SM2-TES: Functional Programming and Property-Based Testing, Day 9

Jan Midtgaard

MMMI, SDU

SDU

Last time. . .

Case study: PBT of Computational Geometry

Coverage (Generally and to inform PBT)

A generator of syntactically correct programs

- following the language grammar
- with several non-terminals
- passing an environment of variables
- used for testing the `bc` calculator (with timeouts)

Caveat: only one type of integers. . .

Outline

Intermezzo: ML typing

Typed Program Generation

Shrinking programs

Intermezzo: ML typing

Formal reasoning

Before we get to compiler testing, I want to talk a bit about OCaml's type system.

To do so, I need to talk a bit about formal reasoning.

One approach to express a formal system for reasoning is by means of inference rules:

$$\frac{P}{Q}\ (\textsc{Rule name})$$

P is the premise and Q is the conclusion.

You should read it as *"if P holds then Q holds"*

(A rule without any premises is called an *axiom*)

Example: A parity system

This system formally decides a natural number's parity:

$$\frac{}{0\ isEven}\ (\textsc{ZeroEven}) \qquad \frac{n\ isOdd}{n+1\ isEven}\ (\textsc{SuccOdd})$$

$$\frac{n\ isEven}{n+1\ isOdd}\ (\textsc{SuccEven})$$

The axiom ZeroEven tells us the parity of base case 0.

The two rules SuccOdd and SuccEven tell us the parity of successor numbers, e.g., for SuccEven:

> *If we've established that some number n is even, then we can conclude that $n+1$ is odd*

A derivation tree

By instantiating the variables (replacing an n with an actual number) we can build a derivation (or proof) tree:

A derivation tree

By instantiating the variables (replacing an n with an actual number) we can build a derivation (or proof) tree:

$$\frac{}{0\ isEven}\ (\textsc{ZeroEven})$$

A derivation tree

By instantiating the variables (replacing an n with an actual number) we can build a derivation (or proof) tree:

$$\dfrac{\dfrac{}{0\ isEven}\ (\textsc{ZeroEven})}{1\ isOdd}\ (\textsc{SuccEven})$$

A derivation tree

By instantiating the variables (replacing an n with an actual number) we can build a derivation (or proof) tree:

$$\dfrac{\dfrac{\dfrac{}{0\ isEven}\ (\textsc{ZeroEven})}{1\ isOdd}\ (\textsc{SuccEven})}{2\ isEven}\ (\textsc{SuccOdd})$$

A derivation tree

By instantiating the variables (replacing an n with an actual number) we can build a derivation (or proof) tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0\ isEven}\,(\textsc{ZeroEven})}{1\ isOdd}\,(\textsc{SuccEven})}{2\ isEven}\,(\textsc{SuccOdd})}{3\ isOdd}\,(\textsc{SuccEven})$$

A derivation tree

By instantiating the variables (replacing an n with an actual number) we can build a derivation (or proof) tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0\ isEven}\ (\textsc{ZeroEven})}{1\ isOdd}\ (\textsc{SuccEven})}{2\ isEven}\ (\textsc{SuccOdd})}{3\ isOdd}\ (\textsc{SuccEven})$$

Such a system of inference rules is a useful vehicle to concisely develop, specify, and test(!) type systems

(that aren't too ad hoc)

Example: Grammars in inference form

We can even formulate a grammar

$$e \;::=\; \mathtt{x} \;\mid\; i \;\mid\; e + e \;\mid\; e * e$$

as a system of inference rules:

$$\frac{}{\mathtt{x}\ isExp}\ (\textsc{Var}) \qquad \frac{}{i\ isExp}\ (\textsc{Literal})$$

$$\frac{e\ isExp \qquad e'\ isExp}{e + e'\ isExp}\ (\textsc{Sum})$$

$$\frac{e\ isExp \qquad e'\ isExp}{e * e'\ isExp}\ (\textsc{Prod})$$

Back to type systems

Formally we can study a simplified subset of OCaml defined by this grammar of expressions:

$$
\begin{array}{rcll}
e & ::= & \texttt{x} & \text{(variables)} \\
& | & \texttt{fun x -> } e & \text{(functions)} \\
& | & e_0\ e_1 & \text{(calls)} \\
& | & (e_0, e_1) & \text{(pairs)} \\
& | & \texttt{fst } e & \text{(first projection)} \\
& | & \texttt{snd } e & \text{(snd projection)}
\end{array}
$$

where I have thrown in pairs and `fst` and `snd` from the standard library.

Back to type systems

We first phrase a grammar of types for this language:

$$\begin{array}{rcll} \tau & ::= & bt & \text{(base types)} \\ & \mid & \tau_1 \to \tau_2 & \text{(arrow types)} \\ & \mid & \tau_1 * \tau_2 & \text{(pair types)} \end{array}$$

I haven't specified base types bt so imagine it includes `unit`, `int`, `...`

Function types are written with arrows, e.g., $\texttt{int} \to \texttt{unit}$ and pair types are written with an asterisk, e.g., $\texttt{int} * \texttt{int}$.

Back to type systems

We first phrase a grammar of types for this language:

$$\begin{array}{rcll} \tau & ::= & bt & \text{(base types)} \\ & \mid & \tau_1 \to \tau_2 & \text{(arrow types)} \\ & \mid & \tau_1 * \tau_2 & \text{(pair types)} \end{array}$$

I haven't specified base types bt so imagine it includes `unit`, `int`, ...

Function types are written with arrows, e.g., $\texttt{int} \to \texttt{unit}$ and pair types are written with an asterisk, e.g., $\texttt{int} * \texttt{int}$.

Finally we need *type environments* Γ: a map that tell us the type of variables in scope:

$$\begin{array}{rcll} \Gamma & ::= & \cdot & \text{(empty type env.)} \\ & \mid & \Gamma, (\mathtt{x} : \tau) & \text{(extended type env.)} \end{array}$$

Typing rules

$$\frac{(\mathtt{x} : \tau) \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma, (\mathtt{x} : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun\ x\ ->}\ e : \tau_1 \to \tau_2}\ (\textsc{Lam})$$

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

$$\frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash (e_0,\ e_1) : \tau_0 * \tau_1}\ (\textsc{Pair}) \qquad \frac{\Gamma \vdash e : \tau_0 * \tau_1}{\Gamma \vdash \mathtt{fst}\ e : \tau_0}\ (\textsc{Fst})$$

$$\frac{\Gamma \vdash e : \tau_0 * \tau_1}{\Gamma \vdash \mathtt{snd}\ e : \tau_1}\ (\textsc{Snd})$$

These are the typing rules of *"simply-typed λ-calculus"*

The VAR rule

$$\frac{(\mathtt{x} : \tau) \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau}\ (\textsc{Var})$$

"If in type environment Γ we have recorded that $\mathtt{x}$ is in scope and has type τ, then we can conclude it"

The APP rule

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

"If in type environment Γ the receiver e_0 type checks with some function type $\tau_1 \to \tau_2$ and the argument e_1 type checks with the same argument type τ_1 then the call $e_0\ e_1$ type checks with type τ_2."

The LAM rule

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

"If in an extended type environment Γ
(where the parameter $\mathtt{x}$ *is assigned some type* τ_1*)*
the function body e *type checks with type* τ_2
then the function type checks with type $\tau_1 \to \tau_2$*."*

The PAIR rule

$$\frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash (e_0,\ e_1) : \tau_0 * \tau_1}\ \text{(PAIR)}$$

"If in type environment Γ
the first component e_0 type checks with type τ_0 and
the second component e_1 type checks with type τ_1
then the pair $(e_0,\ e_1)$ type checks with type $\tau_0 * \tau_1$."

The FST rule

$$\frac{\Gamma \vdash e : \tau_0 * \tau_1}{\Gamma \vdash \texttt{fst}\ e : \tau_0}\ (\textsc{Fst})$$

"If in type environment Γ
the expression e type checks with pair type $\tau_0 * \tau_1$
then the first projection type checks with type τ_0."

The SND rule

$$\frac{\Gamma \vdash e : \tau_0 * \tau_1}{\Gamma \vdash \texttt{snd}\ e : \tau_1}\ (\textsc{Snd})$$

"If in type environment Γ
the expression e type checks with pair type $\tau_0 * \tau_1$
then the second projection type checks with type τ_1."

An example derivation tree

Running the type checker corresponds to building a derivation tree:

$$\frac{}{\Gamma \vdash (\texttt{fun x -> x})\ \texttt{max\_int} : \texttt{int}}\ (\textsc{App})$$

An example derivation tree

Running the type checker corresponds to building a derivation tree:

$$\dfrac{\dfrac{}{\Gamma \vdash (\texttt{fun x -> x}) : \texttt{int} \to \texttt{int}}\,(\textsc{Lam}) \qquad \cdots}{\Gamma \vdash (\texttt{fun x -> x})\ \texttt{max\_int} : \texttt{int}}\,(\textsc{App})$$

An example derivation tree

Running the type checker corresponds to building a derivation tree:

$$\dfrac{(\text{LAM})\ \dfrac{(\text{VAR})\ \dfrac{(\texttt{x} : \texttt{int}) \in \Gamma, (\texttt{x} : \texttt{int})}{\Gamma, (\texttt{x} : \texttt{int}) \vdash \texttt{x} : \texttt{int}}}{\Gamma \vdash (\texttt{fun x -> x}) : \texttt{int} \to \texttt{int}} \qquad}{\Gamma \vdash (\texttt{fun x -> x})\ \texttt{max\_int} : \texttt{int}}\ (\text{APP})$$

An example derivation tree

Running the type checker corresponds to building a derivation tree:

$$\dfrac{(\text{LAM})\ \dfrac{(\text{VAR})\ \dfrac{(\texttt{x} : \texttt{int}) \in \Gamma, (\texttt{x} : \texttt{int})}{\Gamma, (\texttt{x} : \texttt{int}) \vdash \texttt{x} : \texttt{int}}}{\Gamma \vdash (\texttt{fun x -> x}) : \texttt{int} \to \texttt{int}} \qquad \dfrac{(\texttt{max\_int} : \texttt{int}) \in \Gamma}{\Gamma \vdash \texttt{max\_int} : \texttt{int}}\ (\text{VAR})}{\Gamma \vdash (\texttt{fun x -> x})\ \texttt{max\_int} : \texttt{int}}\ (\text{APP})$$

An example derivation tree

Running the type checker corresponds to building a derivation tree:

$$\dfrac{\dfrac{\dfrac{(\texttt{x} : \texttt{int}) \in \Gamma, (\texttt{x} : \texttt{int})}{\Gamma, (\texttt{x} : \texttt{int}) \vdash \texttt{x} : \texttt{int}}\,(\textsc{Var})}{\Gamma \vdash (\texttt{fun x -> x}) : \texttt{int} \to \texttt{int}}\,(\textsc{Lam}) \qquad \dfrac{(\texttt{max\_int} : \texttt{int}) \in \Gamma}{\Gamma \vdash \texttt{max\_int} : \texttt{int}}\,(\textsc{Var})}{\Gamma \vdash (\texttt{fun x -> x})\ \texttt{max\_int} : \texttt{int}}\,(\textsc{App})$$

This is a valid derivation tree if $(\texttt{max\_int} : \texttt{int}) \in \Gamma$.

