Effect-Driven QuickChecking of Compilers

Jan Midtgaard
Mathias Nygaard Justesen
Patrick Kasting
Flemming Nielson
Hanne Riis Nielson

ICFP'17, Oxford

```
exit (-369469277365242727 2)
```

```
int_of_string (exit ((/) ((/) (-1) 5) (abs 5)))
```

What are these?

```
(fun e -> pred 10) (if if true then false else false then
   fun n -> [] else fun o -> [])
```

What are these?

```
let a = let s = (let x = exit (lnot (-47628762612628909 7))
    in fun r -> fun k -> "zsqykify") () in exit (exit (let c
     = (mod) 516587874847912894 (-45340071501207836 48) in 0)
    ) in (asr) (if (fun x -> let h = 2 in false) (let r = (@
    ) [] [] in (lsl) 2273200848021743825 0) then 466 else if
     let b = "" in false then ( * ) 2 (-4611686018427387904)
     else let m = 2 in m) ((fun e -> if exit 5 then
    int_of_string "vgjgme" else (land) 4 7) (fun m -> (&&) (
    let i = () in false) (if true then true else true)))
```

What are these?

```
(-) (let a = (^) ((let o = fun c -> let s = (^) "srtwbawql"
    "fbe" in List.hd [] in let i = (fun x -> []) (fun s ->
   "r") in fun k -> string_of_int (-38239439737450152 93)) (
   ignore [8239; 51660285336868583 0; ])) (if (=) (if false
   then true else false) ((||) (not false) (not false))
   then string_of_int (if true then 8 else (-1)) else if
   exit 2 then string_of_int (-309435526309293195 5) else
   (^) "yvvqp" "el") in (fun b -> 92960247249054152 7) (let
   z = 5 in fun q -> (let c = if false then () else () in
   fun f -> not) (string_of_int (let t = true in 0)))) ((+)
    (let z = let g = let u = string_of_int (let e = () in
   let p = false in 332628035944507958 0) in 2 in let t =
   let i = let c = "ng" in 18 in g in fun b -> (fun w ->
   false) (if false then () else ()) in (fun a -> let e =
   [] in let t = string_of_bool false in if false then
   425635096218227740 else (-116261721431635920 7)) ()) ((
   fun e -> 68) (let x = (fun p -> fun d -> 6) ((fun p ->
   487301623237005908) (let o = "bxztzffio" in [])) ((let i
    = let g = 818 in 1 in fun h -> []) (fun b -> succ 59))
   in (let t = let m = fun p -> "taodmjmcrd" in true in (
   fun o -> string_of_bool) (fun g ->
   (-159253077253349266 7))) ((&&) true (let v = [] in (||)
```

What are these?

They are arbitrary ML programs

(with terrible indentation)

What are these?

What do they do?

What are these?

No idea, we didn't write them

What are these?

No idea, we didn't write them

We wrote a **generator**
that **wrote them**

What good are they then?

What good are they then?

Useful for (automated) testing, e.g., two compilers against each other:

What good are they then?

Useful for (automated) testing, e.g., two compilers against each other:

```
$ ocamlc -o prog.byte prog.ml
```

What good are they then?

Useful for (automated) testing, e.g., two compilers against each other:

```
$ ocamlc -o prog.byte prog.ml
$ ocamlopt -o prog.native prog.ml
```

What good are they then?

Useful for (automated) testing, e.g., two compilers against each other:

```
$ ocamlc -o prog.byte prog.ml
$ ocamlopt -o prog.native prog.ml
$ ./prog.byte > byte.out
```

What good are they then?

Useful for (automated) testing, e.g., two compilers against each other:

```
$ ocamlc -o prog.byte prog.ml
$ ocamlopt -o prog.native prog.ml
$ ./prog.byte > byte.out
$ ./prog.native > native.out
```

What good are they then?

Useful for (automated) testing, e.g., two compilers against each other:

```
$ ocamlc -o prog.byte prog.ml
$ ocamlopt -o prog.native prog.ml
$ ./prog.byte > byte.out
$ ./prog.native > native.out
$ diff -q byte.out native.out
```

What good are they then?

Useful for (automated) testing, e.g., two compilers against each other:

```
$ ocamlc -o prog.byte prog.ml
$ ocamlopt -o prog.native prog.ml
$ ./prog.byte > byte.out
$ ./prog.native > native.out
$ diff -q byte.out native.out
```

Any observed `diff` is suspicious

Surely this can’t work!?

Surely this can’t work!?

We wrote a generator of OCaml programs that does this.

We used it to test this difference property.

Surely this can't work!?

We wrote a generator of OCaml programs that does this.

We used it to test this difference property.

For example, we found:

```
let k = (let i = print_newline ()
         in fun q -> fun i -> "") ()
in 0
```

Bytecode result:

Surely this can't work!?

We wrote a generator of OCaml programs that does this.

We used it to test this difference property.

For example, we found:

```
let k = (let i = print_newline ()
         in fun q -> fun i -> "") ()
in 0
```

Bytecode result: newline printed, `0`
Native code result:

Surely this can't work!?

We wrote a generator of OCaml programs that does this.

We used it to test this difference property.

For example, we found:

```
let k = (let i = print_newline ()
         in fun q -> fun i -> "") ()
in 0
```

Bytecode result: newline printed, `0`
Native code result: `0` (effect delayed indefinitely)

Surely this can’t work!?

We wrote a generator of OCaml programs that does this.

We used it to test this difference property.

For example, we found:

Surely this can't work!?

We wrote a generator of OCaml programs that does this.

We used it to test this difference property.

For example, we found:

```
(/) 0 (let e = not in pred 1)
```

Bytecode result:

Surely this can't work!?

We wrote a generator of OCaml programs that does this.

We used it to test this difference property.

For example, we found:

```
(/) 0 (let e = not in pred 1)
```

Bytecode result: `Exception: Division_by_zero`
Native code result:

Surely this can't work!?

We wrote a generator of OCaml programs that does this.

We used it to test this difference property.

For example, we found:

```
(/) 0 (let e = not in pred 1)
```

Bytecode result: `Exception: Division_by_zero`
Native code result: `0` (effect removed)

Surely this can't work!?

We wrote a generator of OCaml programs that does this.

We used it to test this difference property.

For example, we found:

```
(/) 0 (let e = not in pred 1)
```

Bytecode result: `Exception: Division_by_zero`
Native code result: `0` (effect removed)

This work is the story of how we pulled this off.

For more bugs: read the paper

Program generation: a refinement story

Program generation, take 1

Generate arbitrary strings:

Program generation, take 1

Generate arbitrary strings:

```
"␣𝕎"
```

Program generation, take 1

Generate arbitrary strings:

```
"␣W"
```

```
"Jh/M{8/:m%_}d,w'Tc^$"
```

Program generation, take 1

Generate arbitrary strings:

```
"␣W"

"Jh/M{8/:m%_}d,w'Tc^$"

"y?V&TIW%D$R\\@i5dRh>2EvF\nv<N:0%CGv>\nvJ[KJ1hR_:,M|RBP!aj>
   ymY7|3\"=N1]3r)]gmf[u:01v8Ln;.&`c@q9R;u3Mzczhhn;27\"zU.)
   x|[pIm.=e|DSdXsd1[;3B}2o@(s_|LV0irR'CH-su>8J49h-l-MARkQ
   [4+O(lyQu\"nZ)!K*5Yh#r!;rs+O9,I*oW6BY9VWZ\nan(VnI!N=PKv,
   wJS\\CU298\nzeA3<*Ag<@#Qu_]!T3A6X'^P7s/Q[RP.K}\\#5e+Q`"
```

Program generation, take 1

Generate arbitrary strings:

```
"␣W"
```

```
"Jh/M{8/:m%_}d,w'Tc^$"
```

```
"y?V&TIW%D$R\\@i5dRh>2EvF\nv<N:0%CGv>\nvJ[KJ1hR_:,M|RBP!aj>
   ymY7|3\"=N1]3r)]gmf[u:01v8Ln;.&`c@q9R;u3Mzczhhn;27\"zU.)
   x|[pIm.=e|DSdXsd1[;3B}2o@(s_|LV0irR'CH-su>8J49h-l-MARkQ
   [4+O(lyQu\"nZ)!K*5Yh#r!;rs+O9,I*oW6BY9VWZ\nan(VnI!N=PKv,
   wJS\\CU298\nzeA3<*Ag<@#Qu_]!T3A6X'^P7s/Q[RP.K}\\#5e+Q`"
```

Few of these will make it through the lexer and the parser. . .

