KU LEUVEN

How To Go Eff Without A Hitch

On Efficient Compilation From Eff to OCaml

Stien Vanderhallen
Georgios Karachalias
Tom Schrijvers

EFF

= programming language with algebraic effects

EFF

= programming language with algebraic effects

Computational Effects

Exception throwing	Global state manipulation	I/O
`throw`	`get`	`read`
`try/catch`	`set`	`print`

Computational Effects

Exception throwing	Global state manipulation	I/O
`throw`	`get`	`read`
`try/catch`	`set`	`print`

`operations`: cause impure (effectful) behaviour

Computational Effects

Exception throwing	Global state manipulation	I/O
`throw`	`get`	`read`
`try/catch`	`set`	`print`

`operations`: cause impure (effectful) behaviour

`handlers`: capture impure (effectful) behaviour

EFF

= programming language with algebraic effects

Computational Effects

Exception throwing	Global state manipulation	I/O
`throw`	`get`	`read`
`try/catch`	`set`	`print`

`operations`: cause impure (effectful) behaviour

`handlers`: capture impure (effectful) behaviour

Algebraic Computational Effects

Exception throwing	Global state manipulation	I/O
`throw`	`get`	`read`
~~`try/catch`~~	`set`	`print`
`handle … with …`	`handle … with …`	`handle … with …`

see also: M. Pretnar, *An Introduction to Algebraic Effect Handlers,* MFPS 2015.

Algebraic Computational Effects

Exception throwing	Global state manipulation	I/O
`throw`	`get`	`read`
`try-catch`		
`handle … with …`	`handle … with …`	`handle … with …`

HITCH : how to compile?

see also: M. Pretnar, *An Introduction to Algebraic Effect Handlers,* MFPS 2015.

Algebraic Computational Effects

Exception throwing	Global state manipulation	I/O
`throw`	`get`	`read`
~~`try/catch`~~	`set`	`print`
`handle with …`	`handle with …`	`handle with …`

HITCH : how to compile Eff?

see also: M. Pretnar, *An Introduction to Algebraic Effect Handlers,* MFPS 2015.

What the Eff?

```
effect flip_a_coin : unit -> bool;;

let coin_flipper = handler
    | #flip_a_coin () k -> k true
    | return x -> x
in

handle (if #flip_a_coin ()
            then 1
            else 2 + 3 )
with coin_flipper
```

What the Eff?

```
effect flip_a_coin : unit -> bool;;
```

Effect definition

```
let coin_flipper = handler
    | #flip_a_coin () k -> k true
    | return x -> x
in

handle (if #flip_a_coin ()
           then 1
           else 2 + 3 )
with coin_flipper
```

What the Eff?

```
effect flip_a_coin : unit -> bool;;
```

Effect definition

```
let coin_flipper = handler
    | #flip_a_coin () k -> k true
    | return x -> x
in
```

Handler definition

```
handle (if #flip_a_coin ()
           then 1
           else 2 + 3 )
with coin_flipper
```

What the Eff?

```
effect flip_a_coin : unit -> bool;;
```

Effect definition

```
let coin_flipper = handler
    | #flip_a_coin () k -> k true
    | return x -> x
in
```

Handler definition

```
handle (if #flip_a_coin ()
           then 1
           else 2 + 3 )
with coin_flipper
```

Computation

What the Eff?

```
effect flip_a_coin : unit -> bool;;
```

Effect definition

```
let coin_flipper = handler
    | #flip_a_coin () k -> k true
    | return x -> x
in
```

Handler definition

```
handle (if #flip_a_coin ()
           then 1
           else 2 + 3 )
with coin_flipper
```

Computation

What the Eff?

```
effect flip_a_coin : unit -> bool;;
```

Effect definition

```
let coin_flipper = handler
    | #flip_a_coin () k -> k true
    | return x -> x
in
```

Handler definition

```
handle (if #flip_a_coin ()
           then 1
           else 2 + 3 )
with coin_flipper
```

Computation

What the Eff?

```
effect flip_a_coin : unit -> bool;;
```

Effect definition

```
let coin_flipper = handler
    | #flip_a_coin () k -> k true
    | return x -> x
in
```

Handler definition

```
handle (if #flip_a_coin ()
           then 1
           else 2 + 3 )
with coin_flipper
```

