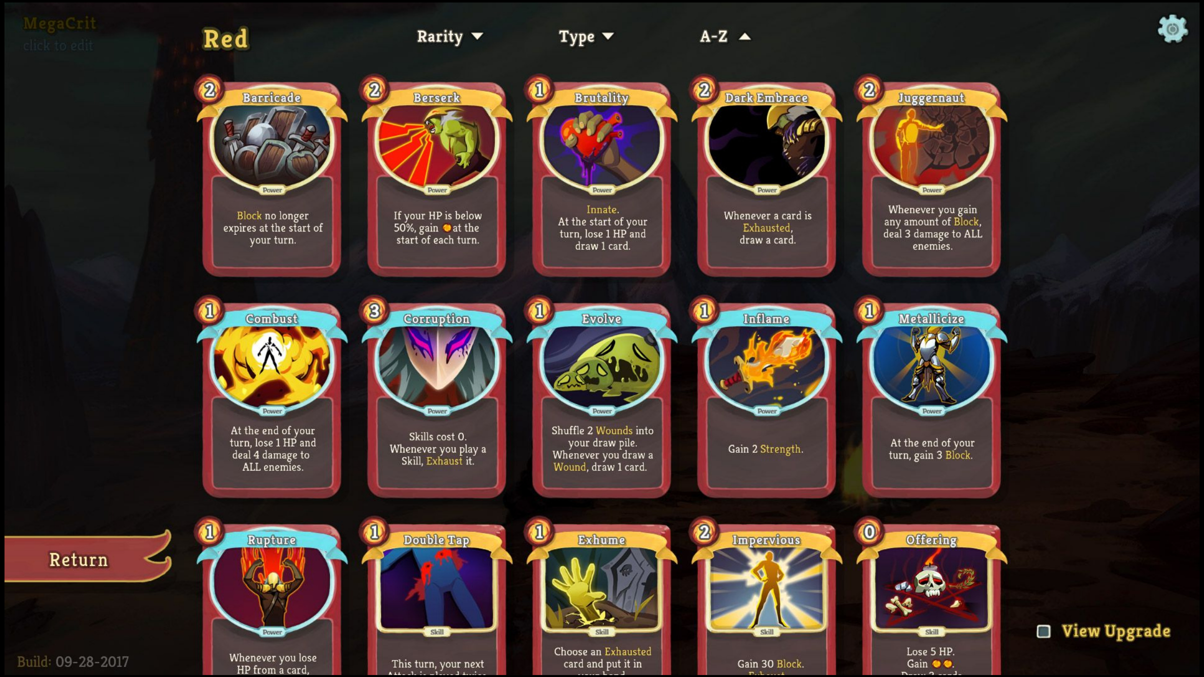
(Non-)Monadic Effect Handlers

Ruben Pieters

Situation: Card Effects



2 Barricade Power
Block no longer expires at the start of your turn.

2 Berserk Power
If your HP is below 50%, gain ● at the start of each turn.

1 Brutality Power
Innate. At the start of your turn, lose 1 HP and draw 1 card.

2 Dark Embrace Power
Whenever a card is Exhausted, draw a card.

2 Juggernaut Power
Whenever you gain any amount of Block, deal 3 damage to ALL enemies.

1 Combust Power
At the end of your turn, lose 1 HP and deal 4 damage to ALL enemies.

3 Corruption Power
Skills cost 0. Whenever you play a Skill, Exhaust it.

1 Evolve Power
Shuffle 2 Wounds into your draw pile. Whenever you draw a Wound, draw 1 card.

1 Inflame Power
Gain 2 Strength.

1 Metallicize Power
At the end of your turn, gain 3 Block.

1 Rupture Power
Whenever you lose HP from a card,

1 Double Tap Skill
This turn, your next

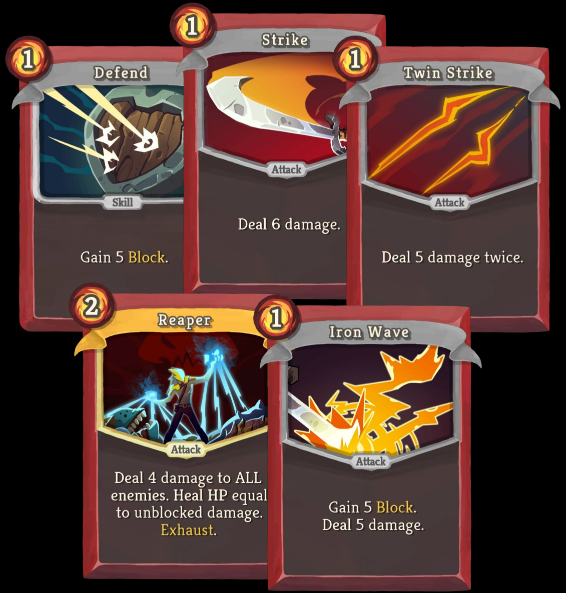
1 Exhume Skill
Choose an Exhausted card and put it in

2 Impervious Skill
Gain 30 Block.

0 Offering Skill
Lose 5 HP. Gain ●●.

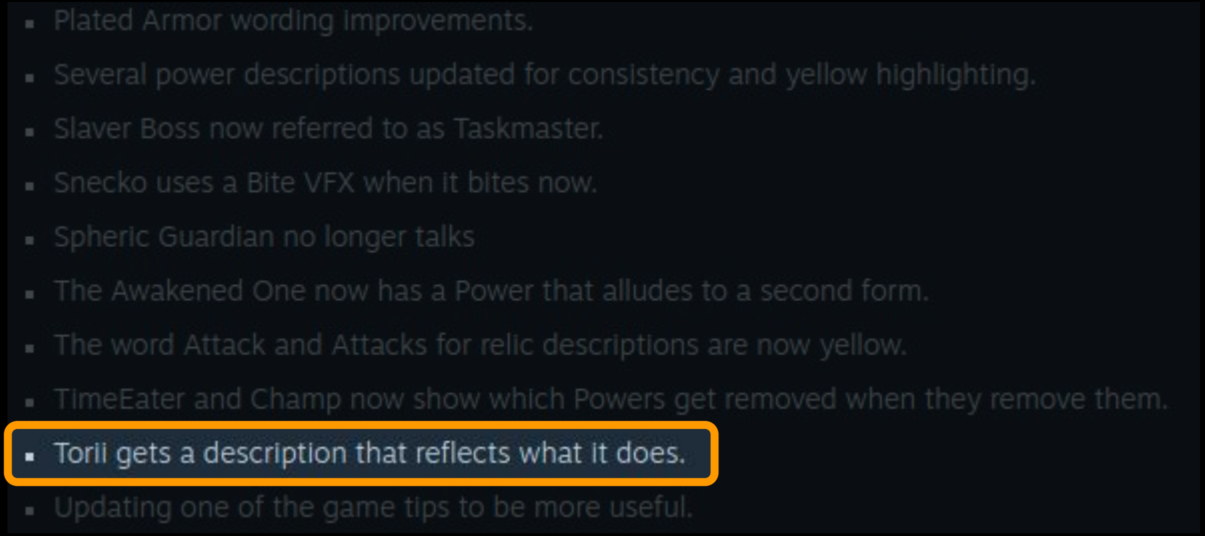
* Game images in this presentation are from *Slay the Spire* (Mega Crit)

Situation: Card Effects



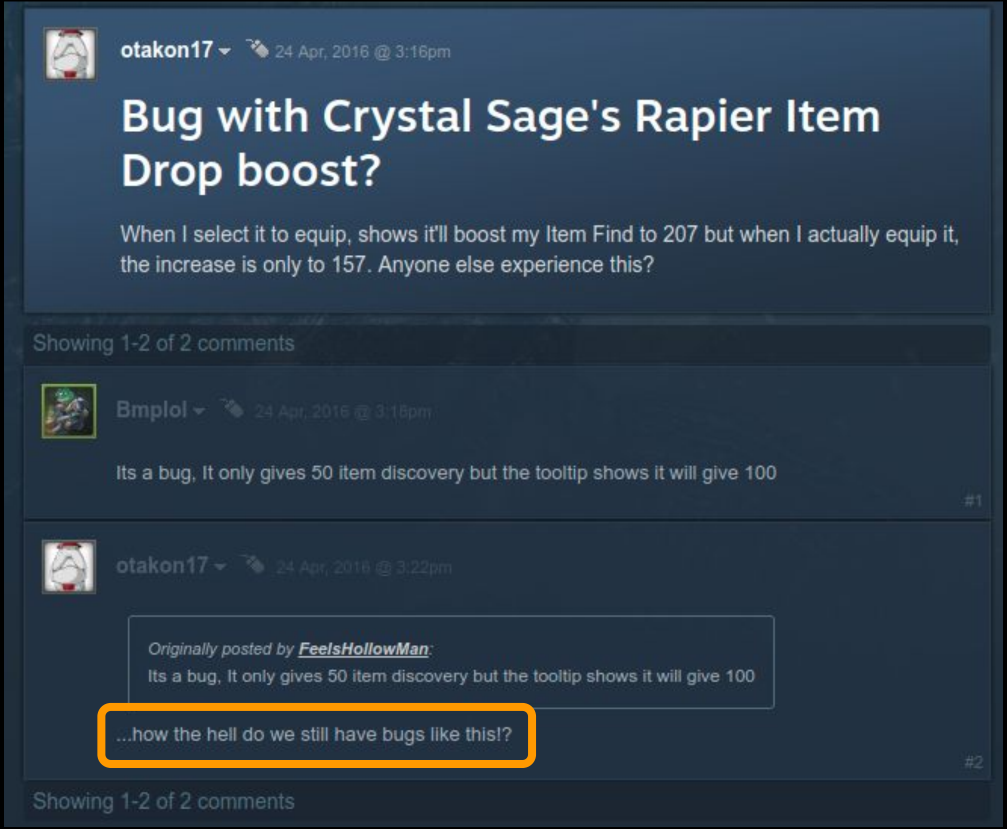
Deal 6 damage.

Bugs



- Plated Armor wording improvements.
- Several power descriptions updated for consistency and yellow highlighting.
- Slaver Boss now referred to as Taskmaster.
- Snecko uses a Bite VFX when it bites now.
- Spheric Guardian no longer talks
- The Awakened One now has a Power that alludes to a second form.
- The word Attack and Attacks for relic descriptions are now yellow.
- TimeEater and Champ now show which Powers get removed when they remove them.
- Torii gets a description that reflects what it does.
- Updating one of the game tips to be more useful.

Bugs



otakon17 24 Apr, 2016 @ 3:16pm

Bug with Crystal Sage's Rapier Item Drop boost?

When I select it to equip, shows it'll boost my Item Find to 207 but when I actually equip it, the increase is only to 157. Anyone else experience this?

Showing 1-2 of 2 comments

Bmplol 24 Apr, 2016 @ 3:18pm

Its a bug, It only gives 50 item discovery but the tooltip shows it will give 100

#1

otakon17 24 Apr, 2016 @ 3:22pm

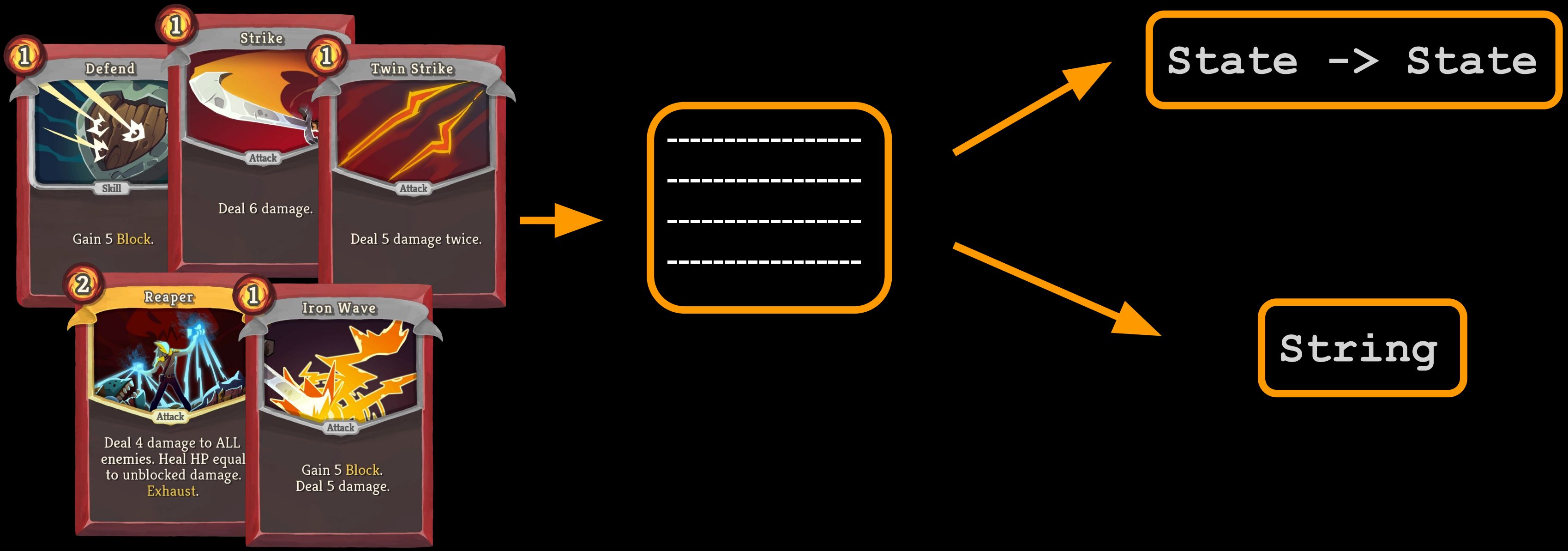
> *Originally posted by* ***FeelsHollowMan***:
> Its a bug, It only gives 50 item discovery but the tooltip shows it will give 100

...how the hell do we still have bugs like this!?

