Effect Handlers for Choice-Based Learning

Gordon Plotkin & Ningning Xie
Google DeepMind

1 Introduction

Machine learning (ML) has achieved many remarkable advances and successes in numerous areas. However, it is difficult for programmers to maintain, reuse, and extend ML programs [25, 3, 2]. In particular, ML programs often come with a wide range of configurable options, including *hyperparameters* [14], different optimization methods [27], or variants of ML models (eg *reinforcement learning* (RL) [28] vs *deep RL* [4]). As a result, programmers often need to define a family of programs. Currently they do this in an ad-hoc way, causing a significant amount of code duplication [29].

Here, we study *ML programming* from a language design perspective. We propose a new *choice-based learning* paradigm which provides a separation of concerns that fosters modularity. On the one hand, programmers define training models with choices and losses; this includes widely-used decision-making models and techniques eg *Markov decision processes*, with actions as choices. On the other hand, they *separately* implement optimization strategies. In contrast, existing learning systems based on decision-making models, eg Spiral [20], SmartChoices [7, 15], come with pre-determined interfaces and built-in learning techniques; so choices cannot be made using user-provided composable language primitives.

The key insight underlying our design of choice-based learning is to combine two programming techniques: *algebraic effects and handlers*, and *loss continuations*. Algebraic effects [21, 22] and handlers [23, 24] provide a flexible mechanism for modular programming with user-defined effects [8, 26, 19, 5]. They achieve modularity by separating user-defined effect operations from their implementation by effect handlers (which may call further effects). Semantically, loss continuations are functions $\gamma \in R^X$ where R is a set of losses. Computations for $x \in X$ are then *selection* functions ie $F \in S(X) =_{\mathrm{def}} R^X \to X$, the *selection monad*. Example such F's are argmax, choosing an x maximising γ, or optimization functions choosing x using γ (eg gradient descent). This monad [12] has been used in such areas as game theory, proof theory, and decision-making models [9, 10, 11, 1]. It can be combined with auxiliary monads T to account for additional effects, yielding $S_T(X) = R^X \to T(X)$ [13].

We implement choice-based learning by combining effect handlers with loss continuations and a loss effect operation, as considered in Section 2 and illustrated in Section 3. The loss operation enables programmers to register losses; choice effect operations enable them to write training models; handlers now have access to the loss continuations, enabling them to define choice operations using optimizations. Semantically, as briefly described below, this corresponds to the monad S_{W_ϵ} where W_ϵ combines the writer monad with one for (as yet) uninterpreted operations. Our design provides a flexible and modular interface for ML programming, as multiple choices and different optimization strategies can be easily nested and combined.

2 Theory

To illustrate our ideas, omitting many details, we present C, a small first-order core language with handlers h and loss continuations l; types σ are basic, b, or products, $(\sigma_1, \ldots, \sigma_n)$.

$$\begin{array}{lcl} e & ::= & x \mid c \mid f(e_1, ..., e_n) \mid \mathbf{let}\ x\!:\!\sigma\ =\ e\ \mathbf{in}\ e \mid (e_1, \ldots, e_n) \mid e.i \\ & & \mid op(e) \mid \mathbf{with}\ h\ \mathbf{from}\ e\ \mathbf{handle}\ e \mid \mathbf{loss}(e) \mid \mathbf{reset}\ e \mid \langle\!\langle e \rangle\!\rangle \end{array}$$

$$h \quad ::= \quad \left\{ \begin{array}{l} \ldots, op_i(p\!:\!par, x\!:\!out_i, l\!:\!(par, in_i) \to \mathbf{real}\,!\,\epsilon, k\!:\!(par, in_i) \to \sigma'\,!\,\epsilon) \mapsto e_i, \ldots \\ \mathbf{return}(p\!:\!par, x\!:\!\sigma) \mapsto e \end{array} \right\}$$

Losses are incurred using the loss construct $\mathbf{loss}(e)$. As in, eg [18], handlers h (invoked using **with**) define all the operations of an effect label ℓ; the difference is that, as well as the usual delimited continuations k, operations have access to loss continuations l; these can be used to perform optimizations. The $\mathbf{reset}\, e$ and $\langle\!\langle e \rangle\!\rangle$ constructs are used to localize effects: $\mathbf{reset}\, e$ resets the loss to 0 and $\langle\!\langle e \rangle\!\rangle$ executes e with the zero loss continuation. Expressions are effect-typed as $\Gamma \vdash e : \sigma \,!\, \epsilon$, where the effects ϵ are multisets of effect labels.