Computation

What the Eff?

```
effect flip_a_coin : unit -> bool;;
```

Effect definition

```
let coin_flipper = handler
    | #flip_a_coin () k -> k true
    | return x -> x
in
```

Handler definition

```
handle (if #flip_a_coin ()
           then 1
           else 2 + 3 )
with coin_flipper
```

Computation

What the Eff?

```
effect flip_a_coin : unit -> bool;;
```

Effect definition

```
let coin_flipper = handler
    | #flip_a_coin () k -> k true
    | return x -> x
in
```

Handler definition

```
handle (if #flip_a_coin ()
           then 1
           else 2 + 3 )
with coin_flipper
```

Computation

What the Eff?

```
effect flip_a_coin : unit -> bool;;
```

Effect definition

```
let coin_flipper = handler
    | #flip_a_coin () k -> k true
    | return x -> x
in
```

Handler definition

```
handle (if #flip_a_coin ()
           then 1
           else 2 + 3 )
with coin_flipper
```

Computation

→1

What the Eff?

```
effect flip_a_coin : unit -> bool;;
```

Effect definition

```
let coin_flipper = handler
  | #flip_a_coin () k -> k true
  | return x -> x
in
```

Handler definition

```
handle (if #flip_a_coin ()
          then 1
          else 2 + 3 )
with coin_flipper
```

Computation

HITCH : how to compile?

What the Eff?

```
effect flip_a_coin : unit -> bool;;

let coin_flipper = handler
  | #flip_a_coin () k -> k true
  | return x -> x
in
handle (if #flip_a_coin () then 1 else 2 + 3)
with coin_flipper
```

Effect definition

Handler definition

Computation

HITCH : how to compile?

HITCH : how to compile efficiently?

Naive compilation

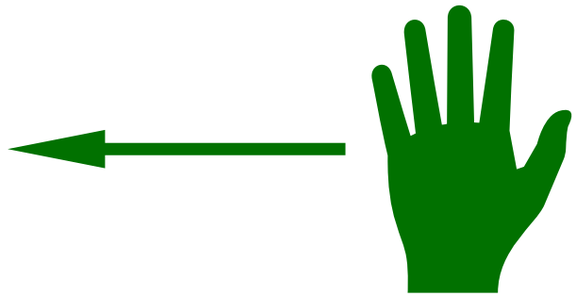
Eff

```
if #flip_a_coin ()
    then 1
    else 2 + 3
```

Naive compilation

Eff

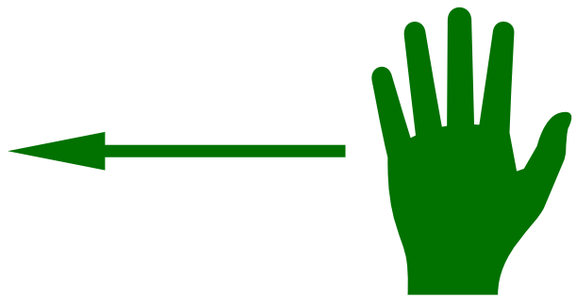
```
if #flip_a_coin ()
    then 1
    else 2 + 3
```



Naive compilation

Eff

```
if #flip_a_coin ()
    then 1
    else 2 + 3
```



SEQUENCING

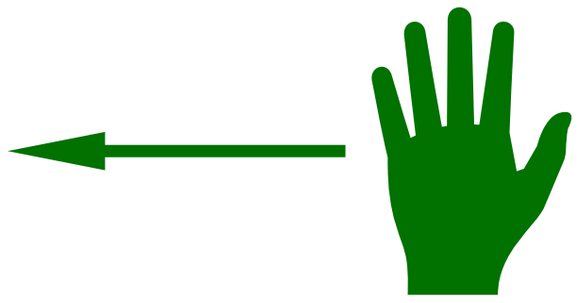
OCaml

```
flip_a_coin () >>= \x ->
if x then (return 1) else
     (+) 3 >>= \p ->
      p  2 >>= \r ->
      return r
```

Naive compilation

Eff

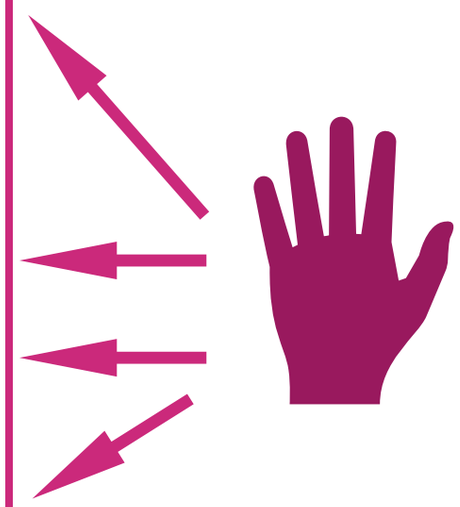
```
if #flip_a_coin ()
    then 1
    else 2 + 3
```