#2

Showing 1-2 of 2 comments

Automate Description



State -> State

String

1.
Data
Types

2.
Effect
Handlers

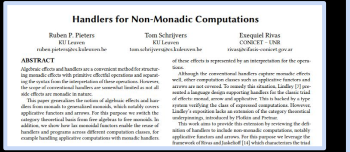
EFF

1.
Data
Types

~~2.
Effect
Handlers~~

3.
Non-
Monadic
Effect
Handlers

EFF



Handlers for Non-Monadic Computations

Ruben P. Pieters
KU Leuven
ruben.pieters@cs.kuleuven.be

Tom Schrijvers
KU Leuven
tom.schrijvers@cs.kuleuven.be

Exequiel Rivas
CONICET – UNR
rivas@cifasis-conicet.gov.ar

ABSTRACT

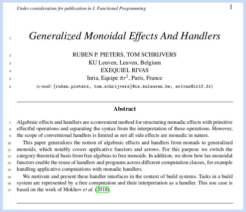
Algebraic effects and handlers are a convenient method for structuring monadic effects with primitive effectful operations and separating the syntax from the interpretation of these operations. However, the scope of conventional handlers are somewhat limited as not all side effects are monadic in nature.

This paper generalizes the notion of algebraic effects and handlers from monads to generalized monoids, which notably covers applicative functors and arrows. For this purpose we switch the category theoretical basis from free algebras to free monoids. In addition, we show how lax monoidal functors enable the reuse of handlers and programs across different computation classes, for example handling applicative computations with monadic handlers.

of these effects is represented by an interpretation for the operations.

Although the conventional handlers capture monadic effects well, other computation classes such as applicative functors and arrows are not covered. To remedy this situation, Lindley [7] presented a language design supporting handlers for the classic triad of effects: monad, arrow and applicative. This is backed by a type system verifying the class of expressed computations. However, Lindley's exposition lacks an extension of the category theoretical underpinnings, introduced by Plotkin and Pretnar.

This work aims to provide this extension by reviewing the definition of handlers to include non-monadic computations, notably applicative functors and arrows. For this purpose we leverage the framework of Rivas and Jaskelioff [14] which characterizes the triad



Under consideration for publication in J. Functional Programming 1

Generalized Monoidal Effects And Handlers

RUBEN P. PIETERS, TOM SCHRIJVERS
KU Leuven, Leuven, Belgium
EXEQUIEL RIVAS
Inria, Equipe πr^2, Paris, France
(*e-mail:* {ruben.pieters, tom.schrijvers}@cs.kuleuven.be, erivas@irif.fr)

Abstract

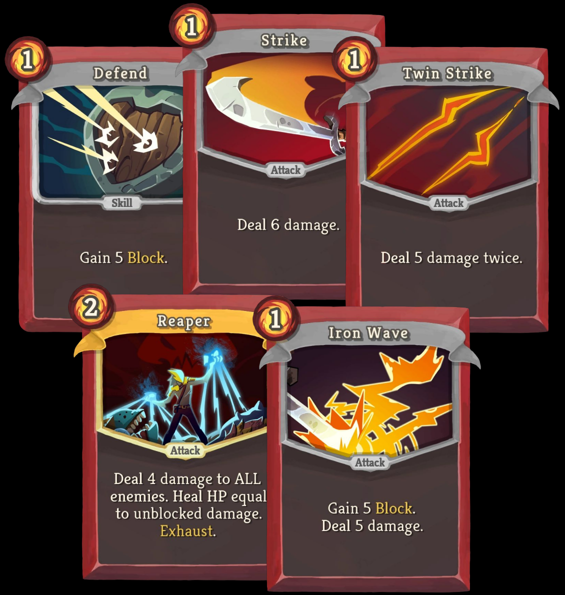
Algebraic effects and handlers are a convenient method for structuring monadic effects with primitive effectful operations and separating the syntax from the interpretation of these operations. However, the scope of conventional handlers is limited as not all side effects are monadic in nature.

This paper generalizes the notion of algebraic effects and handlers from monads to generalized monoids, which notably covers applicative functors and arrows. For this purpose we switch the category theoretical basis from free algebras to free monoids. In addition, we show how lax monoidal functors enable the reuse of handlers and programs across different computation classes, for example handling applicative computations with monadic handlers.

We motivate and present these handler interfaces in the context of build systems. Tasks in a build system are represented by a free computation and their interpretation as a handler. This use case is based on the work of Mokhov *et al.* (2018).

Data Types

Data Types I



```
data Card
```

Data Types I

1
Defend
Skill
Gain 5 Block.
1
Strike
Attack
Deal 6 damage.
Twin Strike
Attack
Deal 5 damage twice.
Attack
Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.
1
Iron Wave
Attack
Gain 5 Block.
Deal 5 damage.

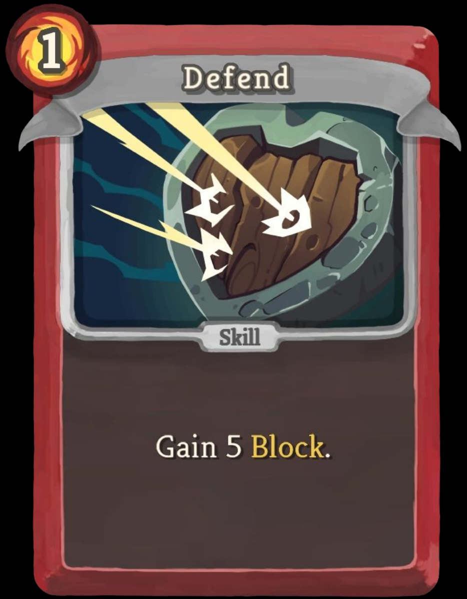
```
data Card
 = Dmg Int
 | Block Int
```

Data Types I

1
Strike
Attack
Deal 6 damage.

Dmg 6

Data Types I



Gain 5 Block.

```
Block 5
```

Data Types I

```
apply :: Card -> State -> State
```

Data Types I

```
apply :: Card -> (State -> State)
```

Data Types I

```
apply :: Card -> State -> State
apply (Dmg x) state = <new state>
apply (Block x) state = <new state>
```

Data Types I

1
Strike
Attack
Deal 6 damage.

```
apply :: Card -> State -> State
apply (Dmg x) state = <new state>
apply (Block x) state = <new state>


> apply (Dmg 6) (10, 10)
(4,10)
```

Data Types I

1
Defend
Skill
Gain 5 Block.

```
apply :: Card -> State -> State
apply (Dmg x) state = <new state>
apply (Block x) state = <new state>


> apply (Block 5) (10, 10)
(10,15)
```

Data Types I

```
desc :: Card -> String
```

Data Types I

```
desc :: Card -> String
desc (Dmg x) =
 "deal " ++ show x ++ " damage"
```

Data Types I

```
desc :: Card -> String
desc (Dmg x) = [i|deal #{x} damage|]
```

Data Types I

```
desc :: Card -> String
desc (Dmg x) = [i|deal #{x} damage|]
desc (Block x) = [i|gain #{x} block|]
```

Data Types I

1
Strike
Attack
Deal 6 damage.

```
desc :: Card -> String
desc (Dmg x) = [i|deal #{x} damage|]
desc (Block x) = [i|gain #{x} block|]
```

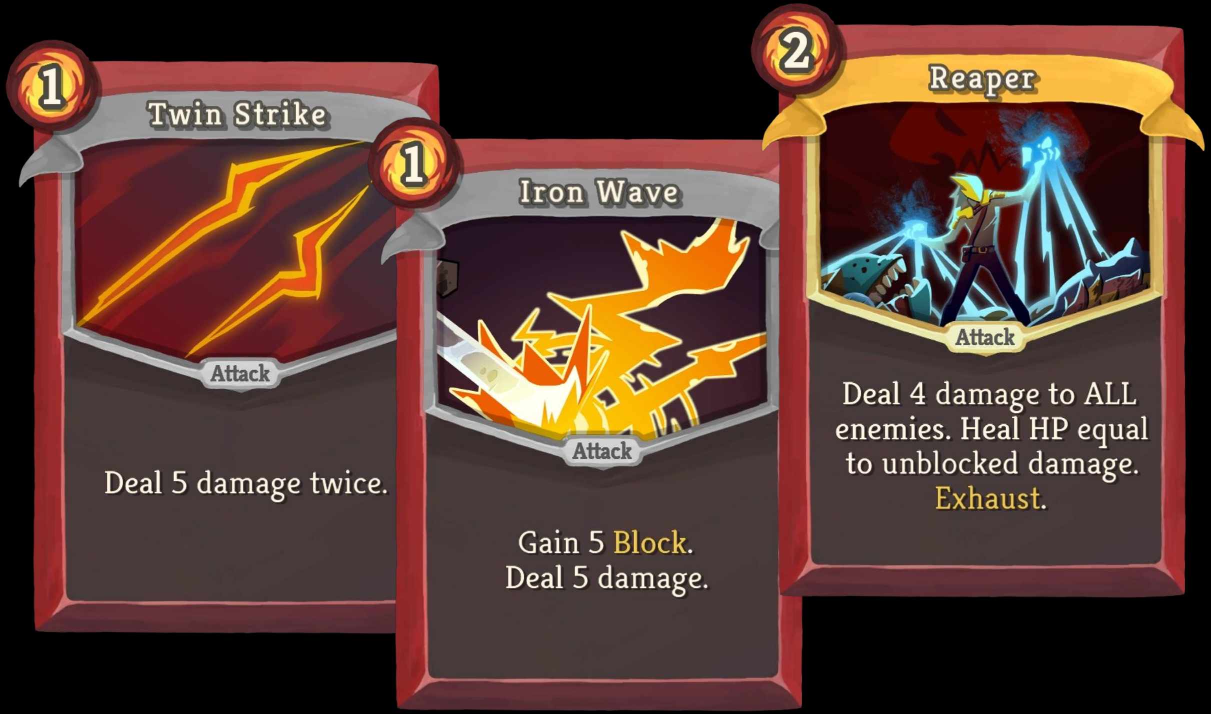
```
> desc (Dmg 6)
"deal 6 damage"
```

Data Types I

1
Defend
Skill
Gain 5 Block.

```
desc :: Card -> String
desc (Dmg x) = [i|deal #{x} damage|]
desc (Block x) = [i|gain #{x} block|]


> desc (Block 5)
"gain 5 block"
```

Data Types II



1 Twin Strike

Attack

Deal 5 damage twice.

1 Iron Wave

Attack

Gain 5 Block.
Deal 5 damage.

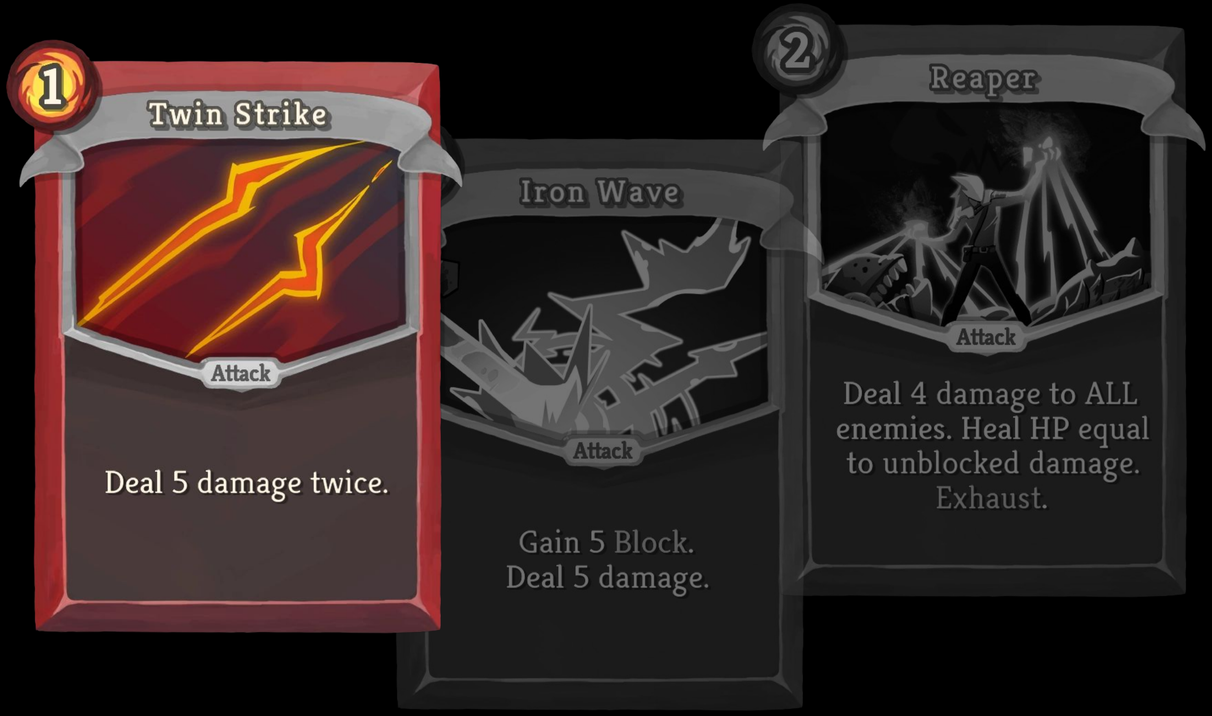
2 Reaper

Attack

Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.

?

Data Types II



1

Twin Strike

Attack

Deal 5 damage twice.

Iron Wave

Attack

Gain 5 Block.
Deal 5 damage.

