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, $\mathrm{F}^\star$)

Combining dependent types and effects

- Known hard problem, various solutions (Ynot/HTT, Idris, Trellys/Zombie, $\mathrm{F}^{\star}$)
- Common approach: encapsulating effectul programs in monads.
  But how to reason about them?

Combining dependent types and effects

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

```
(* No spec *)
val incr : unit → ST unit
(* Hoare triples *)
val incr : unit → ST unit (requires (λ n0 → True))
                          (ensures (λ n0 r n1 → n1 = n0 + 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 → ST unit
(* Hoare triples *)
val incr : unit → ST unit (requires (λ n0 → True))
                          (ensures (λ n0 r n1 → n1 = n0 + 1))
(* Dijkstra's WPs *)
val incr : unit → ST unit (λ post n0 → post () (n0 + 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:

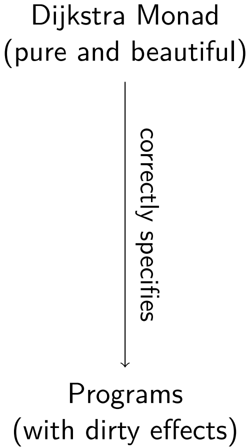
```
(* No spec *)
val incr : unit → ST unit
(* Hoare triples *)
val incr : unit → ST unit (requires (λ n0 → True))
                          (ensures (λ n0 r n1 → n1 = n0 + 1))
(* Dijkstra's WPs *)
val incr : unit → ST unit (λ post n0 → post () (n0 + 1))
```

- **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 $\mathrm{F}^\star$ rigid, in that it had a fixed supply of effects.

Problem...

- The **Dijkstra monad** for each effect needs to be hand-crafted, and proven correct.
- This made $\mathrm{F}^\star$ 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?

Problem... **solution!**

- The **Dijkstra monad** for each effect needs to be hand-crafted, and proven correct.
- This made $\mathrm{F}^{\star}$ 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.

Problem... **solution!**

- The **Dijkstra monad** for each effect needs to be hand-crafted, and proven correct.
- This made $\mathrm{F}^\star$ 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 $\mathrm{F}^\star$... now with user-defined effects.
- Huge boost in simplicity and expressiveness of the effect system.

Problem... **solution!**

- The **Dijkstra monad** for each e hand-crafted, and proven correct.
- This made F⋆ rigid, in th f effects.
- A fundamental questio

What is the **presentation**

f **d?**

- Old dog, new trick: D rm of the representation monad, allowing **autom**
- Simple monadic definition **ruction** WP calculus for it.
- Implemented in F⋆... now with u d effects.
- Huge boost in simplicity and expressiveness of the effect system.

Artifact * Complete * Well Documented * Easy to Reuse * Consistent *

POPL * AEC * Evaluated

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

$$\mathrm{ST}_{wp}\ t = \ (t \to S \to \mathsf{Type}_0) \ \to \ S \to \mathsf{Type}_0$$

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

$$\mathrm{ST}_{wp}\ t = \boxed{(t \to S \to \mathsf{Type}_0)} \to \boxed{S \to \mathsf{Type}_0}$$

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

$$\mathrm{ST}_{wp}\ t = \boxed{(t \to S \to \mathsf{Type}_0)} \to \boxed{S \to \mathsf{Type}_0}$$

  where t is the result type.