Program generation, take 2

Follow the grammar (Celento-al:SPE80):

Program generation, take 2

Follow the grammar (Celento-al:SPE80):

$$e \ ::= \ c \ \mid \ \texttt{x} \ \mid \ \texttt{fun x -> } e \ \mid \ e_0 \ e_1$$

Program generation, take 2

Follow the grammar (Celento-al:SPE80):

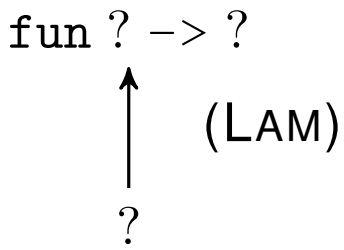
$$e \ ::= \ c \ \mid \ \texttt{x} \ \mid \ \texttt{fun x -> } e \ \mid \ e_0 \ e_1$$

?

Program generation, take 2

Follow the grammar (Celento-al:SPE80):

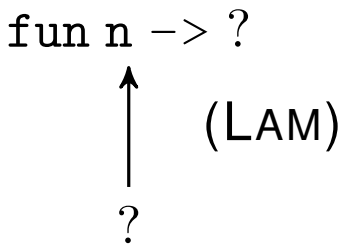
$$e \;::=\; c \;\mid\; \texttt{x} \;\mid\; \texttt{fun x -> } e \;\mid\; e_0\ e_1$$



Program generation, take 2

Follow the grammar (Celento-al:SPE80):

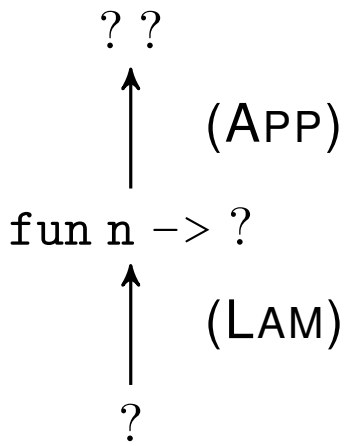
$$e \;::=\; c \;\mid\; \texttt{x} \;\mid\; \texttt{fun x -> } e \;\mid\; e_0\ e_1$$



Program generation, take 2

Follow the grammar (Celento-al:SPE80):

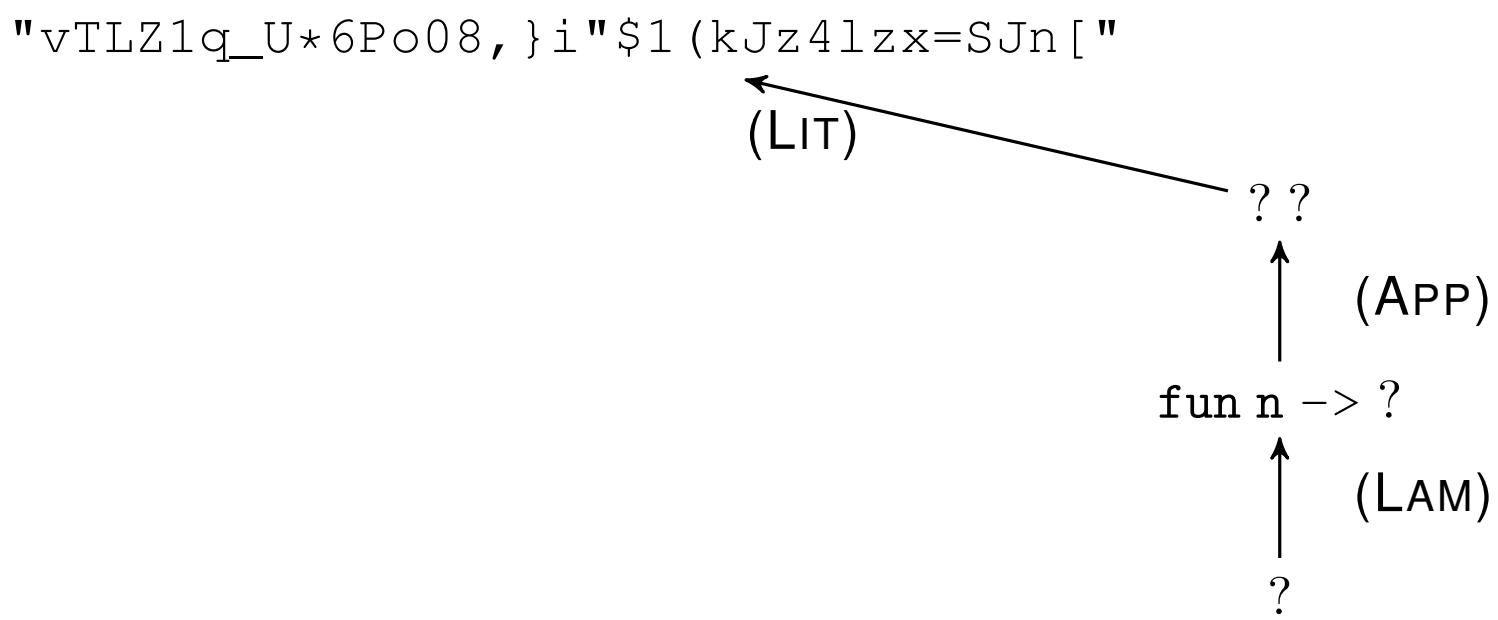
$$e \;::=\; c \;\mid\; \texttt{x} \;\mid\; \texttt{fun x -> } e \;\mid\; e_0\ e_1$$



Program generation, take 2

Follow the grammar (Celento-al:SPE80):

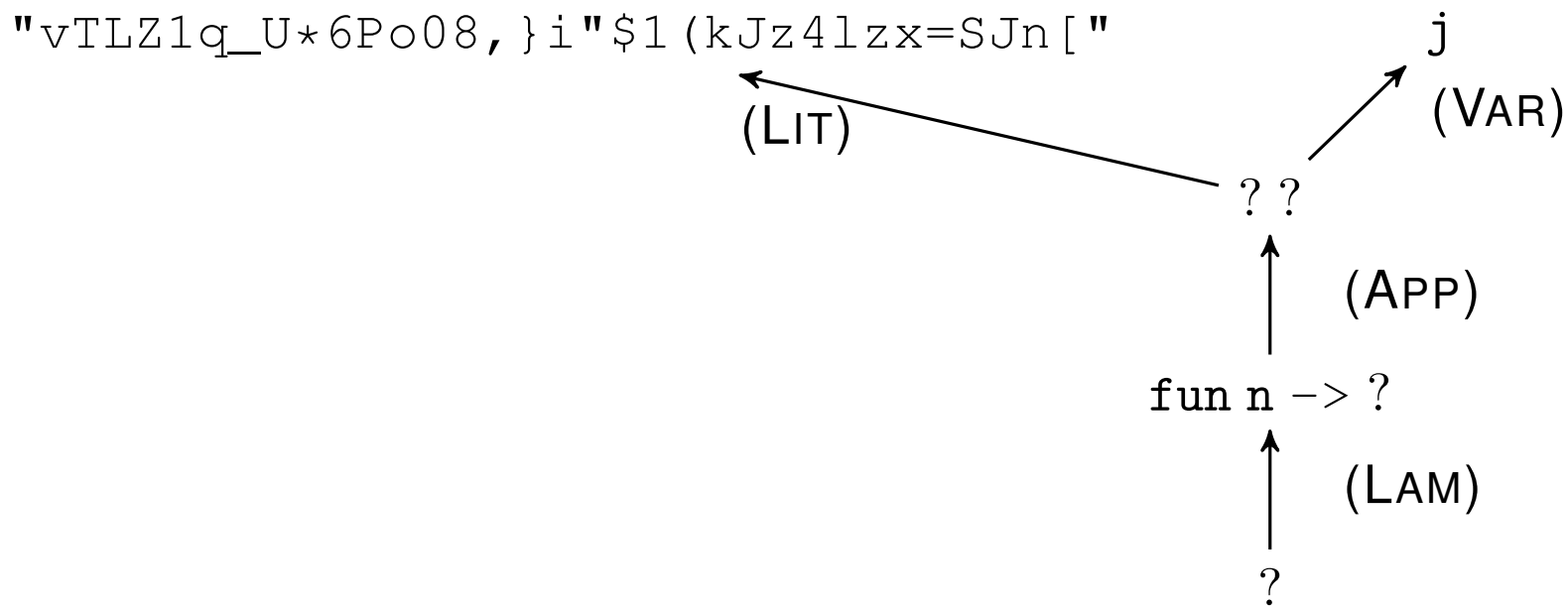
$$e \;::=\; c \;\mid\; \texttt{x} \;\mid\; \texttt{fun x -> } e \;\mid\; e_0\ e_1$$