Given the selection monad origins of this work it is natural to seek a denotational semantics. So, for such expressions e there is a selection monadic semantics $\mathcal{S}\,[\![e]\!]\,(\rho) \in S_{W_\epsilon}([\![\sigma]\!])$. Here W_ϵ is the commutative combination of the writer monad W and the free algebra monad F_ϵ for the ϵ-effect label operations, counted with suitable multiplicities (by [17] $W_\epsilon(X) = F_\epsilon(R \times X)$).

C can be compiled into standard algebraic effect handler languages; in this way we can take advantage of existing effect handler implementations. One such target language is T, with expressions and handlers exactly as above, but without the $\mathbf{loss}(e)$, $\mathbf{reset}\, e$, or $\langle\!\langle e \rangle\!\rangle$ constructs, and where operations do not have access to loss continuations. In order to have a syntax for loss continuations we add a syntactic category of abstractions $ab = \lambda x\!:\!\sigma.\, e$. T also has a monadic semantics, now using F_ϵ. The compiler translates source code C expressions nf in ANF (A-Normal Form) to T target code $\mathcal{T}_\sigma(\mathrm{nf}, ab)$, given a loss continuation ab. (We do not detail ANFs.) Here are some specimen cases for ANF expressions and ANF handlers nh (ie handlers as above, but with ANF operation bodies and return clause):

$$
\begin{array}{lcl}
\mathcal{T}_\sigma((x_1,...,x_n),ab) & = & (0,(x_1,...,x_n)) \\
\mathcal{T}_\sigma(op(x),ab) & = & (0,op(x)) \\
\mathcal{T}_\sigma(\mathbf{with}\,\mathrm{nh}\,\mathbf{from}\,p\,\mathbf{handle}\,\mathrm{nf},ab) & = & \mathbf{with}\,\mathcal{T}(\mathrm{nh},ab)\,\mathbf{from}\,p\,\mathbf{handle}\,\mathcal{T}_\sigma(\mathrm{nf},0_{\sigma_1}) \\
 & & \quad(\mathbf{return}(p\!:\!par,x\!:\!\sigma_1)\mapsto \mathrm{nf}_1 \text{ in } h) \\
\mathcal{T}_\sigma(\mathbf{loss}(x),ab) & = & (x,()) \\
\mathcal{T}_\sigma(\mathbf{reset}\,\mathrm{nf},ab) & = & (0,\mathcal{T}_\sigma(\mathrm{nf},ab).2) \\
\mathcal{T}_\sigma(\langle\!\langle \mathrm{nf}\rangle\!\rangle,ab) & = & \mathcal{T}_\sigma(\mathrm{nf},0_\sigma) \\
\mathcal{T}_\sigma(\mathbf{let}\;x\!:\!\sigma_1\;=\;\mathrm{at}\;\mathbf{in}\;\mathrm{nf},ab) & = & \mathbf{wlet}_\sigma\;x\!:\!\sigma_1\;=\;\mathcal{T}_{\sigma_1}(\mathrm{at},\lambda x\!:\!\sigma_1.\,\mathcal{T}_\sigma(\mathrm{nf},ab)\;\mathbf{wthen}\;ab)\;\mathbf{in}\;\mathcal{T}_\sigma(\mathrm{nf},ab)
\end{array}
$$

