Compiling Algebraic Effects to Javascript in Js_of_ocaml

Armaël Guéneau & KC Sivaramakrishnan

February 17, 2016

ENS Lyon & OCamllabs

Algebraic effects and handlers

Programming and reasonning about computational effects in a pure setting

Computational effects are first-class citizens that can be performed; their behavior is determined by handlers.

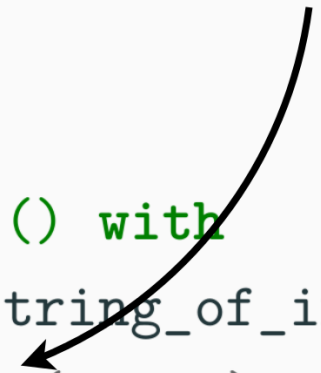
- *Eff* (`http://eff-lang.org`), by A. Bauer and M. Pretnar;
- Implemented in a custom OCaml compiler: `https://github.com/ocamllabs/ocaml-effects`.

Algebraic effects: example

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3))

let res =
  match f () with
  | x -> string_of_int x
  | effect (Foo i) k -> continue k (i + 1)
```

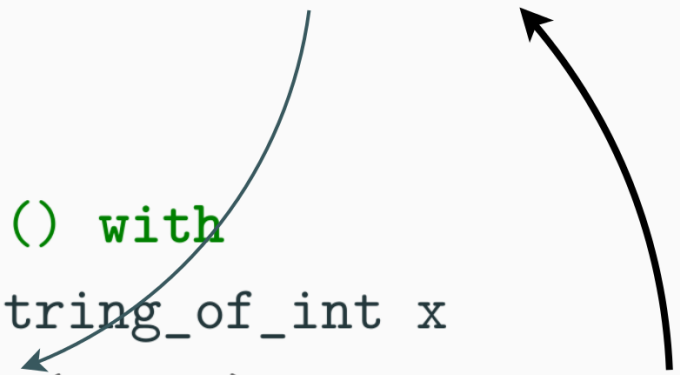
Algebraic effects: example

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3))

let res =
  match f () with
  | x -> string_of_int x
  | effect (Foo i) k -> continue k (i + 1)
```



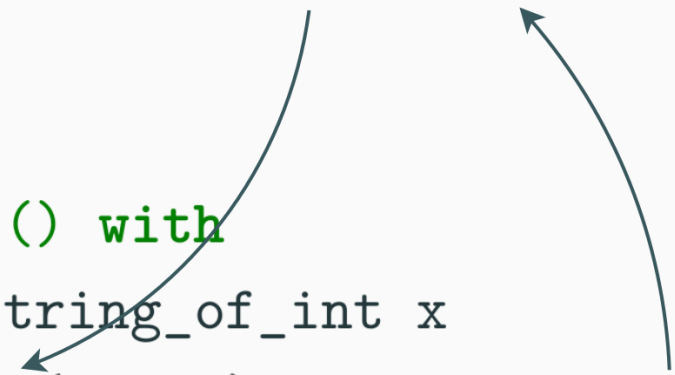
Algebraic effects: example

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3))

let res =
  match f () with
  | x -> string_of_int x
  | effect (Foo i) k -> continue k (i + 1)
```



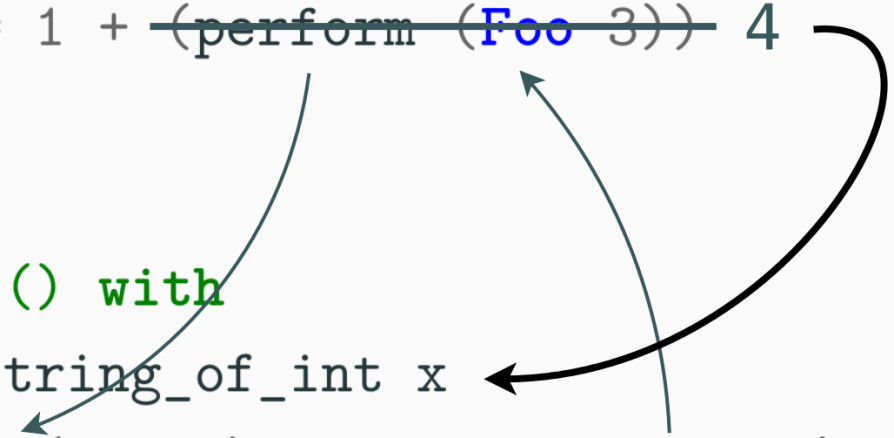
Algebraic effects: example

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3)) 4

let res =
  match f () with
  | x -> string_of_int x
  | effect (Foo i) k -> continue k (i + 1)
```