Program generation, take 2

Follow the grammar (Celento-al:SPE80):

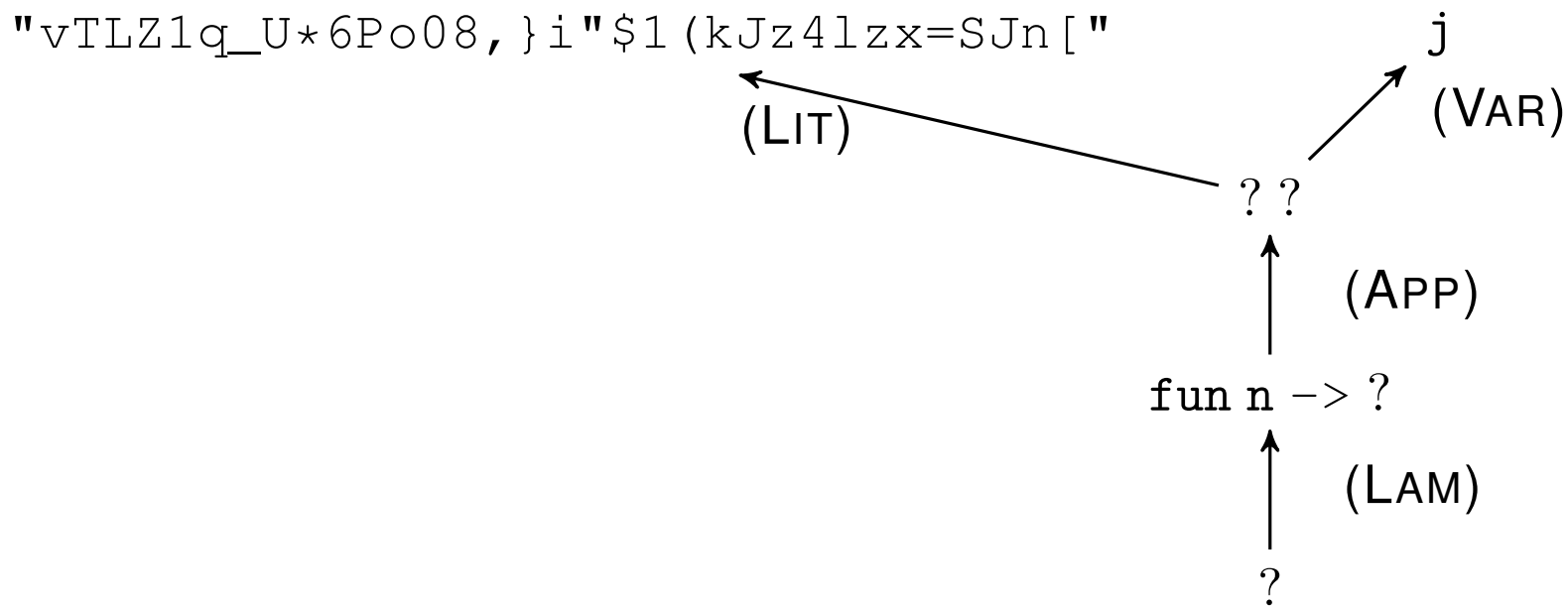
$$e \;::=\; c \;\mid\; \texttt{x} \;\mid\; \texttt{fun x -> } e \;\mid\; e_0\ e_1$$



Program generation, take 2

Follow the grammar (Celento-al:SPE80):

$$e \;::=\; c \;\mid\; \texttt{x} \;\mid\; \texttt{fun x -> } e \;\mid\; e_0\ e_1$$

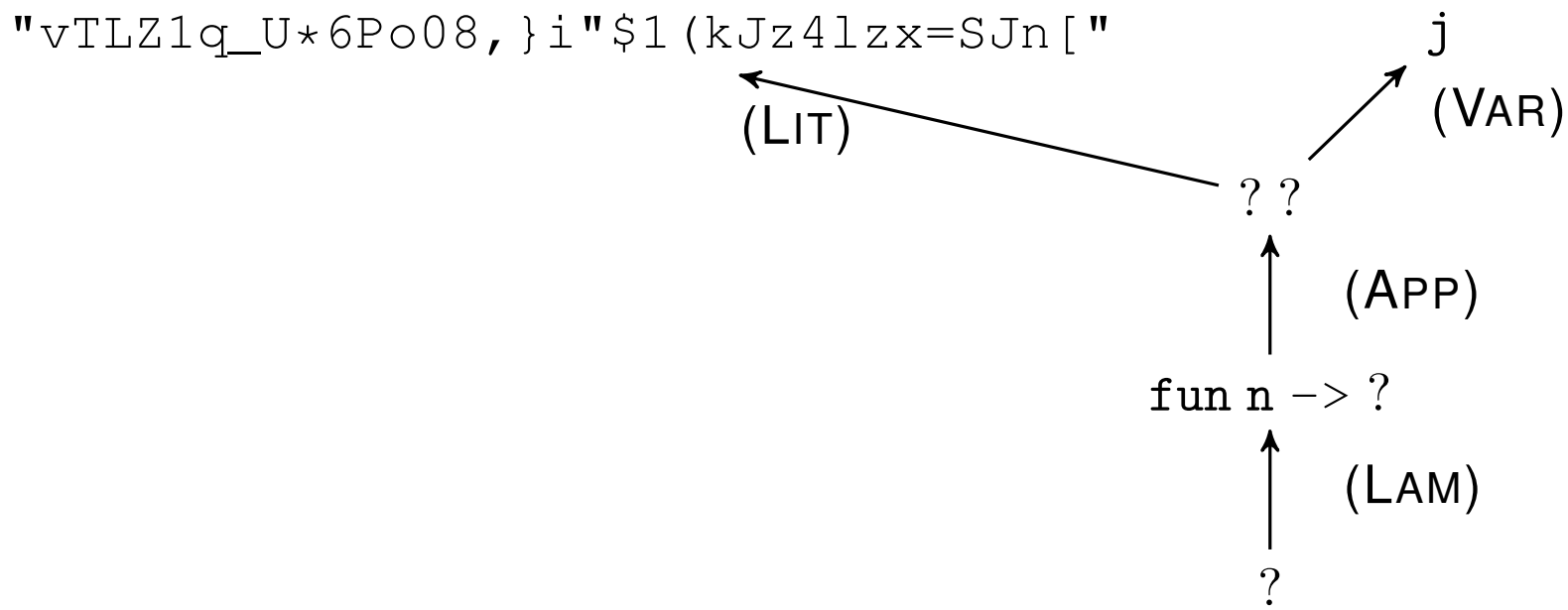


Few of these will make it through the type checker. . .

Program generation, take 2

Follow the grammar (Celento-al:SPE80):

$$e \;::=\; c \;\mid\; \texttt{x} \;\mid\; \texttt{fun x -> } e \;\mid\; e_0\ e_1$$



Few of these will make it through the type checker. . .

Variant: de Bruijn notation

(+ count the number of enclosing binders)

Program generation, take 3

Consider the standard simply-typed λ-calculus rules:

$$\frac{\Delta(c) = \tau}{\Delta; \Gamma \vdash c : \tau}\ (\textsc{Const}) \qquad \frac{(\mathtt{x} : \tau) \in \Gamma}{\Delta; \Gamma \vdash \mathtt{x} : \tau}\ (\textsc{Var})$$

$$\frac{\Delta; \Gamma, (\mathtt{x} : \tau_1) \vdash e : \tau_2}{\Delta; \Gamma \vdash \mathtt{fun\ x\ ->}\ e : \tau_1 \to \tau_2}\ (\textsc{Lam})$$

$$\frac{\Delta; \Gamma \vdash e_0 : \tau_1 \to \tau_2 \qquad \Delta; \Gamma \vdash e_1 : \tau_1}{\Delta; \Gamma \vdash e_0\ e_1 : \tau_2}\ (\textsc{App})$$

Explicit parameter Δ: a type environment for literals

Program generation, take 3

Bottom-up reading of the typing relation (Pałka-al:AST11):

$$\Delta; \Gamma \vdash ? : \texttt{int}$$

Program generation, take 3

Bottom-up reading of the typing relation (Pałka-al:AST11):

