

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
<code>throw</code> <code>try/catch</code>	<code>get</code> <code>set</code>	<code>read</code> <code>print</code>

Computational Effects

Exception throwing	Global state manipulation	I/O
<code>throw</code>	<code>get</code>	<code>read</code>
<code>try/catch</code>	<code>set</code>	<code>print</code>

operations: cause impure (effectful) behaviour

Computational Effects

Exception throwing	Global state manipulation	I/O
<code>throw</code> <code>try/catch</code>	<code>get</code> <code>set</code>	<code>read</code> <code>print</code>

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
<code>throw</code> <code>try/catch</code>	<code>get</code> <code>set</code>	<code>read</code> <code>print</code>

operations: cause impure (effectful) behaviour

handlers: capture impure (effectful) behaviour

Algebraic Computational Effects

Exception throwing	Global state manipulation	I/O
throw try/catch ↓ handle ... with ...	get set handle ... with ...	read print handle ... with ...

Algebraic Computational Effects

Exception throwing	Global state manipulation	I/O
throw	get	read
 HITCH : how to compile?		
handle ... with ...	handle ... with ...	handle ... with ...

Algebraic Computational Effects

Exception throwing	Global state manipulation	I/O
 HITCH : how to compile Eff? throw try/catch ↓ handle with ... get set ↓ handle with ... read print ↓ handle with ...		

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
```

HITCH: how to compile?
Handler
definition

in

```
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
```

Handler
definition

in

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

Computation

with coin_flipper

HITCH: how to compile efficiently?

Naive compilation

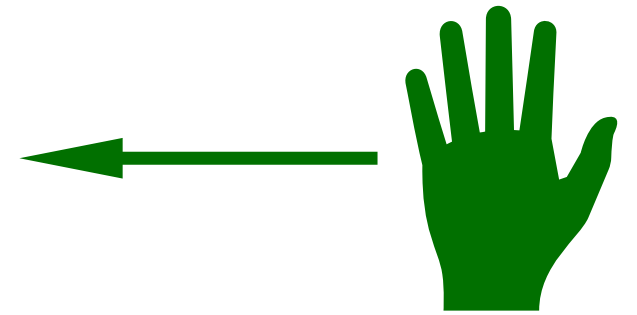
Eff

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

Naive compilation

Eff

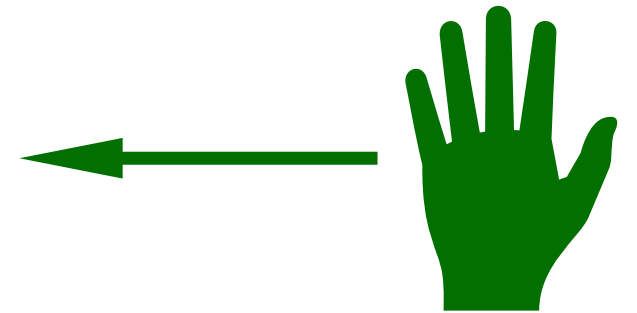
```
if #flip_a_coin ()  
  then 1  
  else 2 + 3
```



Naive compilation

Eff

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



SEQUENCING

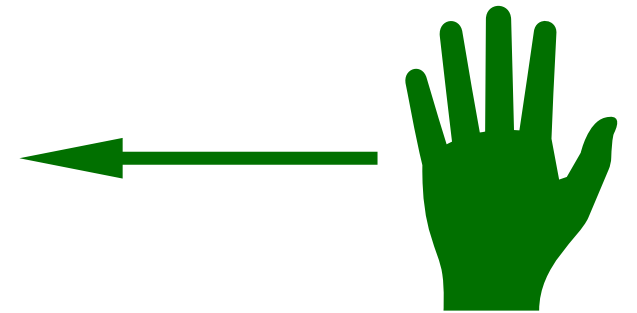
OCaml

```
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
```

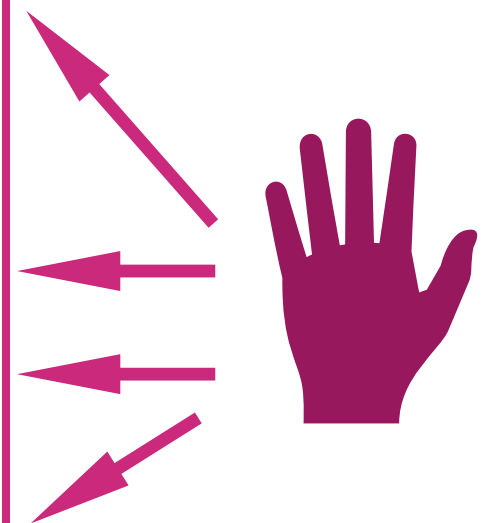


SEQUENCING



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

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
```