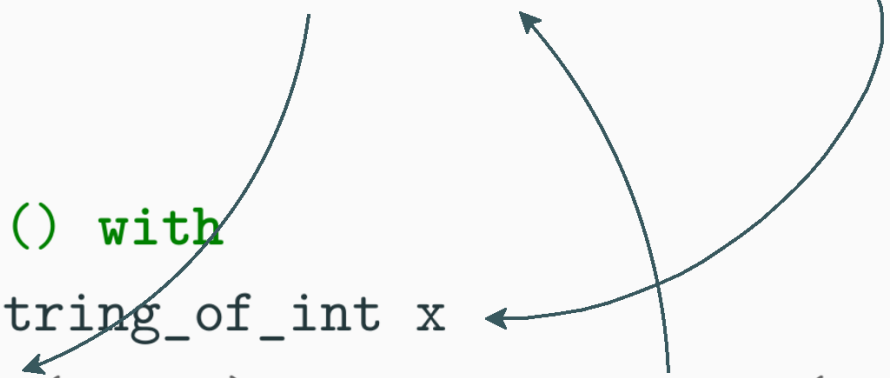
Algebraic effects: example

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3)) 4

let res =
  match f () with
  | x -> string_of_int x
  | effect (Foo i) k -> continue k (i + 1)
```



Algebraic effects: example

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3)) 4

let res =
  match f () with
  | x -> string_of_int x
  | effect (Foo i) k -> continue k (i + 1)
```



```
val res : string = "5"
```

Algebraic effects: more examples

Other examples include:

- Direct-style, lightweight cooperative concurrency
- Deriving a generator from an iteration function
- Anything expressible with delimited continuations

Scheduler Demo

Implementation: in the OCaml run-time

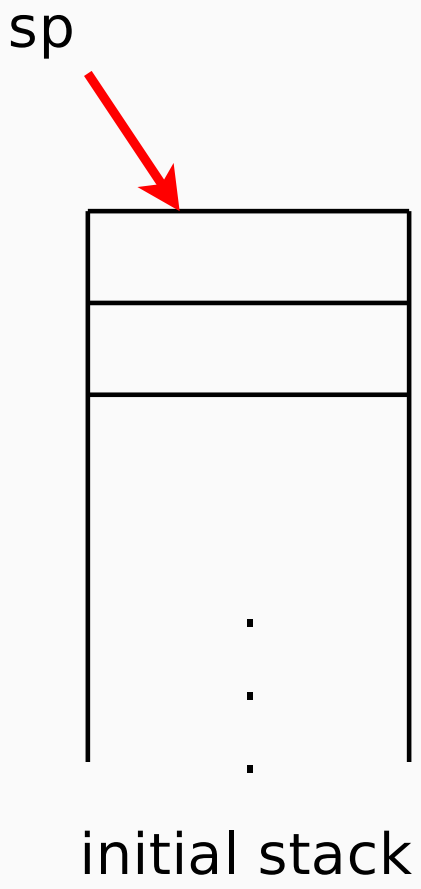
A computation delimited by a handler forms a fiber