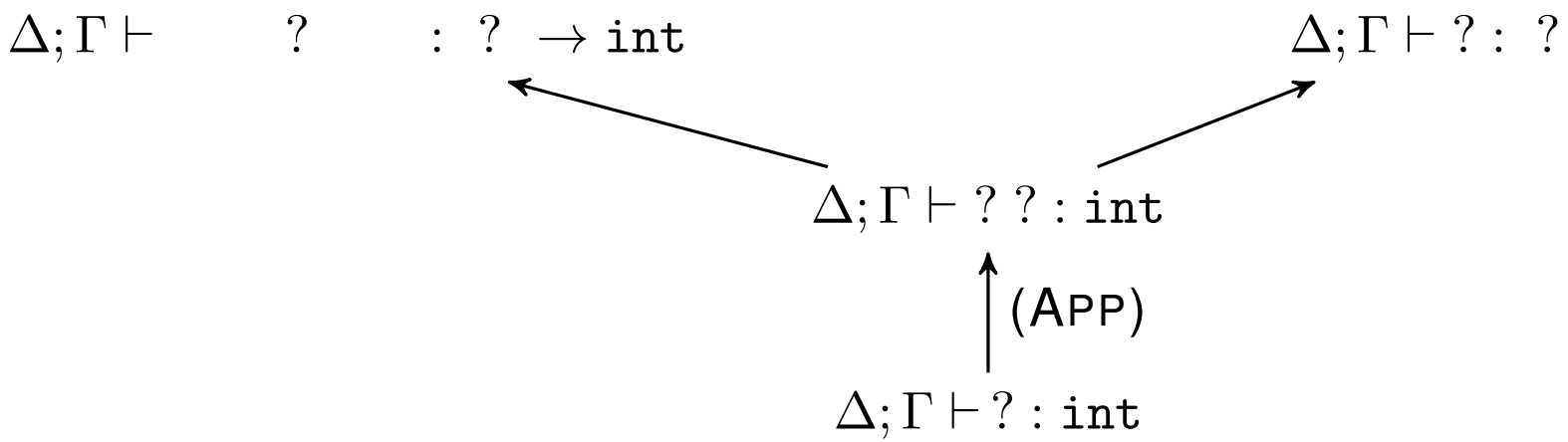
$$\Delta; \Gamma \vdash ?\ ? : \texttt{int}$$

$$\uparrow\ (\textsc{App})$$

$$\Delta; \Gamma \vdash ? : \texttt{int}$$

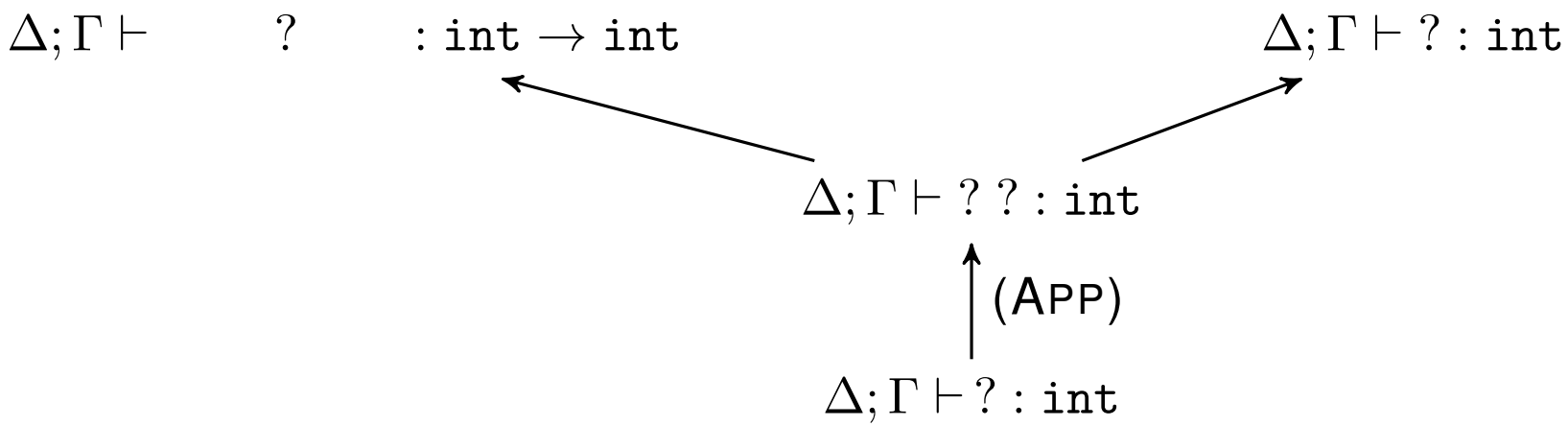
Program generation, take 3

Bottom-up reading of the typing relation (Pałka-al:AST11):



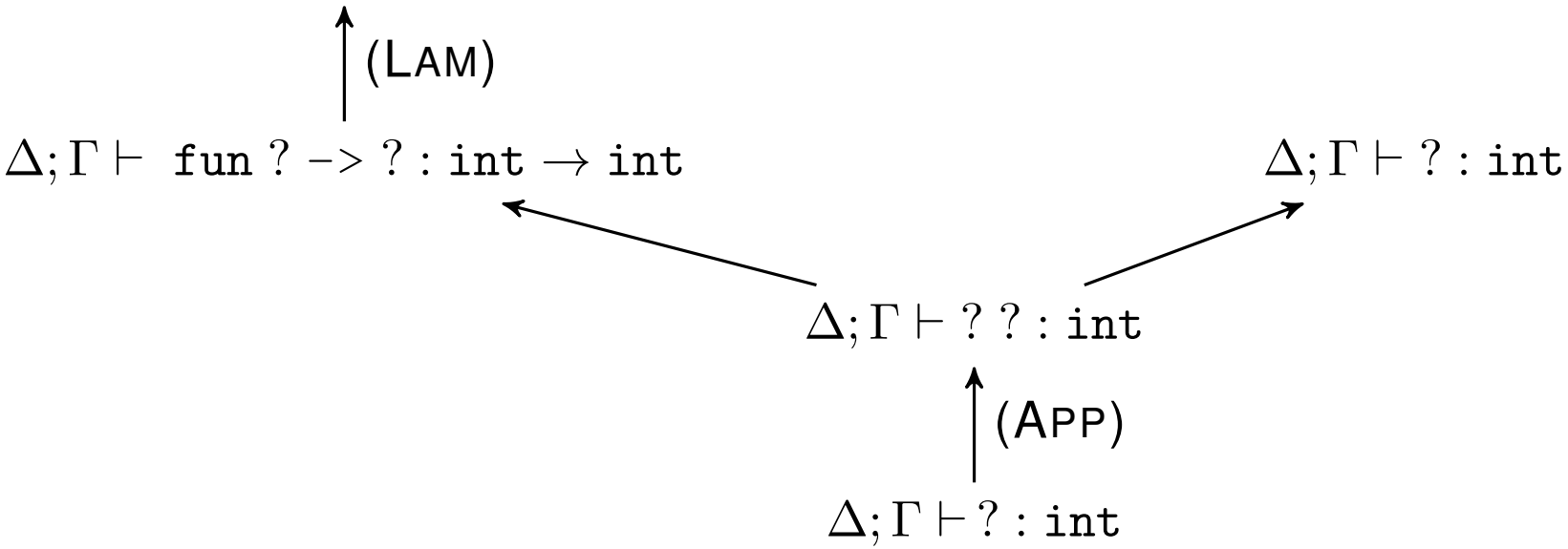
Program generation, take 3

Bottom-up reading of the typing relation (Pałka-al:AST11):