- $\mathrm{F}^\star$'s typing judgment gives a WP to each computation:

$$\Gamma \vdash e : \mathrm{ST}\ t\ \mathit{wp}$$

Verifying code

```
let incr () = let n = get () in put (n + 1)
```

Verifying code

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

- Turn it into explicitly monadic form

Verifying code

let incr () = bind_{st} (get ()) (λ n → put (n + 1))

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

  val get : unit → ST int getwp

  val put : n_1:int → ST unit (setwp n_1)

  val bind_{st} : ∀wa wb. ST a wa → (x:a → ST b (wb x)) → ST b ($\text{bindwp}_{\text{st}}$ wa wb)

Verifying code

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

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

  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$_{\mathrm{st}}$ wa wb)

  to get

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

Verifying code

let incr () = bind_{st} (get ()) (λ n → put (n + 1))

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

  val get : unit → ST int getwp

  val put : n_1:int → ST unit (setwp n_1)

  val bind_{st} : ∀wa wb. ST a wa → (x:a → ST b (wb x)) → ST b (bindwp_{st} wa wb)

  to get

  val incr : unit → ST unit (bindwp_{st} getwp (λ n → setwp (n + 1)))

  = val incr : unit → ST unit (λ post n_0 → post () (n_0 + 1))

Verifying code

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

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

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

val incr : unit $\rightarrow$ ST unit (bindwp$_{st}$ getwp (λ n $\rightarrow$ setwp (n + 1)))
= val incr : unit $\rightarrow$ ST unit (λ post n$_0$ $\rightarrow$ post () (n$_0$ + 1))

Primitive specs

$$
\begin{array}{lcl}
\mathrm{ST}_{wp}\ t & = & (t \to S \to \mathsf{Type}_0) \to S \to \mathsf{Type}_0 \\
\mathrm{returnwp}_{st}\ v & = & \lambda p\ s_0.\ p\ v\ s_0 \\
\mathrm{bindwp}_{st}\ wp\ f & = & \lambda p\ s_0.\ wp\ (\lambda v\ s_1.\ f\ v\ p\ s_1)\ s_0 \\
\mathrm{getwp}_{st} & = & \lambda p\ s_0.\ p\ s_0\ s_0 \\
\mathrm{setwp}_{st}\ s_1 & = & \lambda p\ \_.\ p\ ()\ s_1
\end{array}
$$

Primitive specs

$$\begin{array}{lcl}
\mathrm{ST}_{wp}\ t & = & S \to (t \times S \to \mathsf{Type}_0) \to \mathsf{Type}_0 \\
\mathrm{returnwp}_{st}\ v & = & \lambda s_0\ p.\ p\ (v, s_0) \\
\mathrm{bindwp}_{st}\ wp\ f & = & \lambda s_0\ p.\ wp\ s_0\ (\lambda vs.\ f\,(\mathbf{fst}\ vs)\ (\mathbf{snd}\ vs)\ p) \\
\mathrm{getwp}_{st} & = & \lambda s_0\ p.\ p\ (s_0, s_0) \\
\mathrm{setwp}_{st}\ s_1 & = & \lambda\_\ p.\ p\ ((), s_1)
\end{array}$$

Primitive specs

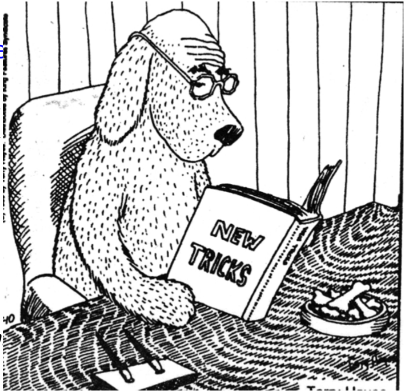
$$\begin{array}{lcl}
\mathrm{ST}_{wp}\ t & = & S \to (t \times S \to \mathsf{Type}_0) \to \mathsf{Type}_0 \\
\mathrm{returnwp}_{st}\ v & = & \lambda s_0\ p.\ p\ (v, s_0) \\
\mathrm{bindwp}_{st}\ wp\ f & = & \lambda s_0\ p.\ wp\ s_0\ (\lambda vs.\ f\ (\mathbf{fst}\ vs)\ (\mathbf{snd}\ vs)\ p) \\
\mathrm{getwp}_{st} & = & \lambda s_0\ p.\ p\ (s_0, s_0) \\
\mathrm{setwp}_{st}\ s_1 & = & \lambda\_\ p.\ p\ ((), s_1)
\end{array}$$

$$\begin{array}{lcl}
\mathrm{ST}\ t & = & S \to t \times S \\
\mathrm{return}_{st}\ v & = & \lambda s_0.\ (v, s_0) \\
\mathrm{bind}_{st}\ m\ f & = & \lambda s_0.\ \mathbf{let}\ vs = m\ s_0\ \mathbf{in}\ f\ (\mathbf{fst}\ vs)\ (\mathbf{snd}\ vs) \\
\mathrm{get} & = & \lambda s_0.\ (s_0, s_0) \\
\mathrm{set}\ s_1 & = & \lambda\_.\ ((), s_1)
\end{array}$$

Primitive specs

$$
\begin{array}{lcl}
\mathrm{ST}_{wp}\ t & = & S \to (t \times S \to \mathsf{Type}_0) \to \mathsf{Type} \\
\mathrm{returnwp}_{st}\ v & = & \lambda s_0\ p.\ p\ (v, s_0) \\
\mathrm{bindwp}_{st}\ wp\ f & = & \lambda s_0\ p.\ wp\ s_0\ (\lambda vs.\ f\ (\mathbf{fst}\ \ vs)\ (\\
\mathrm{getwp}_{st} & = & \lambda s_0\ p.\ p\ (s_0, s_0) \\
\mathrm{setwp}_{st}\ s_1 & = & \lambda\_\ p.\ p\ ((), s_1)
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{ST}\ t & = & S \to t \times S \\
\mathrm{return}_{st}\ v & = & \lambda s_0.\ (v, s_0) \\
\mathrm{bind}_{st}\ m\ f & = & \lambda s_0.\ \mathbf{let}\ \ vs = m\ s_0\ \mathbf{in}\ f\ (\mathbf{fst}\ \ v \\
\mathrm{get} & = & \lambda s_0.\ (s_0, s_0) \\
\mathrm{set}\ s_1 & = & \lambda\_.\ ((), s_1)
\end{array}
$$



Can be derived automatically!