Intuition:

"this OCaml program type checks if variable `max_int` *is bound in the initial environment with type* `int`*"*

Compare this approach to a textual specification

§4.2.2. Integer Operations

The Java programming language provides a number of operators that act on integral values:

- The comparison operators, which result in a value of type boolean:
  - The numerical comparison operators <, <=, >, and >= (§15.20.1)
  - The numerical equality operators == and != (§15.21.1)
- The numerical operators, which result in a value of type int or long:
  - The unary plus and minus operators + and - (§15.15.3, §15.15.4)
  - The multiplicative operators *, /, and % (§15.17)
  - The additive operators + and - (§15.18)

[...]

If an integer operator other than a shift operator has at least one operand of type long, then the operation is carried out using 64-bit precision, and the result of the numerical operator is of type long. If the other operand is not long, it is first widened (§5.1.5) to type long by numeric promotion (§5.6).

Otherwise, the operation is carried out using 32-bit precision, and the result of the numerical operator is of type int. If either operand is not an int, it is first widened to type int by numeric promotion.

From `https://docs.oracle.com/javase/specs/jls/se8/html/`, Sec.4.2.2 of *'The Java Language Specification'*

Typing rules, reconsidered

$$\frac{(\mathtt{x} : \tau) \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma, (\mathtt{x} : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun\ x\ ->}\ e : \tau_1 \to \tau_2}\ (\textsc{Lam})$$

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

$$\frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash (e_0,\ e_1) : \tau_0 * \tau_1}\ (\textsc{Pair}) \qquad \frac{\Gamma \vdash e : \tau_0 * \tau_1}{\Gamma \vdash \mathtt{fst}\ e : \tau_0}\ (\textsc{Fst})$$

$$\frac{\Gamma \vdash e : \tau_0 * \tau_1}{\Gamma \vdash \mathtt{snd}\ e : \tau_1}\ (\textsc{Snd})$$

Suppose we focus on the types

Typing rules, reconsidered

$$\frac{(\mathtt{x} : \tau) \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma, (\mathtt{x} : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun\ x\ ->}\ e : \tau_1 \to \tau_2}\ (\textsc{Lam})$$

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

$$\frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash (e_0, e_1) : \tau_0 * \tau_1}\ (\textsc{Pair}) \qquad \frac{\Gamma \vdash e : \tau_0 * \tau_1}{\Gamma \vdash \mathtt{fst}\ e : \tau_0}\ (\textsc{Fst})$$

$$\frac{\Gamma \vdash e : \tau_0 * \tau_1}{\Gamma \vdash \mathtt{snd}\ e : \tau_1}\ (\textsc{Snd})$$

Suppose we focus on the types

Typing rules, reconsidered

$$\frac{(\quad \tau) \in \Gamma}{\Gamma \vdash \quad \tau}\ (\textsc{Var}) \qquad \frac{\Gamma, (\quad \tau_1) \vdash \quad \tau_2}{\Gamma \vdash \quad \tau_1 \to \tau_2}\ (\textsc{Lam})$$

$$\frac{\Gamma \vdash \quad \tau_1 \to \tau_2 \qquad \Gamma \vdash \quad \tau_1}{\Gamma \vdash \quad \tau_2}\ (\textsc{App})$$

$$\frac{\Gamma \vdash \quad \tau_0 \qquad \Gamma \vdash \quad \tau_1}{\Gamma \vdash \quad \tau_0 * \tau_1}\ (\textsc{Pair}) \qquad \frac{\Gamma \vdash \quad \tau_0 * \tau_1}{\Gamma \vdash \quad \tau_0}\ (\textsc{Fst})$$

$$\frac{\Gamma \vdash \quad \tau_0 * \tau_1}{\Gamma \vdash \quad \tau_1}\ (\textsc{Snd})$$

Suppose we focus on the types

Typing rules, reconsidered

$$\frac{\tau \in \Gamma}{\Gamma \vdash \tau}\ (\textsc{Var}) \qquad \frac{\Gamma, \tau_1 \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2}\ (\textsc{Lam})$$

$$\frac{\Gamma \vdash \tau_1 \to \tau_2 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash \tau_2}\ (\textsc{App}) \qquad \frac{\Gamma \vdash \tau_0 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash \tau_0 * \tau_1}\ (\textsc{Pair})$$

$$\frac{\Gamma \vdash \tau_0 * \tau_1}{\Gamma \vdash \tau_0}\ (\textsc{Fst}) \qquad \frac{\Gamma \vdash \tau_0 * \tau_1}{\Gamma \vdash \tau_1}\ (\textsc{Snd})$$

What is this system?

Typing rules, reconsidered

$$\frac{\tau \in \Gamma}{\Gamma \vdash \tau}\ (\textsc{Var}) \qquad \frac{\Gamma, \tau_1 \vdash \tau_2}{\Gamma \vdash \tau_1 \Rightarrow \tau_2}\ (\textsc{Lam})$$

$$\frac{\Gamma \vdash \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash \tau_2}\ (\textsc{App}) \qquad \frac{\Gamma \vdash \tau_0 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash \tau_0 \wedge \tau_1}\ (\textsc{Pair})$$

$$\frac{\Gamma \vdash \tau_0 \wedge \tau_1}{\Gamma \vdash \tau_0}\ (\textsc{Fst}) \qquad \frac{\Gamma \vdash \tau_0 \wedge \tau_1}{\Gamma \vdash \tau_1}\ (\textsc{Snd})$$

What is this system?

Suppose we write function and pair types differently. . .

Typing rules, reconsidered

$$\frac{\tau \in \Gamma}{\Gamma \vdash \tau}\ (\textsc{Var}) \qquad \frac{\Gamma, \tau_1 \vdash \tau_2}{\Gamma \vdash \tau_1 \Rightarrow \tau_2}\ (\textsc{Lam})$$

$$\frac{\Gamma \vdash \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash \tau_2}\ (\textsc{App}) \qquad \frac{\Gamma \vdash \tau_0 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash \tau_0 \wedge \tau_1}\ (\textsc{Pair})$$

$$\frac{\Gamma \vdash \tau_0 \wedge \tau_1}{\Gamma \vdash \tau_0}\ (\textsc{Fst}) \qquad \frac{\Gamma \vdash \tau_0 \wedge \tau_1}{\Gamma \vdash \tau_1}\ (\textsc{Snd})$$

What is this system?

Suppose we write function and pair types differently. . .

It looks like some kind of logic!

The VAR rule, reconsidered

$$\frac{\tau \in \Gamma}{\Gamma \vdash \tau}\ (\textsc{Var})$$

"If in our assumptions Γ we have recorded that τ holds, then we can conclude it"

The APP rule, reconsidered

$$\frac{\Gamma \vdash \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash \tau_2}\ (\textsc{App})$$

"If under assumptions Γ we can prove that τ_1 implies τ_2 and that τ_1 holds then we can conclude τ_2."

The LAM rule, reconsidered

$$\frac{\Gamma, \tau_1 \vdash \tau_2}{\Gamma \vdash \tau_1 \Rightarrow \tau_2}\ (\textsc{Lam})$$

"If under the assumptions Γ and τ_1 we can prove τ_2 then we can conclude that τ_1 implies τ_2."

The PAIR rule, reconsidered

$$\frac{\Gamma \vdash \tau_0 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash \tau_0 \wedge \tau_1}\ (\textsc{Pair})$$

"If under the assumptions Γ we can prove τ_0 and τ_1 then we can conclude that τ_0 and τ_1 holds."

The FST rule, reconsidered

$$\frac{\Gamma \vdash \tau_0 \wedge \tau_1}{\Gamma \vdash \tau_0}\ (\textsc{Fst})$$

"If under the assumptions Γ
we can prove the conjunction (and) of τ_0 and τ_1
then we can conclude τ_0."

The SND rule, reconsidered

$$\frac{\Gamma \vdash \tau_0 \wedge \tau_1}{\Gamma \vdash \tau_1}\ (\textsc{Snd})$$

"If under the assumptions Γ
we can prove the conjunction (and) of τ_0 and τ_1
then we can conclude τ_1."

The Curry-Howard correspondence

So in an OCaml-like language (F#, SML, Haskell, . . .)

- we can think of types as a form of logical statements (“proposition”)
- where a type check of a program then corresponds to a proof of the statement

This is called the *Curry-Howard correspondence*

The Curry-Howard correspondence

So in an OCaml-like language (F#, SML, Haskell, . . .)

- we can think of types as a form of logical statements (“proposition”)
- where a type check of a program then corresponds to a proof of the statement

This is called the *Curry-Howard correspondence*

Some people say

> *“Propositions-as-types, proofs-as-programs”*

The Curry-Howard correspondence

So in an OCaml-like language (F#, SML, Haskell, . . .)

- we can think of types as a form of logical statements (“proposition”)
- where a type check of a program then corresponds to a proof of the statement

This is called the *Curry-Howard correspondence*

Some people say

> *“Propositions-as-types, proofs-as-programs”*

It extends to sum types, polymorphic types, . . .

The Curry-Howard correspondence

So in an OCaml-like language (F#, SML, Haskell, . . .)

- we can think of types as a form of logical statements (“proposition”)
- where a type check of a program then corresponds to a proof of the statement

This is called the *Curry-Howard correspondence*

Some people say

“Propositions-as-types, proofs-as-programs”

It extends to sum types, polymorphic types, . . .

Bottom line: A type system *can* have a solid foundation!
It doesn’t have to look like it was put together in a garage. . .

Numbering variables: de Bruijn indices

Variables are a can of worms when working with programs.

Consider the following two functions:

```
fun x -> x          fun y -> y
```

In traditional lambda calculus we would write them as:

$$\lambda x.\ x \qquad\qquad \lambda y.\ y$$

Numbering variables: de Bruijn indices

Variables are a can of worms when working with programs.

