Dijkstra Monads for Free

Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy

Microsoft Research, University of Edinburgh, Inria, ENS Paris, UNR Argentina

POPL ’17
Combining dependent types and effects

- Known hard problem, various solutions (Ynot/HTT, Idris, Trellys/Zombie, F\(^*\))
Combining dependent types and effects

- Known hard problem, various solutions (Ynot/HTT, Idris, Trellys/Zombie, F*)
- Common approach: encapsulating effectful programs in monads. But how to reason about them?
Combining dependent types and effects

- Known hard problem, various solutions (Ynot/HTT, Idris, Trellys/Zombie, F*)
- Common approach: encapsulating effectful programs in monads. But how to reason about them?
- One idea (HTT/F*) is to index the monad with a specification:

  \[
  \text{val incr : unit } \rightarrow \text{ST unit}
  \]

\[
\text{val incr : unit } \rightarrow \text{ST unit (requires } (\lambda n_0 \rightarrow \text{True}))
\]
\[
(\text{ensures } (\lambda n_0 r n_1 \rightarrow n_1 = n_0 + 1))
\]
Combining dependent types and effects

- Known hard problem, various solutions (Ynot/HTT, Idris, Trellys/Zombie, F*)
- Common approach: encapsulating effectful programs in monads. But how to reason about them?
- One idea (HTT/F*) is to index the monad with a specification:

  (*) No spec *)
  val incr : unit \rightarrow ST unit

  (*) Hoare triples *)
  val incr : unit \rightarrow ST unit (requires (\lambda n_0 \rightarrow True))
                   (ensures (\lambda n_0 r n_1 \rightarrow n_1 = n_0 + 1))

  (*) Dijkstra’s WPs *)
  val incr : unit \rightarrow ST unit (\lambda post n_0 \rightarrow post () (n_0 + 1))
Combining dependent types and effects

- Known hard problem, various solutions (Ynot/HTT, Idris, Trellys/Zombie, F*)
- Common approach: encapsulating effectful programs in monads. But how to reason about them?
- One idea (HTT/F*) is to index the monad with a specification:

  \[(*) \text{No spec} (*)\]
  \[
  \text{val} \ \text{incr} : \text{unit} \rightarrow \text{ST unit}
  \]

  \[(*) \text{Hoare triples} (*)\]
  \[
  \text{val} \ \text{incr} : \text{unit} \rightarrow \text{ST unit} \left( \text{requires} \ (\lambda \ n_0 \rightarrow \text{True}) \right)
  \left( \text{ensures} \ (\lambda \ n_0 \ r \ n_1 \rightarrow n_1 = n_0 + 1) \right)
  \]

  \[(*) \text{Dijkstra’s WPs} (*)\]
  \[
  \text{val} \ \text{incr} : \text{unit} \rightarrow \text{ST unit} \left( \lambda \ \text{post} \ n_0 \rightarrow \text{post} () \ (n_0 + 1) \right)
  \]

- Dijkstra monads are a generalization of Dijkstra’s predicate transformers to arbitrary effects, and are the bread and butter of F*’s reasoning about effects.
Dijkstra Monad
(pure and beautiful)

correctly specifies

Programs
(with dirty effects)
Dijkstra Monad
(pure and beautiful)
correctly specifies

Programs
(with dirty effects)
Problem...

- The Dijkstra monad for each effect needs to be hand-crafted, and proven correct.
- This made F\(^\star\) rigid, in that it had a fixed supply of effects.
The Dijkstra monad for each effect needs to be hand-crafted, and proven correct.
This made F* rigid, in that it had a fixed supply of effects.
A fundamental question arises:

**What is the relation between the monadic representation for an effect and its Dijkstra monad?**
The Dijkstra monad for each effect needs to be hand-crafted, and proven correct.

This made F* rigid, in that it had a fixed supply of effects.

A fundamental question arises:

What is the relation between the monadic representation for an effect and its Dijkstra monad?

Old dog, new trick: Dijkstra monads are a CPS transform of the representation monad, allowing automatic derivation.

Simple monadic definition gives correct-by-construction WP calculus for it.
The Dijkstra monad for each effect needs to be hand-crafted, and proven correct.

This made F* rigid, in that it had a fixed supply of effects.

A fundamental question arises:

What is the relation between the monadic representation for an effect and its Dijkstra monad?

Old dog, new trick: Dijkstra monads are a CPS transform of the representation monad, allowing automatic derivation.

Simple monadic definition gives correct-by-construction WP calculus for it.

Implemented in F*... now with user-defined effects.

Huge boost in simplicity and expressiveness of the effect system.
The Dijkstra monad for each effect needs to be hand-crafted, and proven correct.