OCaml

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 ()
```

```
else 2 + 3
```

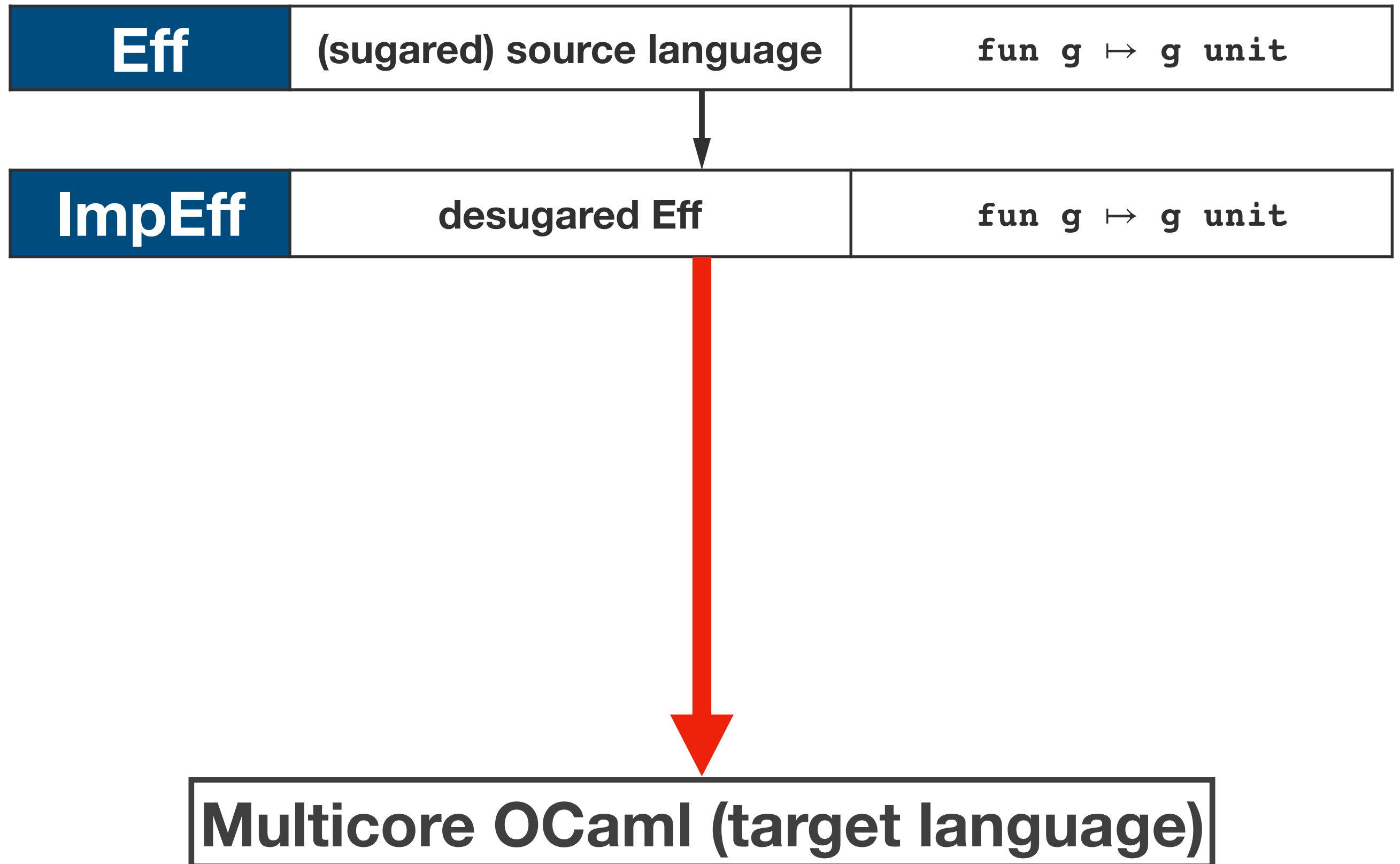
**SOLUTION: keep track
of effect information**

SEQUENCING

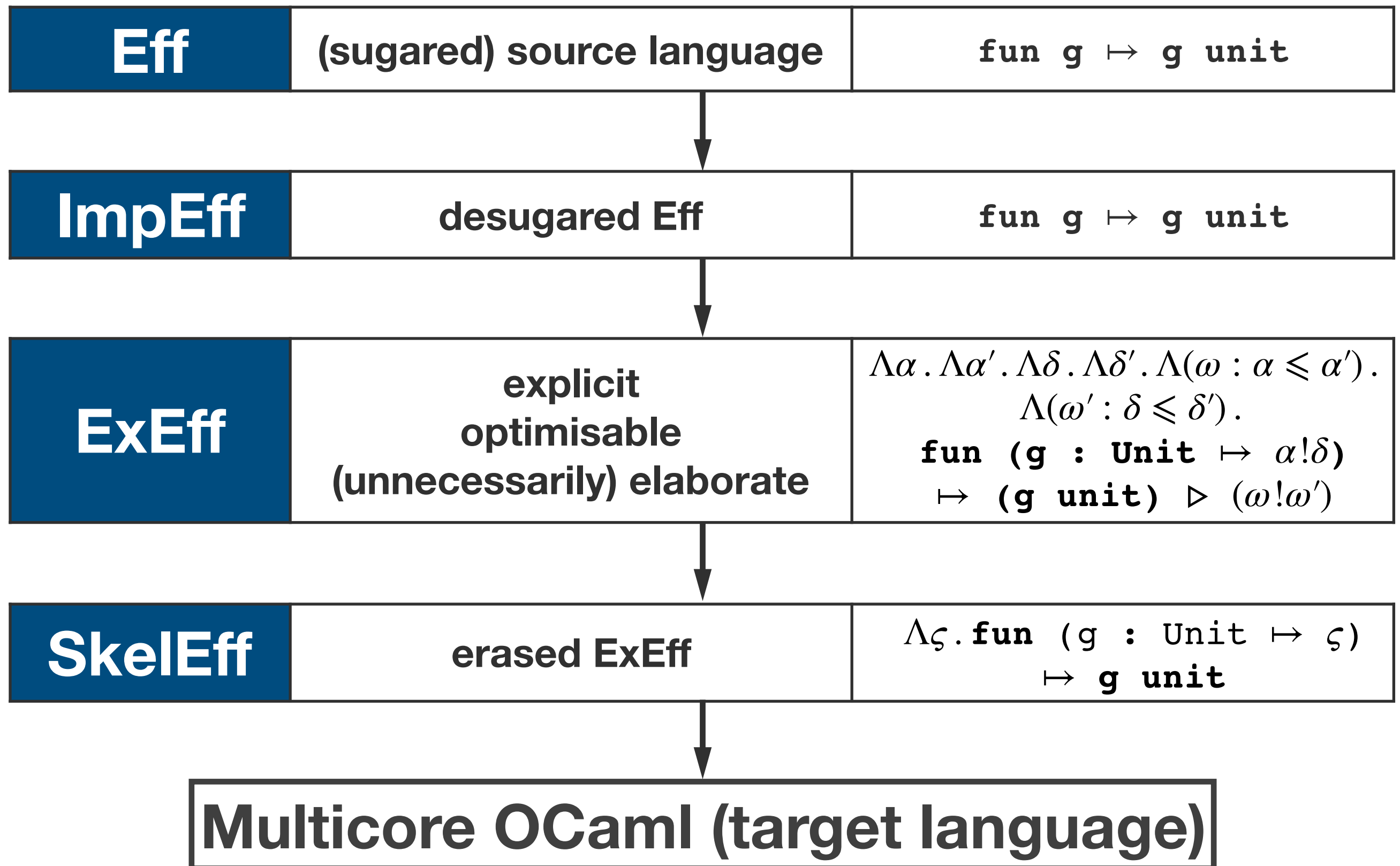