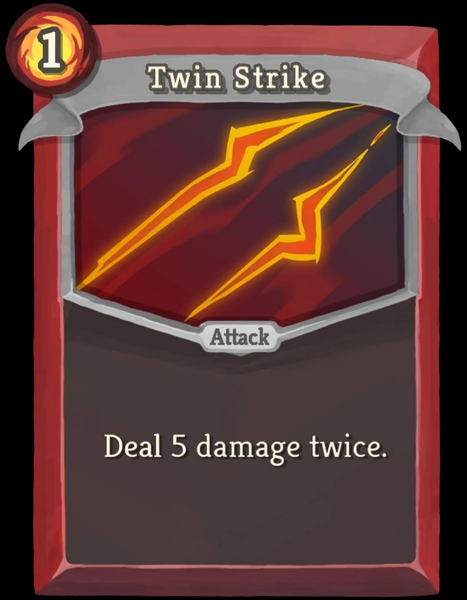
2

Reaper

Attack

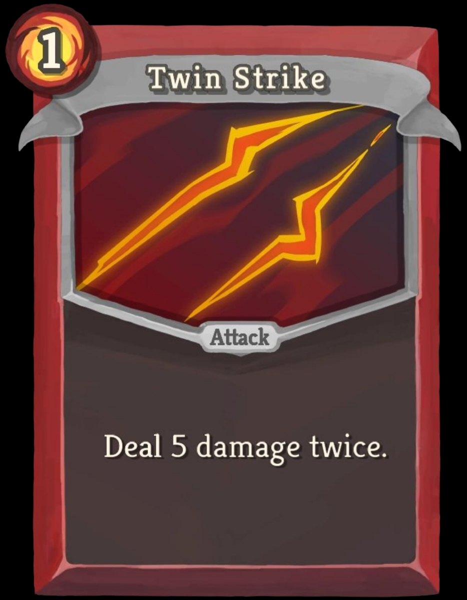
Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.

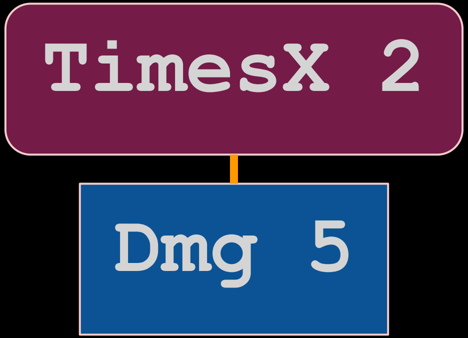
Data Types II



```
data Card
  = Dmg Int
  | Block Int
  | TimesX Int Card
```

Data Types II





TimesX 2

Dmg 5

Data Types II

```
apply :: Card -> State -> State
apply (...) = ...
apply (TimesX 0 a) s = s
apply (TimesX x c) s = let
 s' = apply c s
 in apply (TimesX (x - 1) c) s'
```

Data Types II

```
apply :: Card -> State -> State
apply (...) = ...
apply (TimesX 0 a) s = s
apply (TimesX x c) s = let
 s' = apply c s
 in apply (TimesX (x - 1) c) s'
```

Data Types II

```
apply :: Card -> State -> State
apply (...) = ...
apply (TimesX 0 c) s = s
apply (TimesX x c) s = let
 s' = apply c s
 in apply (TimesX (x - 1) c) s'
```

Data Types II

1
Twin Strike
Attack
Deal 5 damage twice.

```
apply :: Card -> State -> State
apply (...) = ...



> apply (TimesX 2 (Dmg 5)) (10, 10)
(0,10)
```

Data Types II

```
desc :: Card -> String
desc (...) = ...
desc (TimesX x c) =
 [i|#{desc c}, #{x} times|]
```

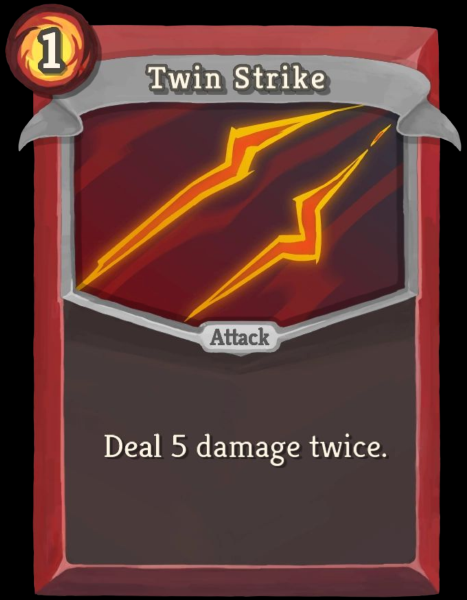
Data Types II

```
desc :: Card -> String
desc (...) = ...
desc (TimesX x c) =
  [i|#{desc c}, #{x} times|]
```

Data Types II

```
desc :: Card -> String
desc (...) = ...
desc (TimesX x c) =
  [i|#{desc c}, #{x} times|]
```

Data Types II



```
desc :: Card -> String
desc (...) = ...
desc (TimesX x c) =
 [i|#{desc c}, #{x} times|]
```

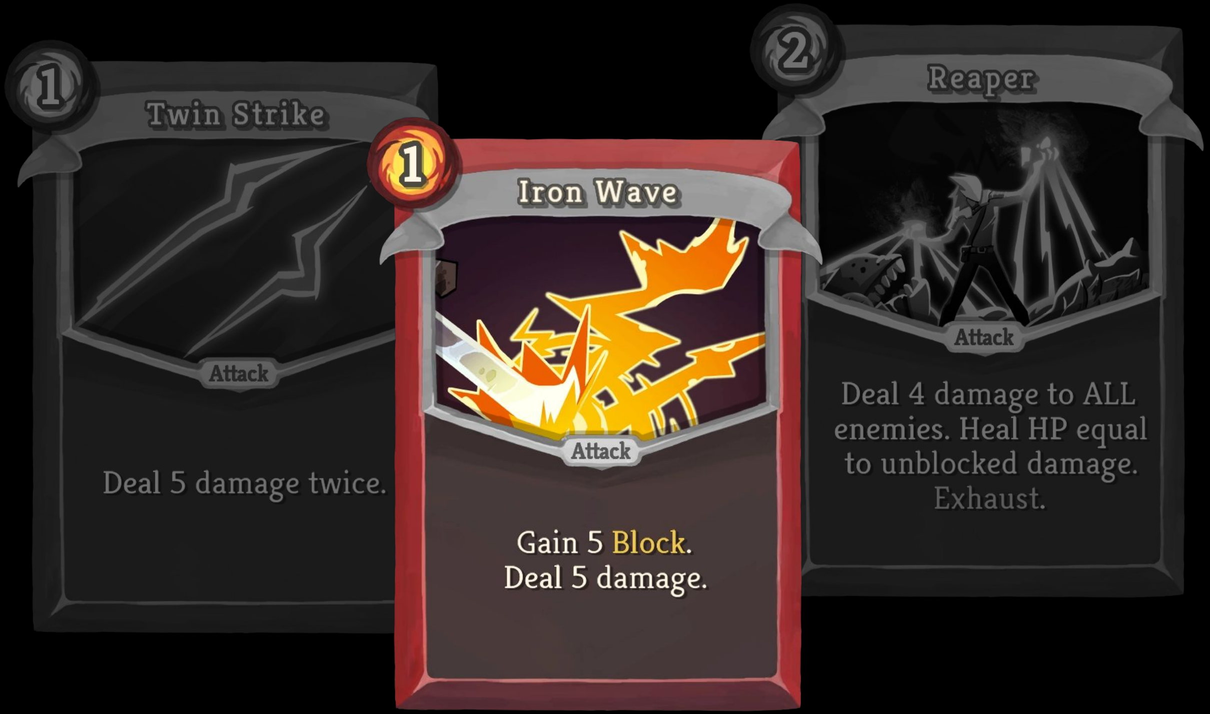
```
> desc (TimesX 2 (Dmg 5))
"deal 5 damage, 2 times"
```

Data Types II

1
Twin Strike
Attack
Deal 5 damage twice.

```
desc :: Card -> String
desc (...) = ...
desc (TimesX 2 c) =
  [i|#{desc c} twice|]


> desc (TimesX 2 (Dmg 5))
"deal 5 damage twice"
```

Data Types III



Deal 5 damage twice.

Gain 5 Block.
Deal 5 damage.

Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.

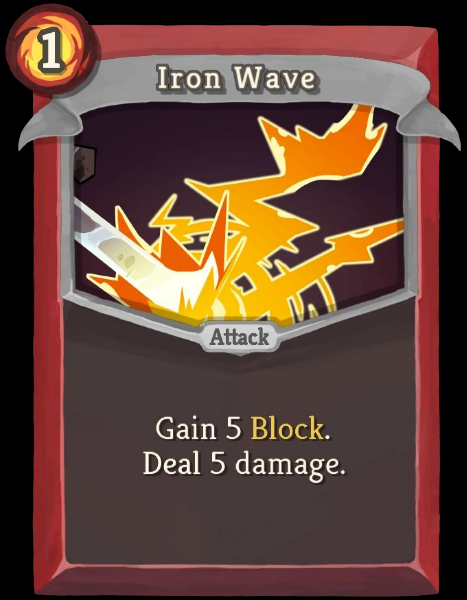
Data Types III

1
Iron Wave
Attack

Gain 5 Block.
Deal 5 damage.

```
data Card
 = Dmg Int
 | Block Int
 | And Card Card
```

Data Types III

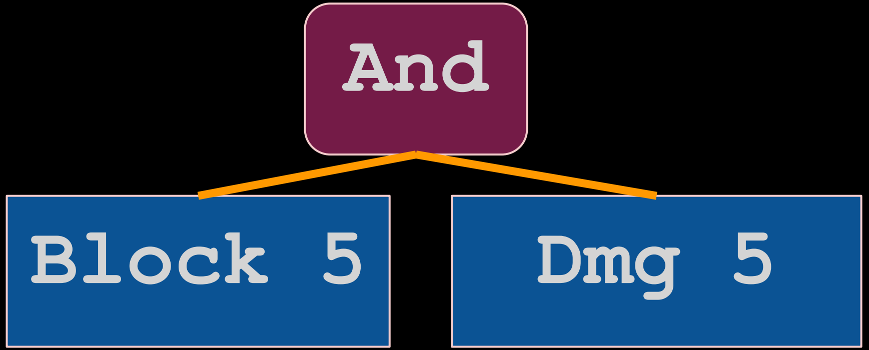


1

Iron Wave

Attack

Gain 5 Block.
Deal 5 damage.



And

Block 5

Dmg 5

Data Types III

```
apply :: Card -> State -> State
apply (...) = ...
apply (And c1 c2) s = let
 after1 = apply c1 s
 in apply c2 after1
```

Data Types III

```
apply :: Card -> State -> State
apply (...) = ...
apply (And c1 c2) s = let
 after1 = apply c1 s
 in apply c2 after1
```

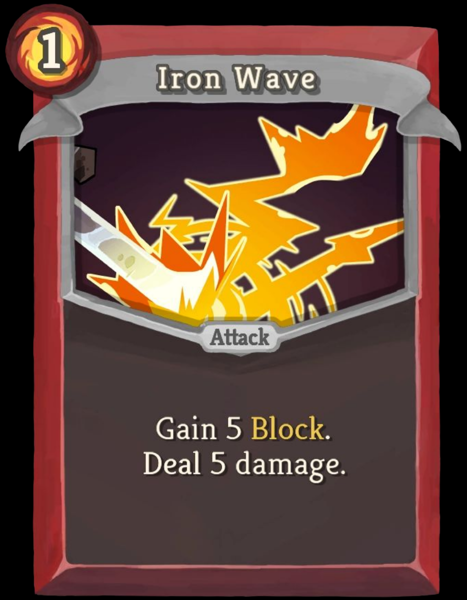
Data Types III

```
apply :: Card -> State -> State
apply (...) = ...
apply (And c1 c2) s = let
 after1 = apply c1 s
 in apply c2 after1
```

Data Types III

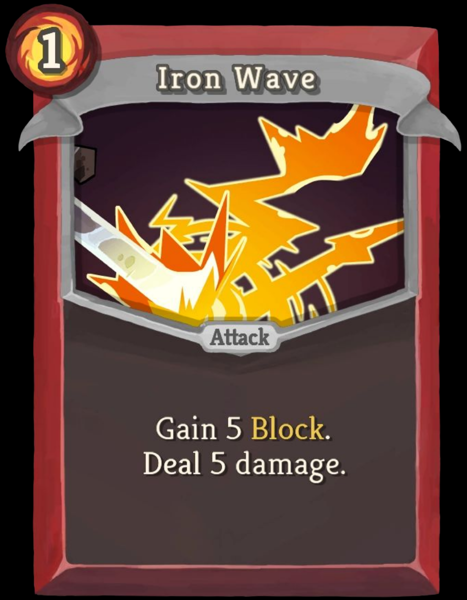
```
desc :: Card -> String
desc (...) = ...
desc (And c1 c2) =
 [i|#{desc c1}, then #{desc c2}|]
```

Data Types III



```
apply :: Card -> State -> State
apply (...) = ...


> apply
    (And (Block 5) (Dmg 5))
    (10, 10)
(5,15)
```

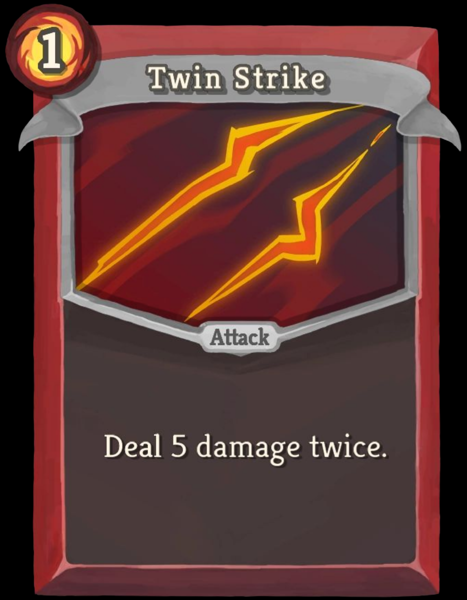
Data Types III



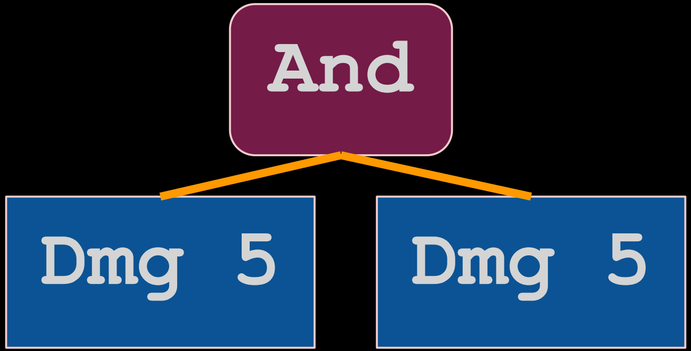
```
desc :: Card -> String
desc (...) = ...
desc (And c1 c2) =
 [i|#{desc c1}, then #{desc c2}|]
```

```
> desc (And (Block 5) (Dmg 5))
"gain 5 block, then deal 5 damage"
```

Data Types III



Deal 5 damage twice.