This made $F^\star$ rigid, in the sense of having a fixed supply of effects.

A fundamental question arises: What is the relation between the monadic representation for an effect and its Dijkstra monad?

Old dog, new trick: Dijkstra monads are a CPS transform of the representation monad, allowing automatic derivation.

Simple monadic definition gives correct-by-construction WP calculus for it.

Implemented in $F^\star$... now with user-defined effects.

Huge boost in simplicity and expressiveness of the effect system.
A reminder on WPs

- Dijkstra monads are essentially monads over **weakest-preconditions** (WP).
- A WP is a **predicate transformer** mapping a postcondition on the outputs of a computation to a precondition on its inputs.
A reminder on WPs

- Dijkstra monads are essentially monads over weakest-preconditions (WP).
- A WP is a **predicate transformer** mapping a postcondition on the outputs of a computation to a precondition on its inputs.
- Example: for stateful computations, WPs are of type

\[ \text{ST}_{wp} \ t = (t \rightarrow S \rightarrow \text{Type}_0) \rightarrow S \rightarrow \text{Type}_0 \]

where \( t \) is the result type.
A reminder on WPs

- Dijkstra monads are essentially monads over **weakest-preconditions** (WP).
- A WP is a **predicate transformer** mapping a **postcondition** on the outputs of a computation to a **precondition** on its inputs.
- Example: for stateful computations, WPs are of type

\[
\text{ST}_{wp} \ t = \ (t \to S \to \text{Type}_0) \to S \to \text{Type}_0
\]

where \( t \) is the result type.
A reminder on WPs

- Dijkstra monads are essentially monads over **weakest-preconditions** (WP).
- A WP is a **predicate transformer** mapping a **postcondition** on the outputs of a computation to a **precondition** on its inputs.
- Example: for stateful computations, WPs are of type

\[
\text{ST}_{wp} \ t = (t \rightarrow S \rightarrow \text{Type}_0) \rightarrow S \rightarrow \text{Type}_0
\]

where \( t \) is the result type.
- \( \text{F*}'s \) typing judgment gives a WP to each computation:

\[
\Gamma \vdash e : \text{ST} \ t \ wp
\]
let incr () = let n = get () in put (n + 1)
Verifying code

\[
\text{let incr () = bind}_{st} (\text{get} ()) (\lambda n \to \text{put} (n + 1))
\]

- Turn it into explicitly monadic form
Verifying code

```plaintext
let incr () = bind_{st} (get ()) (\n \rightarrow put (n + 1))
```

- Turn it into explicitly monadic form
- Compute a WP by simple type inference

```plaintext
val get : unit \rightarrow ST int getwp
val put : n_1 : int \rightarrow ST unit (setwp n_1)
val bind_{st} : \forall wa wb. ST a wa \rightarrow (x:a \rightarrow ST b (wb x)) \rightarrow ST b (bindwp_{st} wa wb)
```
Verifying code

let incr () = bind \( st \) (get ()) (\( \lambda n \rightarrow put (n + 1) \))

- Turn it into explicitly monadic form
- Compute a WP by simple type inference

```plaintext
val get : unit \( \rightarrow \) \( ST \) int getwp
val put : \( n_1 : \) int \( \rightarrow \) \( ST \) unit (setwp \( n_1 \))
val bind_{st} : \( \forall \) wa wb. \( ST \) a wa \( \rightarrow \) (\( x : a \rightarrow ST \) b (wb x)) \( \rightarrow \) ST b (bindwp_{st} wa wb)
```

to get

```plaintext
val incr : unit \( \rightarrow \) \( ST \) unit (bindwp_{st} getwp (\( \lambda n \rightarrow setwp (n + 1) \)))
```
Verifying code

let incr () = bind\(_{st}\) (get ()) (\(\lambda\) \(n\) \(\rightarrow\) put (\(n + 1\)))

- Turn it into explicitly monadic form
- Compute a WP by simple type inference

\[
\begin{align*}
\text{val get : unit} & \rightarrow \text{ST int getwp} \\
\text{val put : n_1:int} & \rightarrow \text{ST unit ( setwp n_1)} \\
\text{val bind}_{st} : \forall wa wb. \text{ST a wa} & \rightarrow (x:a \rightarrow \text{ST b (wb x)}) \rightarrow \text{ST b ( bindwp}_{st} \text{ wa wb)}
\end{align*}
\]

to get

\[
\begin{align*}
\text{val incr : unit} & \rightarrow \text{ST unit (bindwp}_{st} \text{ getwp (\(\lambda\) \(n\) \(\rightarrow\) setwp (\(n + 1\))))} \\
= \text{val incr : unit} & \rightarrow \text{ST unit (\(\lambda\) post n_0 \(\rightarrow\) post () (n_0 + 1))}
\end{align*}
\]
Verifying code

let incr () = bind \( st \) (get ()) (\( \lambda n \rightarrow \text{put} (n + 1) \))