OCaml

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

Original Approach

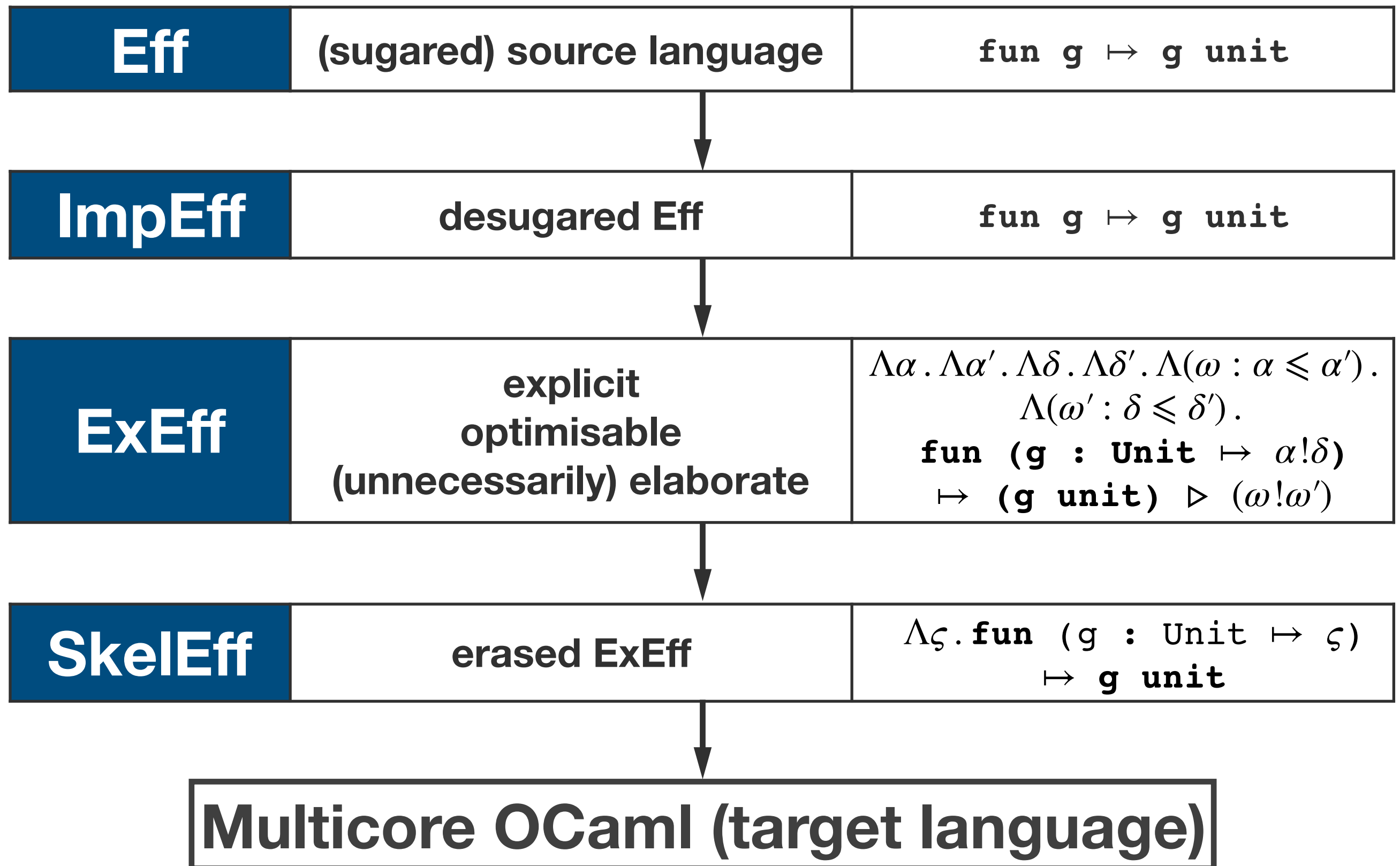


Current Approach



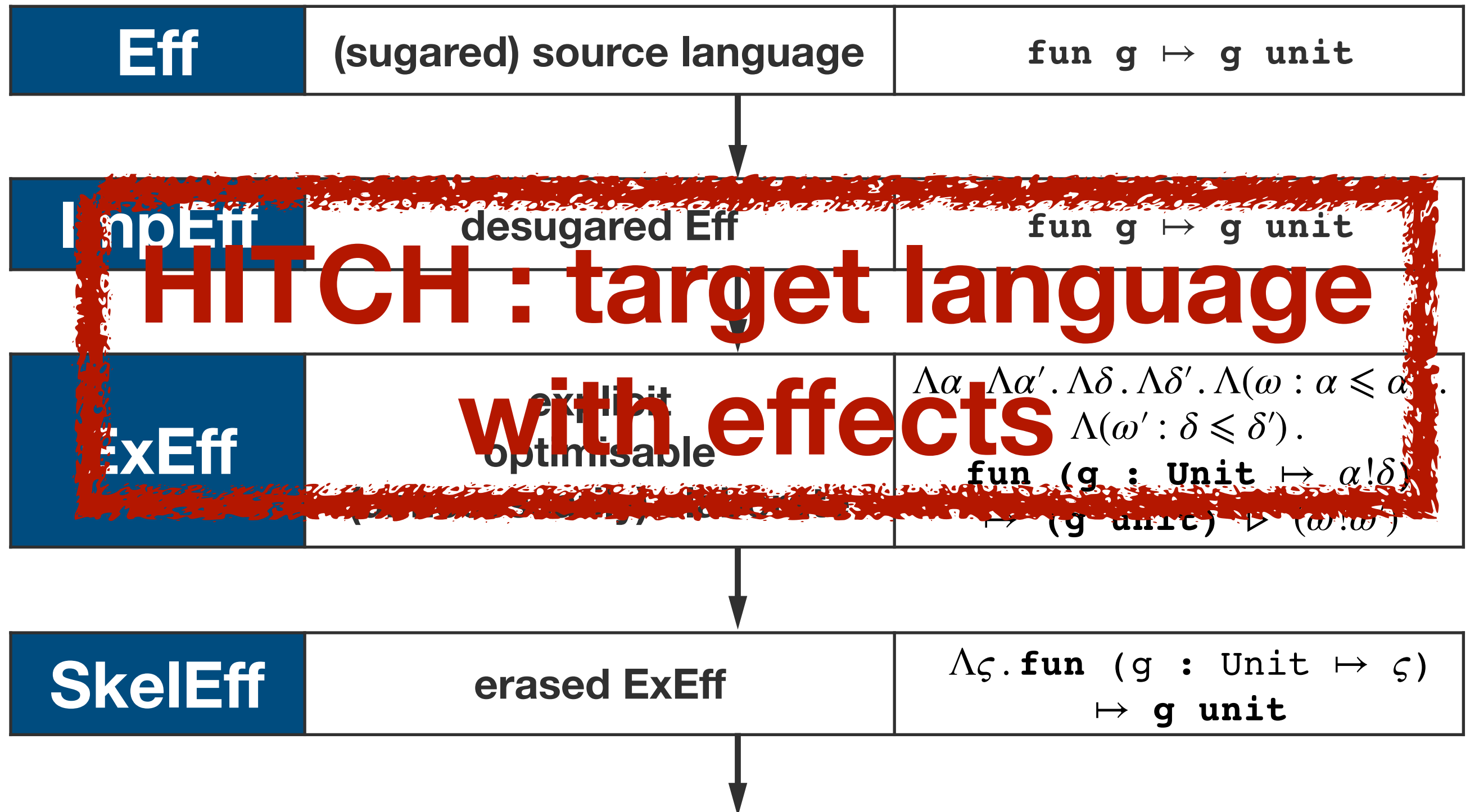