Consider the following two functions:

```
fun x -> x          fun y -> y
```

In traditional lambda calculus we would write them as:

$$\lambda x.\ x \qquad\qquad \lambda y.\ y$$

The two are equivalent up to renaming of variables.
Hence we can number the variable according to the nearest function binding it: $\lambda.\ 0$

When more variables are present this becomes clearer:

$$\lambda f.\ \lambda x.\ \lambda y.\ f(x+y) \quad \text{becomes} \quad \lambda.\ \lambda.\ \lambda.\ 2(1+0)$$

[End-of-Intermezzo]

Typed Program Generation

Inference rules for generation

Our starting point is the following well-known typing rules to guide our generator:

$$\frac{(\mathtt{x} : \tau) \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma, (\mathtt{x} : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun\ x\ ->}\ e : \tau_1 \to \tau_2}\ (\textsc{Lam})$$

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

Inference rules for generation

Our starting point is the following well-known typing rules to guide our generator:

$$\frac{(\mathtt{x} : \tau) \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma, (\mathtt{x} : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun\ x\ ->}\ e : \tau_1 \to \tau_2}\ (\textsc{Lam})$$

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

In addition we throw in two rules for constants and let-bindings:

$$\frac{c \in \tau}{\Gamma \vdash c : \tau}\ (\textsc{Const}) \qquad \frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma, (\mathtt{x} : \tau_0) \vdash e_1 : \tau_1}{\Gamma \vdash \mathtt{let\ x} = e_0\ \mathtt{in}\ e_1 : \tau_1}\ (\textsc{Let})$$

Inference rules for generation

Our starting point is the following well-known typing rules to guide our generator:

$$\frac{(\mathtt{x} : \tau) \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma, (\mathtt{x} : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun\ x\ ->}\ e : \tau_1 \to \tau_2}\ (\textsc{Lam})$$

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

In addition we throw in two rules for constants and let-bindings:

$$\frac{c \in \tau}{\Gamma \vdash c : \tau}\ (\textsc{Const}) \qquad \frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma, (\mathtt{x} : \tau_0) \vdash e_1 : \tau_1}{\Gamma \vdash \mathtt{let\ x} = e_0\ \mathtt{in}\ e_1 : \tau_1}\ (\textsc{Let})$$

Actually we can view let-binding as "syntactic sugar":

$$\mathtt{let\ x} = e_0\ \mathtt{in}\ e_1 \quad \equiv \quad (\mathtt{fun\ x\ ->}\ e_1)\ e_0$$

Typed program generation w/inference rules

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

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

Typed program generation w/inference rules

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

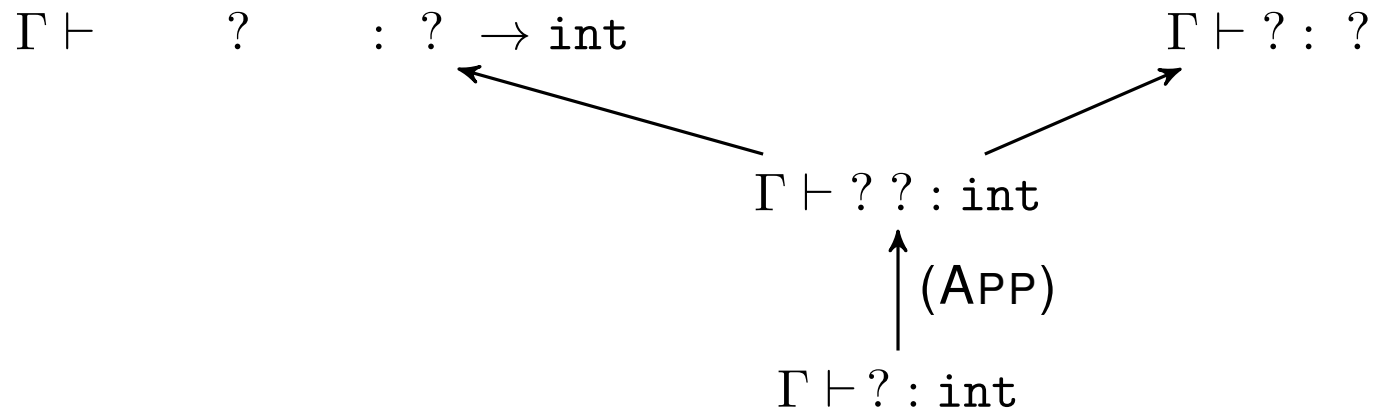
$$\Gamma \vdash ?\ ? : \texttt{int}$$

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

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

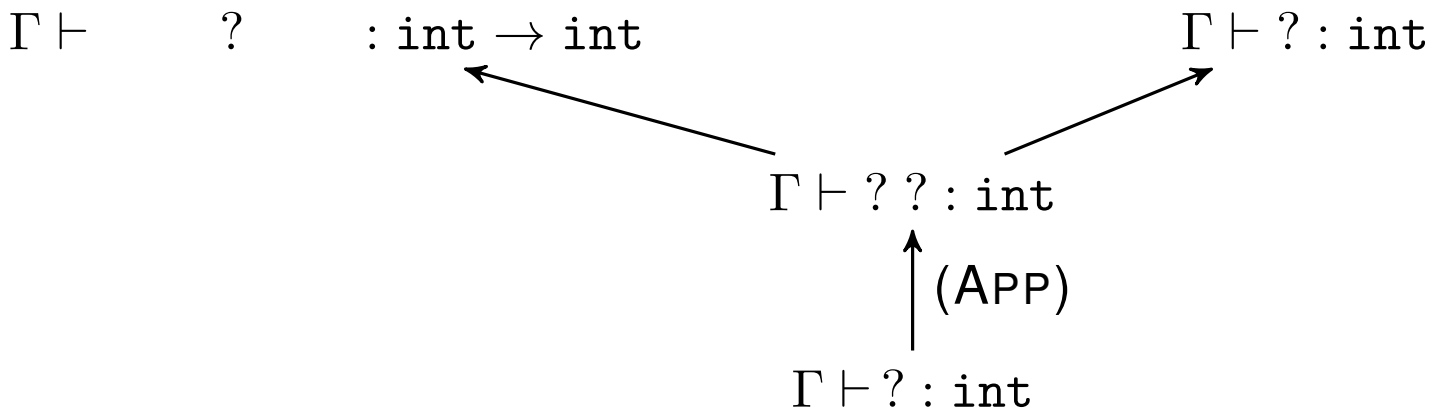
Typed program generation w/inference rules

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



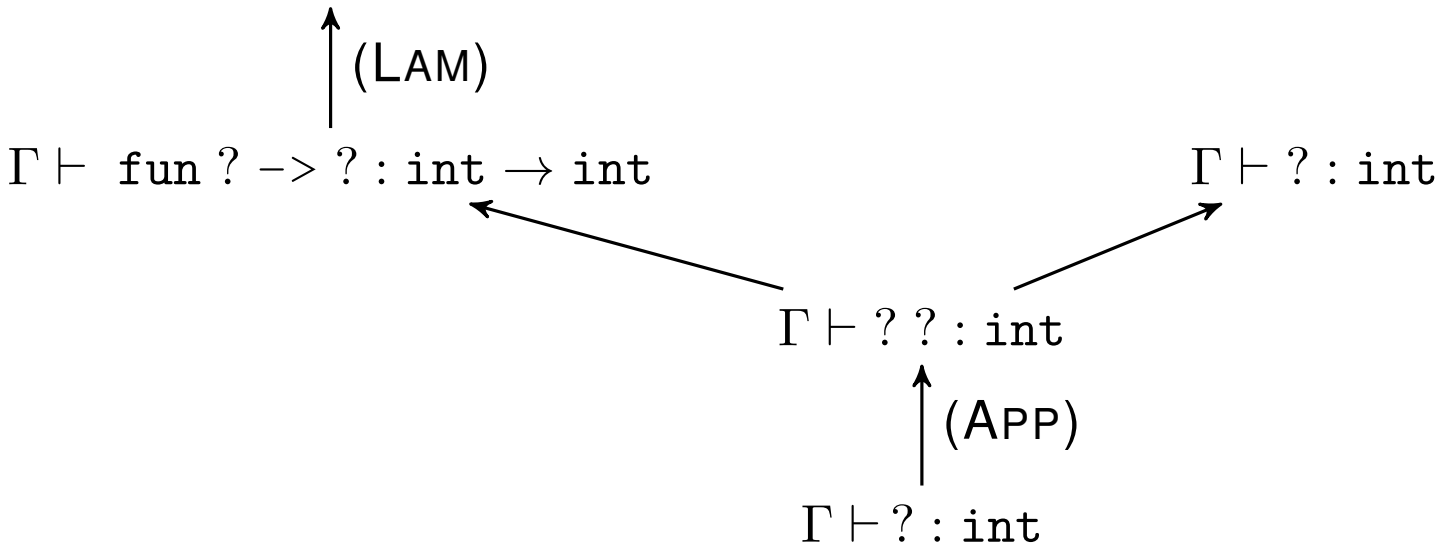
Typed program generation w/inference rules

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



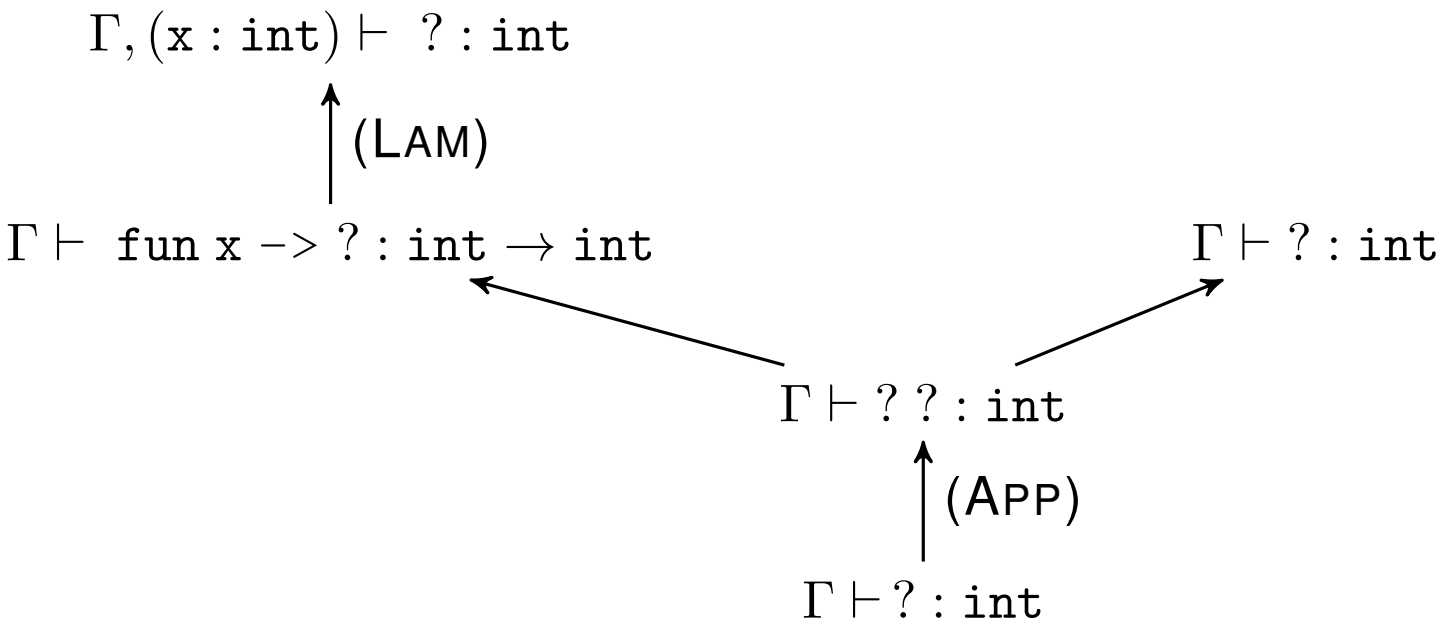
Typed program generation w/inference rules