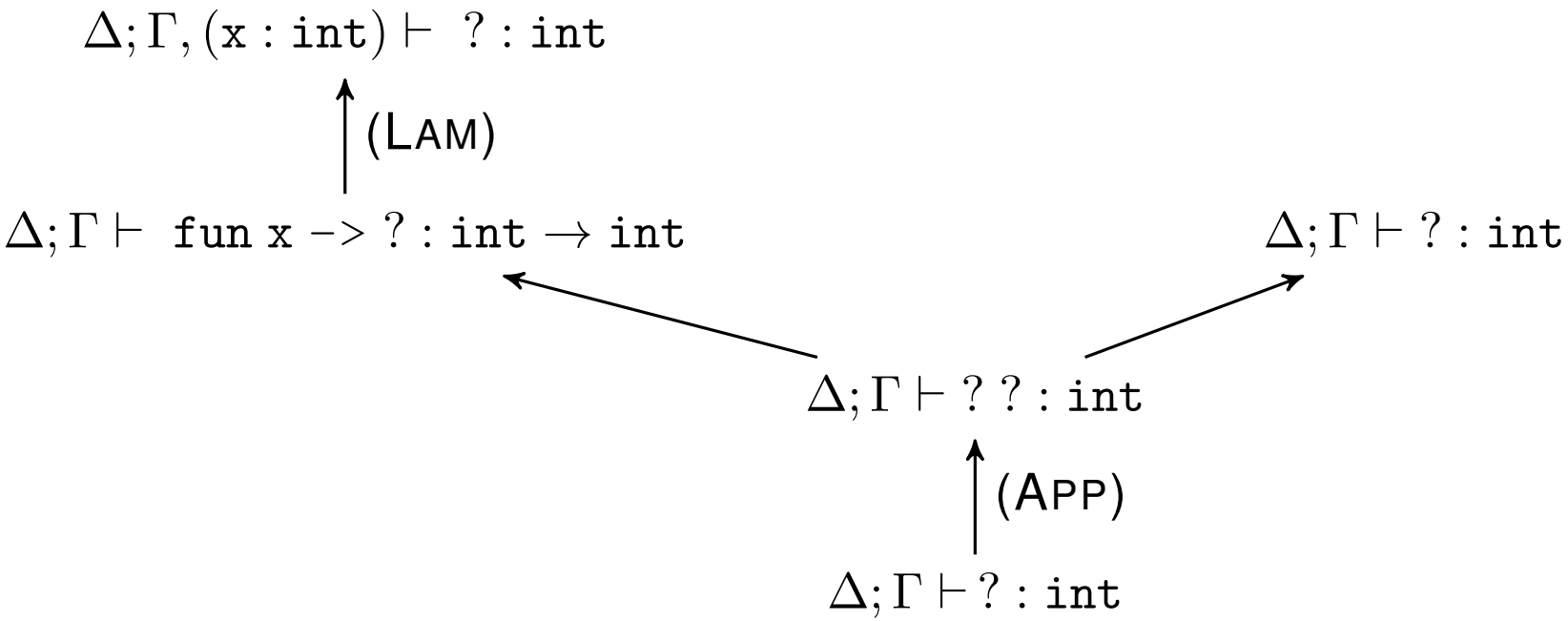
Program generation, take 3

Bottom-up reading of the typing relation (Pałka-al:AST11):



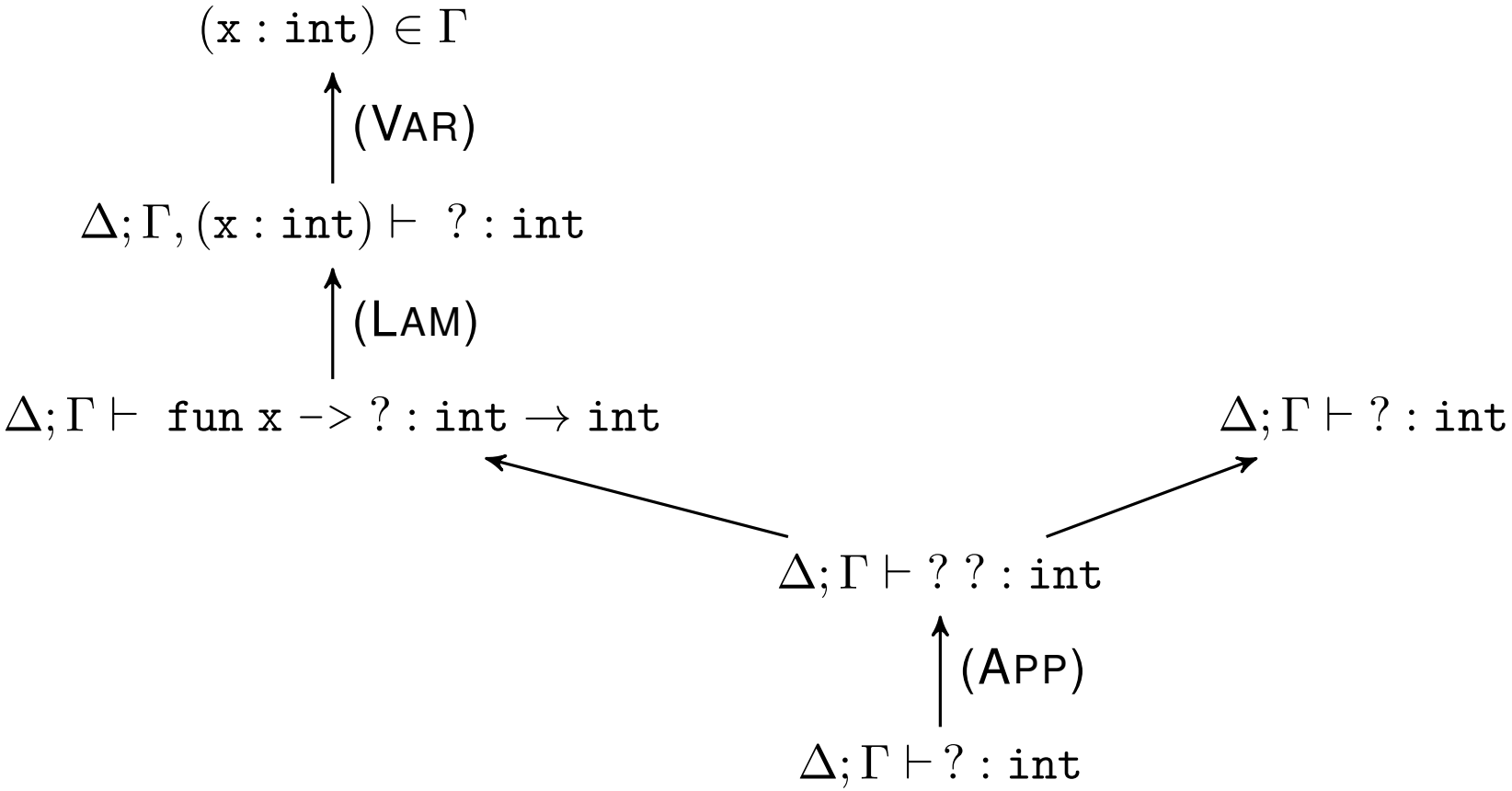
Program generation, take 3

Bottom-up reading of the typing relation (Pałka-al:AST11):