$$
\mathcal{T}(\mathrm{nh},ab)\quad=\quad\left\{\begin{array}{l}
\ldots,op_i(p\!:\!par,x\!:\!out_i\;k\!:\!(par,in_i)\to(\mathbf{real},\sigma')\,!\,\epsilon) \\
\qquad\mapsto\mathcal{T}_{\sigma'}(\mathrm{nf}_i,0_{\sigma'})[\lambda(p'\!:\!par,y\!:\!in_i).\,k(p',y)\;\mathbf{wthen}\;g/l],\ldots \\
\mathbf{return}(p\!:\!par,x\!:\!(\mathbf{real},\sigma)\mapsto\mathbf{wlet}_{\sigma'}\;x\!:\!\sigma\;=\;x\;\mathbf{in}\;\mathcal{T}_{\sigma'}(\mathrm{nf},ab)
\end{array}\right\}
$$

Here **wlet** simulates W_ϵ-binding, **wthen** is used to construct complex loss continuations from simpler ones, and 0_σ is the zero loss continuation $\lambda x:\sigma.\,0$. Note how the loss continuation needed for the handler translation is constructed from the delimited one k and the available global one ab. The translation is *correct* w.r.t. the semantics: for $\vdash \mathrm{nf}:\sigma\,!\,\epsilon$ we have:

$$\mathcal{S}\,[\![\mathrm{nf}]\!]\,(\lambda x\in[\![\sigma]\!]\,.\,0)=\mathcal{F}\,[\![\mathcal{T}_\sigma(\mathrm{nf},0_\sigma)]\!]$$

3 Practice

We provide an implementation of our design as an effect handler library in Haskell. In this article we briefly present two examples to demonstrate the design; interested reader may refer to the appendix for more examples and detailed explanations.

Following our design, the training program can be written as an effectful computation using the choose and loss operations, while the optimization algorithm is implemented as an effect handler for the choose operation that makes use of losses. As an example, the code below on the left defines the training program for linear regression using our library.

```
[effect | data Choose = Choose {choose :: Op [Param] [Param]}]

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

gradDesc = handler Choose {choose =
  operation (λws lk k → do
    ds ← autodiff lk ws
    let ws' = zipWith (λw d → (w − 0.01 ∗ d)) ws ds
    k ws')}
```

The effect *Choose* has an operation *choose* that takes the current parameters and returns new ones, where a parameter *Param* is a datatype used for automatic differentiation. The program *linearReg* defines the linear regression model, taking the current weight and bias $[w, b]$ and a data point x and y. It first performs *choose* to get new parameters, then calculates the model and the loss, before returning the new parameters. As this example shows, with choice-based learning, the system needs to associate each choice with its resulting loss. Notably, the program does not specify how the new parameters are chosen. Instead, we must write a handler for handling the *choose* operation. The *gradDesc* handler on the right defines how *choose* is handled. Inside the handler, we first differentiate the loss continuation using *autodiff* (whose definition is omitted), getting the gradients *ds*. We then do gradient descent by returning, essentially, $(ws - 0.01 * ds)$, where 0.01 is the *learning rate*. Finally, we resume with the new parameters. By combining the training program with the handler, we can get a complete definition of linear regression.

By separating training and optimization, we provide a modular interface where different optimization algorithms can be easily reused, nested, and composed. For example, we can use the interface to implement *Generative adversarial networks* (GAN) [16], a prominent framework for generative AI. Its key idea is to simultaneously train two models that contest with each other: a *generative model* that takes noise and learns to generate samples, and a *discriminative model* that evaluates samples and estimates the probability that a sample comes from a real data distribution rather than the generative distribution. GAN is an interesting example for our framework, as it corresponds nicely to two handlers for the same loss

```
hGenerator = gradDesc   -- gradient descent
hDiscriminator = handler DChoose {dChoose = operation (λws lk k → ...)}   -- gradient ascent

gan sample noise = hDiscriminator α1 $ hGenerator α2 $ do ...
```

In the future, we would like to integrate our design into existing machine learning frameworks such as JAX [6] which has built-in mechanisms for (effect-free) AD and parallelism, and apply the integrated framework to large-scale applications. We would also like to investigate AD for handler languages.