see also: A. H. Saleh et al., *Explicit Effect Subtyping*, ESOP 2018.

Current Approach



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

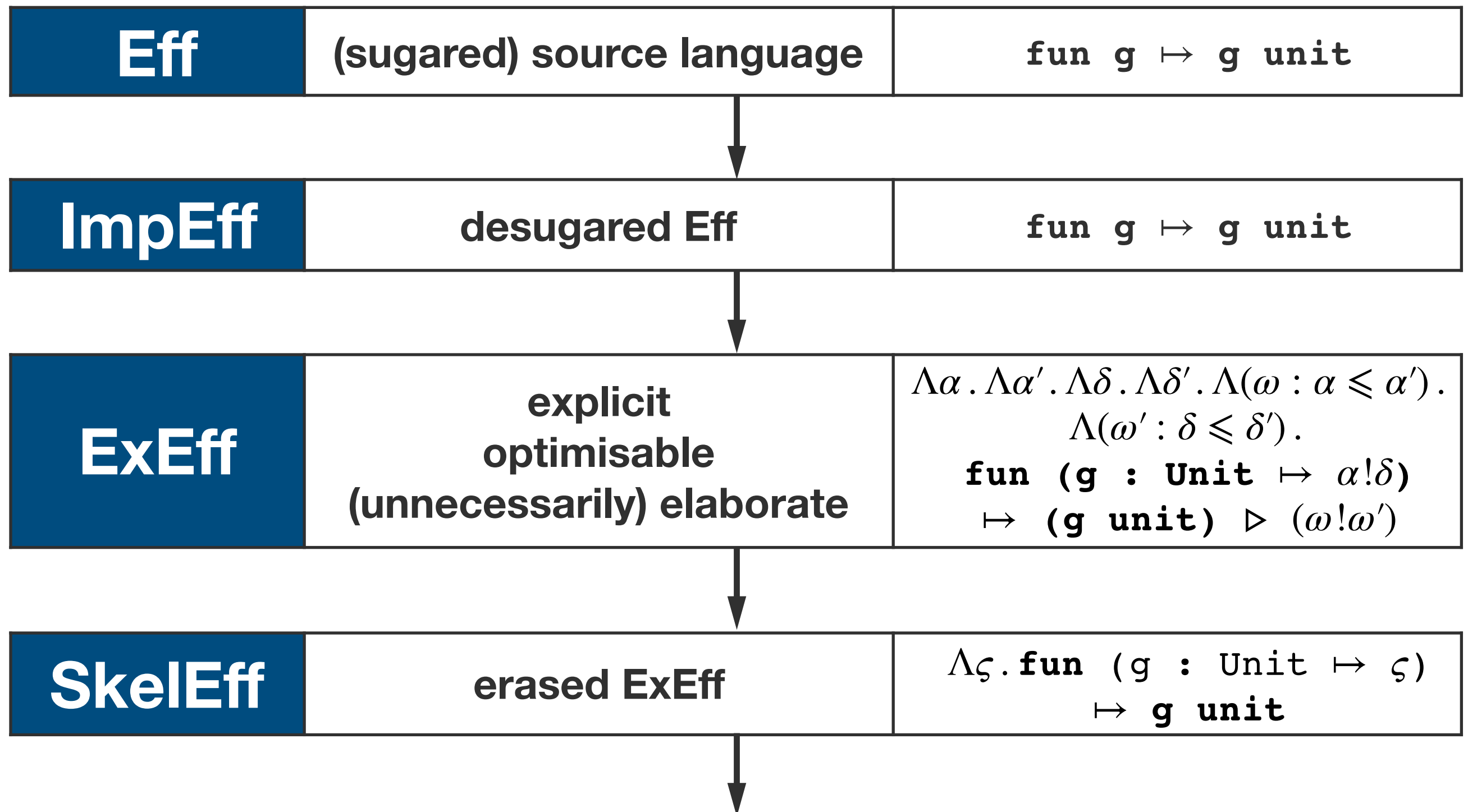
Current Approach



Multicore OCaml (target language)