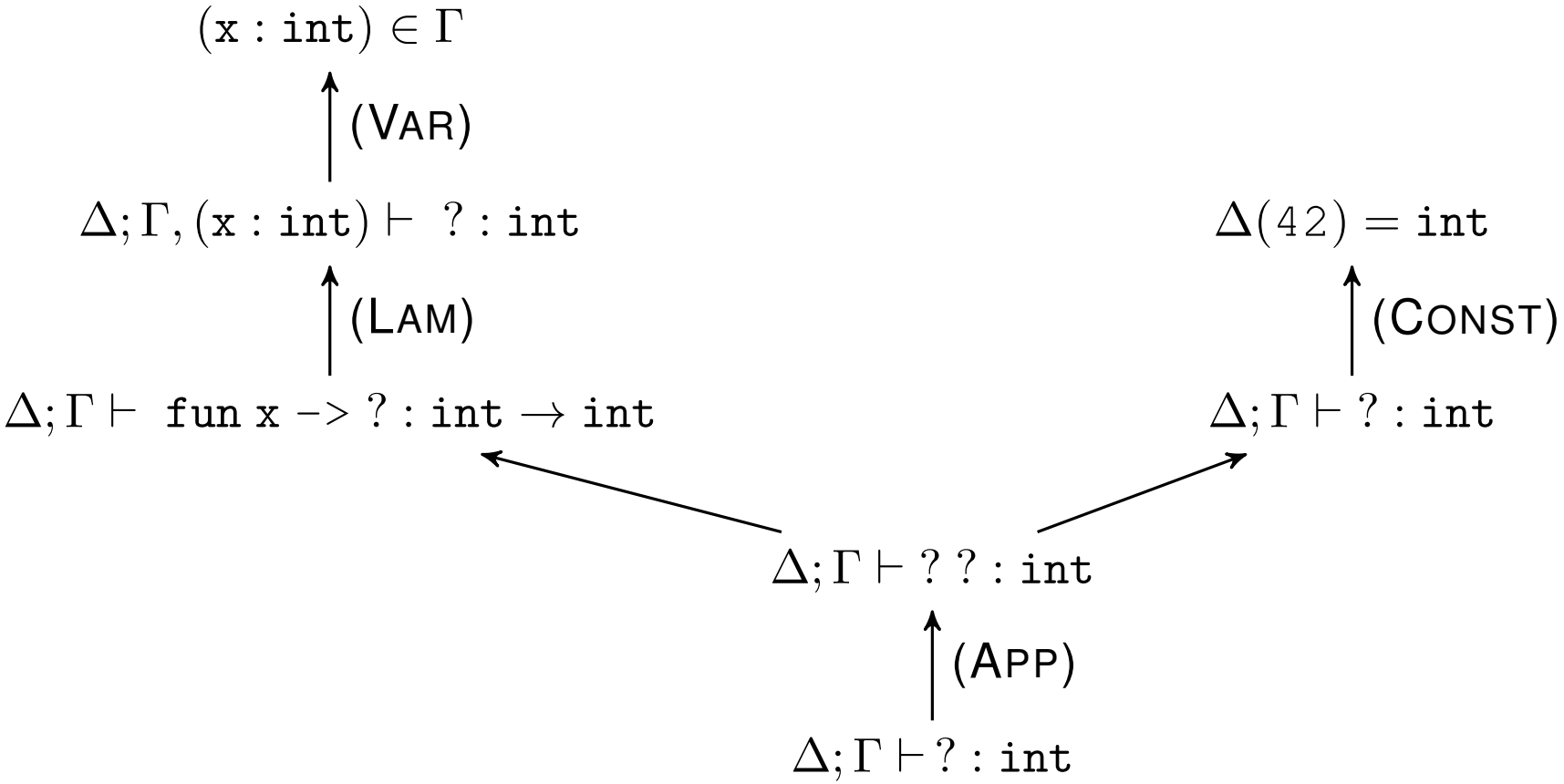
Program generation, take 3

Bottom-up reading of the typing relation (Pałka-al:AST11):



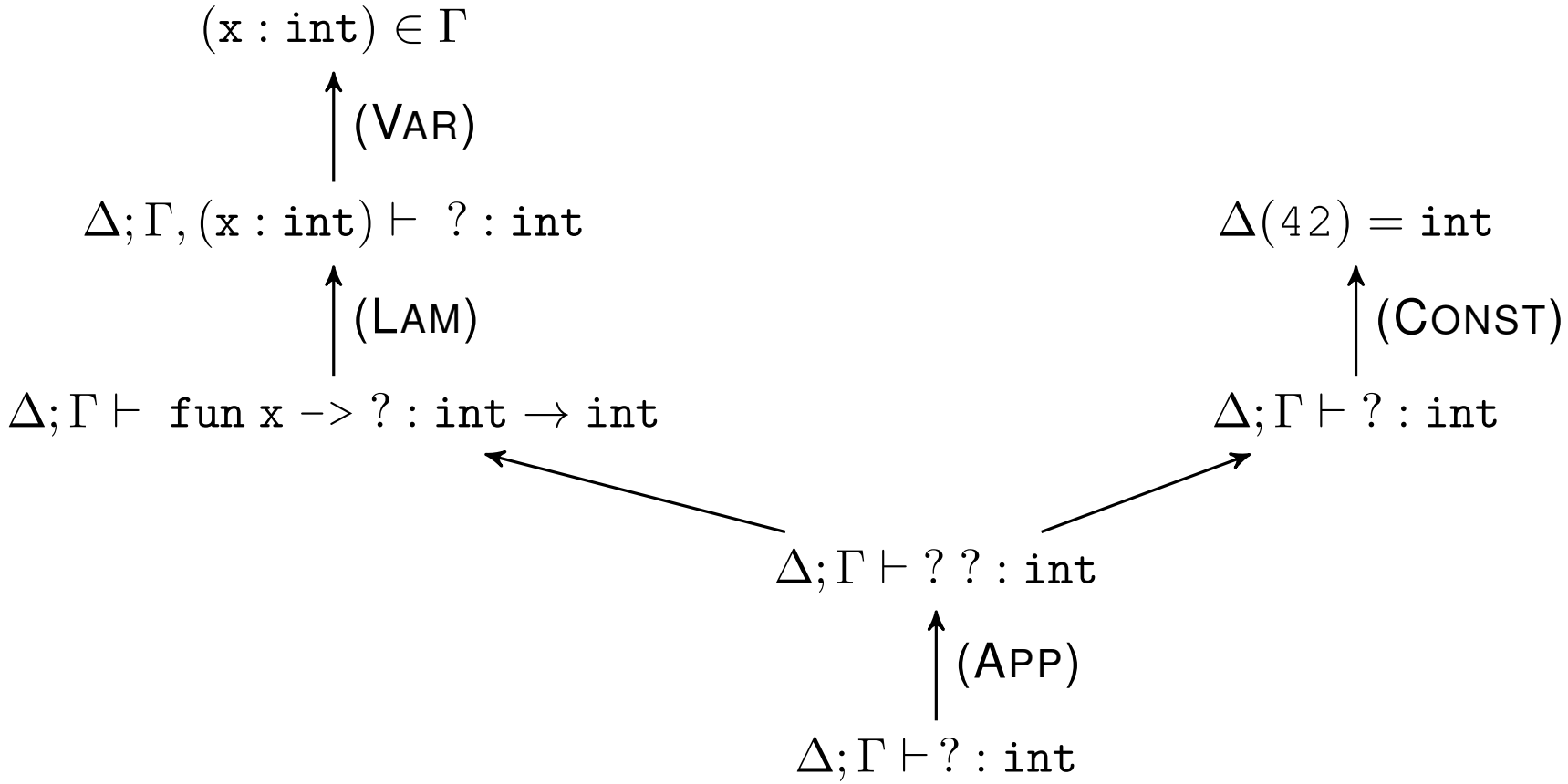
Program generation, take 3

Bottom-up reading of the typing relation (Pałka-al:AST11):



Program generation, take 3

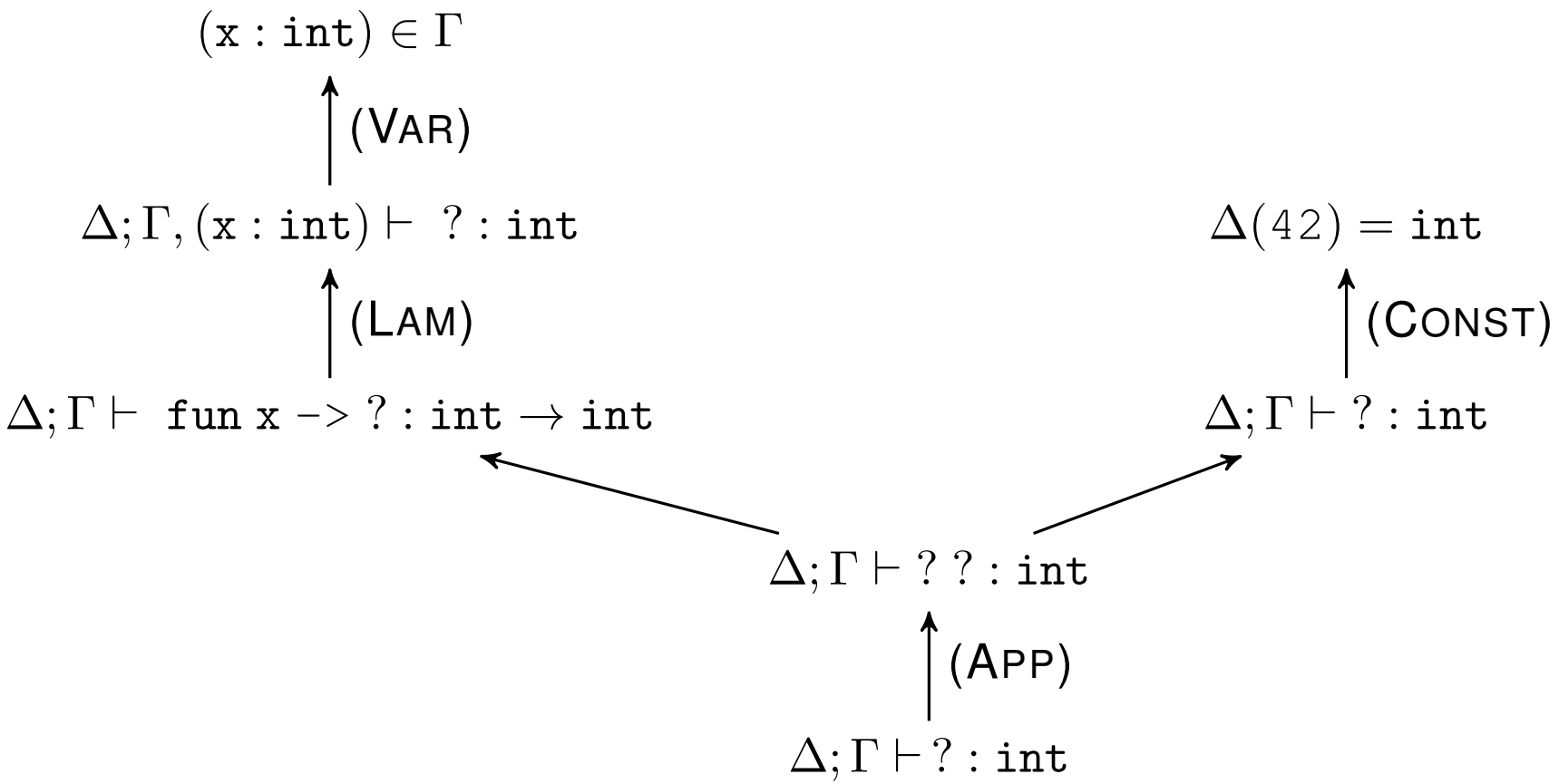
Bottom-up reading of the typing relation (Pałka-al:AST11):