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3))

let res =
  match f () with
  | x -> string_of_int x
  | effect (Foo i) k -> continue k (i + 1)
```

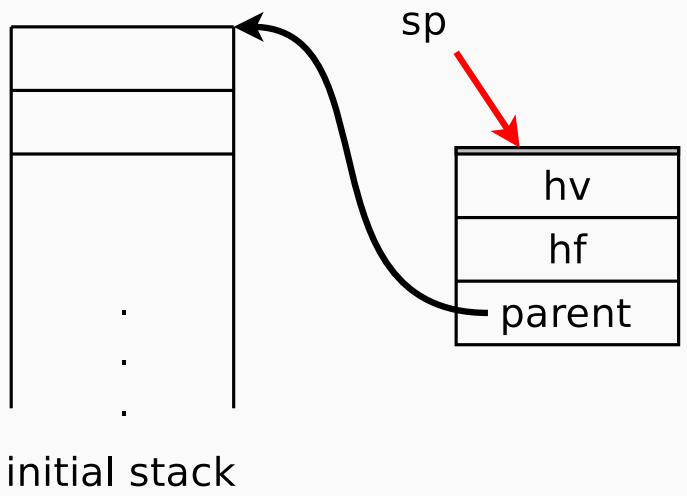
Implementation in the OCaml runtime

- Fibers are heap allocated stacks
- Handlers are attached to the corresponding fiber
- Delimited continuations are one-shot
- Alive fibers form a linked list: switching stacks is a matter of swapping pointers

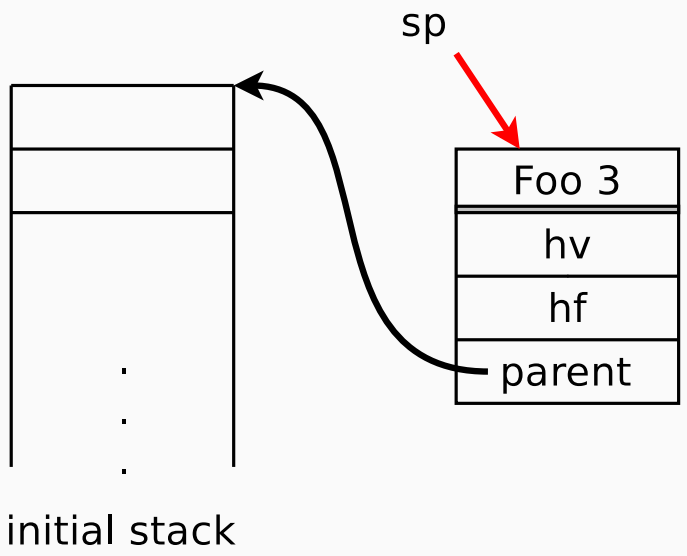
```
let f () =
  1 + (perform (Foo 3))

let res =
  ▶ match f () with
  | x ->
    string_of_int x
  | effect (Foo i) k ->
    continue k (i + 1)
```



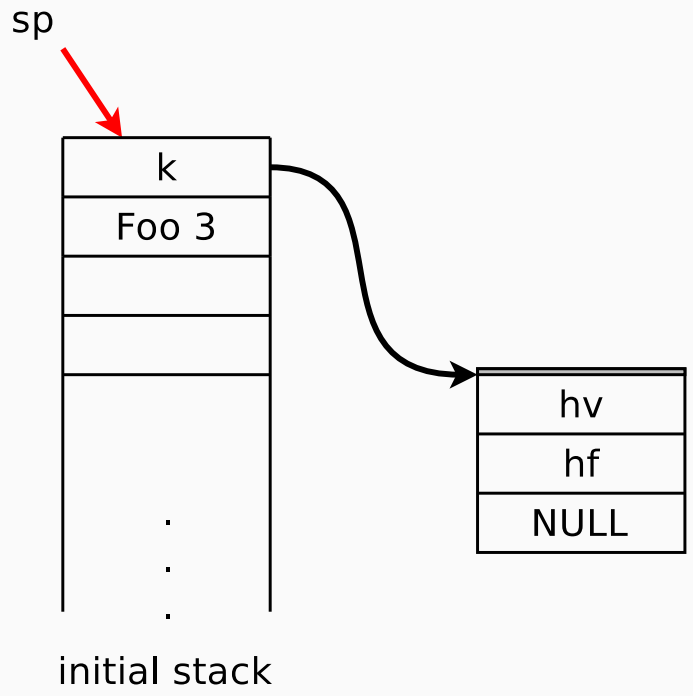
```
let f () =
  1 + (perform (Foo 3))