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

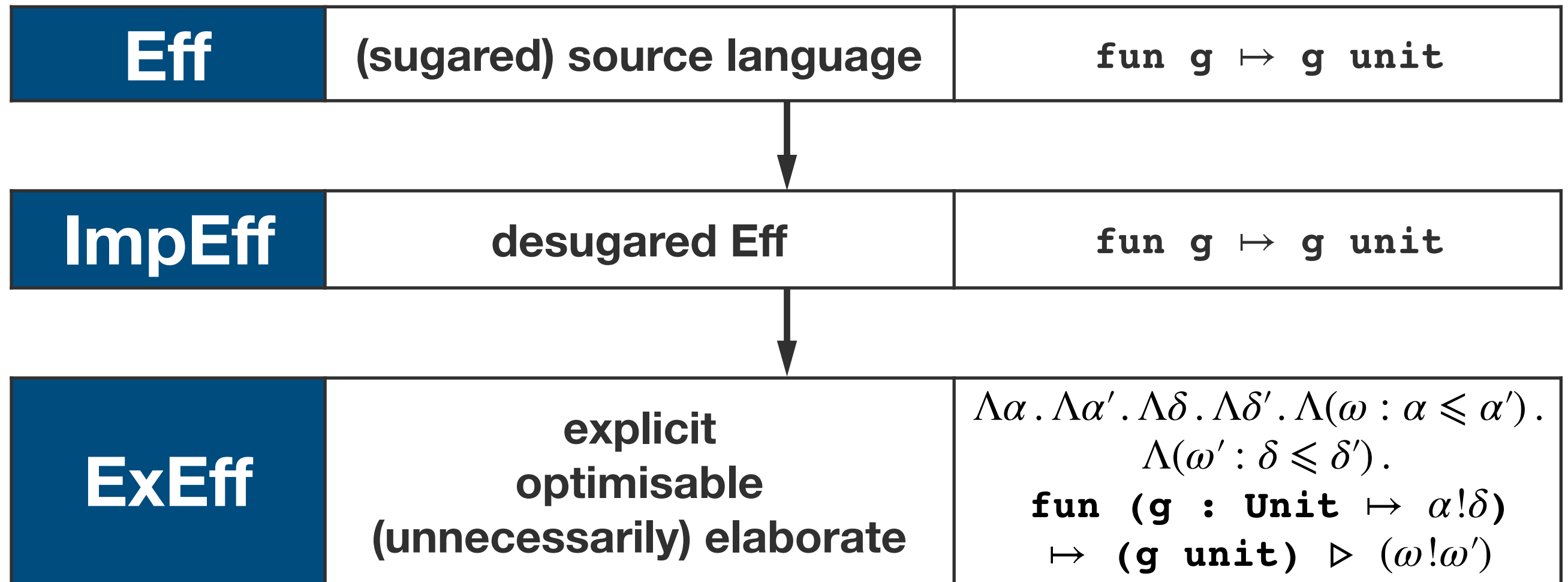
Ongoing Work



Pure languages (Plain OCaml)

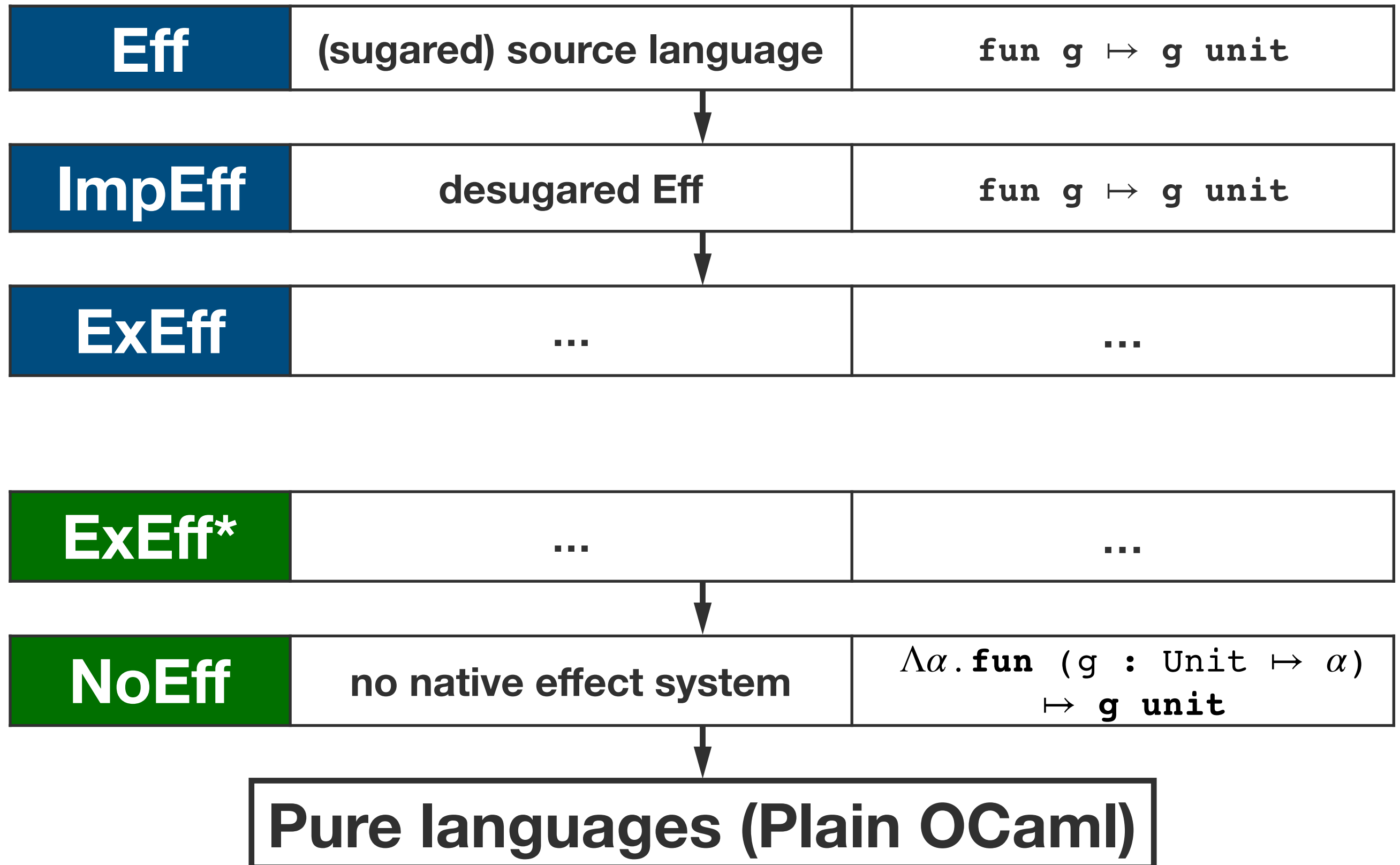
see also: A. H. Saleh et al., *Explicit Effect Subtyping*, ESOP 2018.