- Turn it into explicitly monadic form
- Compute a WP by simple type inference

val get : unit \( \rightarrow \text{ST} \) \( \text{int} \)

val put : \( n_1 : \text{int} \) \( \rightarrow \text{ST} \) \( \text{unit} \) \( (\text{setwp} \ n_1) \)

val \( \text{bind}_{st} \) : \( \forall \text{wa wb. ST} \) \( \text{a wa} \rightarrow (x:a \rightarrow \text{ST} \ b (\text{wb} \ x)) \rightarrow \text{ST} \ b \) \( (\text{bindwp}_{st} \ \text{wa wb}) \)

to get

val incr : unit \( \rightarrow \text{ST} \) \( \text{unit} \) \( (\text{bindwp}_{st} \ \text{getwp} \ (\lambda n \rightarrow \text{setwp} (n + 1))) \)

= val incr : unit \( \rightarrow \text{ST} \) \( \text{unit} \) \( (\lambda \text{post} \ n_0 \rightarrow \text{post} () (n_0 + 1)) \)
Primitive specs

\[
\begin{align*}
\text{ST}_{wp} t & = (t \rightarrow S \rightarrow \text{Type}_0) \rightarrow S \rightarrow \text{Type}_0 \\
\text{returnwp}_{st} v & = \lambda p\ s_0.\ p\ v\ s_0 \\
\text{bindwp}_{st} wp f & = \lambda p\ s_0.\ wp\ (\lambda v\ s_1.\ f\ v\ p\ s_1)\ s_0 \\
\text{getwp}_{st} & = \lambda p\ s_0.\ p\ s_0\ s_0 \\
\text{setwp}_{st} s_1 & = \lambda p\ _\ .\ p\ ()\ s_1
\end{align*}
\]
Primitive specs

\[
\begin{align*}
\text{ST}_{wp} t & = S \rightarrow (t \times S \rightarrow \text{Type}_0) \rightarrow \text{Type}_0 \\
\text{returnwp}_{st} v & = \lambda s_0 \ p. \ p \ (v, s_0) \\
\text{bindwp}_{st} wp f & = \lambda s_0 \ p. \ wp \ s_0 \ (\lambda vs. \ f (\text{fst \ vs}) (\text{snd \ vs}) \ p) \\
\text{getwp}_{st} & = \lambda s_0 \ p. \ p \ (s_0, s_0) \\
\text{setwp}_{st} s_1 & = \lambda p. \ p \ ((), s_1)
\end{align*}
\]
Primitive specs

\[
\begin{align*}
ST_{wp} t &= S \to (t \times S \to \text{Type}_0) \to \text{Type}_0 \\
\text{return}_{wp}{}^{st} v &= \lambda s_0. p. p (v, s_0) \\
\text{bind}_{wp}{}^{st} wp f &= \lambda s_0. p. wp s_0 (\lambda vs. f (\text{fst} vs) (\text{snd} vs) p) \\
\text{get}_{wp}{}^{st} &= \lambda s_0. p. p (s_0, s_0) \\
\text{set}_{wp}{}^{st} s_1 &= \lambda_. p. p ((), s_1) \\
\end{align*}
\]

\[
\begin{align*}
ST t &= S \to t \times S \\
\text{return}_{st} v &= \lambda s_0. (v, s_0) \\
\text{bind}_{st} m f &= \lambda s_0. \text{let} vs = m s_0 \text{ in } f (\text{fst} vs) (\text{snd} vs) \\
\text{get} &= \lambda s_0. (s_0, s_0) \\
\text{set} s_1 &= \lambda_. ((), s_1) \\
\end{align*}
\]
Primitive specs

ST\_wp t = \( S \rightarrow (t \times S \rightarrow \text{Type}_0) \rightarrow \text{Type} \)

return\_wp\_st v = \( \lambda s_0. \ p. \ p \ (v, s_0) \)

bind\_wp\_st wp f = \( \lambda s_0. \ p. \ wp \ s_0 \ (\lambda vs. \ f (\text{fst} \ vs) \ (s_0, s_0)) \)

get\_wp\_st = \( \lambda s_0. \ p. \ p \ (s_0, s_0) \)

set\_wp\_st s_1 = \( \lambda \_ p. \ p \ ((), s_1) \)

ST t = \( S \rightarrow t \times S \)

return\_st v = \( \lambda s_0. \ (v, s_0) \)

bind\_st m f = \( \lambda s_0. \ \text{let} \ vs = m \ s_0 \ \text{in} \ f \ (\text{fst} \ vs) \)

get = \( \lambda s_0. \ (s_0, s_0) \)

set s_1 = \( \lambda \_ \ ((), s_1) \)

Can be derived automatically!
We introduce two calculi: \textsc{dm} and a new \textsc{F}* formalization called \textsc{EMF}*.
We introduce two calculi: DM and a new F* formalization called EMF*.