SEQUENCING

OCaml

```
flip_a_coin () >>= \x ->
if x then (return 1) else
     (+) 3 >>= \p ->
      p  2 >>= \r ->
      return r
```



Naive compilation

Eff

```
if #flip_a_coin ()
    then 1
```

HITCH : expensive!

OCaml

```
flip_a_coin () >>= \x ->
if x then (return 1) else
     (+) 3 >>= \p ->
      p  2 >>= \r ->
      return r
```

Naive compilation

Eff

```
if #flip_a_coin ()
    then 1
```

HITCH : unnecessary!

OCaml

```
flip_a_coin () >>= \x ->
if x then (return 1) else
     (+) 3 >>= \p ->
      p  2 >>= \r ->
      return r
```

Naive compilation

Eff

```
if #flip_a_coin ()
    then 1
    else 2 + 3
```

SEQUENCING

OCaml

```
flip_a_coin () >>= \x ->
if x then (return 1) else
    let r = 2 + 3
    in return r
```

Naive compilation

```
if #flip_a_coin ()
then return 1
else 2 + 3
```

Eff

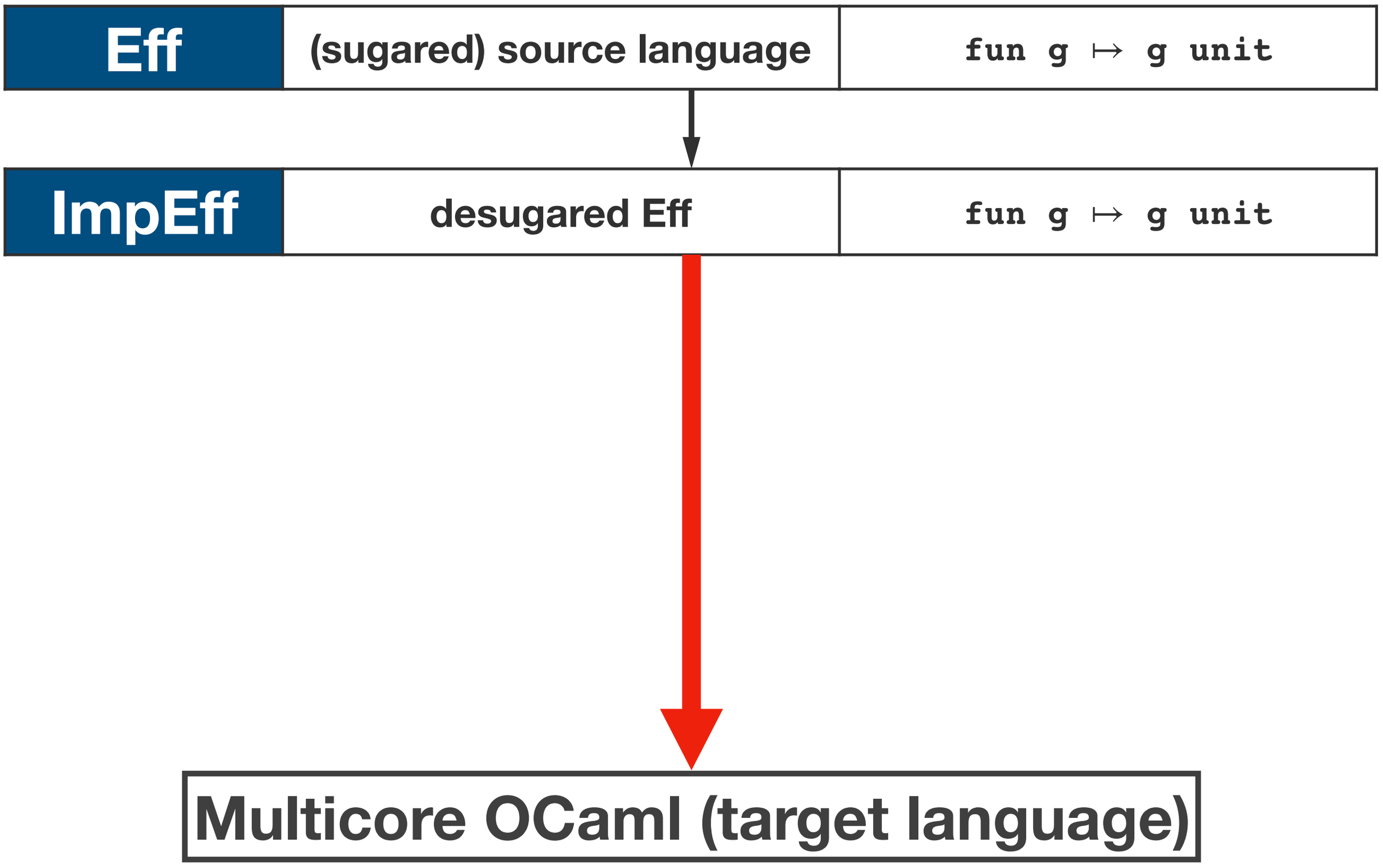
SOLUTION : keep track of effect information