Ongoing Work



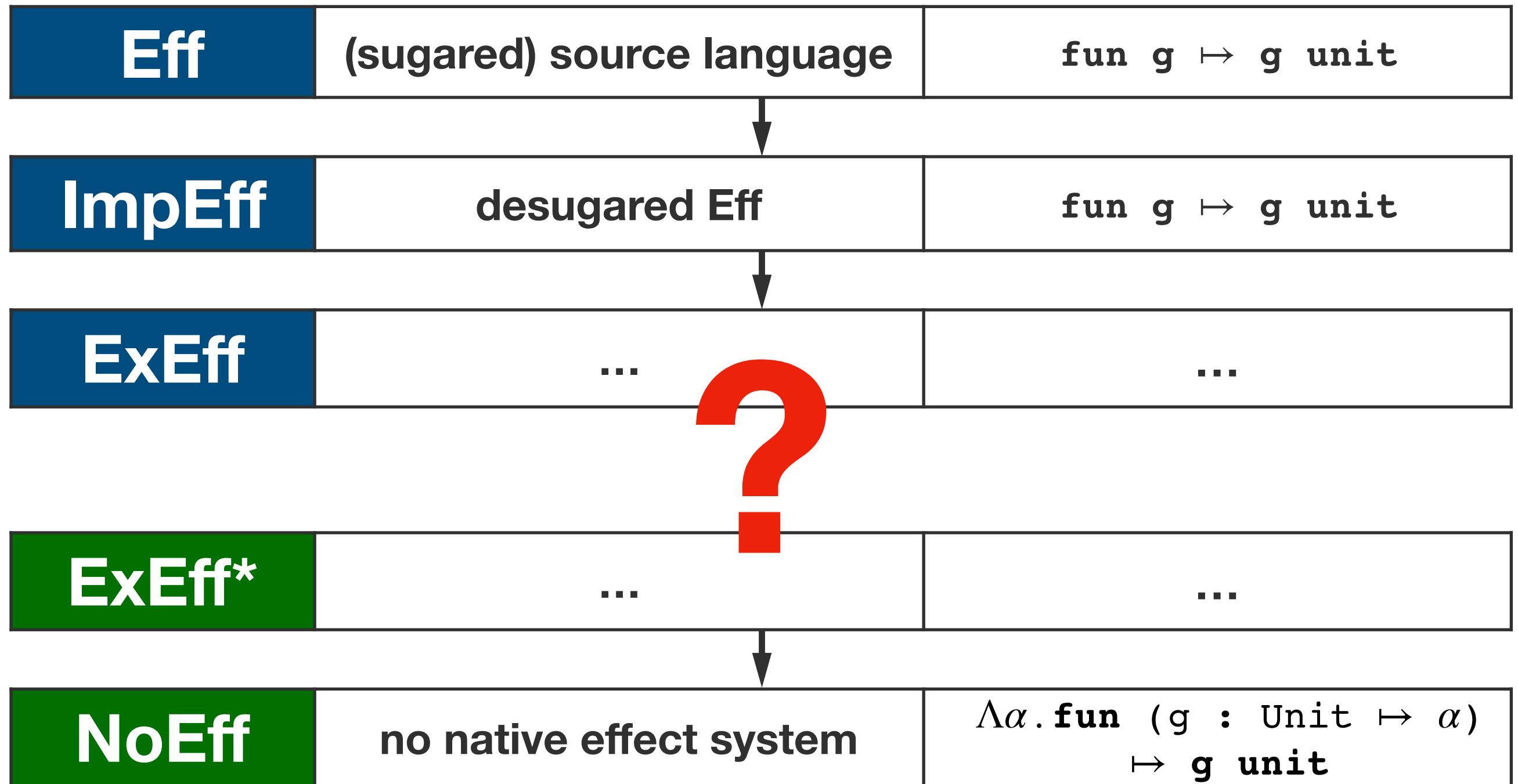
Pure languages (Plain OCaml)

Ongoing Work



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

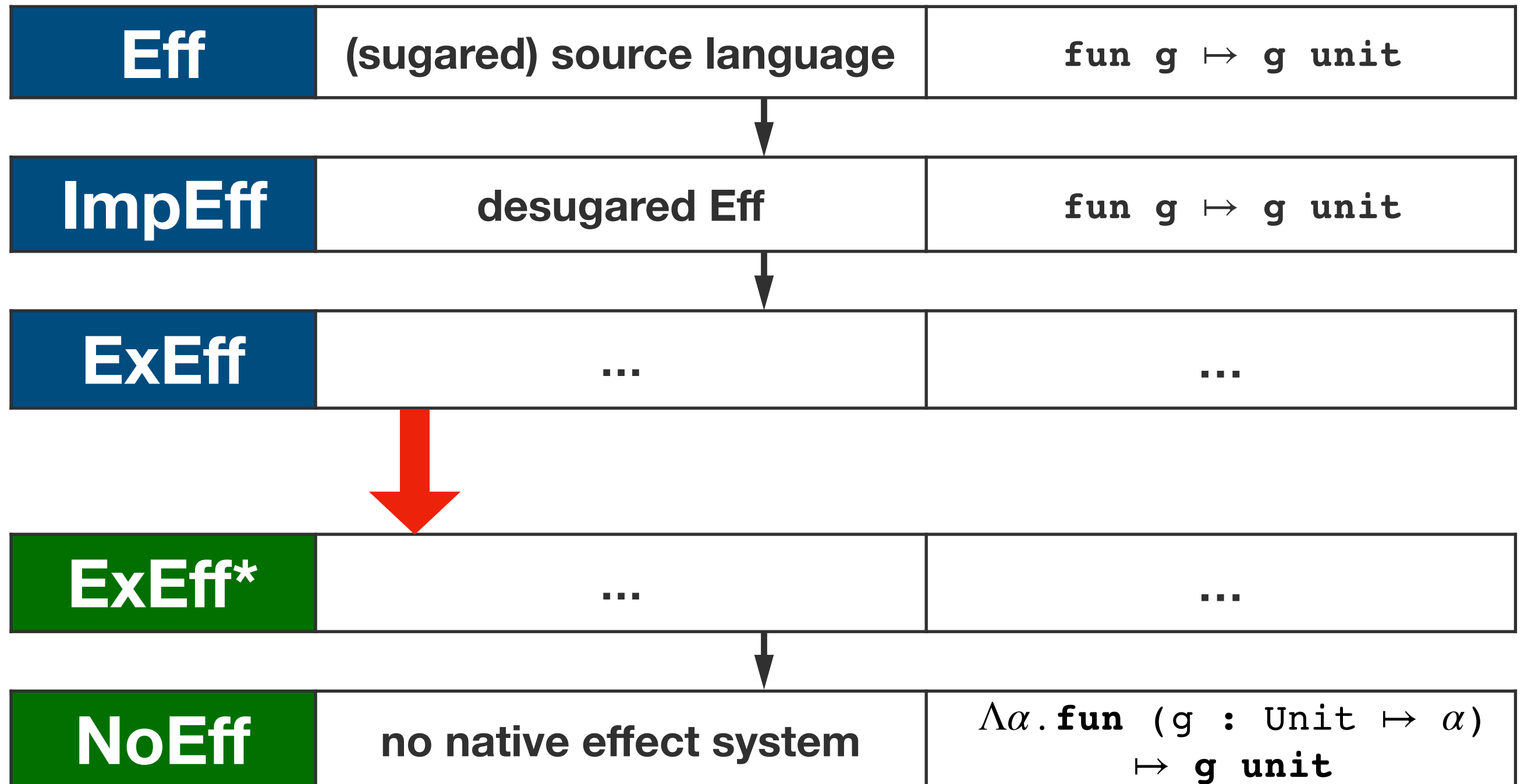