Output guaranteed to make it through the type checker!

Program generation, take 3

Bottom-up reading of the typing relation (Pałka-al:AST11):



Output guaranteed to make it through the type checker!

Parameters: initial type environment and the goal type

Evaluation-order dependence

The observed behavior of an OCaml program may depend on the evaluation order:

```
((fun x -> fun y -> ())
              (print_int 0)) (print_int 1)
```

Evaluation-order dependence

The observed behavior of an OCaml program may depend on the evaluation order:

```
((fun x -> fun y -> ())
              (print_int 0)) (print_int 1)
```

Under left-to-right evaluation: prints `01`

Evaluation-order dependence

The observed behavior of an OCaml program may depend on the evaluation order:

```
((fun x -> fun y -> ())
              (print_int 0)) (print_int 1)
```

Under left-to-right evaluation: prints `01`
Under right-to-left evaluation: prints `10`

Evaluation-order dependence

The observed behavior of an OCaml program may depend on the evaluation order:

```
((fun x -> fun y -> ())
              (print_int 0)) (print_int 1)
```

Under left-to-right evaluation: prints `01`
Under right-to-left evaluation: prints `10`

In OCaml (and many other languages, like C, C++, Scheme, . . .) the evaluation order is unspecified

- OCaml's bytecode backend uses right-to-left
- The native code backend sometimes uses left-to-right

Evaluation-order dependence

The observed behavior of an OCaml program may depend on the evaluation order:

```
((fun x -> fun y -> ())
                (print_int 0)) (print_int 1)
```

Under left-to-right evaluation: prints `01`
Under right-to-left evaluation: prints `10`

In OCaml (and many other languages, like C, C++, Scheme, . . .) the evaluation order is unspecified

- OCaml's bytecode backend uses right-to-left
- The native code backend sometimes uses left-to-right

For testing purposes: Signal-to-noise ratio too low

(How) Can we avoid generating such programs?

First refinement: expressions may have effect

One bit $ef \in \{\texttt{tt}, \texttt{ff}\}$ tracks potential run-time effects:

First refinement: expressions may have effect

One bit $ef \in \{\texttt{tt}, \texttt{ff}\}$ tracks potential run-time effects:

$$\frac{\Delta(c) = \tau \qquad \tau \sqsubseteq \tau'}{\Delta; \Gamma \vdash c : \tau' \,\&\, ef} \text{ (ECONST)} \qquad \frac{(\texttt{x} : \tau) \in \Gamma \qquad \tau \sqsubseteq \tau'}{\Delta; \Gamma \vdash \texttt{x} : \tau' \,\&\, ef} \text{ (EVAR)}$$

$$\frac{\Delta; \Gamma, (\texttt{x} : \tau_1) \vdash e : \tau_2 \,\&\, ef_1}{\Delta; \Gamma \vdash \texttt{fun x -> } e : \tau_1 \xrightarrow{ef_1} \tau_2 \,\&\, ef} \text{ (ELAM)}$$

$$\frac{\Delta; \Gamma \vdash e_0 : \tau_1 \xrightarrow{ef} \tau_2 \,\&\, ef_0 \qquad \Delta; \Gamma \vdash e_1 : \tau_1 \,\&\, ef_1 \qquad ef \vee ef_0 \vee ef_1 \implies ef'}{\Delta; \Gamma \vdash e_0\ e_1 : \tau_2 \,\&\, ef'} \text{ (EAPP)}$$

Sub-effecting allows non-effectful e to replace effectful e

From 'may-effects' to evaluation-order dep.

If both e_0 and e_1 in $e_0\ e_1$ may have effects

then the application is evaluation-order dependent

We can model this with a second bit ev

The effect is now a pair of bits:

$$\varphi ::= ef\,/\,ev \qquad \text{with } ef, ev \in \{\texttt{tt}, \texttt{ff}\}$$

These are both ordered by implication ordering

(componentwise lifted to pairs)

(Sub-)Effect-system

$$\frac{\Delta(c) = \tau \qquad \tau \sqsubseteq \tau'}{\Delta; \Gamma \vdash c : \tau' \,\&\, \varphi}\ (\textsc{EConst}) \qquad \frac{(\mathtt{x} : \tau) \in \Gamma \qquad \tau \sqsubseteq \tau'}{\Delta; \Gamma \vdash \mathtt{x} : \tau' \,\&\, \varphi}\ (\textsc{EVar})$$

$$\frac{\Delta; \Gamma, (\mathtt{x} : \tau_1) \vdash e : \tau_2 \,\&\, \varphi_1}{\Delta; \Gamma \vdash \mathtt{fun\ x\ ->}\ e : \tau_1 \xrightarrow{\varphi_1} \tau_2 \,\&\, \varphi}\ (\textsc{ELam})$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \,\&\, \varphi_1 \qquad \Delta; \Gamma \vdash e_0 : \tau_1 \xrightarrow{\varphi} \tau_2 \,\&\, \varphi_0 \qquad \varphi \sqcup \varphi_0 \sqcup \varphi_1 \sqsubseteq \varphi' \qquad \varphi_0 \sqcap \varphi_1 \Rrightarrow \varphi'}{\Delta; \Gamma \vdash e_0\ e_1 : \tau_2 \,\&\, \varphi'}\ (\textsc{EApp})$$

The $\Rrightarrow$ operator over pairs propagates from ef to ev

Intermezzo: soundness

Operational semantics of non-det. eval. order

$$\frac{e_0 \xrightarrow{\eta} e_0'}{e_0\ e_1 \xrightarrow{\eta} e_0'\ e_1}\ (\textsc{AppL}) \qquad \frac{e_1 \xrightarrow{\eta} e_1'}{e_0\ e_1 \xrightarrow{\eta} e_0\ e_1'}\ (\textsc{AppR})$$

$$\frac{}{(\texttt{fun x -> } e)\ val \xrightarrow{\epsilon} e[\texttt{x} \mapsto val]}\ (\textsc{AppLam})$$

$$\frac{\delta(c\ val_1\ \ldots\ val_n) = (val, \eta)}{c\ val_1\ \ldots\ val_n \xrightarrow{\eta} val}\ (\textsc{AppDelta})$$

This specifies a call-by-value semantics

Note how (AppL) and (AppR) allows for both left-to-right and right-to-left evaluation (and even interleaved evaluation)

Soundness of formalization (1/2)

Theorem: Preservation and progress

- if $\Delta; \cdot \vdash e : \tau \,\&\, \varphi$ and there exists e' such that $e \xrightarrow{\eta} e'$ then $\Delta; \cdot \vdash e' : \tau \,\&\, \varphi$

Soundness of formalization (1/2)

Theorem: Preservation and progress

- if $\Delta; \cdot \vdash e : \tau \,\&\, \varphi$ and there exists e' such that $e \xrightarrow{\eta} e'$ then $\Delta; \cdot \vdash e' : \tau \,\&\, \varphi$
- if $\Delta; \cdot \vdash e : \tau \,\&\, \varphi$ then either $e = val$ or there exists e' such that $e \xrightarrow{\eta} e'$

This ensures correctness of the underlying type system

It says nothing about effects

Soundness of formalization (2/2)