SEQUENCING

OCaml

```
flip_a_coin () >>= \x ->
if x then (return 1) else
    let r = 2 + 3
    in return r
```

Original Approach



Current Approach

Eff	(sugared) source language	`fun g ↦ g unit`
ImpEff	desugared Eff	`fun g ↦ g unit`
ExEff	**explicit optimisable (unnecessarily) elaborate**	$\Lambda\alpha\,.\,\Lambda\alpha'\,.\,\Lambda\delta\,.\,\Lambda\delta'\,.\,\Lambda(\omega : \alpha \leqslant \alpha')\,.$ $\Lambda(\omega' : \delta \leqslant \delta')\,.$ **fun (g : Unit** $\mapsto$ $\alpha!\delta$**)** $\mapsto$ **(g unit)** $\triangleright$ $(\omega!\omega')$
SkelEff	**erased ExEff**	$\Lambda\varsigma\,.$ **fun** (g : Unit $\mapsto$ ς) $\mapsto$ **g unit**

Multicore OCaml (target language)

see also: A. H. Saleh et al., *Explicit Effect Subtyping,* ESOP 2018.

Current Approach

Eff	**(sugared) source language**	`fun g ↦ g unit`
ImpEff	**desugared Eff**	`fun g ↦ g unit`
ExEff	**explicit optimisable (unnecessarily) elaborate**	$\Lambda\alpha\,.\,\Lambda\alpha'\,.\,\Lambda\delta\,.\,\Lambda\delta'\,.\,\Lambda(\omega : \alpha \leqslant \alpha')\,.$ $\Lambda(\omega' : \delta \leqslant \delta')\,.$ **fun (g : Unit** $\mapsto$ $\alpha!\delta$**)** $\mapsto$ **(g unit)** $\triangleright$ $(\omega!\omega')$
SkelEff	**erased ExEff**	$\Lambda\varsigma\,.$ **fun** (g : Unit $\mapsto$ ς) $\mapsto$ **g unit**

Multicore OCaml (target language)

see also: A. H. Saleh et al., *Explicit Effect Subtyping,* ESOP 2018.

Current Approach

Eff	(sugared) source language	`fun g ↦ g unit`
ImpEff	desugared Eff	`fun g ↦ g unit`
ExEff	explicit optimisable	$\Lambda\alpha\,\Lambda\alpha'.\,\Lambda\delta\,.\,\Lambda\delta'\,.\,\Lambda(\omega : \alpha \leqslant \alpha').\,\Lambda(\omega' : \delta \leqslant \delta')\,.$ `fun (g : Unit ↦ α!δ)` $\mapsto$ `(g unit)` $\triangleright$ $(\omega!\omega')$
SkelEff	erased ExEff	$\Lambda\varsigma\,.$ `fun (g : Unit ↦ ς) ↦ g unit`

HITCH : target language with effects

Multicore OCaml (target language)

see also: A. H. Saleh et al., *Explicit Effect Subtyping,* ESOP 2018.