Data Types III

1
Twin Strike
Attack
Deal 5 damage twice.

```
apply :: Card -> State -> State
apply (...) = ...
```

```
> apply (And (Dmg 5) (Dmg 5)) (10, 10)
(0,10)
```

Data Types III

1
Twin Strike
Attack
Deal 5 damage twice.

```
desc :: Card -> String
desc (...) = ...
desc (And c1 c2) =
  [i|#{desc c1}, then #{desc c2}|]
```

```
> desc (And (Dmg 5) (Dmg 5))
"deal 5 damage, then deal 5 damage"
```

Data Types III

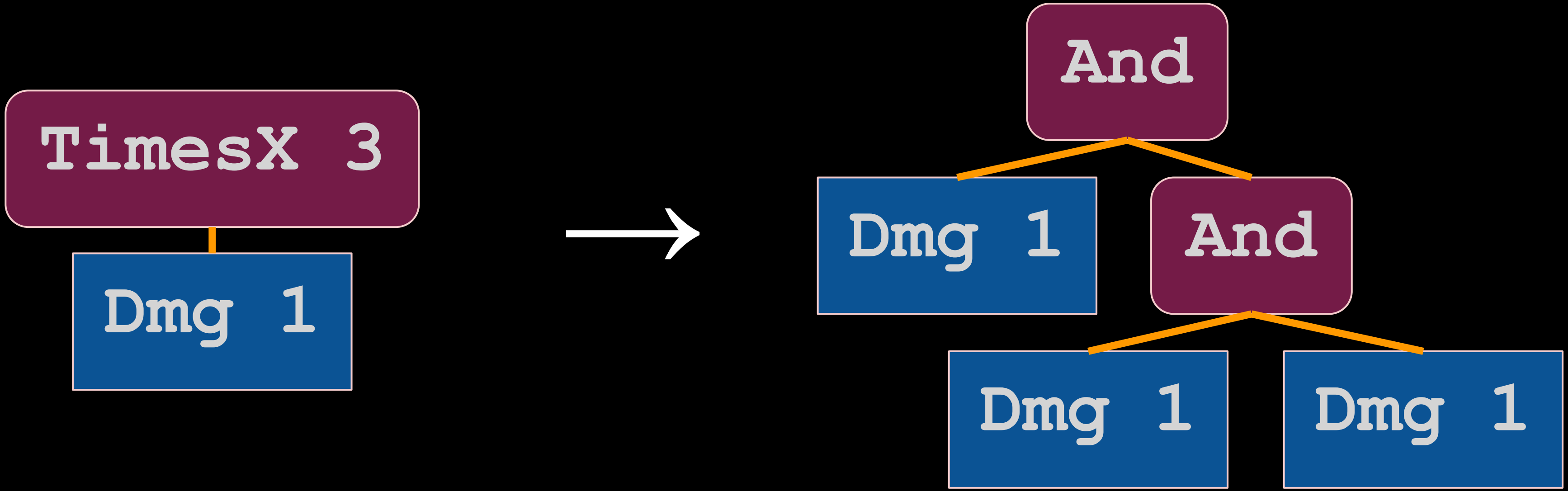
1
Twin Strike
Attack
Deal 5 damage twice.

```
desc :: Card -> String
desc (...) = ...
desc (And c1 c2) =
 [i|#{desc c1}, then #{desc c2}|]


> desc (And (Dmg 5) (Dmg 5))
"deal 5 damage, then deal 5 damage"
```

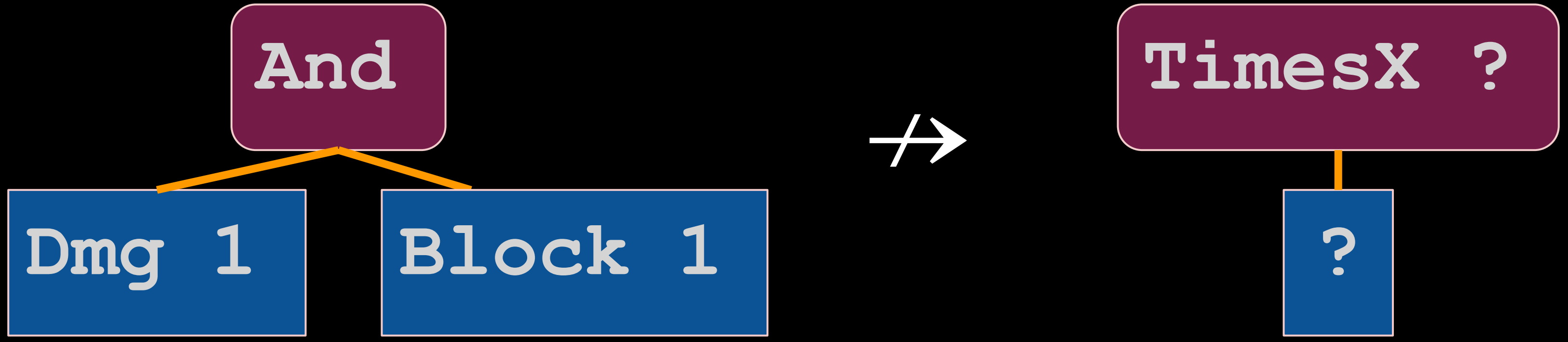
Data Types III

```
desc :: Card -> String
desc (...) = ...
desc (And c1 c2) | <condition> =
 <#{desc c} twice>
desc (And c1 c2) =
 [i|#{desc c1}, then #{desc c2}|]

> desc (And (Dmg 5) (Dmg 5))
"deal 5 damage twice"
```

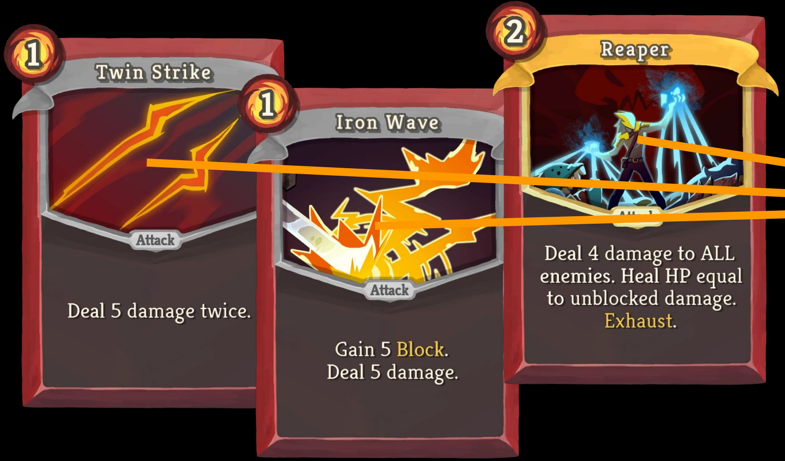
Data Types III



Data Types III



Data Types IV



```
data Card
  = Dmg Int
  | Block Int
  | ???
```

Data Types IV

```
data Card a where
 Dmg :: Int -> Card Int
 Block :: Int -> Card Int
 Bind :: Card a -> (a -> Card b)
      -> Card b
```

Data Types IV

```
data Card a where
 Dmg :: Int -> Card Int
 Block :: Int -> Card Int
 Bind :: Card a -> (a -> Card b)
      -> Card b
```

Data Types IV

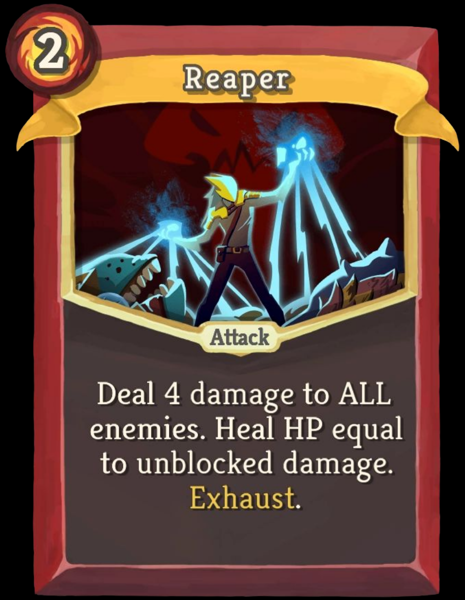
damage dealt

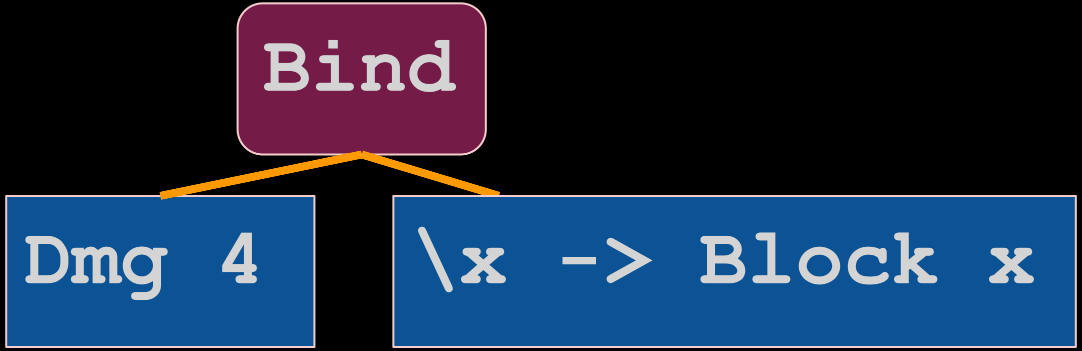
```
data Card a where
 Dmg :: Int -> Card Int
 Block :: Int -> Card Int
 Bind :: Card a -> (a -> Card b)
      -> Card b
```

Data Types IV

```
data Card a where
 Dmg :: Int -> Card Int
 Block :: Int -> Card Int
 Bind :: Card a -> (a -> Card b)
      -> Card b
```

Data Types IV


2
Reaper
Attack
Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage.
Exhaust.



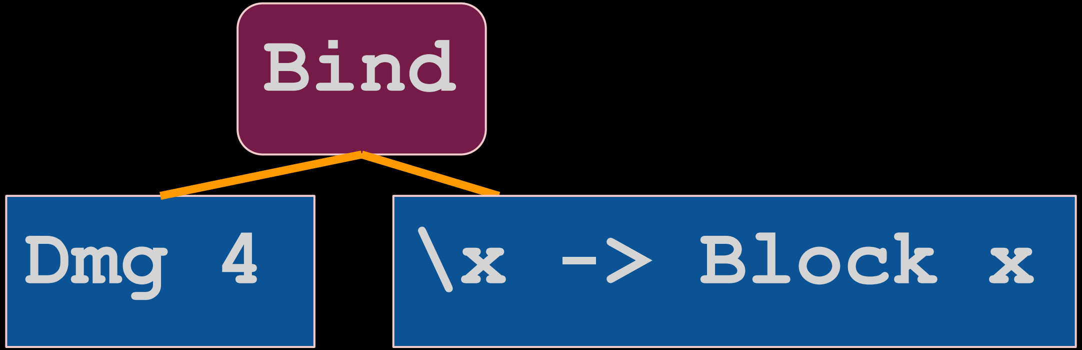
Bind

Dmg 4

\x -> Block x

Data Types IV

2
Reaper
Attack
Deal 4 damage .
Gain block equal
to unblocked damage.



Bind

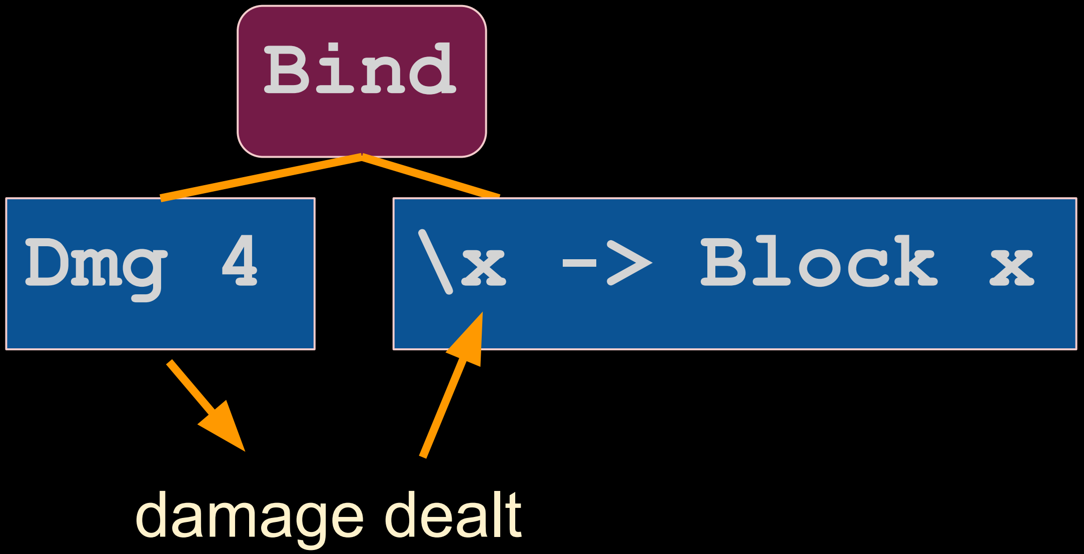
Dmg 4

\x -> Block x

Data Types IV

2
Reaper
Attack

Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.