References

[1] Martín Abadi and Gordon D. Plotkin. Smart choices and the selection monad. In *Proceedings of the Thirty sixth Annual IEEE Symposium on Logic in Computer Science (LICS 2021)*, pages 1–14. IEEE Computer Society Press, June 2021.

[2] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[3] Anders Arpteg, Björn Brinne, Luka Crnkovic-Friis, and Jan Bosch. Software engineering challenges of deep learning. In *2018 44th euromicro conference on software engineering and advanced applications (SEAA)*, pages 50–59. IEEE, 2018.

[4] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

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

[6] 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. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

[7] Victor Carbune, Thierry Coppey, Alexander Daryin, Thomas Deselaers, Nikhil Sarda, and Jay Yagnik. Smartchoices: Hybridizing programming and machine learning, 2019. URL `https://arxiv.org/abs/1810.00619`.

[8] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. Concurrent system programming with effect handlers. In *Trends in Functional Programming: 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers 18*, pages 98–117. Springer, 2018.

[9] Martín Hötzel Escardó and Paulo Oliva. The peirce translation and the double negation shift. In Fernando Ferreira, Benedikt Löwe, Elvira Mayordomo, and Luís Mendes Gomes, editors, *Programs, Proofs, Processes*, pages 151–161, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-13962-8.

[10] Martín Hötzel Escardó and Paulo Oliva. Computational interpretations of analysis via products of selection functions. In *CiE*, pages 141–150. Springer, 2010.

[11] Martín Hötzel Escardó and Paulo Oliva. 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*, MSFP '10, page 21–32, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450302555. doi: 10.1145/1863597.1863605. URL `https://doi.org/10.1145/1863597.1863605`.

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

[13] Martín Hötzel Escardó and Paulo Oliva. The herbrand functional interpretation of the double negation shift. *The Journal of Symbolic Logic*, 82(2):590–607, 2017.

[14] Matthias Feurer and Frank Hutter. Hyperparameter optimization. *Automated machine learning: Methods, systems, challenges*, pages 3–33, 2019.

[15] Daniel Golovin, Gabor Bartok, Eric Chen, Emily Donahue, Tzu-Kuo Huang, Efi Kokiopoulou, Ruoyan Qin, Nikhil Sarda, Justin Sybrandt, and Vincent Tjeng. Smartchoices: Augmenting software with learned implementations. *arXiv preprint arXiv:2304.13033*, 2023.

[16] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

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

[18] Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 486–499, 2017.

[19] Daan Leijen. Structured asynchrony with algebraic effects. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, pages 16–29, 2017.

[20] Meta. Spiral: Self-tuning services via real-time machine learning, 2018. URL `https://engineering.fb.com/2018/06/28/data-infrastructure/spiral-self-tuning-services-via-real-time-machine`

[21] Gordon Plotkin and John Power. Adequacy for algebraic effects. In *Foundations of Software Science and Computation Structures: 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001 Proceedings 4*, pages 1–24. Springer, 2001.

[22] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied categorical structures*, 11:69–94, 2003.

[23] Gordon Plotkin and Matija Pretnar. 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*, pages 80–94. Springer, 2009.

[24] Matija Pretnar and Gordon D Plotkin. Handling algebraic effects. *Logical methods in computer science*, 9, 2013.

[25] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. *Advances in neural information processing systems*, 28, 2015.

[26] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto ocaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 206–221, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454039. URL `https://doi.org/10.1145/3453483.3454039`.

[27] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. A survey of optimization methods from a machine learning perspective. *IEEE transactions on cybernetics*, 50(8):3668–3681, 2019.

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

[29] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. An empirical study of refactorings and technical debt in machine learning systems. In *2021 IEEE/ACM 43rd international conference on software engineering (ICSE)*, pages 238–250. IEEE, 2021.