Ongoing Work

Eff	(sugared) source language	`fun g ↦ g unit`
ImpEff	desugared Eff	`fun g ↦ g unit`
ExEff	explicit optimisable (unnecessarily) elaborate	$\Lambda\alpha\,.\,\Lambda\alpha'\,.\,\Lambda\delta\,.\,\Lambda\delta'\,.\,\Lambda(\omega : \alpha \leqslant \alpha')\,.$ $\Lambda(\omega' : \delta \leqslant \delta')\,.$ `fun (g : Unit ↦ α!δ)` `↦ (g unit)` $\triangleright$ $(\omega!\omega')$
SkelEff	erased ExEff	$\Lambda\varsigma\,.$ `fun (g : Unit ↦ ς)` `↦ g unit`

Pure languages (Plain OCaml)

see also: A. H. Saleh et al., *Explicit Effect Subtyping,* ESOP 2018.

Ongoing Work

Eff	**(sugared) source language**	`fun g ↦ g unit`
ImpEff	**desugared Eff**	`fun g ↦ g unit`
ExEff	**explicit** **optimisable** **(unnecessarily) elaborate**	$\Lambda\alpha\,.\,\Lambda\alpha'\,.\,\Lambda\delta\,.\,\Lambda\delta'\,.\,\Lambda(\omega : \alpha \leqslant \alpha')\,.$ $\Lambda(\omega' : \delta \leqslant \delta')\,.$ `fun (g : Unit ↦` $\alpha!\delta$`)` `↦ (g unit) ▷` $(\omega!\omega')$

?

Pure languages (Plain OCaml)