let res =
  match ▶ f () with
  | x ->
    string_of_int x
  | effect (Foo i) k ->
    continue k (i + 1)
```



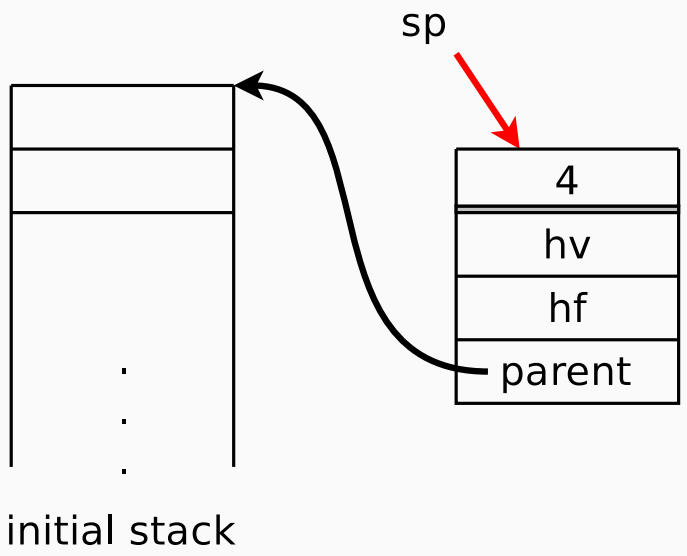
```
let f () =
  1 + ▶ (perform (Foo 3))

let res =
  match f () with
  | x ->
    string_of_int x
  | effect (Foo i) k ->
    continue k (i + 1)
```



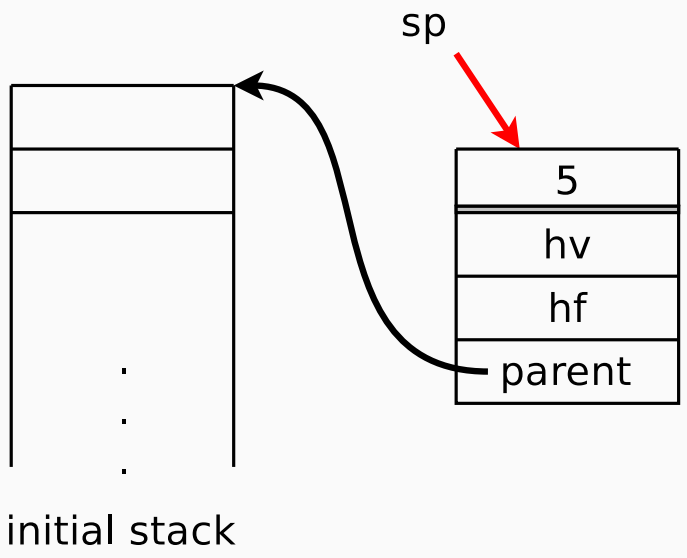
```
let f () =
  1 + (perform (Foo 3))

let res =
  match f () with
  | x ->
    string_of_int x
  | effect (Foo i) k ->
    ▶ continue k (i + 1)
```



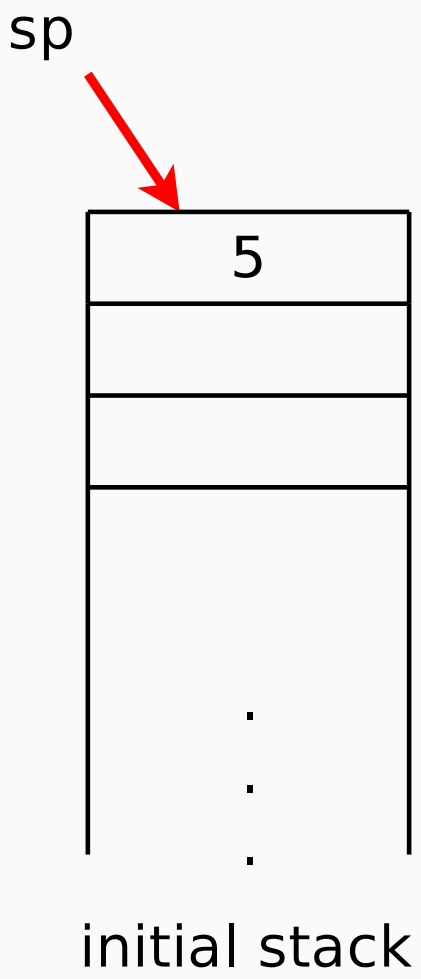
```
let f () =
  1 + ▶ (perform (Foo 3))

