Partial Dijkstra Monads for All

Théo Winterhalter\textsuperscript{1}, Cezar-Constantin Andrici\textsuperscript{1}, Cătălin Hriţcu\textsuperscript{1}, Kenji Maillard\textsuperscript{2}, Guido Martínez\textsuperscript{3}, and Exequiel Rivas\textsuperscript{4}

\textsuperscript{1}MPI-SP \quad \textsuperscript{2}Inria Rennes \quad \textsuperscript{3}CIFASIS-CONICET and UNR Argentina \quad \textsuperscript{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\textsuperscript{*}. We show that this issue can be overcome when the computation and specification monads support partiality by providing a way to \textit{require} pre-conditions, and we provide several techniques to build such monads.

Dijkstra monads are indexed monad structures that are used in F\textsuperscript{*} for verifying effectful programs [SHK\textsuperscript{+16}, SWS\textsuperscript{+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 $W^{ST} A = (A \times S \to P) \to (S \to P)$, where $P$ 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:

- $\text{return}^{ST} (x : A) : ST A (\text{return}^{W} x) \quad \text{return}^{W} = \lambda p s_0. p (x, s_0)$
- $\text{get}^{ST} () : ST S \text{get}^{W} \quad \text{get}^{W} = \lambda p s_0. p (s_0, s_0)$
- $\text{put}^{ST} (s : S) : ST \text{unit} (\text{put}^{W} s) \quad \text{put}^{W} = \lambda p s_0. p ((), s)$
- $\text{bind}^{ST} (c : ST A w_c) (f : (x : A) \to ST B (w_f x)) : ST B (\text{bind}^{W} w_c w_f)$

If we take a post-condition $p : A \times S \to P$, we say it holds on program $\text{return}^{ST} x$ if we can prove $\text{return}^{W} x p$ on the initial state $s_0$ or in other words if $p (x, s_0)$ holds. For $p$ to hold on $\text{get}^{ST} ()$ 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 $\text{put}^{ST} s$ 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\textsuperscript{*} to compute verification conditions simply by dependent type inference.

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

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

For instance, from the usual state computation monad $\text{State} A = S \to A \times S$ 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 $M A := \text{list} A$ as computation monad, $W A = (A \to P) \to P$ as specification monad, and $\theta^{W} c = \lambda p. (\forall x \in c. p x)$ 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
\( \theta^\exists \) instead. This construction of Dijkstra monads neatly separates the
syntax \((M)\) from the specification \((W)\) and semantics \((\theta)\) 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 \( \text{PURE} \) 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:

\[
\text{require } (p : \mathcal{P}) : \text{PURE } p (\lambda q. (\exists (h : p). q h))
\]

It returns a proof of the proposition \( p \) that can then be used by the continuation. The specifi-
cation 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
\( \text{require}^M (p : \mathcal{P}) : M p \) and that a Dijkstra monad \( D \) supports partiality when its specification
monad does too and we have \( \text{require}^D (p : \mathcal{P}) : D p \) (\( \text{require}^W \) \( p \)).

In F*, one can define such a require in \( \text{PURE} \) and because F* expects to be able to lift
computations in \( \text{PURE} \) 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 \( \theta c \leq^W w \) 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, \( \text{require} \) \( p \) would need to have type \( \text{State} \ p = S \to p \times S \), which one cannot
inhabit for an arbitrary \( p : \mathcal{P} \).

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 \( \theta (\text{require}^M \ p) \leq^W \text{require}^W \ p \).

We provide several ways to build computation (and specification) monads that sup-
port 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 par-
tiality monad \( G A = \sum (p : \mathcal{P}). (p \to A) \) and the specification monad obtained by applying \( T \) to
the continuation monad \( W^\text{Cont} A = (A \to \mathcal{P}) \to \mathcal{P} \). 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 occur-
cences 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