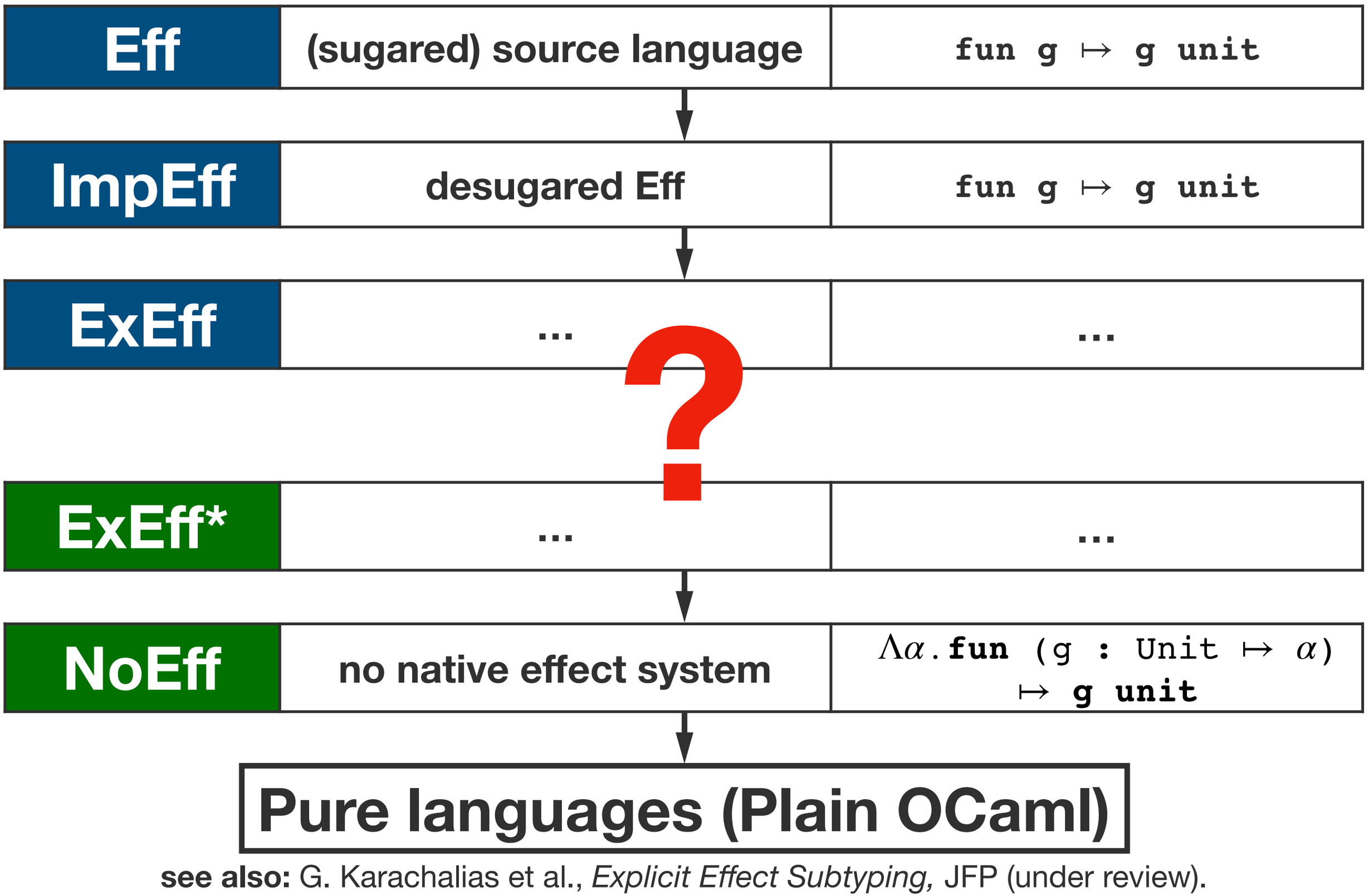
Ongoing Work

Eff	(sugared) source language	`fun g ↦ g unit`
ImpEff	desugared Eff	`fun g ↦ g unit`
ExEff	...	...
ExEff*	...	...
NoEff	**no native effect system**	$\Lambda\alpha.$ **fun** (g : Unit ↦ α) ↦ **g unit**

Pure languages (Plain OCaml)

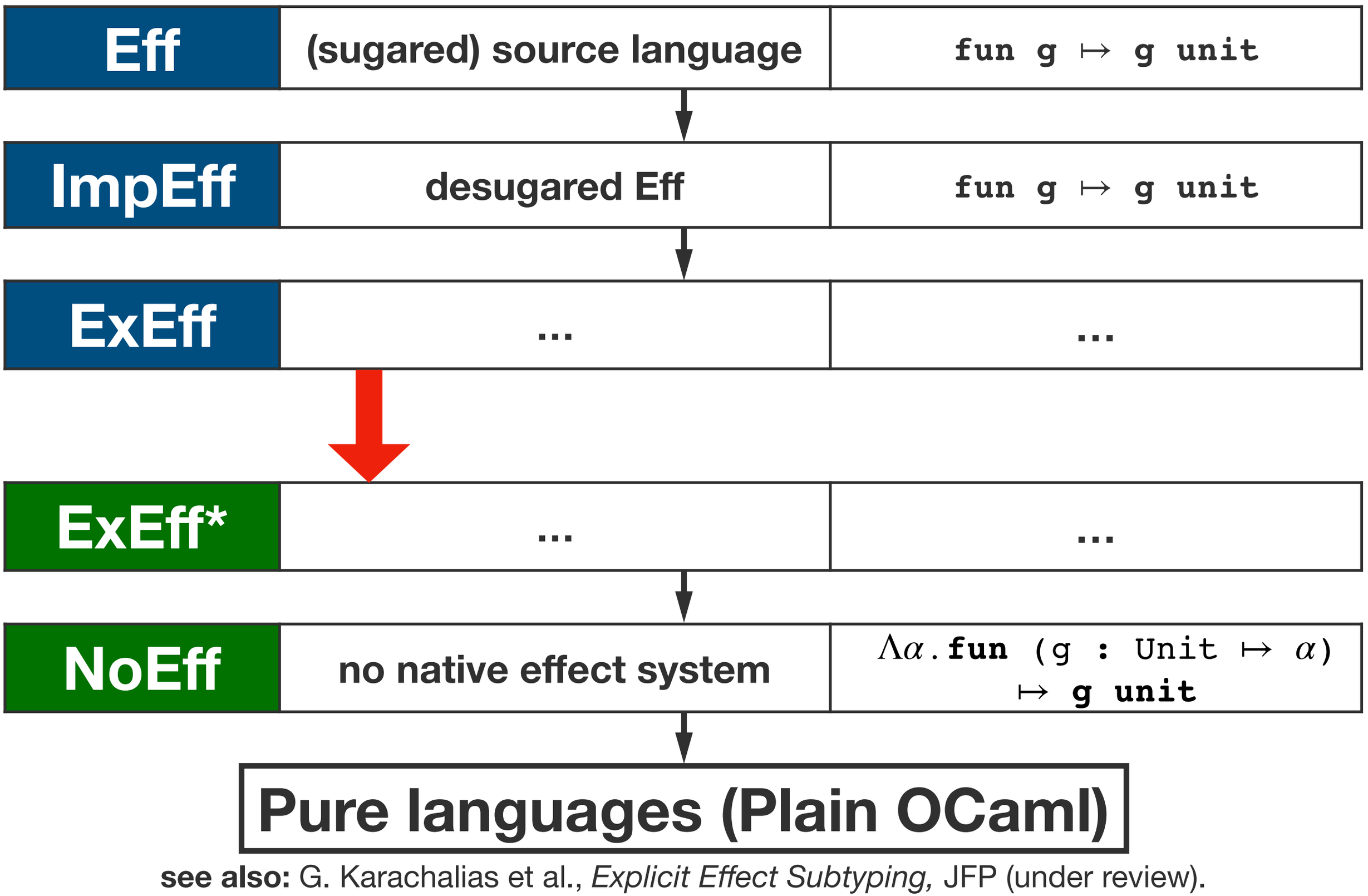
see also: G. Karachalias et al., *Explicit Effect Subtyping,* JFP (under review).