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



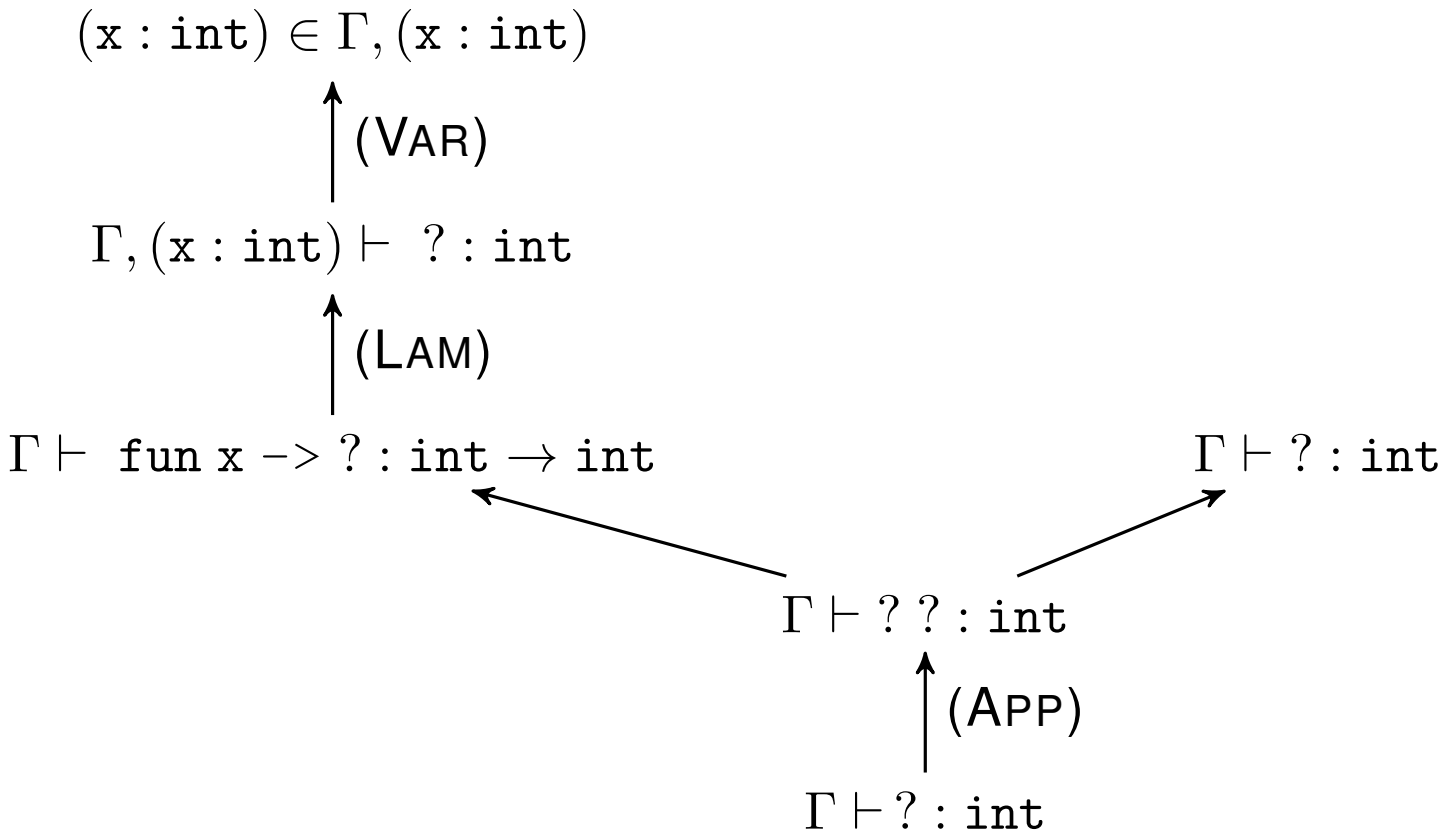
Typed program generation w/inference rules

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



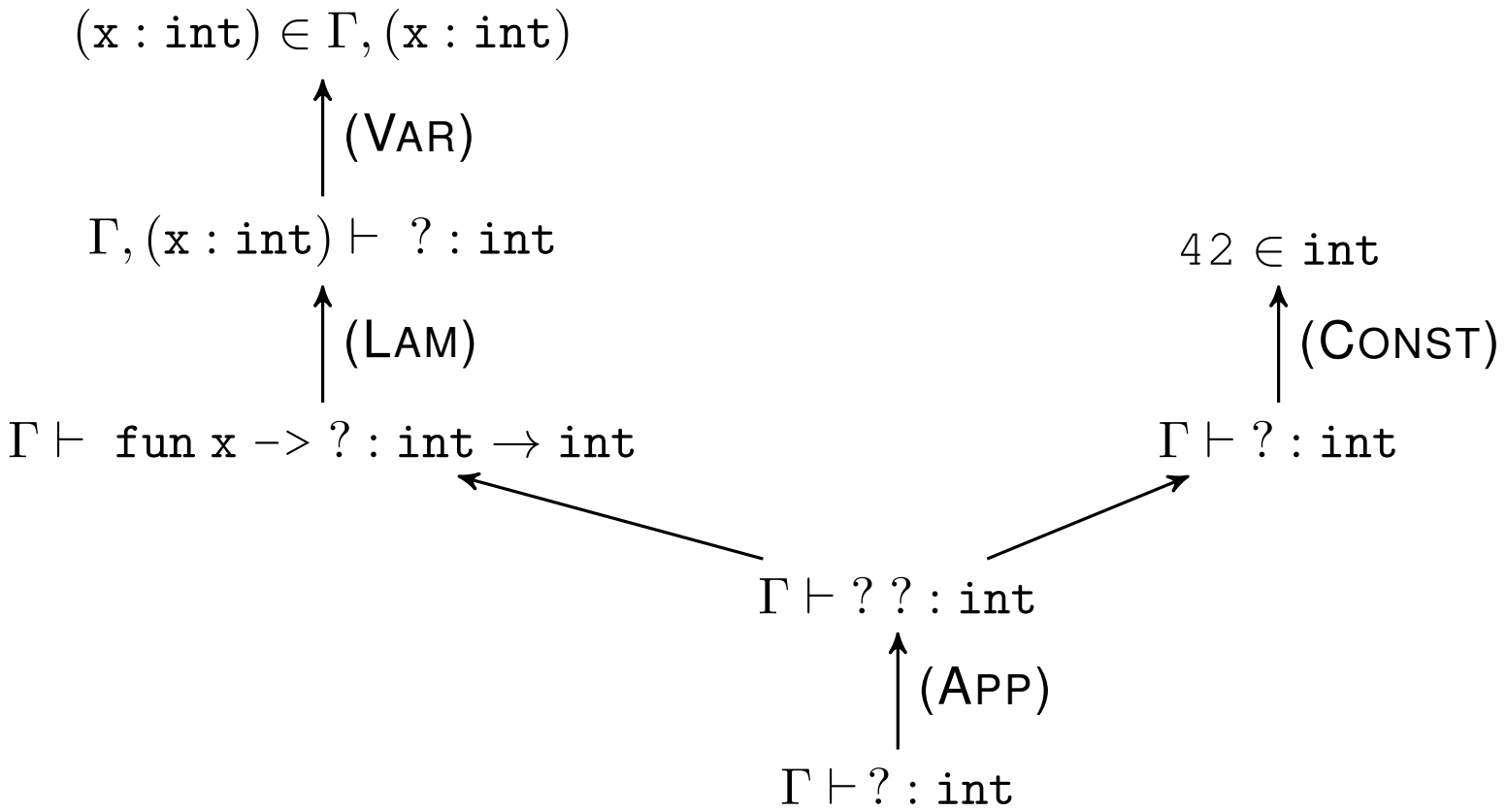
Typed program generation w/inference rules

