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

Jan Midtgaard

MMMI, SDU

SDU

Outline

Intermezzo: ML typing

Compiler Testing

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 tell us the parity of base case 0.

The two rules SUCCODD and SUCCEVEN tells us the parity of successor numbers, e.g., for SUCCODD:

If we've established that some number n is even, then we are allowed to 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)

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:

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

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 *"simply-typed λ-calculus"* typing rules

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 $\mathtt{x}$ 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."

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, . . .)

- 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, . . .)

- 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, . . .)

- 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”

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]

Compiler Testing

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

Fun with the compiler tester

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 compiler tester

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 compiler tester

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 compiler tester

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 compiler tester

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

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 a range of approaches to generating programs
- Coupled with *differential testing* this yields a powerful approach for automated compiler testing