let res =
  match f () with
  | x ->
    string_of_int x
  | effect (Foo i) k ->
    continue k (i + 1)
```



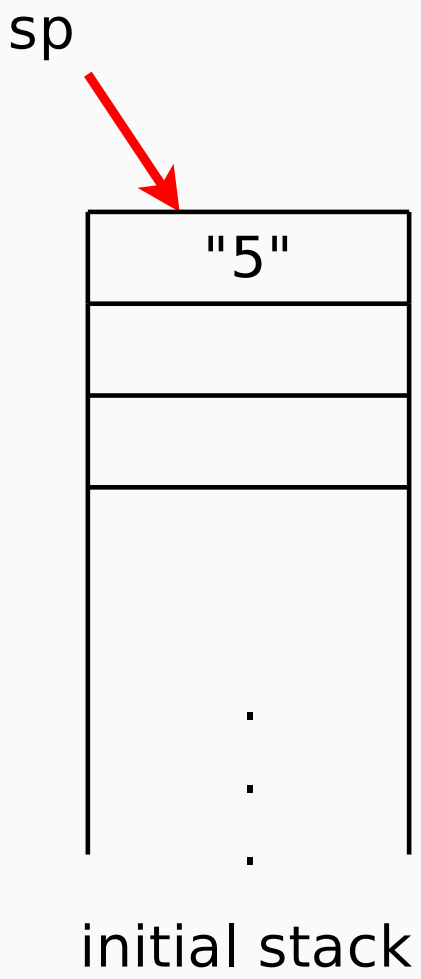
```
let f () =
  ▶ 1 + (perform (Foo 3))

let res =
  match f () with
  | x ->
    string_of_int x
  | effect (Foo i) k ->
    continue k (i + 1)
```



```
let f () =
  1 + (perform (Foo 3))

let res =
  match f () with
  | x ->
    string_of_int ▶ x
  | effect (Foo i) k ->
    continue k (i + 1)
```



```
let f () =
  1 + (perform (Foo 3))