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



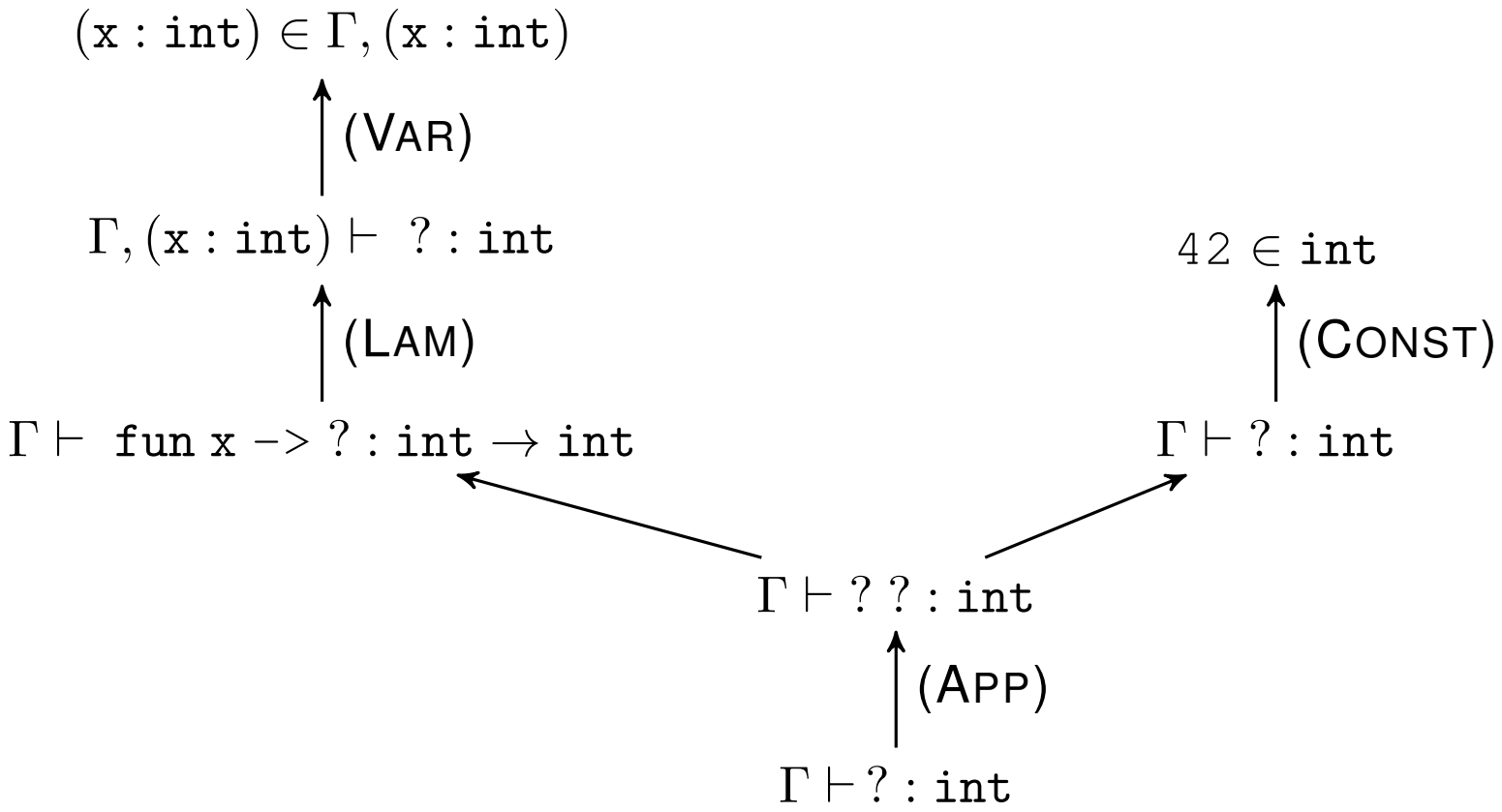
Typed program generation w/inference rules

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



Typed program generation w/inference rules

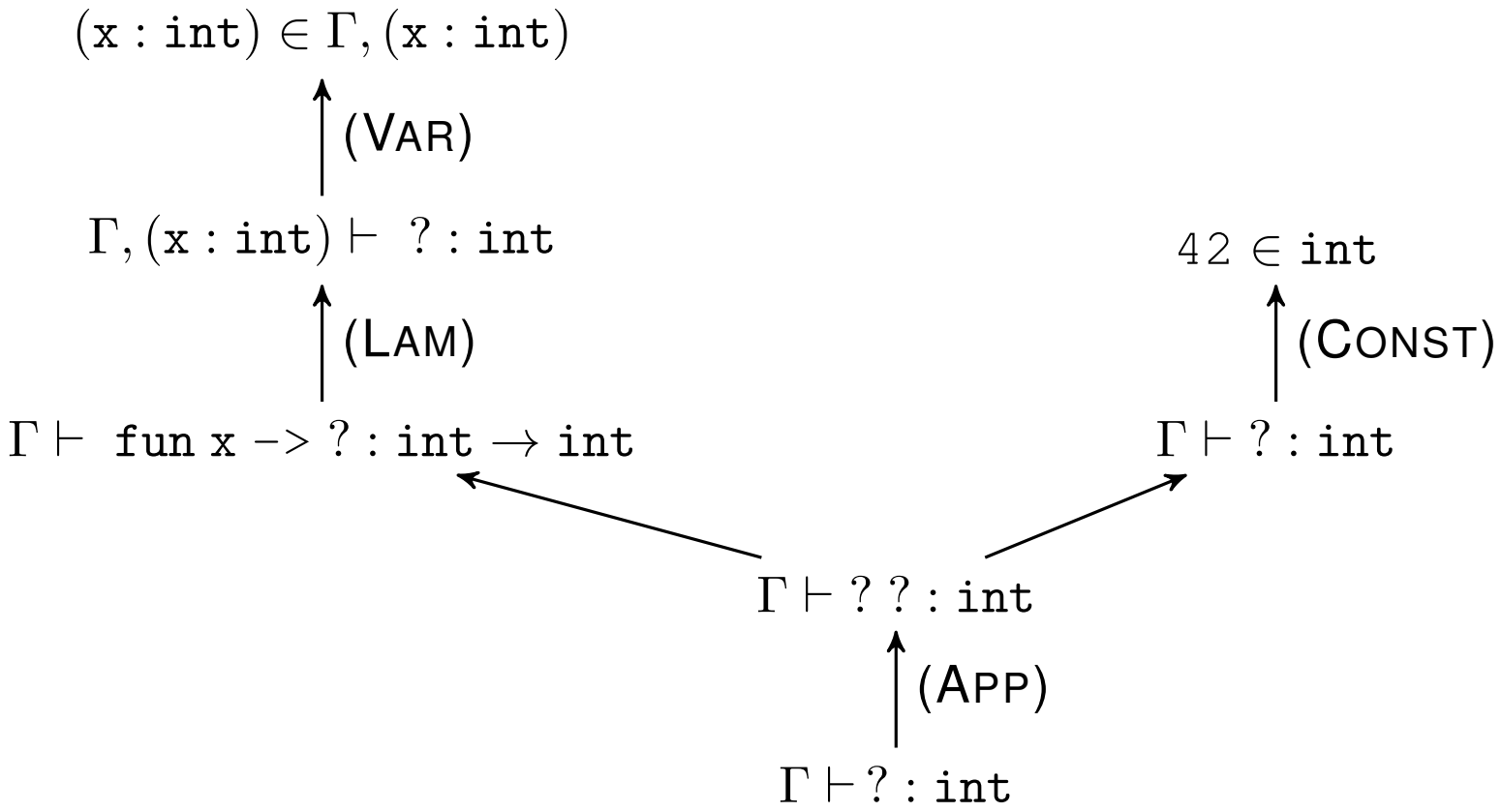
Bottom-up reading of the typing relation (Pałka-al:AST11):



Output guaranteed to make it through the type checker!

Typed program generation w/inference rules

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

A type for types

We first declare a type representing types:

```
type typ =
  | Unit
  | Int
  | String
  | Fun of typ * typ

let rec typ_to_string t = match t with
  | Unit   -> "unit"
  | Int    -> "int"
  | String -> "string"
  | Fun (t,t') -> "(" ^ typ_to_string t ^ "␣->␣" ^ typ_to_string t' ^ ")"

let leaf_gen = Gen.oneofl [Unit; Int; String]
let typ_gen = Gen.(sized (fix (fun rgen n -> match n with
    | 0 -> leaf_gen
    | _ ->
      oneof
        [leaf_gen;
         map2 (fun t t' -> Fun(t,t')) (rgen (n/2)) (rgen (n/2))]
    )))
```

A type for types

We first declare a type representing types:

```
type typ =
  | Unit
  | Int
  | String
  | Fun of typ * typ

let rec typ_to_string t = match t with
  | Unit   -> "unit"
  | Int    -> "int"
  | String -> "string"
  | Fun (t,t') -> "(" ^ typ_to_string t ^ "␣->␣" ^ typ_to_string t' ^ ")"

let leaf_gen = Gen.oneofl [Unit; Int; String]
let typ_gen = Gen.(sized (fix (fun rgen n -> match n with
    | 0 -> leaf_gen
    | _ ->
      oneof
        [leaf_gen;
         map2 (fun t t' -> Fun(t,t')) (rgen (n/2)) (rgen (n/2))]
    )))
```

This straightforward generator seems to work well:

```
# List.map typ_to_string (Gen.generate ~n:5 typ_gen);;
["string"; "(int␣->␣(unit␣->␣int))"; "(string␣->␣unit)"; "string";
 "(string␣->␣int)"]
```

Generating constants

We write a type and a generator for constants (literals):

```
type lit =
  | Unitlit
  | Intlit of int
  | Strlit of string

let lit_to_string l = match l with
  | Unitlit  -> "()"
  | Intlit i ->
    let s = string_of_int i in    (* put parens around negative ints *)
    if i < 0 then "(" ^ s ^ ")" else s
  | Strlit s -> "\"" ^ String.escaped s ^ "\""   (* escape strings *)

open Gen

(*  lit_gen : typ -> (lit option) Gen.t  *)
let lit_gen t = match t with
  | Unit   -> return (Some Unitlit)
  | Int    -> map (fun i -> Some (Intlit i)) small_signed_int
  | String -> let str_gen = string_size ~gen:printable small_nat in
              map (fun s -> Some (Strlit s)) str_gen
  | Fun (_,_) -> return None
```

This generator takes a type as argument and returns an `option`: `None` signals that generation failed.

Expression types

To setup for generation of type-correct expressions, we declare an expression type and write a printer:

```
type exp =
  | Lit of lit
  | Var of string
  | Lam of string * exp
  | App of exp * exp
  | Let of string * exp * exp

let rec exp_to_string e = match e with
  | Lit l -> lit_to_string l
  | Var x -> x
  | Lam (x,e) -> "(fun␣" ^ x ^ "␣->␣" ^ exp_to_string e ^ ")"
  | App (f,arg) -> "(" ^ exp_to_string f ^ "␣" ^ exp_to_string arg ^ ")"
  | Let (x,e,e') ->
    "(let␣" ^ x ^ "␣=␣" ^ exp_to_string e ^ "␣in␣" ^ exp_to_string e' ^ ")"

let var_gen = map (fun c -> String.make 1 c) (char_range 'a' 'z')
```

This also builds a generator of 1-character variable names.

Generator structure, take 1

The generator takes an environment, a goal type, and a fuel parameter:

```
(*  exp_gen : env -> typ -> int -> (exp option) Gen.t  *)
let rec exp_gen env t n =
  let const_rule env t = (* ... *) in
  let var_rule env t   = (* ... *) in
  let lam_rule env t   = (* ... *) in
  let app_rule env t   = (* ... *) in
  let let_rule env t   = (* ... *) in

  let rules = match n with
    | 0 -> [const_rule; var_rule]
    | _ -> [const_rule; var_rule; lam_rule; app_rule; let_rule] in

  oneofl rules >>= fun rule -> rule env t
```

When we are out of fuel we choose among leaf rules. Otherwise we choose among all of them.