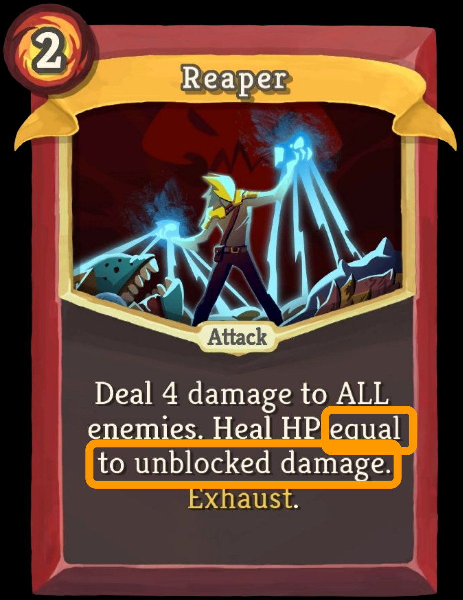
`Bind`

`Dmg 4`

`\x -> Block x`

damage dealt

Data Types IV

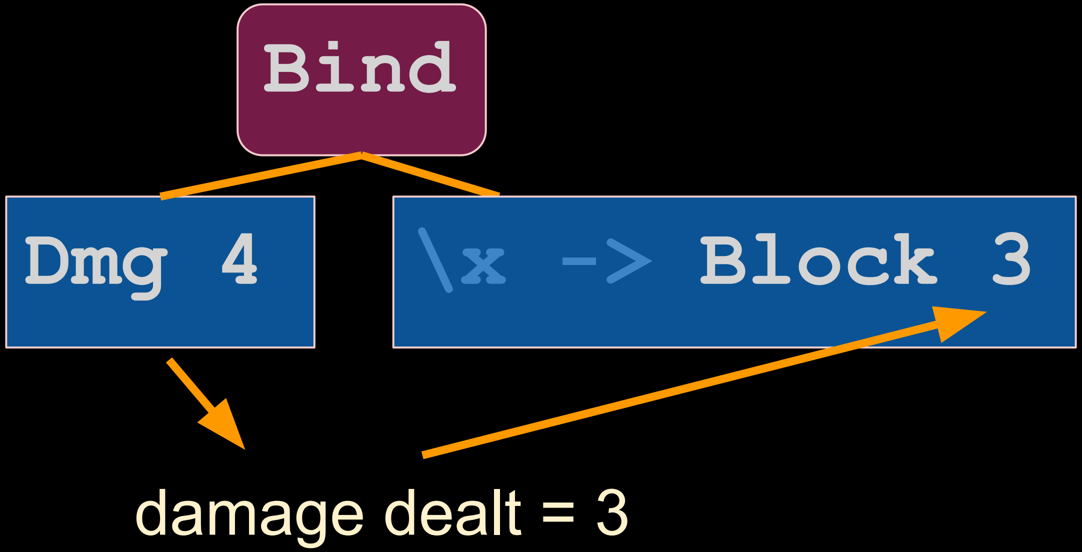


2

Reaper

Attack

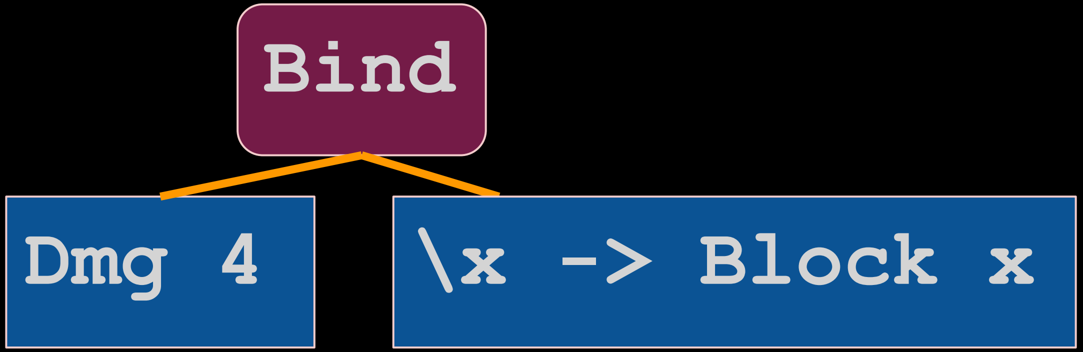
Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.



```
Bind
Dmg 4
\x -> Block 3
```

damage dealt = 3

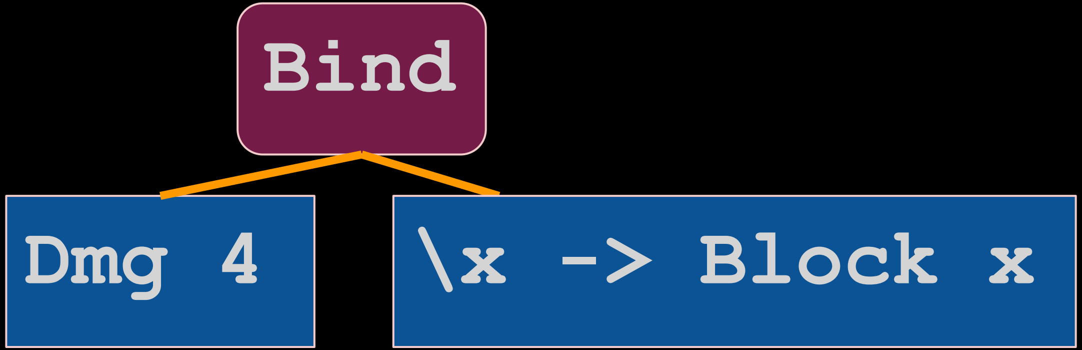
Data Types IV



```
Bind
Dmg 4
\x -> Block x
```

```
:: Card Int
```

Data Types IV



```
Bind
Dmg 4
\x -> Block x
```

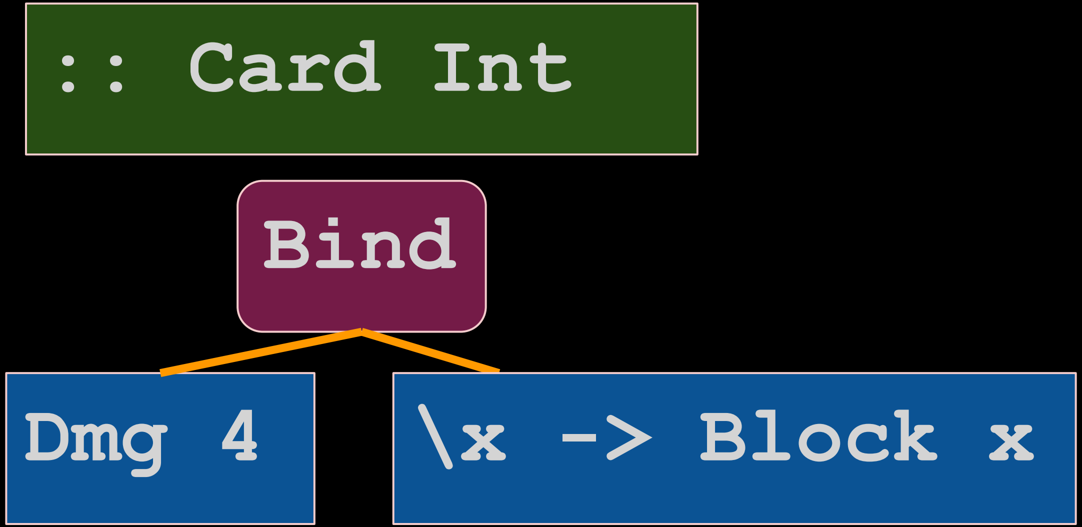
```
:: Card Int
```

```
:: Int ->
Card Int
```

Data Types IV

2
Reaper
Attack

Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.



```
:: Card Int
Bind
Dmg 4
\x -> Block x
```

Data Types IV

```
apply ::
 Card a -> State -> (a, State)
apply (Dmg x) s = (x, <new state>)
apply (Block x) s = (x, <new state>)
```

Data Types IV

```
apply ::
 Card a -> State -> (a, State)
apply (Dmg x) s = (x, <new state>)
apply (Block x) s = (x, <new state>)
```

Data Types IV

```
apply ::
 Card a -> State -> (a, State)
apply (...) = (...)
apply (Bind c1 c2) s = let
 (a, after1) = apply c1 s
 in apply (c2 a) after1
```

Data Types IV

```
apply ::
 Card a -> State -> (a, State)
apply (...) = (...)
apply (Bind c1 c2) s = let
  (a, after1) = apply c1 s
 in apply (c2 a) after1
```

eg.
damage dealt

Data Types IV

2
Reaper
Attack

Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.

```
apply :: Card a -> State -> (a, State)
apply (...) = (...)



> apply
    (Bind (Dmg 4) (\x -> Block x))
    (10, 10)
(4,(6,14))
```

Data Types IV

```
desc :: Card a -> String
desc (Dmg x) = [i|deal #{x} damage|]
desc (Block x) = [i|gain #{x} block|]
```

Data Types IV

```
desc :: Card a -> String
desc (...) = (...)
desc (Bind c1 c2) =
 [i|#{desc c1}, then ?|]
```

Data Types IV

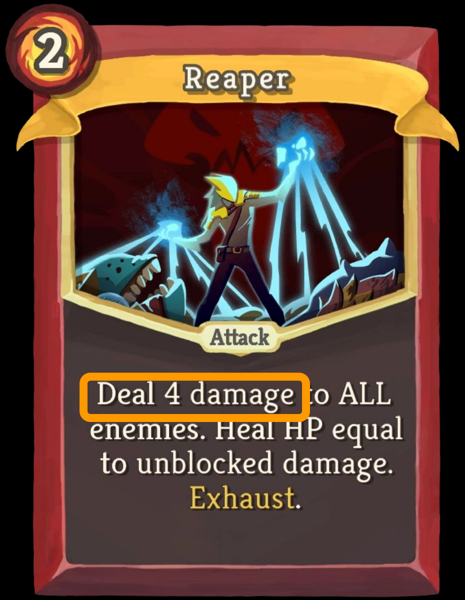
```
desc :: Card a -> String
desc (...) = (...)
desc (Bind c1 c2) =
 [i|#{desc c1}, then ?|]
```

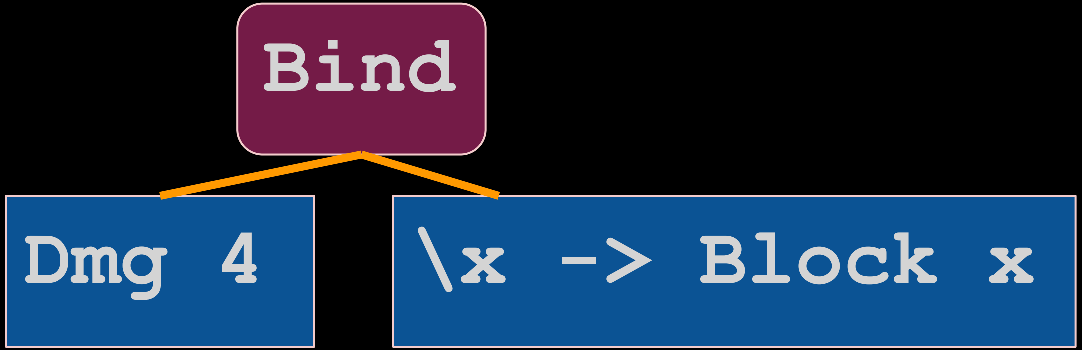
Data Types IV

2
Reaper
Attack
Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.

```
desc :: Card a -> String
desc (...) = (...)
desc (Bind c1 c2) =
  [i|#{desc c1}, then ?|]


> desc (Bind (Dmg 4) (\x -> Block x))
"deal 4 damage, then ?"
```

Data Types IV


2
Reaper
Attack
Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.



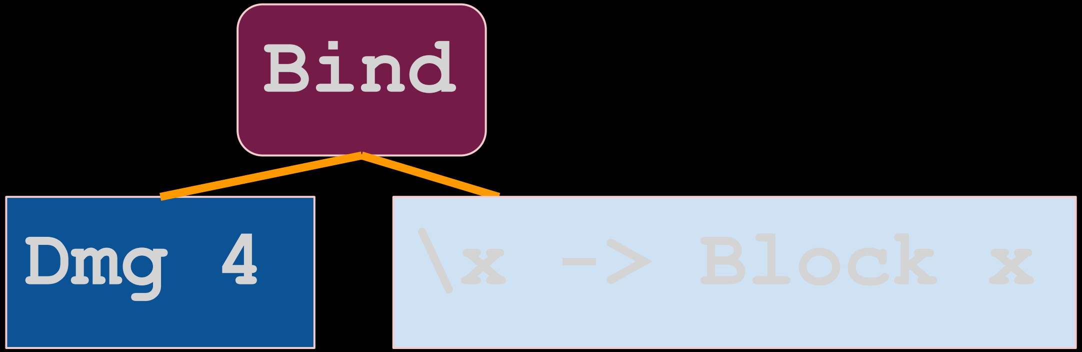
Bind

Dmg 4

\x -> Block x

Data Types IV

2
Reaper
Attack
Deal 4 damage to ALL



```
Bind
Dmg 4
\x -> Block x
```

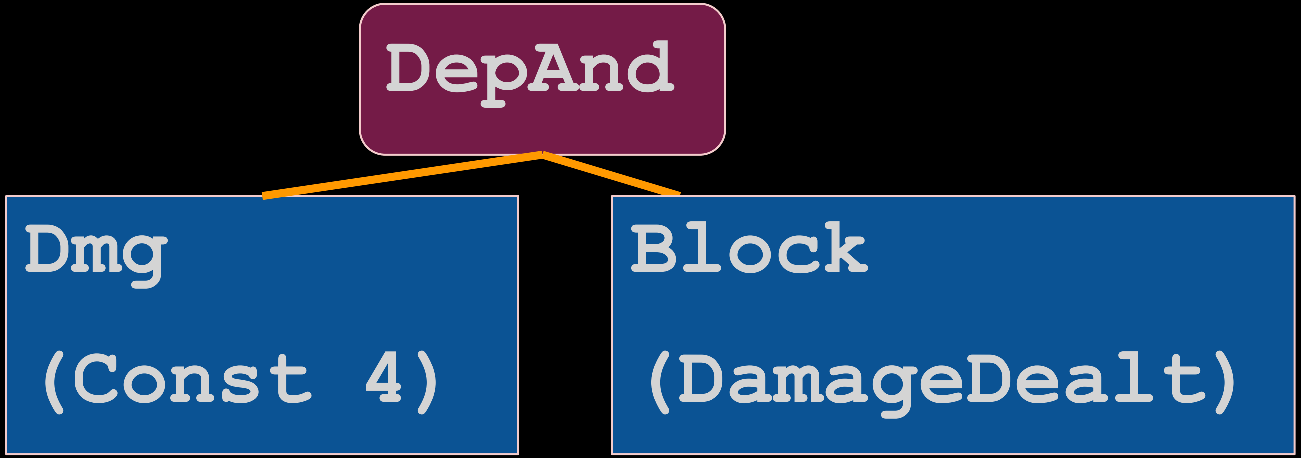
Data Types V

```
data Card a b where
 Dmg :: From a Int -> Card a Int
 Block :: From a Int -> Card a Int
 DepAnd :: Card () a -> Card a b
        -> Card () b

data From i o where
 Const :: a -> From () a
 DamageDealt :: From Int Int
```

Data Types V

2
Reaper
Attack

Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.