let res =
  match f () with
  | x ->
    ▶ string_of_int x
  | effect (Foo i) k ->
    continue k (i + 1)
```



Implementation: in Js_of_ocaml

Js_of_ocaml: a compiler from OCaml bytecode to Javascript

- Bytecode produced by ocaml-effects' `ocamlc` contains effects primitives
- To be translated directly, effects need stack manipulation primitives
- Not available in Javascript

$\Rightarrow$ We do a *whole program selective CPS-transform*.
Implemented as a first pass, right after parsing bytecode into intermediate representation

Full CPS-transform

λ-calculus + effect primitives

$$
\begin{aligned}
t, u ::= {} & x \mid \lambda x.t \mid t\ u \\
& \mid \text{alloc\_stack}\ (v \mapsto h_v)\ (e\ k \mapsto h_f) \\
& \mid \text{resume stack}\ f\ v \\
& \mid \text{perform}\ e \\
& \mid \text{delegate}\ e\ k
\end{aligned}
$$

Syntactic sugar:

- `continue stack e` =
  resume stack $(\lambda x.x)$ e
- `match body with v -> hv | effect e k -> hf` =
  resume (alloc_stack $(v \mapsto h_v)$ $(e\ k \mapsto h_f)$) $(\lambda().body)$ ()

λ-calculus + effect primitives: CPS translation

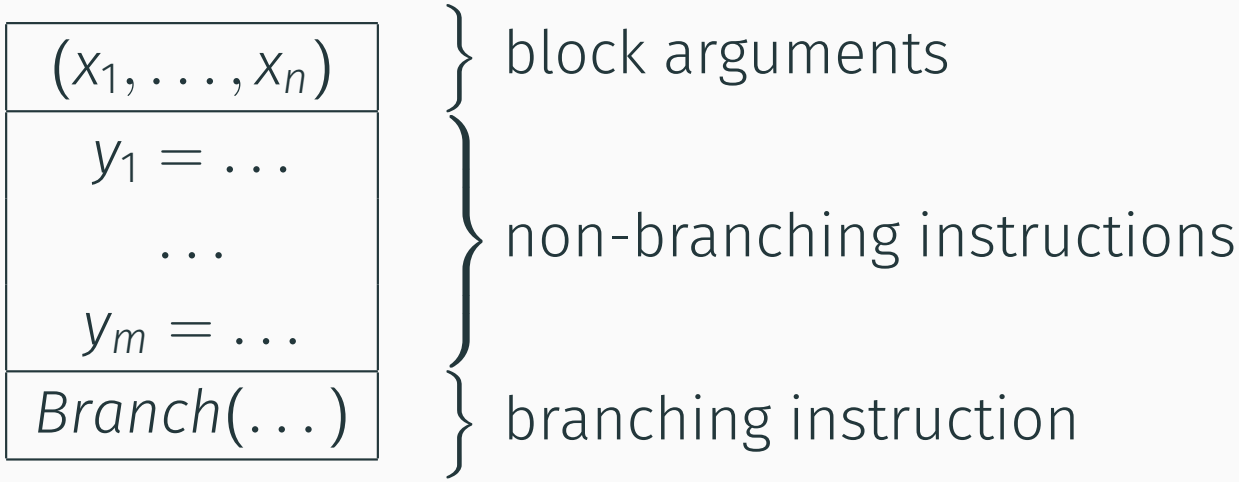
- $[\![$ alloc_stack $(v \mapsto h_v)$ $(e\ k \mapsto h_f)$ $]\!]$ =
  $\lambda k\ k_f.\ k\ (\lambda f\ x.$
  $f\ x \qquad (\lambda v.\ [\![h_v]\!]\ (\lambda v.\ v)\ k_f)$
  $(\lambda e\ k.\ [\![h_f]\!]\ (\lambda v.\ v)\ k_f))$
- $[\![$ resume stack f v $]\!]$ = $\lambda k\ k_f.$ k (stack f v)
- $[\![$ perform e $]\!]$ = $\lambda k\ k_f.\ k_f$ e $(\lambda f\ v.$ f v $k\ k_f)$

Some intuitions:

- A fiber is a $(\lambda f\ v. \ldots)$; $(f\ v)$ will be the computation to run on top of it
- Delimited continuations return a value
- *Abtracting Control*, Danvy & Filinski. Metacontinuations = continuation of the parent

Js_of_ocaml intermediate representation: a variant of SSA

Array of blocks, where a block is:



Js_of_ocaml intermediate representation: CPS translation

CPS and SSA are closely related, so the transformation is minimal:

- Add two arguments to each block, for both continuations;
- Every time one needs to capture the current continuation, allocate a closure instead of branching.

$$Branch(pc, args) \quad \rightarrow \quad \begin{array}{lcl} k & = & Closure(pc, args) \\ \ldots & & \\ ret & = & Apply(k, args) \\ Return(ret) & & \end{array}$$

Js_of_ocaml intermediate representation: CPS translation

Actual bytecode includes additional control structures (for-loops, ifs, etc.) that need to be translated.

For example,