Ongoing Work



see also: G. Karachalias et al., *Explicit Effect Subtyping,* JFP (under review).

Ongoing Work



see also: G. Karachalias et al., *Explicit Effect Subtyping,* JFP (under review).

Ongoing Work

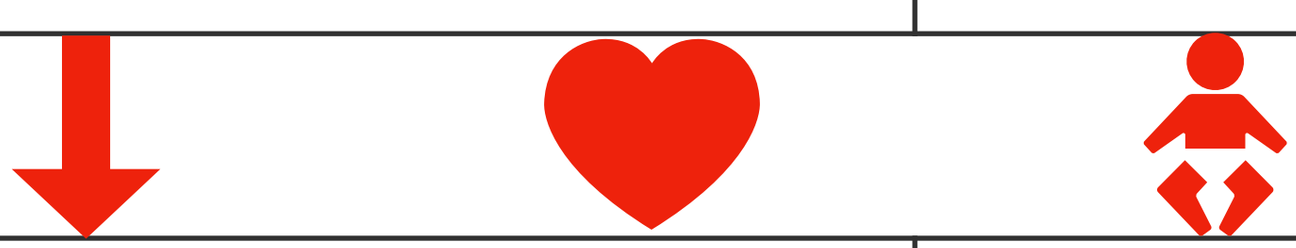
Eff	(sugared) source language	`fun g ↦ g unit`
ImpEff	desugared Eff	`fun g ↦ g unit`
ExEff	...	...
ExEff*	...	...
NoEff	no native effect system	$\Lambda\alpha.$ `fun (g : Unit ↦ α) ↦ g unit`

Pure languages (Plain OCaml)

see also: G. Karachalias et al., *Explicit Effect Subtyping,* JFP (under review).

Ongoing Work

Eff	**(sugared) source language**	`fun g ↦ g unit`
ImpEff	**desugared Eff**	`fun g ↦ g unit`
ExEff	**...**	**...**
ExEff*	**...**	**...**
NoEff	**no native effect system**	$\Lambda\alpha.$ **fun** (g : Unit ↦ α) ↦ **g unit**



Pure languages (Plain OCaml)

see also: G. Karachalias et al., *Explicit Effect Subtyping,* JFP (under review).

Thank you!

Any questions?

References

(1) **Bauer, A., Pretnar, M.**: *Programming with algebraic effects and handlers*. J. Log. Algebr. Meth. Program. 84(1), 108–123 (2015)

(2) **Karachalias, G., Pretnar, M., Saleh, A.H., Schrijvers, T.**: *Explicit effect subtyping* (2019), under consideration for publication in J. Functional Programming

(3) **Pretnar, M., Saleh, A.H.S., Faes, A., Schrijvers, T.**: *Efficient compilation of algebraic effects and handlers*. Tech. Rep. CW 708, KU Leuven, Informatics Section, Department of Computer Science (2017)

(4) **Saleh, A.H.**: *Efficient Algebraic Effect Handlers*. Ph.D. thesis, KU Leuven, Informatics Section, Department of Computer Science, Faculty of Engineering Science (2019)

(5) **Saleh, A.H., Karachalias, G., Pretnar, M., Schrijvers, T.**: *Explicit effect subtyping*. In: 27th European Symposium on Programming (ESOP) (2018)

(6) **Serckx, B.**: *Optimalisaties in de Eff Compiler*. Master's thesis, KU Leuven, Faculteit Ingenieurswetenschappen (2019)