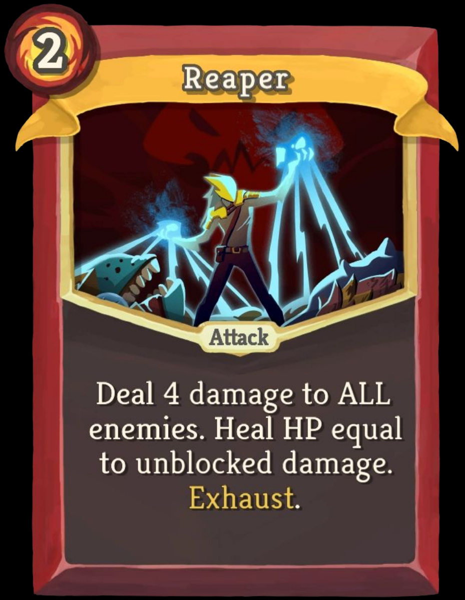


DepAnd

Dmg (Const 4)

Block (DamageDealt)

Data Types V



Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.

```
> apply (DepAnd (Dmg (Const 4))
      (Block (DamageDealt))) (10, 10)
(4,(6,14))
```

```
> desc (DepAnd (Dmg (Const 4)) (Block
      (DamageDealt)))
"deal 4 damage, then block equal to
damage dealt"
```

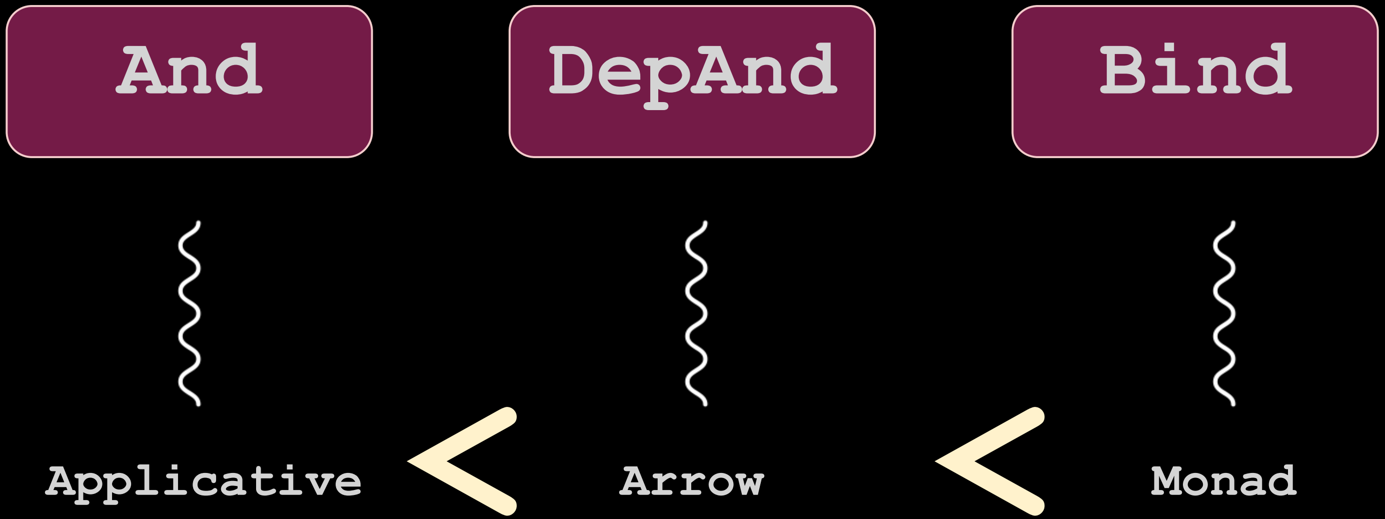
Relation

`And`

`DepAnd`

`Bind`

Relation



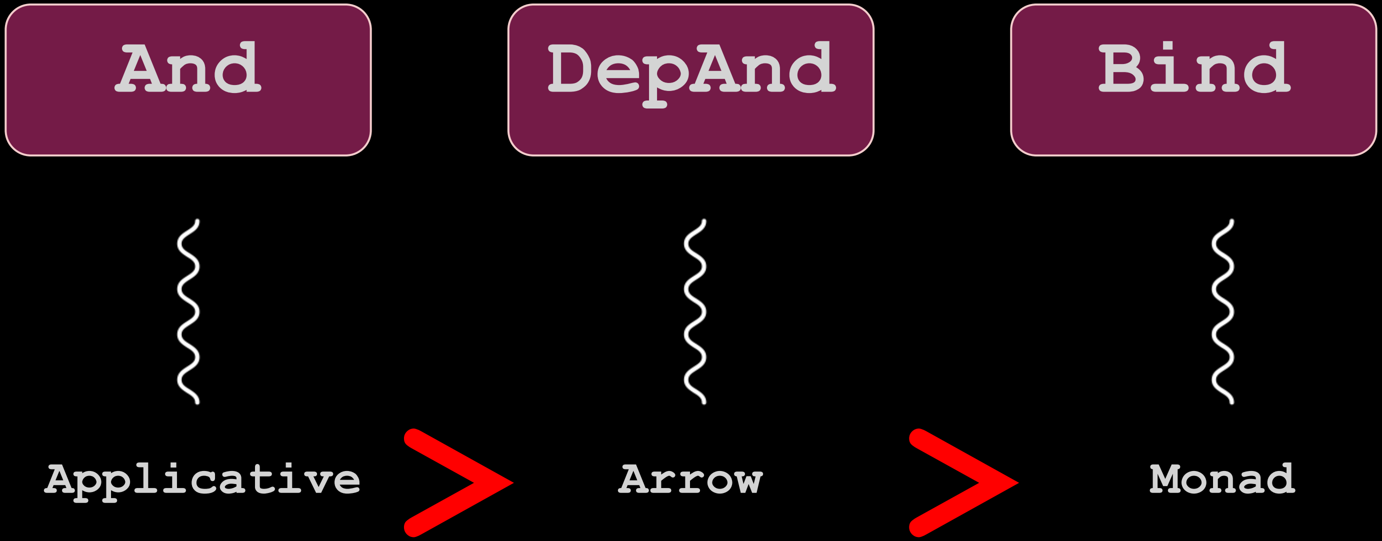
And

DepAnd

Bind

Applicative < Arrow < Monad

Relation



And

DepAnd

Bind

Applicative > Arrow > Monad

Effect Handlers

Handler Languages/Libraries

As Library:

Scala

Idris

…

As Language Feature:

EFF

koka

OCaml

…

Handler Languages/Libraries

As Library:

Scala

Idris

...

As Language Feature:

EFF

koka

OCaml

...

Handlers

```
data Card a where
 Dmg :: Int -> Card Int
 Block :: Int -> Card Int
```

```
effect Dmg: int -> int
effect Block: int -> int
```

EFF

Handlers

```
data Card a where
 Dmg :: Int -> Card Int
 Block :: Int -> Card Int
```

```
effect Dmg: int -> int
effect Block: int -> int
```

EFF

Handlers

1
Strike
Attack
Deal 6 damage.

```
perform (Dmg 6)
```

Handlers

1
Strike
Attack
Deal 6 damage.

```
perform (Dmg 6)
```

```
error: uncaught effect ‘Dmg 6’.
```

Handlers

1
Strike
Attack
Deal 6 damage.

```
handle
 perform (Dmg 6)
with
  | effect (Dmg x) k -> ...
```

Handlers

1
Strike
Attack
Deal 6 damage.

```
handle
 perform (Dmg 6)
with
  | effect (Dmg x) k ->
     "deal #{x} damage"
```

Handlers

1
Strike
Attack
Deal 6 damage.

```
handle
 perform (Dmg 6)
with
  | effect (Dmg x) k -> (fun s
    -> (continue k x) <new s>)
```

Handlers

```
apply (Dmg 1) where
 apply (Dmg x) s = <new s>
```

```
handle
 perform (Dmg 1)
with
 | effect (Dmg x) k -> ...
```

EFF

Handlers

```
apply (Dmg 1) where
 apply (Dmg x) s = <new s>
```

```
handle
 perform (Dmg 1);
 perform (Block 1)
with
 | effect (Dmg x) k -> ...
```

EFF

Handlers

```
apply (Dmg 1) where
 apply (Dmg x) s = <new s>
```

```
handle
 perform (Dmg 1);
 perform (Block 1)
with
 | effect (Dmg x) k -> ...
 | effect (Block x) k -> ...
```

EFF

Handlers

1
Iron Wave
Attack
Gain 5 Block.
Deal 5 damage.

```
handle
 perform (Dmg 1);
 perform (Block 1)
with
  | effect (Dmg x) k -> (fun s
    -> (continue k x) <new s>)
  | effect (Block x) k -> (fun s
    -> (continue k x) <new s>)
  | x -> (fun s -> s)
```

Handlers

```
f = handle
 perform (Dmg 1);
 perform (Block 1)
with
  | effect (Dmg x) k -> (fun s
    -> <new s>)
  | effect (Block x) k -> (fun s
    -> <new s>)
  | x -> (fun s -> s)
```

```
> f (5, 5)
```

Handlers

```
f = handle
 perform (Dmg 1);
 perform (Block 1)
with
 | effect (Dmg x) k -> (fun s
   -> <new s>)
 | effect (Block x) k -> (fun s
   -> <new s>)
 | x -> (fun s -> s)
```

```
> f (5, 5)
(4,5)
```

Handlers

```
f = handle
 perform (Dmg 1);
 perform (Block 1)
with
 | effect (Dmg x) k -> (fun s
   -> <new s>)
 | effect (Block x) k -> (fun s
   -> <new s>)
 | x -> (fun s -> s)
```

```
> f (5, 5)
(4,5)
```

Handlers

```
f = handle
 perform (Dmg 1);
 perform (Block 1)    ?
with
  | effect (Dmg x) k -> (fun s
    -> <new s>)
  | effect (Block x) k -> (fun s
    -> <new s>)
  | x -> (fun s -> s)
```

```
> f (5, 5)
(4, 5)
```

Handlers

```
f = handle
  perform (Dmg 1);
  perform (Block 1)
with
  | effect (Dmg x) k -> (fun s
    -> (continue k x) <new s>)
  | effect (Block x) k -> (fun s
    -> (continue k x) <new s>)
  | x -> (fun s -> s)
```

```
> f (5, 5)
```

Handlers

```
f = handle
 perform (Dmg 1);
 perform (Block 1)
with
 | effect (Dmg x) k -> (fun s
   -> (continue k x) <new s>)
 | effect (Block x) k -> (fun s
   -> (continue k x) <new s>)
 | x -> (fun s -> s)
```

```
> f (5, 5)
(4,5)
```

Handlers

```
f = handle
 perform (Dmg 1);
 perform (Block 1)
with
  | effect (Dmg x) k -> (fun s
    -> (continue k x) <new s>)
  | effect (Block x) k -> (fun s
    -> (continue k x) <new s>)
  | x -> (fun s -> s)
```

```
> f (5, 5)
```

~~(4,5)~~

Handlers

```
f = handle
 perform (Dmg 1);
 perform (Block 1)
with
  | effect (Dmg x) k -> (fun s
    -> (continue k x) <new s>)
  | effect (Block x) k -> (fun s
    -> (continue k x) <new s>)
  | x -> (fun s -> s)
```

```
> f (5, 5)
```

~~(4,5)~~

(4,6)

Handlers

```
f = handle
 perform (Dmg 1);
 perform (Block 1)
with
  | effect (Dmg x) k -> (fun s
    -> (continue k x) <new s>)
  | effect (Block x) k -> (fun s
    -> (continue k x) <new s>)
  | x -> (fun s -> s)
```

```
> f (5, 5)
```

~~(4,5)~~

~~(4,6)~~

Handlers

```
f = handle
 perform (Dmg 1);
 perform (Block 1)
with
  | effect (Dmg x) k -> (fun s
    -> (continue k x) <new s>)
  | effect (Block x) k -> (fun s
    -> (continue k x) <new s>)
  | x -> (fun s -> s)
```