Ongoing Work



Pure languages (Plain OCaml)

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

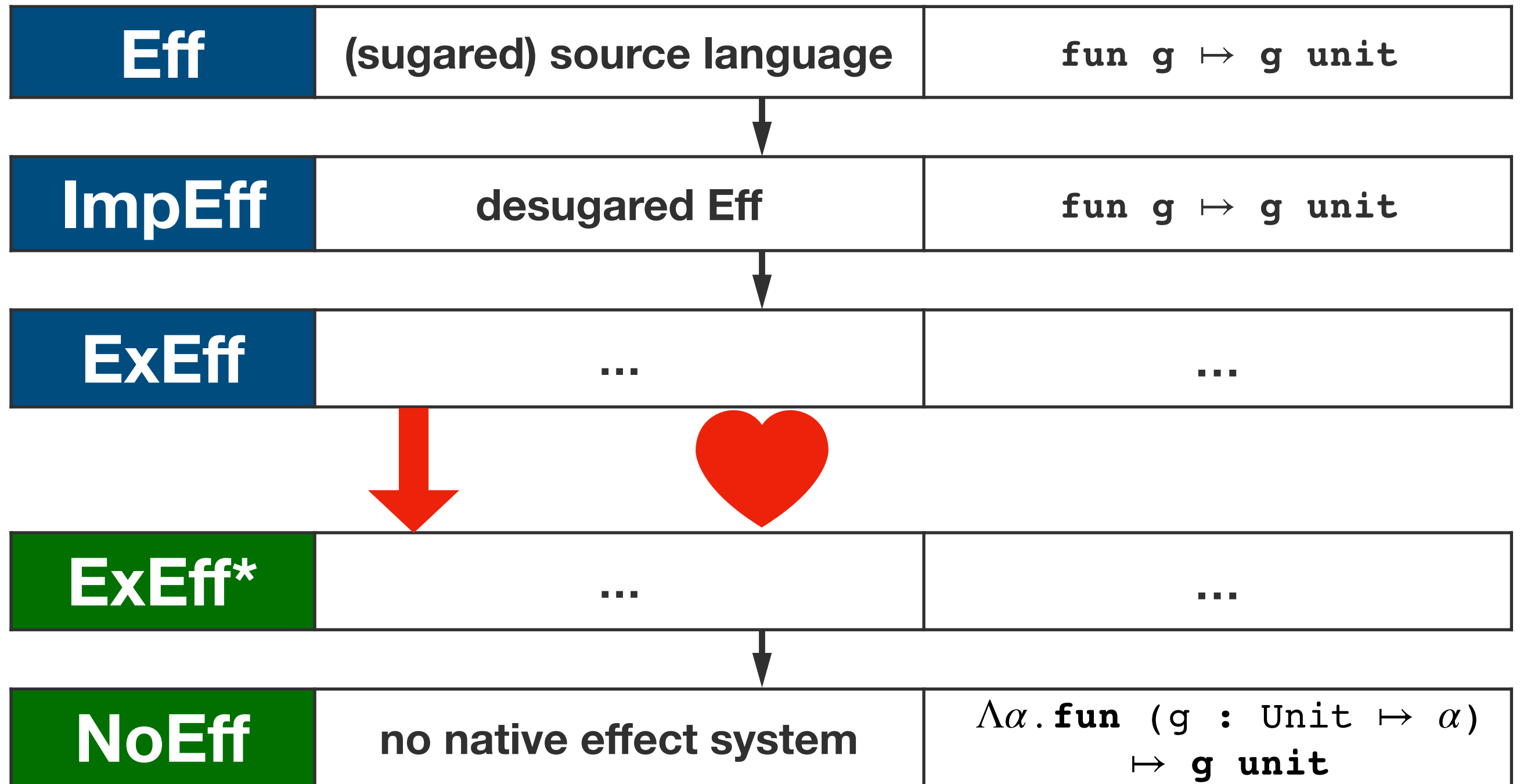
Ongoing Work



Pure languages (Plain OCaml)

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