DM: simply-typed with an abstract base monad $\tau$ (and somewhat restricted)
  - Used to define monads, actions, lifts

EMF*: dependently-typed, allows for user-defined effects
We introduce two calculi: $DM$ and a new $F^*$ formalization called $EMF^*$.

$DM$: simply-typed with an abstract base monad $\tau$ (and somewhat restricted)
- Used to define monads, actions, lifts

$EMF^*$: dependently-typed, allows for user-defined effects

Two translations from well-typed $DM$ terms to $EMF^*$
- $\star$-translation: gives specification (selective CPS)
- Elaboration: gives implementation (essentially an identity)
We introduce two calculi: DM and a new F* formalization called EMF*.

DM: simply-typed with an abstract base monad $\tau$ (and somewhat restricted)
- Used to define monads, actions, lifts

EMF*: dependently-typed, allows for user-defined effects

Two translations from well-typed DM terms to EMF*
- $*$-translation: gives specification (selective CPS)
- Elaboration: gives implementation (essentially an identity)

$*$-translation gives a correct Dijkstra monad for elaborated terms.
Examples: state, exceptions, continuations...
Logical relation correctly specifies

\[ e : C \]

\[ \ast \text{-translation} \]

\[ \text{elaboration} \]

\[ e^* : C^* \]

\[ \text{correctly specifies} \]

\[ e : F_C e^* \]

\[ (\text{DM}) \]

\[ (\text{EMF}^*) \]
Logical relation

\[
\begin{align*}
\text{(DM)} & \quad e : C \\
& \quad \text{elaboration} \\
\text{(EMF}^*\text{)} & \quad e^* : C^* \\
\text{correctly specifies} & \\
\end{align*}
\]
Pure in EMF*

- Pure is the only primitive EMF* effect.
- A WP for Pure $t$ is of type

$$(t \rightarrow \text{Type}_0) \rightarrow \text{Type}_0$$
Pure in EMF*

- Pure is the only primitive EMF* effect.
- A WP for Pure \( t \) is of type

\[
(t \rightarrow \text{Type}_0) \rightarrow \text{Type}_0
\]

- The Dijkstra monad for Pure is exactly the continuation monad.
Pure in EMF*

- Pure is the only primitive EMF* effect.
- A WP for Pure $t$ is of type

$$ (t \rightarrow \text{Type}_0) \rightarrow \text{Type}_0 $$

- The Dijkstra monad for Pure is exactly the continuation monad.

Lemma (Correctness of Pure)

If $\vdash e : \text{Pure } t \text{ wp and } \models \text{wp } p$, then $e \leadsto^* v \text{ s.t. } \models p v.$
Reasoning about ST

- Say we have a term $e$ such that

$$e : S \rightarrow t \times S$$
Reasoning about $ST$

- Say we have a term $e$ such that

$$e : S \rightarrow t \times S$$

- From logical relation, we get

$$e : s_0 : S \rightarrow \text{Pure} \ (t \times S) \ (e^* \ s_0)$$
Reasoning about $\mathcal{ST}$

- Say we have a term $e$ such that

$$e : S \rightarrow t \times S$$

- From logical relation, we get

$$\underline{e} : s_0 : S \rightarrow \text{Pure} \left( t \times S \right) \ (e^* \ s_0)$$

- From previous and correctness of Pure, we get

**Corollary (Correctness of $\mathcal{ST}$)**

If $\vdash e : S \rightarrow t \times S$, and $\models e^* \ s_0 \ p$, then $\underline{e} \ s_0 \ \rightsquigarrow^* (v, s)$ s.t. $\models p(v, s)$. 
- In DM, we can also provide a lift between two monads.

\[
\begin{align*}
\text{ST} \ t &= S \rightarrow t \times S \\
\text{EXNST} \ t &= S \rightarrow (1 + t) \times S \\
\text{lift} &: \text{ST} \ t \rightarrow \text{EXNST} \ t \\
\text{lift} \ m &= \lambda s_0. \ \text{let} \ vs = m \ s_0 \ \text{in} \ (\text{inr} \ (\text{fst} \ vs), \text{snd} \ vs)
\end{align*}
\]
Relating effects