Downside: if the chosen rule fails (returning `None`) the generator fails. . .

A generator with backtracking

We can easily turn it into a backtracking generator:

```
(*  exp_gen : env -> typ -> int -> (exp option) Gen.t  *)
let rec exp_gen env t n =
  let const_rule env t = (* ... *) in
  let var_rule env t   = (* ... *) in
  let lam_rule env t   = (* ... *) in
  let app_rule env t   = (* ... *) in
  let let_rule env t   = (* ... *) in

  let rules = match n with
    | 0 -> [const_rule; var_rule]
    | _ -> [const_rule; var_rule; lam_rule; app_rule; let_rule] in

  let rec try_each_loop rules = match rules with
    | [] -> return None
    | rule::rest ->
      rule env t >>= fun res -> match res with
      | None -> try_each_loop rest
      | _    -> return res in

  shuffle_l rules >>= try_each_loop
```

This first shuffles the rules, then tries them one by one.

Does it matter?

Let's try to measure the generator over 100.000 calls:

```
Test.make ~name:"failure␣stats" ~count:100000
  (set_collect
    (fun opt -> if opt = None then "fail" else "succ") prog_arb)
  (fun _ -> true)
```

We then classify the output as `"fail"` or `"succ"`.

Does it matter?

Let's try to measure the generator over 100.000 calls:

```
Test.make ~name:"failure␣stats" ~count:100000
  (set_collect
    (fun opt -> if opt = None then "fail" else "succ") prog_arb)
  (fun _ -> true)
```

We then classify the output as `"fail"` or `"succ"`.

Without backtracking:

```
 generated  error    fail    pass /  total     time test name
[✓] 100000      0       0 100000 / 100000     0.3s failure stats

fail: 69253 cases
succ: 30747 cases
```

With backtracking:

```
 generated  error    fail    pass /  total     time test name
[✓] 100000      0       0 100000 / 100000    47.5s failure stats

succ: 100000 cases
```

Does it matter?

Let's try to measure the generator over 100.000 calls:

```
Test.make ~name:"failure stats" ~count:100000
  (set_collect
    (fun opt -> if opt = None then "fail" else "succ") prog_arb)
  (fun _ -> true)
```

We then classify the output as `"fail"` or `"succ"`.

Without backtracking:

```
 generated  error    fail    pass /  total     time test name
[✓] 100000      0       0 100000 / 100000     0.3s failure stats

fail: 69253 cases
succ: 30747 cases
```

With backtracking:

```
 generated  error    fail    pass /  total     time test name
[✓] 100000      0       0 100000 / 100000    47.5s failure stats

succ: 100000 cases
```

With backtracking it never fails – without it fails 69% of the time!

Does it matter?

Let's try to measure the generator over 100.000 calls:

```
Test.make ~name:"failure stats" ~count:100000
  (set_collect
    (fun opt -> if opt = None then "fail" else "succ") prog_arb)
  (fun _ -> true)
```

We then classify the output as `"fail"` or `"succ"`.

Without backtracking:

```
 generated  error    fail    pass /  total     time test name
[✓] 100000      0       0 100000 / 100000     0.3s failure stats

fail: 69253 cases
succ: 30747 cases
```

With backtracking:

```
 generated  error    fail    pass /  total     time test name
[✓] 100000      0       0 100000 / 100000    47.5s failure stats

succ: 100000 cases
```

With backtracking it never fails – without it fails 69% of the time!

Now, compare the times: backtracking is **not** free!

The constant rule

With `lit_gen` it is easy to write `const_rule`:

```
(*  const_rule : env -> typ -> (exp option) Gen.t  *)
let const_rule env t =
  lit_gen t >>= fun res -> match res with
    | None   -> return None
    | Some c -> return (Some (Lit c)) in
```

Compare with the inference rule:

$$\frac{c \in \tau}{\Gamma \vdash c : \tau}\ (\textsc{Const})$$

It is `lit_gen`'s job to satisfy the premise.

When it succeeds, we wrap its result up in `Lit`.

The lambda rule

The lambda rule reads as follows:

```
(*  lam_rule : env -> typ -> (exp option) Gen.t  *)
let lam_rule env t = match t with
  | Unit
  | Int
  | String -> return None
  | Fun (t1,t2) ->
    var_gen >>= fun x ->
    exp_gen ((x,t1)::env) t2 (n-1) >>= fun res -> match res with
    | None   -> return None
    | Some e -> return (Some (Lam (x,e))) in
```

Compare with the inference rule:

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

The first three cases say that the goal type has to be a function type.

The lambda rule

The lambda rule reads as follows:

```
(*  lam_rule : env -> typ -> (exp option) Gen.t  *)
let lam_rule env t = match t with
  | Unit
  | Int
  | String -> return None
  | Fun (t1,t2) ->
    var_gen >>= fun x ->
    exp_gen ((x,t1)::env) t2 (n-1) >>= fun res -> match res with
    | None   -> return None
    | Some e -> return (Some (Lam (x,e))) in
```

Compare with the inference rule:

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

The first three cases say that the goal type has to be a function type. Otherwise we generate a variable, extend the `env` and try to fulfill the premise recursively.

The application rule

The application rule reads as follows:

```
(*  app_rule : env -> typ -> (exp option) Gen.t  *)
let app_rule env t =
  typ_gen >>= fun t1 ->
  exp_gen env (Fun (t1,t)) (n/2) >>= fun res -> match res with
  | None    -> return None
  | Some e0 ->
    exp_gen env t1 (n/2) >>= fun res -> match res with
    | None    -> return None
    | Some e1 -> return (Some (App (e0,e1))) in
```

Compare again with the inference rule:

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

We start by generating an arbitrary argument type τ_1.
If we ignore the `None` cases representing failure,
the two recursive calls match the premises exactly.

The `let` rule

Finally consider the **`let`** rule:

```
(*  let_rule : env -> typ -> (exp option) Gen.t  *)
let let_rule env t =
  pair var_gen typ_gen >>= fun (x,t0) ->
  exp_gen env t0 (n/2) >>= fun res -> match res with
  | None    -> return None
  | Some e0 ->
      exp_gen ((x,t0)::env) t (n/2) >>= fun res -> match res with
      | None    -> return None
      | Some e1 -> return (Some (Let (x,e0,e1))) in
```

and compare with the corresponding inference rule:

$$\frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma, (\mathtt{x} : \tau_0) \vdash e_1 : \tau_1}{\Gamma \vdash \mathtt{let\ x} = e_0 \mathtt{\ in\ } e_1 : \tau_1}\ (\textsc{Let})$$

We first generate an arbitrary variable `x` and type τ_0. In the `Some`-cases we call the generator recursively twice.

Again this matches the premises precisely.

The variable rule

The `var_rule` reads as follows:

```
(*  var_rule : env -> typ -> (exp option) Gen.t  *)
let var_rule env t =
  match List.filter (fun (_,t') -> t=t') (uniq_env env) with
  | []  -> return None
  | env ->
    let vars = List.map fst env in
    map (fun x -> Some (Var x)) (oneofl vars) in
```

Compared to the rule, `List.filter` and `oneofl` fulfills the premise:

$$\frac{(\mathtt{x} : \tau) \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau}\ (\textsc{Var})$$

The variable rule

The `var_rule` reads as follows:

```
(*  var_rule : env -> typ -> (exp option) Gen.t  *)
let var_rule env t =
  match List.filter (fun (_,t') -> t=t') (uniq_env env) with
  | []  -> return None
  | env ->
    let vars = List.map fst env in
    map (fun x -> Some (Var x)) (oneofl vars) in
```

Compared to the rule, `List.filter` and `oneofl` fulfills the premise:

$$\frac{(\mathtt{x} : \tau) \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau}\ (\textsc{Var})$$

`uniq_env` handles shadowing of duplicate variable names.
E.g., in `env = [("x",Int); ("x",String); ("x",Unit)]` we should choose among the first occurrences (in scope). So, we extract the unique variables and build an environment of those:

```
let uniq_env env =
  let uniq_vars = List.sort_uniq String.compare (List.map fst env) in
  List.map (fun x -> (x,List.assoc x env)) uniq_vars
```

Initial type environment

To start off the generator we define an initial environment:

```
let init_env =
  [ ("min_int",Int);
    ("max_int",Int);
    ("succ", Fun(Int,Int));
    ("pred", Fun(Int,Int));
    ("string_of_int", Fun(Int,String));
    ("int_of_string", Fun(String,Int));
    ("print_endline", Fun(String,Unit));
    ("print_newline", Fun(Unit,Unit));
    ("(+)",   Fun(Int,Fun(Int,Int)));
    ("(-)",   Fun(Int,Fun(Int,Int)));
    ("(␣*␣)", Fun(Int,Fun(Int,Int)));
    ("(/)",   Fun(Int,Fun(Int,Int)));
    ("(mod)", Fun(Int,Fun(Int,Int)));
    ("(^)",   Fun(String,Fun(String,String))) ]
```

We then use it along with a random type and a random amount of fuel as parameters to `exp_gen`:

```
let prog_gen =
  pair nat (oneofl [Unit;Int;String]) >>= fun (size,typ) ->
  exp_gen init_env typ size
```

Testing the generator (1/2)

It seems to work nicely:

```
utop # #require "qcheck";;

utop # #use "typegen.ml";;

utop # Gen.generate1 prog_gen;;
- : exp option =
Some
 (Let ("w", Lam ("f", Lam ("k", Lit Unitlit)),
   Let ("w",
    App (Var "print_endline",
     App (Var "string_of_int", Let ("d", Lit Unitlit, Lit (Intlit (-5))))),
    Let ("q", Var "print_newline", Lit Unitlit))))

utop # Print.option exp_to_string (Gen.generate1 prog_gen);;
- : string = "Some␣(())"

utop # Print.option exp_to_string (Gen.generate1 prog_gen);;
- : string =
"Some␣((let␣r␣=␣(let␣q␣=␣\"\"␣in␣(((mod)␣max_int)␣(-1)))␣in␣(((let␣b␣=
(fun␣x␣->␣(print_newline␣(let␣d␣=␣(((^)␣(let␣j␣=␣min_int␣in␣\"\"))
\"p]2C|!]1r\")␣in␣()))))␣in␣(let␣f␣=␣((let␣n␣=␣(int_of_string␣(let␣p␣=
\"AwLOVRPj(OFuMgsop9C7]#7#[d\"␣in␣p))␣in␣(fun␣l␣->␣()))␣(let␣j␣=␣(fun␣r
->␣r)␣in␣\"f+3IuL\"))␣in␣((fun"... (* string length 1384; truncated *)
```

Testing the generator (2/2)

The generator code so far spans ∼160 LOC.
It is supposed to output type-correct programs, so we should test that the output is accepted by OCaml:

```
(* the full generator of typed programs *)
let prog_arb = make ~print:(Print.option exp_to_string) prog_gen

let write_prog src filename =
  let ostr = open_out filename in
  let () = output_string ostr src in
  close_out ostr

let typecheck_test =
  Test.make ~name:"output␣typechecks" ~count:1000
    prog_arb
    (fun prog_opt -> match prog_opt with
       | None -> true
       | Some prog ->
         let file = "testdir/test.ml" in
         write_prog (exp_to_string prog) file;
         0 = Sys.command ("ocamlc␣-w␣-5@20-26␣" ^ file))
```

This way, I found and revised a buggy variable rule. . .