```
for i = 0 to 100 do
  perform (Foo i)
done
```

cannot be translated to a Javascript for-loop.

$\Rightarrow$ Perform some analysis on the CFG and translate multiple jumps to a block to function calls.

Js_of_ocaml intermediate representation: CPS translation

To do: Trampolining.

Javascript engines do not perform *Tail Call Elimination*: programs in CPS style tend to blow up the stack very quickly.

Where to insert trampolines to preserve tail call optimisation in presence of effects?

Selective CPS-transform

(WIP)

Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform, Odersky et al.

- Uses Scala's type system to type expressions as performing computational effects ("*impure*") or not ("*pure*")
- In our case, we lost typing information for the program: we'll try to recover an approximation of it, then perform a purity analysis
- We do not consider contraction in our analysis: we do not try to know whenever a handler handles all possible effects

A Selective CPS Transformation, L. Nielsen.

- Gives safety constraints for purity annotations on types and terms (T = "*pure*", N = "*impure*", $N < T$)

$$
\begin{array}{rcl}
\mathrm{A} & ::= & \mathrm{T} \mid \mathrm{N} \\
e & ::= & \mathrm{c} \mid \mathrm{x} \mid \textsc{fun}^{\mathrm{A}}\ \mathrm{f}\ \mathrm{x}.e \mid (e\ @^{\mathrm{A}}\ e)^{\mathrm{A}} \mid \textsc{callcc}\ \mathrm{x}.e \mid \textsc{throw}\ e\ e \\
\tau & ::= & b \mid \tau \xrightarrow{\mathrm{A}} \tau \mid \langle \tau \rangle
\end{array}
$$

$$\frac{\Gamma(\mathrm{x}) = \tau}{\Gamma \vdash \mathrm{x} : \tau, \mathrm{T}} \qquad \frac{\textsc{ConstType}(\mathrm{c}) = b}{\Gamma \vdash \mathrm{c} : b, \mathrm{T}}$$

$$\frac{\Gamma[\mathrm{x} : \tau_1][\mathrm{f} : (\tau_1 \xrightarrow{\mathrm{A}_1} \tau_2)] \vdash e : \tau_2, \mathrm{A} \quad \mathrm{A}_1 \leq \mathrm{A}}{\Gamma \vdash \textsc{fun}\ \mathrm{f}\ \mathrm{x}.e : \tau_1 \xrightarrow{\mathrm{A}_1} \tau_2, \mathrm{T}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\mathrm{A}_3} \tau_2, \mathrm{A}_1 \quad \Gamma \vdash e_2 : \tau_1, \mathrm{A}_2 \quad \mathrm{A} \leq \min(\mathrm{A}_1, \mathrm{A}_2, \mathrm{A}_3)}{\Gamma \vdash e_1\ @\ e_2 : \tau_2, \mathrm{A}}$$

$$\frac{\Gamma[\mathrm{x} : \langle \tau \rangle] \vdash e : \tau, \mathrm{A}}{\Gamma \vdash \textsc{callcc}\ \mathrm{x}.e : \tau, \mathrm{N}} \qquad \frac{\Gamma \vdash e_1 : \langle \tau \rangle, \mathrm{A}_1 \quad \Gamma \vdash e_2 : \tau, \mathrm{A}_2}{\Gamma \vdash \textsc{throw}\ e_1\ e_2 : \tau_2, \mathrm{N}}$$

Inferring purity annotations

Nielsen's paper only gives checking rules for the purity annotations.

If we know the unannotated types for a program, we can:

- Annotate all the terms and types with fresh "purity variables"
- Collect all constraints
- Propagate all constraints (will instantiate variables to N)
- Instantiate unconstrained variables to T (we have full program information)

Inferring purity annotations

An interesting optimisation: specialisation of higher-order functions.

```
let _ =
  List.iter (fun x -> perform (Foo x)) [1;2;3];
  List.iter (fun x -> print_int x) [1;2;3]
```

Which type for `List.iter`? $(b \xrightarrow{N} b) \to b$

We can do better:

- Generate a CPS version and a direct style version of `List.iter`
- Have distinct "purity variables" for `List.iter` at each call site
- Call site's annotations indicate which version to use

Approximating types of the program

We still need some types before doing the purity analysis.

Idea:

- Compute a 0CFA analysis
- Obtain a set of terms that flow through the arguments and return value of functions
- Deduce a set of possible types for the function
- Extend the previous purity checking rules to work with set of possible types

For the future

- Benchmarks / optimisations
- Interoperability with external libraries
- More precise analysis?