1007/978-3-031-21314-4_7

Johannes Hartmann, Tom Schrijvers, and Jeremy Gibbons. 2024. Towards a more efficient Selection Monad. *Trends in Functional Programming, Proceedings* (2024). doi:10.1007/978-3-031-74558-4_3

Jules Hedges. 2015. The selection monad as a CPS transformation. arXiv:1503.06061 [cs.PL] https://arxiv.org/abs/1503.06061

Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development* (Nara, Japan) *(TyDe 2016)*. Association for Computing Machinery, New York, NY, USA, 15–27. doi:10.1145/2976022.2976033

Martin Hyland, Gordon Plotkin, and John Power. 2006. Combining effects: Sum and tensor. *Theoretical Computer Science* 357, 1 (2006), 70–99. doi:10.1016/j.tcs.2006.03.013 Clifford Lectures and the Mathematical Foundations of Programming Semantics.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 145–158. doi:10.1145/2500365.2500590

Wiktor Kuchta. 2022. *Normalisation for Algebraic Effect Handlers*. Master's thesis. University of Wroclaw. Available at https://github.com/wikku/normalization-effect-handlers/blob/main/fscd-term.pdf.

Ugo Dal Lago, Francesco Gavazzo, and Alexis Ghyselen. 2022. On Reinforcement Learning, Effect Handlers, and the State Monad. arXiv:2203.15426 [cs.PL] https://arxiv.org/abs/2203.15426

Adrien Marie Legendre. 1806. *Nouvelles méthodes pour la détermination des orbites des comètes: avec un supplément contenant divers perfectionnemens de ces méthodes et leur application aux deux comètes de 1805*. Courcier.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS, Vol. 153)*, Paul Blain Levy and Neel Krishnaswami (Eds.). 100–126. doi:10.4204/EPTCS.153.8

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 486–499. doi:10.1145/3009837.3009872

Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 238 (Oct. 2023), 26 pages. doi:10.1145/3622814

Dan Piponi. 2022. https://colab.sandbox.google.com/drive/1HGs59anVC2AOsmt7C4v8yD6v8gZSJGm6

Gordon Plotkin. 2009. Adequacy for infinitary algebraic effects. In *Algebra and Coalgebra in Computer Science*, Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–2. doi:10.1007/978-3-642-03741-2_1

Gordon Plotkin and John Power. 2001. Adequacy for algebraic effects. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 1–24. doi:10.1007/3-540-45315-6_1

Gordon Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In *Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 18*. Springer, 80–94. doi:10.1007/978-3-642-00590-9_7

Gordon Plotkin and Ningning Xie. 2025. Handling the Selection Monad (Full Version). arXiv:2504.03890 [cs.PL] https://arxiv.org/abs/2504.03890

Matija Pretnar. 2015. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic notes in theoretical computer science* 319 (2015), 19–35. doi:10.1016/j.entcs.2015.12.003

Matija Pretnar and Gordon D Plotkin. 2013. Handling algebraic effects. *Logical methods in computer science* 9 (2013). doi:10.2168/LMCS-9(4:23)2013

Martin Riedmiller. 2005. Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method. In *Machine Learning: ECML 2005: 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005. Proceedings 16*. Springer, 317–328. doi:10.1007/11564096_32

Sebastian Ruder. 2017. An overview of gradient descent optimization algorithms. arXiv:1609.04747 [cs.LG] https://arxiv.org/abs/1609.04747

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 206–221. doi:10.1145/3453483.3454039

Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

William W Tait. 1967. Intensional interpretations of functionals of finite type I. *The journal of symbolic logic* 32, 2 (1967), 198–212. doi:10.2307/2271658

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (Dec. 2019), 32 pages. doi:10.1145/3371119

Ningning Xie and Daan Leijen. 2021. Generalized Evidence Passing for Effect Handlers: Efficient Compilation of Effect Handlers to C. *Proc. ACM Program. Lang.* 5, ICFP, Article 71 (aug 2021), 30 pages. doi:10.1145/3473576

Received 2024-11-15; accepted 2025-03-06