Theorem: Soundness of effect bit

If $\Delta; \Gamma \vdash e : \tau \,\&\, ef \,/\, ev$ and $e \xrightarrow{\eta_1} e_1 \xrightarrow{\eta_2} \ldots \xrightarrow{\eta_n} e_n$ and there exists i such that $\eta_i \neq \epsilon$ then $ef = \texttt{tt}$

Read: ef correctly anticipates run-time effects

Soundness of formalization (2/2)

Theorem: Soundness of effect bit

If $\Delta; \Gamma \vdash e : \tau \,\&\, ef \,/\, ev$ and $e \xrightarrow{\eta_1} e_1 \xrightarrow{\eta_2} \ldots \xrightarrow{\eta_n} e_n$ and there exists i such that $\eta_i \neq \epsilon$ then $ef = \texttt{tt}$

Read: ef correctly anticipates run-time effects

Theorem: Soundness of evaluation-order bit

If $\Delta; \Gamma \vdash e : \tau \,\&\, ef \,/\, ev$ and $e \xrightarrow{\eta_1} e_1 \xrightarrow{\eta_2} \ldots \xrightarrow{\eta_n} e_n = val$ and $e \xrightarrow{\eta'_1} e'_1 \xrightarrow{\eta'_2} \ldots \xrightarrow{\eta'_{n'}} e'_{n'} = val'$ and $ev = \texttt{ff}$ then $\eta_1 \eta_2 \ldots \eta_n = \eta'_1 \eta'_2 \ldots \eta'_{n'}$

Read: ev correctly anticipates evaluation order dependence (think contrapositively)

Enough refinement: Back to testing

- We can now read the effect-system bottom-up to drive a generator
- If our goal is $\tau \,\&\, \texttt{tt}/\texttt{ff}$ the output programs may have effects but not depend on evaluation order

Enough refinement: Back to testing

- We can now read the effect-system bottom-up to drive a generator
- If our goal is τ `&` `tt/ff` the output programs may have effects but not depend on evaluation order
- Our prototype implementation
  - also supports `let` and `if` and more builtin types
  - uses a selection of bindings from `Pervasives` as its initial typing environment
  - uses `int` `&` `tt/ff` as its goal
  - wraps result in `let i = _ in print_int i` to ensure some output
  - compares the output of 500 programs

From counterexample to bug report

On the need for shrinking (1/2)

Dear Xavier et al.

I believe I've found a bug in one of the compiler backends. Please consider the attached program, which exhibits different behaviour when compiled with the two backends.

Yours truly,

Jan

On the need for shrinking (1/2)

Dear Xavier et al.

I believe I've found a bug in one of the compiler backends. Please consider the attached program, which exhibits different behaviour when compiled with the two backends.

Yours truly,

Jan

```
let i = (let a = let s = (let x = exit (lnot
    (-47628762612628909 7)) in fun r -> fun k -> "zsqykify")
    () in exit (exit (let c = (mod) 516587874847912894
    (-453400715012078364 8) in 0)) in (asr) (if (fun x -> let
     h = 2 in false) (let r = (@) [] [] in (lsl)
    2273200848021743825 0) then 466 else if let b = "" in
    false then ( * ) 2 (-4611686018427387904) else let m = 2
     in m) ((fun e -> if exit 5 then int_of_string "vgjgme"
    else (land) 4 7) (fun m -> (&&) (let i = () in false) (
    if true then true else true)))) in print_int i
```

On the need for shrinking (2/2)

Dear Xavier et al.

I believe I've found a bug in one of the compiler backends. Please consider the attached program, which exhibits different behaviour when compiled with the two backends.

Yours truly,

Jan

```
let s = (let x = exit 0 in fun r -> fun k -> "") () in 0
```

Shrinking counterexample programs

We formulate a type-preserving shrinker

Shrinking counterexample programs

We formulate a type-preserving shrinker
that performs a sequence of “micro-shrinking” steps:

Shrinking counterexample programs

We formulate a type-preserving shrinker
that performs a sequence of “micro-shrinking” steps:

$$\texttt{823987324972} \rightsquigarrow \texttt{0} \qquad \text{(shrink lit.)}$$

Shrinking counterexample programs

We formulate a type-preserving shrinker
that performs a sequence of “micro-shrinking” steps:

$$\texttt{823987324972} \rightsquigarrow \texttt{0} \qquad \text{(shrink lit.)}$$

$$\texttt{e : int} \rightsquigarrow \texttt{0} \qquad \text{(replace by lit.)}$$

Shrinking counterexample programs

We formulate a type-preserving shrinker
that performs a sequence of "micro-shrinking" steps:

```
823987324972 ⇝ 0
```
(shrink lit.)

```
e : int ⇝ 0
```
(replace by lit.)

```
(fun x -> e) e' ⇝ let x = e' in e
```
(imm. app.)

Shrinking counterexample programs

We formulate a type-preserving shrinker
that performs a sequence of "micro-shrinking" steps:

```
823987324972 ⇝ 0
```
(shrink lit.)

```
e : int ⇝ 0
```
(replace by lit.)

```
(fun x -> e) e' ⇝ let x = e' in e
```
(imm. app.)

```
fun x -> e ⇝ fun x -> e'  if e ⇝ e'
```
(rec. shrink body)

Shrinking counterexample programs

We formulate a type-preserving shrinker
that performs a sequence of "micro-shrinking" steps:

$$\texttt{823987324972} \rightsquigarrow \texttt{0} \qquad \text{(shrink lit.)}$$

$$\texttt{e : int} \rightsquigarrow \texttt{0} \qquad \text{(replace by lit.)}$$

$$\texttt{(fun x -> e) e'} \rightsquigarrow \texttt{let x = e' in e} \qquad \text{(imm. app.)}$$

$$\texttt{fun x -> e} \rightsquigarrow \texttt{fun x -> e'} \text{ if } \texttt{e} \rightsquigarrow \texttt{e'} \qquad \text{(rec. shrink body)}$$

$$\texttt{let x = e in e'} \rightsquigarrow \texttt{e'} \text{ if } \texttt{x} \notin FV(\texttt{e'}) \qquad \text{(remove unneeded binding)}$$

Shrinking counterexample programs

We formulate a type-preserving shrinker
that performs a sequence of "micro-shrinking" steps:

$$\texttt{823987324972} \rightsquigarrow \texttt{0} \qquad \text{(shrink lit.)}$$

$$\texttt{e : int} \rightsquigarrow \texttt{0} \qquad \text{(replace by lit.)}$$

$$\texttt{(fun x -> e) e'} \rightsquigarrow \texttt{let x = e' in e} \qquad \text{(imm. app.)}$$

$$\texttt{fun x -> e} \rightsquigarrow \texttt{fun x -> e'} \text{ if } \texttt{e} \rightsquigarrow \texttt{e'} \qquad \text{(rec. shrink body)}$$

$$\texttt{let x = e in e'} \rightsquigarrow \texttt{e'} \text{ if } \texttt{x} \notin FV(\texttt{e'}) \qquad \text{(remove unneeded binding)}$$

$$\texttt{if e then e' else e''} \rightsquigarrow \texttt{e'} \qquad \text{(keep branch)}$$

Summary and conclusion

- An effect system for evaluation order dependence, formalized and proved sound
- We let the effect system drive a program generator in a prototype compiler tester, incl. a shrinker
- A principled approach to FP generation, that combines core techniques from our community.
- We found 4 bugs (and rediscovered 2) in the OCaml native code backend this way. All fixed in 4.05.0.
- This approach has potential as next generation quality assurance of FP compiler optimizations.

Summary and conclusion

- An effect system for evaluation order dependence, formalized and proved sound
- We let the effect system drive a program generator in a prototype compiler tester, incl. a shrinker
- A principled approach to FP generation, that combines core techniques from our community.
- We found 4 bugs (and rediscovered 2) in the OCaml native code backend this way. All fixed in 4.05.0.
- This approach has potential as next generation quality assurance of FP compiler optimizations.

`https://github.com/jmid/efftester`

Summary and conclusion

- An effect system for evaluation order dependence, formalized and proved sound
- We let the effect system drive a program generator in a prototype compiler tester, incl. a shrinker
- A principled approach to FP generation, that combines core techniques from our community.
- We found 4 bugs (and rediscovered 2) in the OCaml native code backend this way. All fixed in 4.05.0.
- This approach has potential as next generation quality assurance of FP compiler optimizations.

`https://github.com/jmid/efftester`

Thanks