Formalization

- We introduce two calculi: DM and a new F* formalization called EMF*.

Formalization

- We introduce two calculi: DM and a new $\mathrm{F}^\star$ formalization called $\mathrm{EMF}^\star$.
- DM: simply-typed with an abstract base monad τ (and somewhat restricted)
    - Used to define monads, actions, lifts
- $\mathrm{EMF}^\star$: dependently-typed, allows for user-defined effects

Formalization

- We introduce two calculi: DM and a new $\mathrm{F}^\star$ formalization called $\mathrm{EMF}^\star$.
- DM: simply-typed with an abstract base monad τ (and somewhat restricted)
    - Used to define monads, actions, lifts
- $\mathrm{EMF}^\star$: dependently-typed, allows for user-defined effects
- Two translations from well-typed DM terms to $\mathrm{EMF}^\star$
    - $\star$-translation: gives specification (selective CPS)
    - Elaboration: gives implementation (essentially an identity)

Formalization

- We introduce two calculi: DM and a new $\mathrm{F}^\star$ formalization called $\mathrm{EMF}^\star$.
- DM: simply-typed with an abstract base monad τ (and somewhat restricted)
  - Used to define monads, actions, lifts
- $\mathrm{EMF}^\star$: dependently-typed, allows for user-defined effects
- Two translations from well-typed DM terms to $\mathrm{EMF}^\star$
  - $\star$-translation: gives specification (selective CPS)
  - Elaboration: gives implementation (essentially an identity)
- $\star$-translation gives a correct Dijkstra monad for elaborated terms.
  Examples: state, exceptions, continuations...

$$\begin{tikzcd}
 & & e^\star : C^\star \arrow[dd, "\text{correctly specifies}"] & \\
(\textsc{dm}) & e : C \arrow[ur, "\star\text{-translation}"] \arrow[dr, "\text{elaboration}"'] & & (\textsc{emf}^\star) \\
 & & \underline{e} : \mathrm{F}_C\ e^\star &
\end{tikzcd}$$

$$\begin{tikzcd}
 & & & e^\star : C^\star \arrow[dd, "\text{correctly specifies}"] & \\
(\textsc{dm}) & e : C \arrow[urr, "\star\text{-translation}"] \arrow[drr, "\text{elaboration}"'] & & & (\textsc{emf}^\star) \\
 & & & \boxed{\underline{e} : \mathrm{F}_C\ e^\star} &
\end{tikzcd}$$

Logical relation

Pure in EMF*

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

$$(t \to \mathsf{Type}_0) \to \mathsf{Type}_0$$

Pure in $\mathrm{EMF}^\star$

- Pure is the only primitive $\mathrm{EMF}^\star$ effect.
- A WP for $\mathrm{Pure}\ t$ is of type

$$(t \to \mathsf{Type}_0) \to \mathsf{Type}_0$$

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

Pure in EMF*

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

$$(t \to \mathsf{Type}_0) \to \mathsf{Type}_0$$

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

Lemma (Correctness of Pure)

If $\vdash e : \mathrm{Pure}\ t\ wp$ *and* $\vDash wp\ p$, *then* $e \rightsquigarrow^* v$ *s.t.* $\vDash p\ v$.