Shrinking programs

A type-preserving shrinker (1/2)

New errors should not be introduced while reducing counterexamples. Hence the shrinker should preserve types and type-correctness of the generated program.

The shrinker is composed of small rewrite steps:

`(fun x -> e) e'` $\Rightarrow$ `let x = e' in e`

`let x = e' in e` $\Rightarrow$ `e` if `x` doesn't occur in `e`

A type-preserving shrinker (1/2)

New errors should not be introduced while reducing counterexamples. Hence the shrinker should preserve types and type-correctness of the generated program.

The shrinker is composed of small rewrite steps:

```
(fun x -> e) e' ⇒ let x = e' in e
```

```
let x = e' in e ⇒ e
```
if `x` doesn't occur in `e`

And 3 rules for lifting out nested **`let`**-bindings:

```
(let x = e in e') e'' ⇒ let x = e in e' e''
```
if `x` doesn't occur in `e''`

```
e (let x = e' in e'') ⇒ let x = e' in e e''
```
if `x` doesn't occur in `e`

```
let x = (let y = e1 in e2) in e' ⇒
let y = e1 in let x = e2 in e'
```
if `y` doesn't occur in `e'`

A type-preserving shrinker (2/3)

We thus need a helper function for finding occurrences of a variable:

```
let rec occurs x e = match e with
  | Lit _ -> false
  | Var y -> x = y
  | Lam (y,e) -> x <> y && occurs x e
  | App (f,arg) -> occurs x f || occurs x arg
  | Let (y,e,e') -> occurs x e || (x <> y && occurs x e')
```

In the `Lam` and `Let` cases we check for duplicates, i.e., a new binding of the same variable.

A type-preserving shrinker (2/3)

We thus need a helper function for finding occurrences of a variable:

```
let rec occurs x e = match e with
  | Lit _ -> false
  | Var y -> x = y
  | Lam (y,e) -> x <> y && occurs x e
  | App (f,arg) -> occurs x f || occurs x arg
  | Let (y,e,e') -> occurs x e || (x <> y && occurs x e')
```

In the `Lam` and `Let` cases we check for duplicates, i.e., a new binding of the same variable.

We can phrase a simple shrinker of literals:

```
let lit_shrink l = match l with
  | Unitlit  -> Iter.empty
  | Intlit i -> Iter.map (fun i' -> Intlit i') (Shrink.int i)
  | Strlit s ->
    Iter.map (fun s' -> Strlit s') (Shrink.string s)
```

A type-preserving shrinker (3/3)

The expression shrinker is now straightforward:

```
let (<+>) = Iter.(<+>)

let rec exp_shrink e = match e with
  | Lit l      -> Iter.map (fun l' -> Lit l') (lit_shrink l)
  | Var x      -> Iter.empty
  | Lam (x,e) -> Iter.map (fun e' -> Lam (x,e')) (exp_shrink e)
  | App (f,arg) ->
    (match f with
     | Lam (x,e) ->
       Iter.return (Let (x,arg,e))
     | Let (x,e,e') when not (occurs x arg) ->
       Iter.return (Let (x,e,App(e',arg)))
     | _ -> Iter.empty)
    <+>
    (match arg with
     | Let (x,e,e') when not (occurs x f) ->
       Iter.return (Let (x,e,App(f,e')))
     | _ -> Iter.empty)
    <+>
    Iter.map (fun f' -> App (f',arg)) (exp_shrink f)
    <+>
    Iter.map (fun arg' -> App (f,arg')) (exp_shrink arg)
  | Let (x,e,e') ->
    (* ... *)
```

Testing compiler backends (1/3)

Recall that OCaml has two compiler backends:

- `ocamlc` – a fast bytecode compiler
- `ocamlopt` – an optimizing native code compiler

If we generate a program, compile it with both backends, and run both output, we expect the same behavior:

Testing compiler backends (1/3)

Recall that OCaml has two compiler backends:

- `ocamlc` – a fast bytecode compiler
- `ocamlopt` – an optimizing native code compiler

If we generate a program, compile it with both backends, and run both output, we expect the same behavior:

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

Testing compiler backends (1/3)

Recall that OCaml has two compiler backends:

- `ocamlc` – a fast bytecode compiler
- `ocamlopt` – an optimizing native code compiler

If we generate a program, compile it with both backends, and run both output, we expect the same behavior:

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

Testing compiler backends (1/3)

Recall that OCaml has two compiler backends:

- `ocamlc` – a fast bytecode compiler
- `ocamlopt` – an optimizing native code compiler

If we generate a program, compile it with both backends, and run both output, we expect the same behavior:

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

Testing compiler backends (1/3)

Recall that OCaml has two compiler backends:

- `ocamlc` – a fast bytecode compiler
- `ocamlopt` – an optimizing native code compiler

If we generate a program, compile it with both backends, and run both output, we expect the same behavior:

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

Testing compiler backends (1/3)

Recall that OCaml has two compiler backends:

- `ocamlc` – a fast bytecode compiler
- `ocamlopt` – an optimizing native code compiler

If we generate a program, compile it with both backends, and run both output, we expect the same behavior:

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

Testing compiler backends (1/3)

Recall that OCaml has two compiler backends:

- `ocamlc` – a fast bytecode compiler
- `ocamlopt` – an optimizing native code compiler

If we generate a program, compile it with both backends, and run both output, we expect the same behavior:

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

Any observed `difference` is suspicious

Testing compiler backends (2/3)

The `run` function compiles and runs a `srcfile` program:

```
let run srcfile compname compcomm =
  let exefile = "testdir/" ^ compname in
  let outfile = exefile ^ ".out" in
  let exitcode = Sys.command (compcomm ^ " -o " ^ exefile ^ " " ^ srcfile) in
  if exitcode <> 0
  then failwith
         (compname ^ " compilation failed with error " ^ string_of_int exitcode)
  else
    let runcode = Sys.command ("./" ^ exefile ^ " >" ^ outfile ^ " 2>&1") in
    (runcode, outfile)

let backend_eq_test =
  Test.make ~name:"backend equiv test" ~count:100
    prog_arb (fun prog_opt -> match prog_opt with
        | None -> true
        | Some prog ->
          let file = "testdir/test.ml" in
          let () = write_prog (exp_to_string prog) file in
          let ncode,nout = run file "native" "ocamlopt -O3 -w -5-26" in
          let bcode,bout = run file "byte" "ocamlc -w -5-26" in
          let comp =
            Sys.command ("diff -q " ^ nout ^ " " ^ bout ^ " > /dev/null") in
          ncode = bcode && comp = 0)
```

SDU We then call `run` twice and compare the results

Testing compiler backends (3/3)

This works nicely to actually find differences:

```
generated error fail pass / total     time test name
[✗]    56    0    1   55 /  100   106.3s backend equiv test

--- Failure ---------------------------------------------------

Test backend equiv test failed (132 shrink steps):

Some ((let f = ((let t = (print_endline "Y") in (fun w -> print_newline))
                                                (print_newline ()))) in ()))
```

A cleaned up version reads:

```
let f =
  (let t = print_endline "Y" in fun w -> print_newline) (print_newline ())
in ()
```

Testing compiler backends (3/3)

This works nicely to actually find differences:

```
generated error fail pass / total     time test name
[✗]    56    0    1   55 /  100   106.3s backend equiv test

--- Failure ------------------------------------------------

Test backend equiv test failed (132 shrink steps):

Some ((let f = ((let t = (print_endline "Y") in (fun w -> print_newline))
                                                (print_newline ()))) in ()))
```

A cleaned up version reads:

```
let f =
  (let t = print_endline "Y" in fun w -> print_newline) (print_newline ())
in ()
```

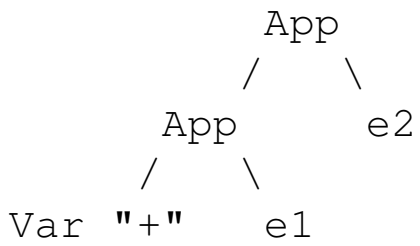
`ocamlopt` evaluates left-to-right: prints `"Y"` then newline

`ocamlc` evaluates right-to-left: prints newline then `"Y"`

A difference? yes A bug? no (according to spec. . .)

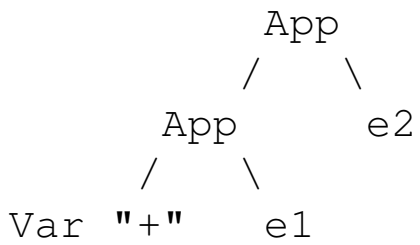
Direct calls (1/3)

The shape of a call to `(+)` is:



Direct calls (1/3)

The shape of a call to `(+)` is:

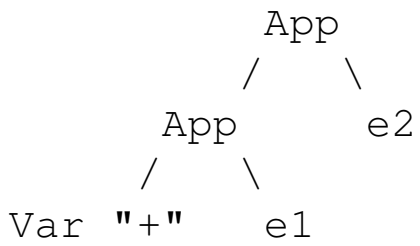


Generating such a call requires

- the goal type to be `Int`
- choosing `app_rule` with an argument type `Int`
- choosing `app_rule` again with an argument type `Int`

Direct calls (1/3)

The shape of a call to `(+)` is:



Generating such a call requires

- the goal type to be `Int`
- choosing `app_rule` with an argument type `Int`
- choosing `app_rule` again with an argument type `Int`

We can measure the chance of doing so:

```
Test.make ~name:"binop stats" ~count:10000
  (set_collect
     (fun opt -> match opt with
       | None   -> "no binop"
       | Some e ->
         if contains_binop_call e then "some binop" else "no binop")
      prog_arb)
  (fun _ -> true)
```

```
no binop: 9885 cases
some binop: 115 cases
```

Only 1.1% contain a call to a binary operation. . .