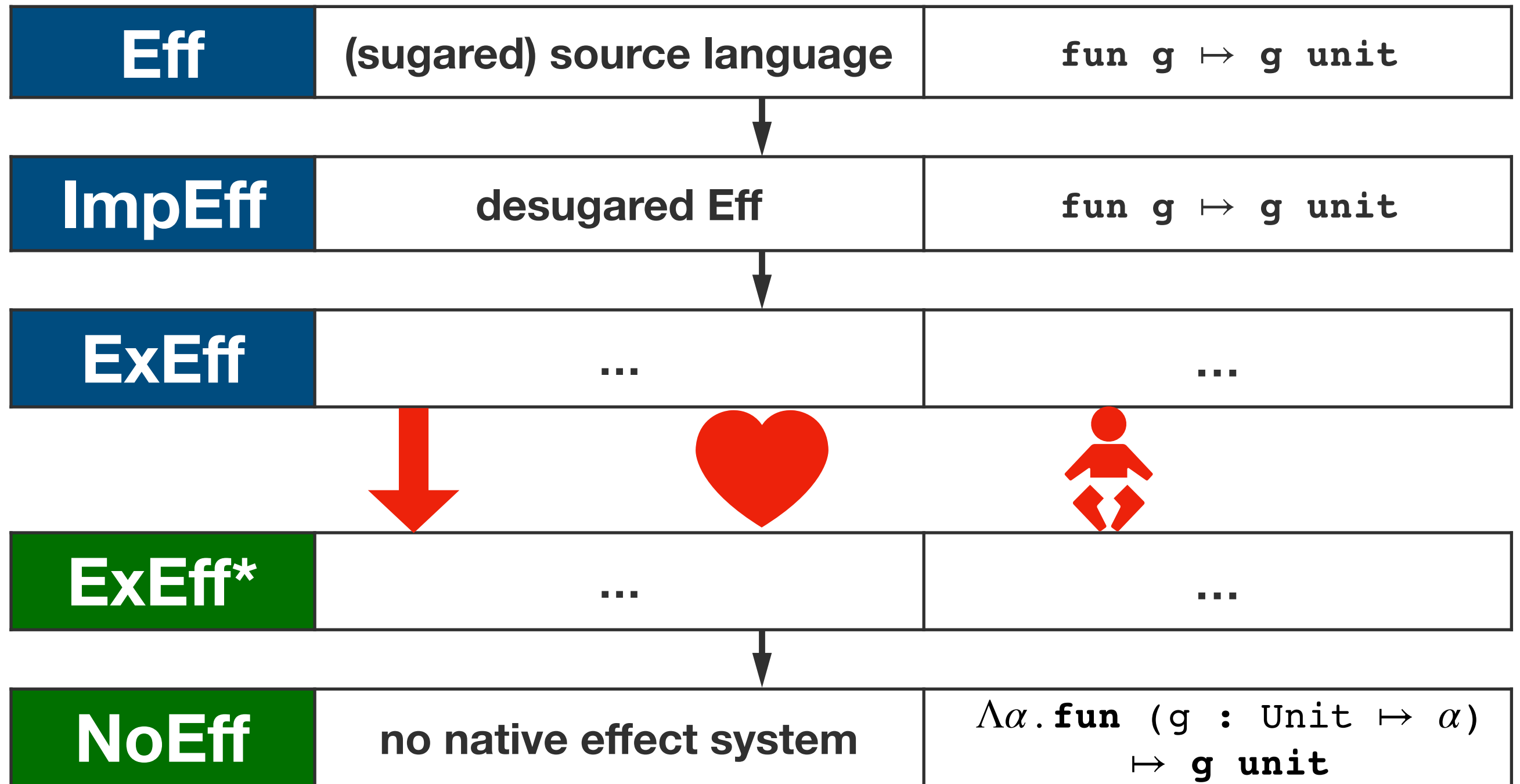
Ongoing Work



Pure languages (Plain OCaml)

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

Ongoing Work



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)