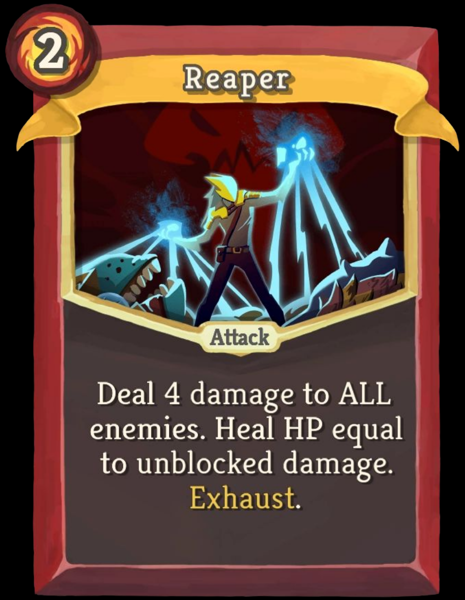
```
> f (5, 5)
```

~~(4,5)~~

~~(4,6)~~

(4,6)

Handlers


2
Reaper
Attack
Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.

```
handler
  | effect (Dmg x) k ->
    "deal #{x} damage, and then ?"
  | effect (Block x) k ->
    "block #{x}, and then ?"
```

Handlers

2
Reaper
Attack

Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.

```
handler
  | effect (Dmg x) k ->
    "deal #{x} damage, and then ?"
  | effect (Block x) k ->
    "block #{x}, and then ?"
```

Handlers

```
data Card a where
 Dmg :: Int -> Card Int
 Block :: Int -> Card Int
 Bind :: Card a
      -> (a -> Card b)
      -> Card b
```

```
| effect (Dmg x) k -> (fun s
   -> (continue k x) <new s>)
| effect (Block x) k -> (fun s
   -> (continue k x) <new s>)
```

EFF

Handlers

```
data Card
  = Dmg Int (Int -> Card)
  | Block Int (Int -> Card)
```

```
| effect (Dmg x) k -> (fun s
   -> (continue k x) <new s>)
| effect (Block x) k -> (fun s
   -> (continue k x) <new s>)
```

EFF

Handlers

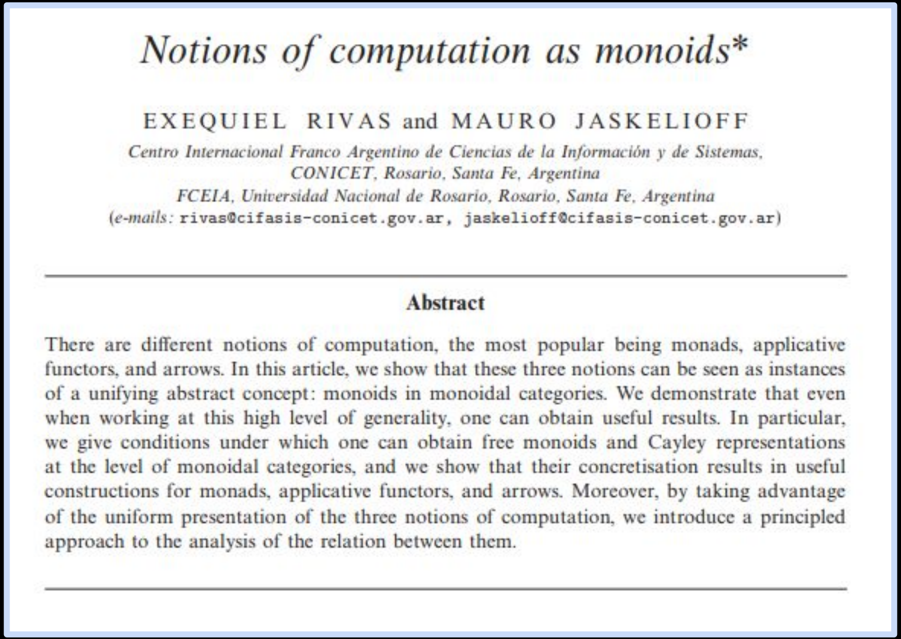
```
data Card a
 = Dmg Int (Int -> Card a)
 | Block Int (Int -> Card a)
 | Return a
```

```
| effect (Dmg x) k -> (fun s
  -> (continue k x) <new s>)
| effect (Block x) k -> (fun s
  -> (continue k x) <new s>)
| x -> (fun s -> s)
```

EFF

Non-Monadic Effect Handlers

Non-Monadic Handlers



*Notions of computation as monoids**

EXEQUIEL RIVAS and MAURO JASKELIOFF

Centro Internacional Franco Argentino de Ciencias de la Información y de Sistemas,
CONICET, Rosario, Santa Fe, Argentina
FCEIA, Universidad Nacional de Rosario, Rosario, Santa Fe, Argentina
(*e-mails:* rivas@cifasis-conicet.gov.ar, jaskelioff@cifasis-conicet.gov.ar)

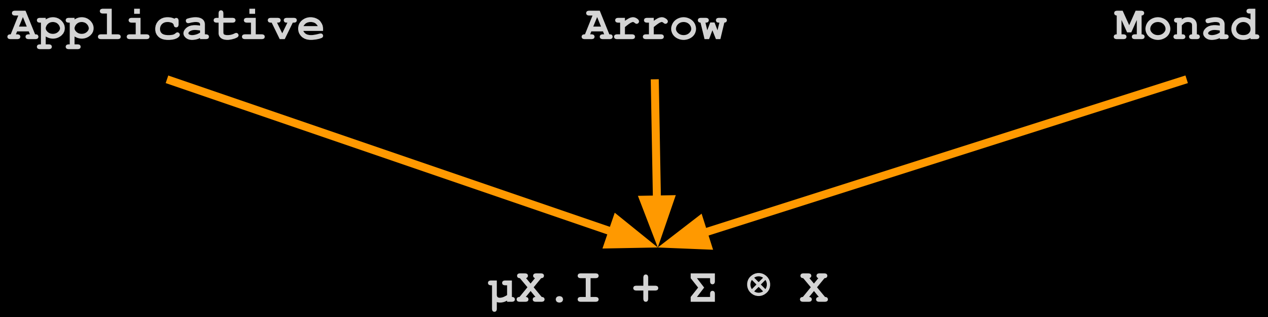
Abstract

There are different notions of computation, the most popular being monads, applicative functors, and arrows. In this article, we show that these three notions can be seen as instances of a unifying abstract concept: monoids in monoidal categories. We demonstrate that even when working at this high level of generality, one can obtain useful results. In particular, we give conditions under which one can obtain free monoids and Cayley representations at the level of monoidal categories, and we show that their concretisation results in useful constructions for monads, applicative functors, and arrows. Moreover, by taking advantage of the uniform presentation of the three notions of computation, we introduce a principled approach to the analysis of the relation between them.

Non-Monadic Handlers

`Applicative` `Arrow` `Monad`

Non-Monadic Handlers



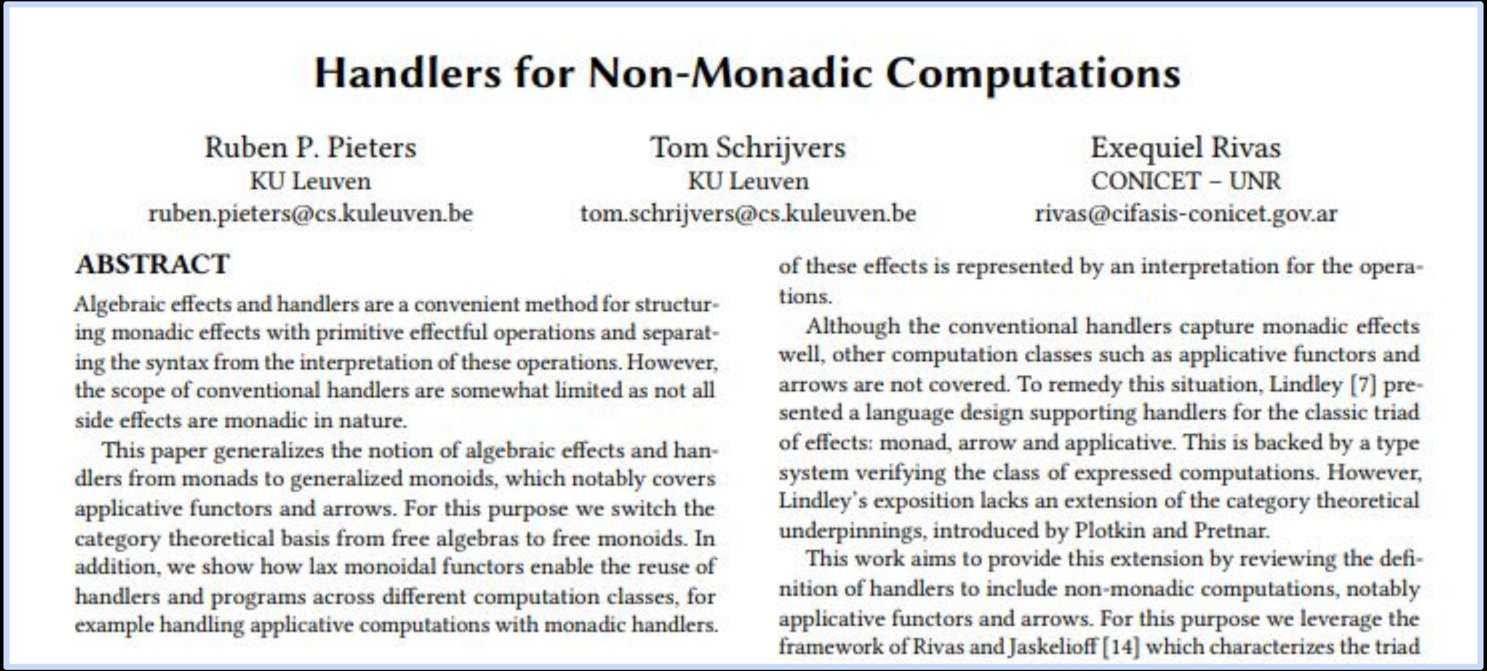
Applicative

Arrow

Monad

$\mu X. I + \Sigma \otimes X$

Non-Monadic Handlers



Handlers for Non-Monadic Computations

Ruben P. Pieters
KU Leuven
ruben.pieters@cs.kuleuven.be

Tom Schrijvers
KU Leuven
tom.schrijvers@cs.kuleuven.be

Exequiel Rivas
CONICET – UNR
rivas@cifasis-conicet.gov.ar

ABSTRACT

Algebraic effects and handlers are a convenient method for structuring monadic effects with primitive effectful operations and separating the syntax from the interpretation of these operations. However, the scope of conventional handlers are somewhat limited as not all side effects are monadic in nature.

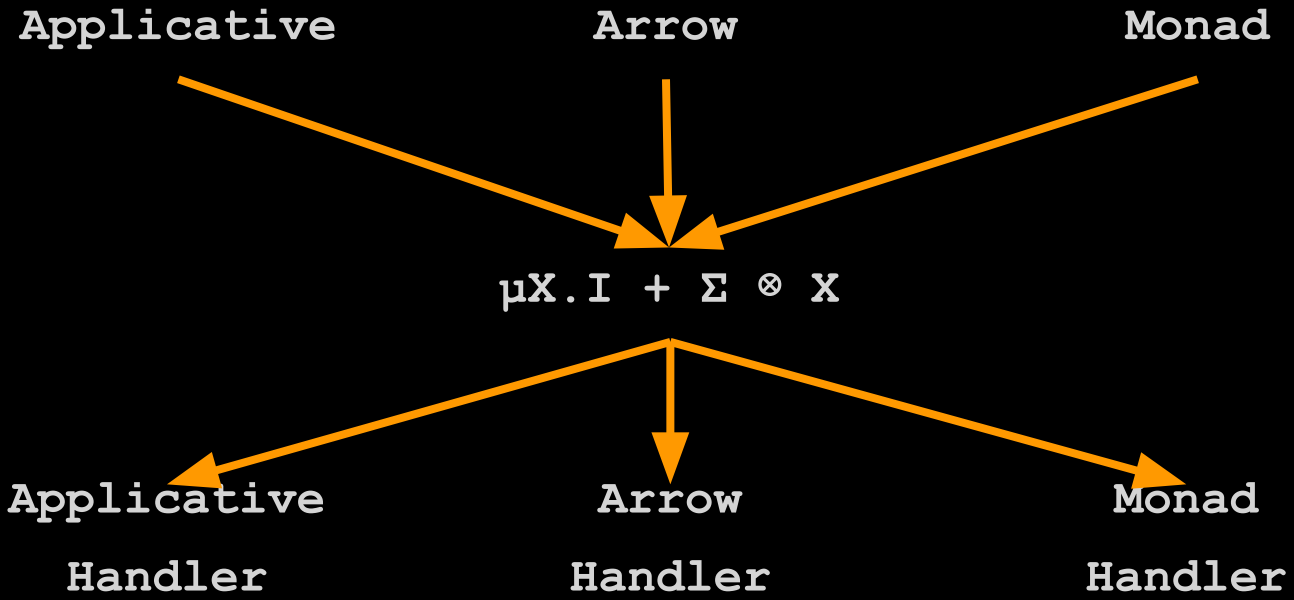
This paper generalizes the notion of algebraic effects and handlers from monads to generalized monoids, which notably covers applicative functors and arrows. For this purpose we switch the category theoretical basis from free algebras to free monoids. In addition, we show how lax monoidal functors enable the reuse of handlers and programs across different computation classes, for example handling applicative computations with monadic handlers.

of these effects is represented by an interpretation for the operations.

Although the conventional handlers capture monadic effects well, other computation classes such as applicative functors and arrows are not covered. To remedy this situation, Lindley [7] presented a language design supporting handlers for the classic triad of effects: monad, arrow and applicative. This is backed by a type system verifying the class of expressed computations. However, Lindley's exposition lacks an extension of the category theoretical underpinnings, introduced by Plotkin and Pretnar.

