Partial Dijkstra Monads for All

Théo Winterhalter[1], Cezar-Constantin Andrici[1], Cătălin Hriţcu[1], Kenji Maillard[2], Guido Martínez[3], and Exequiel Rivas[4]

[1] MPI-SP [2] Inria Rennes [3] CIFASIS-CONICET and UNR Argentina [4] TUT

Abstract

Dijkstra Monads for All introduces a generic method to construct a Dijkstra monad from a monad morphism between a computation and a specification monad. However, applying this construction to usual computation monads yields Dijkstra monads that do not support partiality, which makes them unusable in F*. We show that this issue can be overcome when the computation and specification monads support partiality by providing a way to *require* pre-conditions, and we provide several techniques to build such monads.

Dijkstra monads are indexed monad structures that are used in F* for verifying effectful programs [SHK+16, SWS+13]. Concretely, a Dijkstra monad $D\ A\ w$ represents an effectful computation returning values of type A and obeying specification $w : W\ A$, where W is a specification monad. For instance, the state Dijkstra monad ST is usually specified by the monad $\mathsf{W}^{\mathsf{ST}}\ A = (A \times S \to \mathbb{P}) \to (S \to \mathbb{P})$, where $\mathbb{P}$ is the type of propositions. W^{ST} is the type of a predicate transformer taking a post-condition on the final state and a result value and returning a pre-condition on the initial state. We have the following Dijkstra monad interface:

$$\begin{array}{lcl@{\qquad}lcl}
\mathbf{return}^{\mathsf{ST}}\ (x : A) & : & \mathsf{ST}\ A\ (\mathbf{return}^{\mathsf{W}}\ x) & \mathbf{return}^{\mathsf{W}} & = & \lambda p\ s_0.\ p\ (x, s_0) \\
\mathbf{get}^{\mathsf{ST}}\ () & : & \mathsf{ST}\ S\ \mathbf{get}^{\mathsf{W}} & \mathbf{get}^{\mathsf{W}} & = & \lambda p\ s_0.\ p\ (s_0, s_0) \\
\mathbf{put}^{\mathsf{ST}}\ (s : S) & : & \mathsf{ST}\ \mathbf{unit}\ (\mathbf{put}^{\mathsf{W}}\ s) & \mathbf{put}^{\mathsf{W}} & = & \lambda p\ s_0.\ p\ ((), s)
\end{array}$$

$$\mathbf{bind}^{\mathsf{ST}}\ (c : \mathsf{ST}\ A\ w_c)\ (f : (x : A) \to \mathsf{ST}\ B\ (w_f\ x)) : \mathsf{ST}\ B\ (\mathbf{bind}^{\mathsf{W}}\ w_c\ w_f)$$
$$\mathbf{bind}^{\mathsf{W}}\ w_c\ w_f = \lambda p\ s_0.\ w_c\ (\lambda (x, s_1).\ w_f\ x\ p\ s_1)\ s_0$$

If we take a post-condition $p : A \times S \to \mathbb{P}$, we say it holds on program $\mathbf{return}^{\mathsf{ST}}\ x$ if we can prove $\mathbf{return}^{\mathsf{W}}\ x\ p$ on the initial state s_0 or in other words if $p\ (x, s_0)$ holds. For p to hold on $\mathbf{get}^{\mathsf{ST}}\ ()$ then it must hold on return value and final state both equal to the initial state: $p\ (s_0, s_0)$. For p to hold on $\mathbf{put}^{\mathsf{ST}}\ s$ it must hold on final state s and trivial unit value $()$: $p\ ((), s)$; the initial state is erased so it is ignored. Such typed Dijkstra monad interfaces allow F* to compute verification conditions simply by dependent type inference.

Constructing Dijkstra monads. *Dijkstra Monads for All* (DM4All) [MAA+19] introduces a generic way to construct Dijkstra monads. For any computation monad M, and for any ordered specification monad W with order $\leq^{\mathsf{W}}$, if there is a monad morphism $\theta : M \to W$ then one can define the following Dijkstra monad:

$$D\ A\ w = \{c : M\ A \mid \theta\ c \leq^{\mathsf{W}} w\} \tag{1}$$

For instance, from the usual state computation monad $\mathsf{State}\ A = S \to A \times S$ to the W^{ST} specification monad above one can define the monad morphism $\theta\ c = \lambda p\ s_0.\ p\ (c\ s_0)$. Another example is non-determinism, where we can take $M\ A := \mathbf{list}\ A$ as computation monad, $W\ A = (A \to \mathbb{P}) \to \mathbb{P}$ as specification monad, and $\theta^{\forall}\ c = \lambda p.\ (\forall x \in c.\ p\ x)$ as monad morphism, essentially saying that any post-condition should hold for all values stored in the list, or in other words for every possible outcome of the computation. This gives a demonic interpretation of non-determinism, and one can also chose an angelic interpretation by using $\theta^{\exists}\ c = \lambda p.\ (\exists x \in c.\ p\ x)$ instead. This construction of Dijkstra monads neatly separates the syntax (M) from the specification (W) and semantics (θ) as one can *forget* the refinement and extract the value in $M\ A$ from $D\ A\ w$. While very general, the DM4All construction often produces Dijkstra monads that do not support partiality on standard computational monads, which makes them unusable in F*, as explained below.

F* and the partiality effect. The PURE effect (i.e., Dijkstra monad) of F* represents in fact *partial* computations, as for instance one can use the pre-condition to discard provably unreachable branches of a pattern-matching, and recursive functions can loop on arguments not satisfying the pre-condition. We can model this notion of partiality in a more standard dependent type theory via a require construct with the following type:

$$\textbf{require}\ (p : \mathbb{P}) : \mathsf{PURE}\ p\ (\lambda q.\ (\exists (h : p).\ q\ h))$$

It returns a proof of the proposition p that can then be used by the continuation. The specification requires p as a pre-condition (the $\exists(h : p)$ part) and also asks for the post-condition (q) to hold on the proof of p. We argue that the existence of such an operator is tantamount to supporting partiality. Concretely, we will say that a monad M supports partiality when there is $\textbf{require}^{\mathsf{M}}\ (p : \mathbb{P}) : M\ p$ and that a Dijkstra monad D supports partiality when its specification monad does too and we have $\textbf{require}^{\mathsf{D}}\ (p : \mathbb{P}) : D\ p\ (\textbf{require}^{\mathsf{W}}\ p)$.

In F*, one can define such a **require** in PURE and because F* expects to be able to lift computations in PURE to any other Dijkstra monad, then such Dijkstra monads should also support a **require** operation.

Partial Dijkstra monads for all. As we pointed out above, computations c in $D\ A\ w$ obtained by DM4All (1) can be coerced to type $M\ A$, by just forgetting the $\theta\ c \leq^{\mathsf{W}} w$ refinement. This means that in order for D to support partiality, the underlying computation monad M should already support partiality. Yet most computation monads do not. For instance, for the state monad, **require** p would need to have type $\mathsf{State}\ p = S \to p \times S$, which one cannot inhabit for an arbitrary $p : \mathbb{P}$.

We show that the DM4All construction can be made to produce partial Dijkstra monads—thus usable in F*—when both the monads M and W additionally support a **require** construct such that $\theta\ (\textbf{require}^{\mathsf{M}}\ p) \leq^{\mathsf{W}} \textbf{require}^{\mathsf{W}}\ p$.

We provide several ways to build computation (and specification) monads that support a **require** construct. First, we provide an account of *Dijkstra monads for free* (DM4Free) [AHM+17] that fits in this setting. Basically, DM4Free produces a partial Dijkstra monad from a computation monad obtained by applying a monad transformer T to the partiality monad $\mathsf{G}\ A = \sum(p : \mathbb{P}).\ (p \to A)$ and the specification monad obtained by applying T to the continuation monad $\mathsf{W}^{\mathsf{Cont}}\ A = (A \to \mathbb{P}) \to \mathbb{P}$. This confirms the empirical observation that DM4Free yields Dijkstra monads that are usable in F*. Second, we provide a construction for adding an extra **require** constructor to the signature of a free monad, allowing for deep occurrences of **require** within computations. Together these cases cover many usual effects such as I/O, non-determinism, state, unrecoverable exceptions, etc. We prove formally in Coq that the DM4All construction with **require** yields partial Dijkstra monads and we include examples of the constructions above.[1] We are also investigating how to adapt interaction trees [XZH+19] to support partiality for potentially non-terminating computations in the style of *Dijkstra monads forever* [SZ21].

[1] https://github.com/TheoWinterhalter/pdm4all/releases/tag/types2022

References

[AHM+17] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 515–529, 2017.

[MAA+19] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *PACMPL*, 3(ICFP):104:1–104:29, 2019.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270, 2016.

[SWS+13] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398, 2013.

[SZ21] Lucas Silver and Steve Zdancewic. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2021.

[XZH+19] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.