- In $\mathbb{DM}$, we can also provide a lift between two monads.

$$\text{ST } t = S \rightarrow t \times S \quad \text{EXNST } t = S \rightarrow (1 + t) \times S$$

$$\text{lift } : \quad \text{ST } t \rightarrow \text{EXNST } t$$

$$\text{lift } m = \lambda s_0. \ \text{let } vs = m \ s_0 \ \text{in } (\text{inr } (\text{fst } vs), \text{snd } vs)$$

- It will be translated to a correct Dijkstra monad lift.

$$\text{liftwp } : \quad \text{ST}_{wp} \ t \rightarrow \text{EXNST}_{wp} \ t$$

$$\text{liftwp } wp = \lambda s_0 \ p. \ wp \ s_0 \ (\lambda vs. \ p (\text{inr } (\text{fst } vs), \text{snd } vs))$$
Properties of the translations

Besides correctly specifying programs, the generated WPs enjoys some nice properties

- The $\star$-translation preserves equality

- Monads mapped to Dijkstra monads
- Lifts mapped to Dijkstra lifts
- Laws about actions preserved
- $e^{\star}$ is monotonic: it maps weaker postconditions to weaker preconditions.
  \[
  (\forall x : p_1 x = p_2 x) \Rightarrow e^{\star}p_1 = e^{\star}p_2
  \]
- $e^{\star}$ is conjunctive: it distributes over $\land$ and $\forall$.
  \[
  e^{\star}(x : p_1 \land p_2) = e^{\star}p_1 \land e^{\star}p_2
  \]
- These properties together ensure that any $dm$ monad provides a correct Dijkstra monad, that's also usable within the $F^{\star}$ compiler.
Properties of the translations

Besides correctly specifying programs, the generated WPs enjoys some nice properties

- The $\star$-translation preserves equality
  - Monads mapped to Dijkstra monads
  - Lifts mapped to Dijkstra lifts
  - Laws about actions preserved
Properties of the translations

Besides correctly specifying programs, the generated WPs enjoys some nice properties

- The \( \star \)-translation preserves equality
  - Monads mapped to Dijkstra monads
  - Lifts mapped to Dijkstra lifts
  - Laws about actions preserved

- \( e^\star \) is **monotonic**: it maps weaker postconditions to weaker preconditions.
  \[
  (\forall x. p_1 x \implies p_2 x) \implies e^\star p_1 \implies e^\star p_2
  \]

- \( e^\star \) is **conjunctive**: it distributes over \( \land \) and \( \forall \).
  \[
  e^\star (\lambda x. p_1 x \land p_2 x) \iff e^\star p_1 \land e^\star p_2
  \]
Properties of the translations

Besides correctly specifying programs, the generated WPs enjoys some nice properties

- The $\star$-translation preserves equality
  - Monads mapped to Dijkstra monads
  - Lifts mapped to Dijkstra lifts
  - Laws about actions preserved

- $e^\star$ is **monotonic**: it maps weaker postconditions to weaker preconditions.
  \[
  (\forall x. p_1 x \implies p_2 x) \implies e^\star p_1 \implies e^\star p_2
  \]

- $e^\star$ is **conjunctive**: it distributes over $\land$ and $\forall$.
  \[
  e^\star (\lambda x. p_1 x \land p_2 x) \iff e^\star p_1 \land e^\star p_2
  \]

- These properties together ensure that any $\text{DM}$ monad provides a correct Dijkstra monad, that’s also usable within the $F^\star$ compiler.
Conclusions and further work

- We show a formal connection between WPs and CPS, with good properties.
- New version of $F^*$ with user-defined effects:
  greatly broadens its applications and reduces proof obligations.
- Extrinsic reasoning; primitive effects: *details in paper*. 
Conclusions and further work

- We show a formal connection between WPs and CPS, with good properties.
- New version of F* with user-defined effects: greatly broadens its applications and reduces proof obligations.
- Extrinsic reasoning; primitive effects: details in paper.

Further work:
- Exciting new opportunities in verification: probabilistic computation, concurrency, cost analysis...
- Improve the expressiveness of DM.
Conclusions and further work

- We show a formal connection between WPs and CPS, with good properties.
- New version of F* with user-defined effects: greatly broadens its applications and reduces proof obligations.
- Extrinsic reasoning; primitive effects: **details in paper**.

Further work:
- Exciting new opportunities in verification: probabilistic computation, concurrency, cost analysis...
- Improve the expressiveness of DM.

Thank you!
Thank you!