This work aims to provide this extension by reviewing the definition of handlers to include non-monadic computations, notably applicative functors and arrows. For this purpose we leverage the framework of Rivas and Jaskelioff [14] which characterizes the triad

Non-Monadic Handlers



Applicative

Arrow

Monad

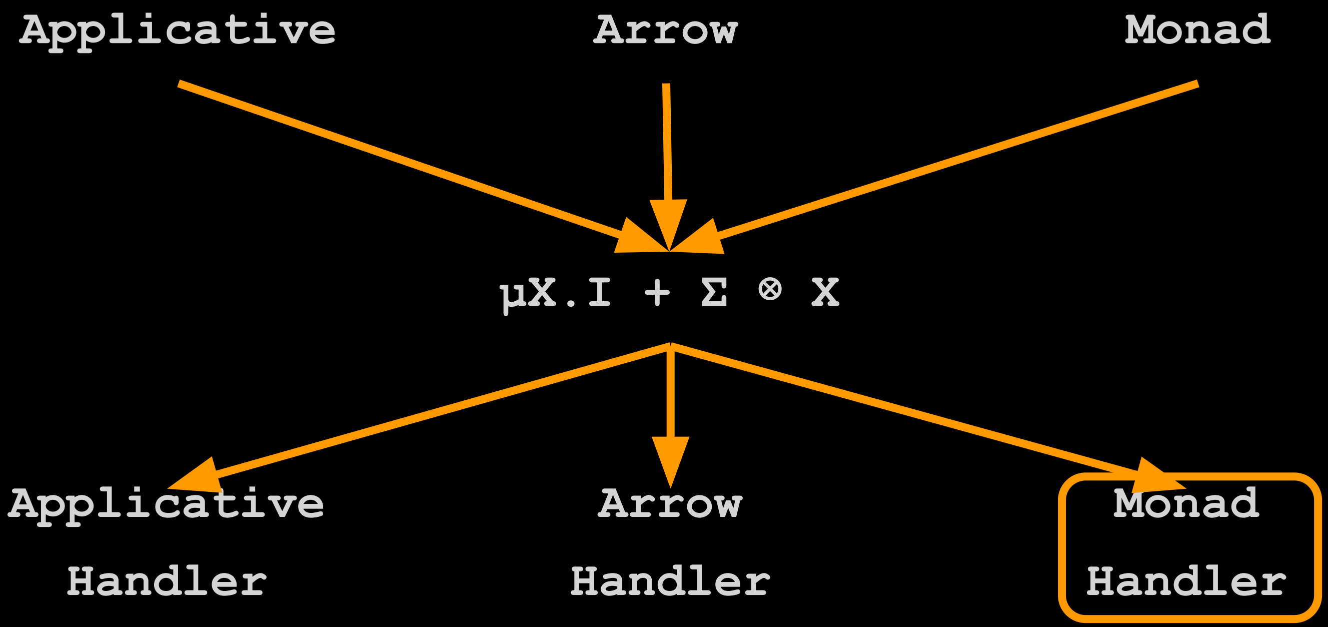
$\mu X . I + \Sigma \otimes X$

Applicative Handler

Arrow Handler

Monad Handler

Non-Monadic Handlers



Applicative

Arrow

Monad

$\mu X. I + \Sigma \otimes X$

Applicative Handler

Arrow Handler

Monad Handler

Non-Monadic Handlers

```
data Card a where
 Dmg :: Int -> Card Int
 Block :: Int -> Card Int
 Bind :: Card a
      -> (a -> Card b)
      -> Card b
 Return :: a -> Card a
```

```
| effect (Dmg x) k -> ...
| effect (Block x) k -> ...
```

EFF

Non-Monadic Handlers

Applicative

Arrow

```
| effect (Dmg x) f k -> ...
```

Monad

```
| effect (Dmg x) k -> ...
```

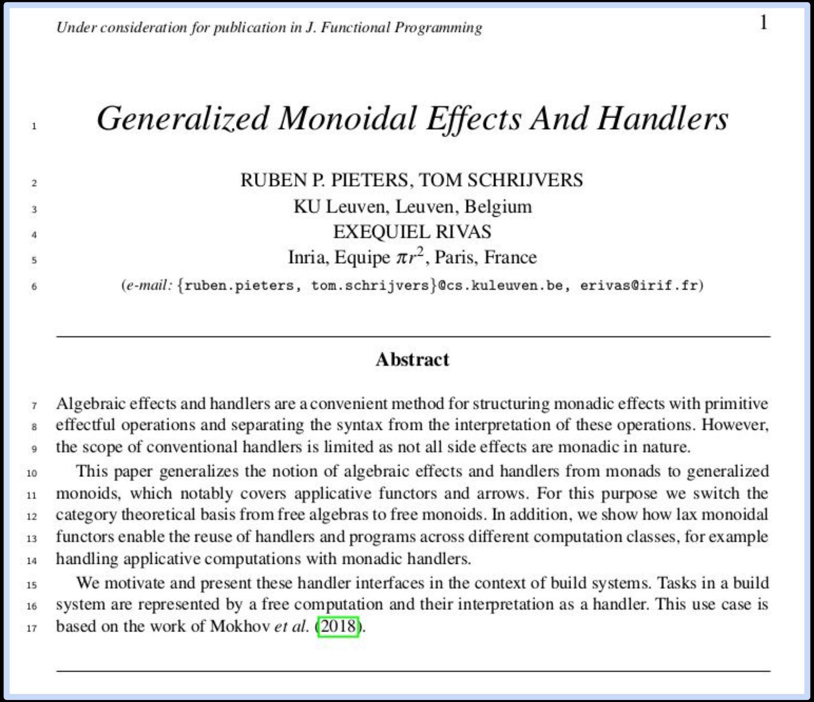
Non-Monadic Handlers

2
Reaper
Attack

Deal 4 damage to ALL enemies. Heal HP equal to unblocked damage. Exhaust.

```
handler
  | effect (Dmg x) f k ->
    "deal #{x} damage, and then #{k}"
  | effect (Block x) f k ->
    "block #{x}, and then #{k}"
```

Non-Monadic Handlers



Under consideration for publication in J. Functional Programming 1

Generalized Monoidal Effects And Handlers

RUBEN P. PIETERS, TOM SCHRIJVERS
KU Leuven, Leuven, Belgium
EXEQUIEL RIVAS
Inria, Equipe πr^2, Paris, France
(*e-mail:* {`ruben.pieters, tom.schrijvers`}`@cs.kuleuven.be, erivas@irif.fr`)

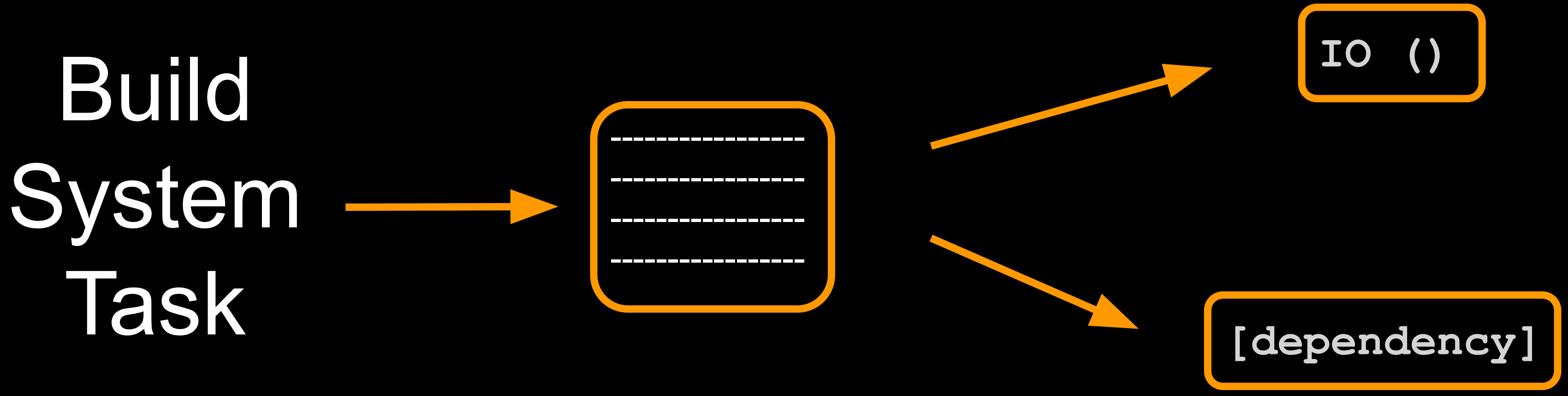
Abstract

Algebraic effects and handlers are a convenient method for structuring monadic effects with primitive effectful operations and separating the syntax from the interpretation of these operations. However, the scope of conventional handlers is limited as not all side effects are monadic in nature.

This paper generalizes the notion of algebraic effects and handlers from monads to generalized monoids, which notably covers applicative functors and arrows. For this purpose we switch the category theoretical basis from free algebras to free monoids. In addition, we show how lax monoidal functors enable the reuse of handlers and programs across different computation classes, for example handling applicative computations with monadic handlers.

We motivate and present these handler interfaces in the context of build systems. Tasks in a build system are represented by a free computation and their interpretation as a handler. This use case is based on the work of Mokhov *et al.* (2018).

Non-Monadic Handlers



Build Systems à la Carte, A. Mokhov et al.

Conclusion

Conclusion

```haskell
data Card a where
 Dmg :: Int -> Card Int
 Block :: Int -> Card Int
```

```
effect Dmg: int -> int
effect Block: int -> int
```

EFF

Conclusion

```
data Card a where
 Dmg :: Int -> Card Int
 Block :: Int -> Card Int
```

Monad

Applicative

Arrow

```
effect Dmg: int -> int
effect Block: int -> int
```

Conclusion

```
data Card a where
 Dmg :: Int -> Card Int
 Block :: Int -> Card Int
```

```
effect Dmg: int -> int
effect Block: int -> int

| effect (Dmg x) k -> ...
```

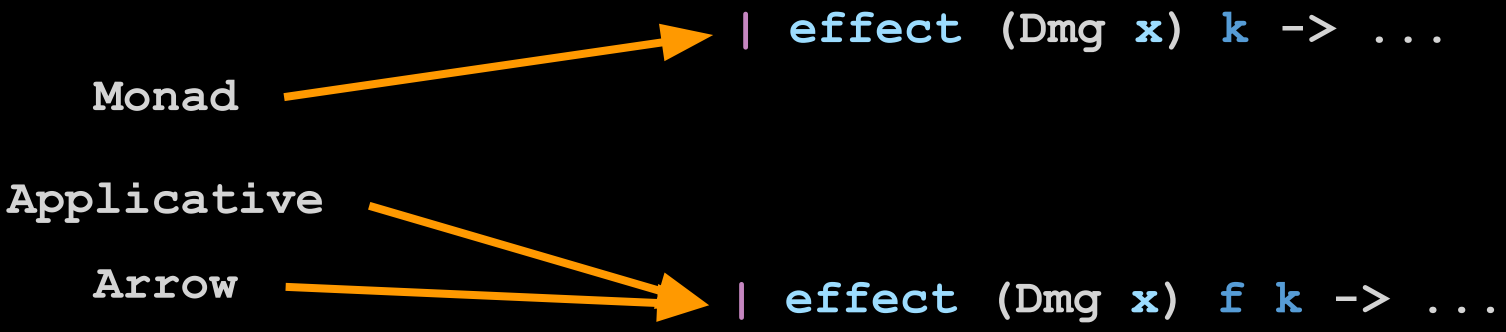
```
        Monad

     Applicative

        Arrow
```

EFF

Conclusion

```
data Card a where
 Dmg :: Int -> Card Int
 Block :: Int -> Card Int
```

```
effect Dmg: int -> int
effect Block: int -> int
```



Monad

```
| effect (Dmg x) k -> ...
```

Applicative

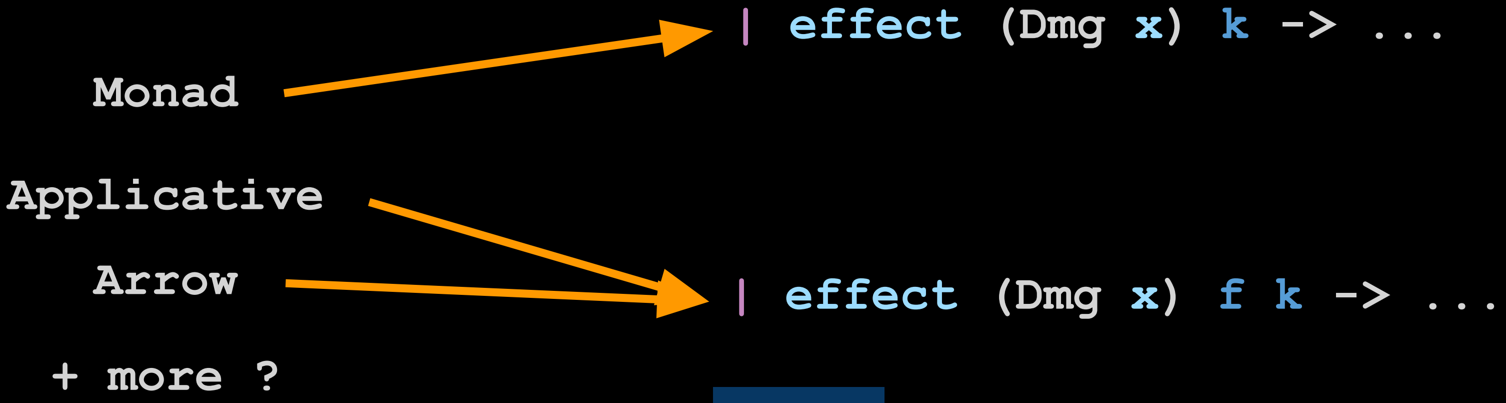
Arrow

```
| effect (Dmg x) f k -> ...
```

Conclusion

```
data Card a where
 Dmg :: Int -> Card Int
 Block :: Int -> Card Int
```

```
effect Dmg: int -> int
effect Block: int -> int
```



Monad

```
| effect (Dmg x) k -> ...
```

Applicative

Arrow

```
| effect (Dmg x) f k -> ...
```

+ more ?

EFF