Reasoning about ST

- Say we have a term e such that

$$e \ : \ S \to t \times S$$

Reasoning about ST

- Say we have a term e such that

$$e \;:\; S \to t \times S$$

- From logical relation, we get

$$\underline{e} \;:\; s_0{:}S \to \mathrm{Pure}\ (t \times S)\ (e^{\star}\ s_0)$$

Reasoning about ST

- Say we have a term e such that

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

- From logical relation, we get

$$\underline{e} \;:\; s_0{:}S \rightarrow \mathrm{Pure}\ (t \times S)\ (e^{\star}\ \ s_0)$$

- From previous and correctness of Pure, we get

Corollary (Correctness of ST)

If $\vdash e : S \rightarrow t \times S$*, and* $\vDash e^{\star}\ \ s_0\ \ p$*, then* $\underline{e}\ \ s_0 \rightsquigarrow^{*} (v, s)$ *s.t.* $\vDash p\ (v, s)$*.*

Relating effects

- In DM, we can also provide a lift between two monads.

$$\mathsf{ST}\ \mathsf{t} = \mathsf{S} \to \mathsf{t} \times \mathsf{S} \qquad \mathsf{EXNST}\ \mathsf{t} = \mathsf{S} \to (1 + \mathsf{t}) \times \mathsf{S}$$

$$\begin{array}{lcl} \mathsf{lift} & : & \mathsf{ST}\ \mathsf{t} \to \mathsf{EXNST}\ \mathsf{t} \\ \mathsf{lift}\ \mathsf{m} & = & \lambda \mathsf{s}_0.\ \mathbf{let}\ \mathsf{vs} = \mathsf{m}\ \mathsf{s}_0\ \mathbf{in}\ (\mathbf{inr}\ (\mathbf{fst}\ \mathsf{vs}), \mathbf{snd}\ \mathsf{vs}) \end{array}$$

Relating effects

- In DM, we can also provide a lift between two monads.

$$\mathsf{ST\ t = S \to t \times S} \qquad \mathsf{EXNST\ t = S \to (1 + t) \times S}$$

$$\begin{array}{lcl} \mathsf{lift} & : & \mathsf{ST\ t \to EXNST\ t} \\ \mathsf{lift\ m} & = & \lambda \mathsf{s_0}.\ \mathbf{let}\ \mathsf{vs} = \mathsf{m\ s_0}\ \mathbf{in}\ (\mathbf{inr}\ (\mathbf{fst}\ \mathsf{vs}), \mathbf{snd}\ \mathsf{vs}) \end{array}$$

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

$$\begin{array}{lcl} \mathrm{liftwp} & : & \mathrm{ST}_{wp}\ t \to \mathrm{EXNST}_{wp}\ t \\ \mathrm{liftwp}\ wp & = & \lambda s_0\ p.\ wp\ s_0\ (\lambda vs.\ p\ (\mathbf{inr}\ (\mathbf{fst}\ vs), \mathbf{snd}\ vs)) \end{array}$$

$$\begin{tikzcd}
 & \mathsf{Div} \arrow[r] \arrow[dr] & \mathsf{DivExn} \arrow[dr] & \\
\mathsf{Pure} \arrow[ur] \arrow[r] \arrow[dr] & \mathsf{Exn} \arrow[ur] \arrow[dr] & \mathsf{DivSt} \arrow[r] & \mathsf{All} \\
 & \mathsf{St} \arrow[ur] \arrow[r] & \mathsf{StExn} \arrow[ur] &
\end{tikzcd}$$

Properties of the translations

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

- The $\star$-translation preserves equality

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 $\wedge$ and $\forall$.

$$e^\star \ (\lambda x. \ p_1 \ x \wedge p_2 \ x) \iff e^\star \ p_1 \wedge e^\star \ p_2$$

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 $\textsc{dm}$ monad provides a correct Dijkstra monad, that's also usable within the $\mathrm{F}^\star$ compiler.

Conclusions and further work

- We show a formal connection between WPs and CPS, with good properties.
- New version of $\mathrm{F}^\star$ 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 $\mathrm{F}^{\star}$ 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 expresiveness of DM.

Conclusions and further work

- We show a formal connection between WPs and CPS, with good properties.
- New version of $\mathrm{F}^\star$ 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 expresiveness of $\textsc{dm}$.

Thank you!

Thank you!