Direct calls (2/3)

To increase the chance, Pałka-al:AST11 suggest to add an additional rule:

$$\frac{(\texttt{f} : \tau_1 \to \ldots \to \tau_n \to \tau) \in \Gamma \qquad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \texttt{f}\ e_1 \ldots e_n : \tau}\ (\textsc{Indir})$$

Reading it bottom up as a generator:

- Choose a function `f` from the environment with the right result type τ
- Generate argument expressions of the right types
- Glue the result together as a call

  (Formally, this rule is redundant)

Direct calls (3/3)

The corresponding code is a bit more complex:

```
let rec collect_args t arg_acc = match t with
(* helper for collecting args *)
  | Unit
  | Int
  | String -> (List.rev arg_acc, t)
  | Fun (t,t') -> collect_args t' (t::arg_acc)
```

Direct calls (3/3)

The corresponding code is a bit more complex:

```
let rec collect_args t arg_acc = match t with
(* helper for collecting args *)
  | Unit
  | Int
  | String -> (List.rev arg_acc, t)
  | Fun (t,t') -> collect_args t' (t::arg_acc)

let indir_rule env t =
  let fenv = List.map (fun (x,t) -> (x, collect_args t [])) (uniq_env env) in
  match List.filter (fun (x,(args,ret)) -> args <> [] && ret = t) fenv with
   | []   -> return None
   | fenv ->
     oneofl fenv >>= fun (f,(ts,t)) ->
     let arglen = List.length ts in
     let gen_list = List.map (fun ti -> exp_gen env ti (n/arglen)) ts in
     flatten_l gen_list >>= fun res ->
     if List.mem None res
     then return None
     else
       let exps = List.filter_map (fun e -> e) res in (*keep only Some's*)
       return (Some (Indir (f,exps)))
```

Direct calls (3/3)

The corresponding code is a bit more complex:

```
let rec collect_args t arg_acc = match t with
(* helper for collecting args *)
  | Unit
  | Int
  | String -> (List.rev arg_acc, t)
  | Fun (t,t') -> collect_args t' (t::arg_acc)

let indir_rule env t =
  let fenv = List.map (fun (x,t) -> (x, collect_args t [])) (uniq_env env) in
  match List.filter (fun (x,(args,ret)) -> args <> [] && ret = t) fenv with
   | []   -> return None
   | fenv ->
     oneofl fenv >>= fun (f,(ts,t)) ->
     let arglen = List.length ts in
     let gen_list = List.map (fun ti -> exp_gen env ti (n/arglen)) ts in
     flatten_l gen_list >>= fun res ->
     if List.mem None res
     then return None
     else
       let exps = List.filter_map (fun e -> e) res in (*keep only Some's*)
       return (Some (Indir (f,exps)))
```

Adding it increases the frequency of binary operations:

```
no binop: 6455 cases
some binop: 3545 cases
```

Now 35% contain a binary operation

Extending with type variables

If we want to add `list` types, they are straightforward to model and generate.

However, which type should `List.length` have?

Extending with type variables

If we want to add `list` types, they are straightforward to model and generate.

However, which type should `List.length` have?

`int list -> int`, `string list -> int`,
`(int list) list -> int`, ...?

Extending with type variables

If we want to add `list` types, they are straightforward to model and generate.

However, which type should `List.length` have?

`int list -> int`, `string list -> int`, `(int list) list -> int`, ...?

Rather than try to enumerate them all, we need a type variable! `'a list -> int` for any `'a`

Extending with type variables

If we want to add `list` types, they are straightforward to model and generate.

However, which type should `List.length` have?

`int list -> int`, `string list -> int`, `(int list) list -> int`, ...?

Rather than try to enumerate them all, we need a type variable! `'a list -> int` for any `'a`

If we add pair types, which type should `fst` have?

Extending with type variables

If we want to add `list` types, they are straightforward to model and generate.

However, which type should `List.length` have?

`int list -> int`, `string list -> int`, `(int list) list -> int`, ...?

Rather than try to enumerate them all, we need a type variable! `'a list -> int` for any `'a`

If we add pair types, which type should `fst` have?

`'a * 'b -> 'a` for any `'a` and `'b`

Extending with type variables

If we want to add `list` types, they are straightforward to model and generate.

However, which type should `List.length` have?

```
int list -> int, string list -> int,
(int list) list -> int, ...?
```

Rather than try to enumerate them all, we need a type variable! `'a list -> int` for any `'a`

If we add pair types, which type should `fst` have?

```
'a * 'b -> 'a
```
for any `'a` and `'b`

With type variables, types match up to variables:
`int list * string` matches `'a * 'b` by choosing `'a = int list` and `'b = string`

(this matching algorithm is called unification)

Beyond our type-driven tester

Two students from DTU made a similar generator as their course project.

Beyond many examples of different evaluation order, they also found a bug where the compiler optimized away a division by 0:

```
0 / List.hd [0]
```

Beyond our type-driven tester

Two students from DTU made a similar generator as their course project.

Beyond many examples of different evaluation order, they also found a bug where the compiler optimized away a division by 0:

```
0 / List.hd [0]
```

Later, to avoid generating evaluation-order dependent programs, we devised a type and effects system and wrote a generator following it. Result: 5 more bugs. . .

The extended generator is described in more detail in:

Midtgaard, Justesen, Kasting, Nielson, Nielson, ICFP 2017: *Effect-driven QuickChecking of Compilers*

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

Fun with the generator since the paper

There are other OCaml compilers,
e.g., js_of_ocaml and BuckleScript

When testing them against the bytecode backend I found:

- 2 bugs in js_of_ocaml
- 8 bugs in BuckleScript

For example:

Fun with the generator since the paper

There are other OCaml compilers,
e.g., js_of_ocaml and BuckleScript

When testing them against the bytecode backend I found:

- 2 bugs in js_of_ocaml
- 8 bugs in BuckleScript

For example:

```
let m = (<>) (fun g -> "") (fun v -> "") in 0
```

Bytecode result:

Fun with the generator since the paper

There are other OCaml compilers,
e.g., js_of_ocaml and BuckleScript

When testing them against the bytecode backend I found:

- 2 bugs in js_of_ocaml
- 8 bugs in BuckleScript

For example:

```
let m = (<>) (fun g -> "") (fun v -> "") in 0
```

Bytecode result:

```
Exception: Invalid_argument "compare: functional value"
```

js_of_ocaml result:

Fun with the generator since the paper

There are other OCaml compilers,
e.g., js_of_ocaml and BuckleScript

When testing them against the bytecode backend I found:

- 2 bugs in js_of_ocaml
- 8 bugs in BuckleScript

For example:

```
let m = (<>) (fun g -> "") (fun v -> "") in 0
```

Bytecode result:

```
Exception: Invalid_argument "compare: functional value"
```

js_of_ocaml result: `0`

Fun with the generator since the paper

There are other OCaml compilers,
e.g., js_of_ocaml and BuckleScript

When testing them against the bytecode backend I found:

- 2 bugs in js_of_ocaml
- 8 bugs in BuckleScript

For example:

```
let m = (<>) (fun g -> "") (fun v -> "") in 0
```

Bytecode result:

```
Exception: Invalid_argument "compare: functional value"
```

js_of_ocaml result: `0`

There were also false alarms, e.g., due to diff. int width

Compiler testing more broadly (1/2)

There's a research subfield of program generation.

"Differential testing" was originally introduced in the context of testing C compilers (McKeeman:DTJ98).
Generator: rec. generator w/weights ("stochastic grammar")

Compiler testing more broadly (1/2)

There's a research subfield of program generation.

"Differential testing" was originally introduced in the context of testing C compilers (McKeeman:DTJ98).
Generator: rec. generator w/weights ("stochastic grammar")

CSmith `https://embed.cs.utah.edu/csmith/`
Generator: C programs free of undefined behavior (compute+print a checksum).
CSmith had a great impact (476 GCC + LLVM bugs)

Compiler testing more broadly (1/2)

There's a research subfield of program generation.

"Differential testing" was originally introduced in the context of testing C compilers (McKeeman:DTJ98).
Generator: rec. generator w/weights ("stochastic grammar")

CSmith `https://embed.cs.utah.edu/csmith/`
Generator: C programs free of undefined behavior (compute+print a checksum).
CSmith had a great impact (476 GCC + LLVM bugs)

Understood as property-based testing they both test:
Property: a compiled program behaves the same with and without optimization (or w/diff. compilers)

Compiler testing more broadly (1/2)

There's a research subfield of program generation.

"Differential testing" was originally introduced in the context of testing C compilers (McKeeman:DTJ98).
Generator: rec. generator w/weights ("stochastic grammar")

CSmith `https://embed.cs.utah.edu/csmith/`
Generator: C programs free of undefined behavior (compute+print a checksum).
CSmith had a great impact (476 GCC + LLVM bugs)

Understood as property-based testing they both test:
Property: a compiled program behaves the same with and without optimization (or w/diff. compilers)

Both come with test case reducers (aka. shrinkers)...

Compiler testing more broadly (2/2)

CSmith inspired other work:

Equivalence Modulo Input (EMI)

`https://people.inf.ethz.ch/suz/emi/index.html`

(1622 GCC+LLVM bugs)

Generator: a pair of C programs p, p' and an input i

Property: p and p' are equivalent when a variable $\mathtt{x}$ is i

Compiler testing more broadly (2/2)

CSmith inspired other work:

Equivalence Modulo Input (EMI)

`https://people.inf.ethz.ch/suz/emi/index.html`

(1622 GCC+LLVM bugs)

Generator: a pair of C programs p, p' and an input i

Property: p and p' are equivalent when a variable $\mathtt{x}$ is i

Graphics compiler testing ("Metamorphic testing")

`http://multicore.doc.ic.ac.uk/projects/clsmith/`

(+50 OpenCL bugs, startup + Google acquisition)

Generator: a pair of OpenCL programs p, p' where p' contains some dead code

Property: p and p' produces (sufficiently) identical images

Summary and conclusion

- We've covered inference rules and how they can be used to formalize a type system
- We've seen the correspondence between such a type system and formal logic
- We've seen how we can use such rules to guide a program generator
- Coupled with *differential testing* this yields a powerful approach for automated compiler testing