Typing MiniFun

Lorenzo Ceragioli

October 30, 2024

IMT Lucca

Type Analysis

Static analysis

- Proves properties about the behaviour of the program by looking at the code (i.e. without executing it)
- Decidable, therefore approximated (Rice's theorem)

Type Analysis

- The most common static analysis
- Property of a term is its type
- Enforces *no type error at run-time*
- For MiniFun it also implies *no deadlock*

More Concretely

1. We will define a deduction system for deciding the type of constructs and check their *consistency*
   - property $t \triangleright \tau$, i.e. t is of type τ
   - `int` and `bool` atomic types
   - functional types
   - inconsistency implies contradiction in the properties

$$t \triangleright \tau \text{ and } t \triangleright \tau' \text{ with } \tau \text{ and } \tau' \text{ incompatible}$$

2. You will implement a procedure for performing the type analysis

An Example of Inconsistency

```
letfun f x = x and true
  in f 5
```

Note that

- `f 5` is a legal term, per se
- `x and true` is a legal term, per se
- but they are not consistent (x must be both bool and int)
- **idea:** the deduction system computes and propagates the constraints

Approximation

Before seeing the formal treatment, be aware that the following

- is not problematic
- will be deemed inconsistent by the type system

```
letfun f x = f (x - 1) in
  fun x => (x + (f x)) and true
```

There will always be cases like this, no matter how hard you try!

Formally...

Syntax of MiniFun's Types

A type τ is either *int*, *bool*, or a function

$$\tau ::= \ \mathit{int} \mid \mathit{bool} \mid \tau \longrightarrow \tau$$

We write $\mathbb{T}$ for the set of all types.

We assume different types to be incompatible: $t \rhd \tau$ and $t \rhd \tau'$ with $\tau \neq \tau'$ is always a contradiction.

The typing is contextual, the type of `fun x => x + y` *is* $\mathit{int} \longrightarrow \mathit{int}$ *if* `y` *is an integer, it is not defined (an error) otherwise.*

A typing context (or environment) Γ is a partial function associating variables with types $\Gamma : X \longrightarrow \mathbb{T}$.

Typing Judgments

A typing judgment is a sequent of the form $\Gamma \vdash t \triangleright \tau$

- Read as "the term t has type τ in the context Γ"
- In the deduction system, τ is inferred by the types of the subterms of t and of the non-local names
- The type of t is needed to infer the types of the terms of which t is subterm and in which its name is used
- In theory we are mainly interested in wether t has a type or not (i.e. if it is correct)

Deduction System

Literals are trivially typed.

$$\frac{}{\Gamma \vdash n \rhd \mathit{int}}\ \textsc{Num} \qquad \frac{}{\Gamma \vdash v \rhd \mathit{bool}}\ \textsc{Bool}$$

Variables are typed according to the context

$$\frac{}{\Gamma[x \mapsto \tau] \vdash x \rhd \tau}\ \textsc{Var}$$

This notation is the same as requiring in the premise that $\Gamma(x) = \tau$.

Deduction System

We assume builtin operations ($+$, -, and, ...) to be implicitly typed

$$\frac{\Gamma \vdash t_1 \rhd \mathit{int} \quad \Gamma \vdash t_2 \rhd \mathit{int}}{\Gamma \vdash t_1 \; + \; t_2 \rhd \mathit{int}} \textsc{ Plus}$$

Are we approximating something here?

$$\frac{\Gamma \vdash t_1 \rhd \mathit{bool} \quad \Gamma \vdash t_2 \rhd \tau \quad \Gamma \vdash t_3 \rhd \tau}{\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \rhd \tau} \textsc{ If}$$

Deduction System

Notice, very similar to computing the semantics.

$$\frac{\Gamma[x \mapsto \tau] \vdash t \rhd \tau'}{\Gamma \vdash \texttt{fun}\ x\ \texttt{=>}\ t \rhd \tau \longrightarrow \tau'}\ \textsc{Fun}$$

$$\frac{\Gamma \vdash t_1 \rhd \tau \longrightarrow \tau' \quad \Gamma \vdash t_2 \rhd \tau}{\Gamma \vdash t_1\ t_2 \rhd \tau'}\ \textsc{FunApp}$$

Deduction System

$$\frac{\Gamma \vdash t_1 \rhd \tau \quad \Gamma[x \mapsto \tau] \vdash t_2 \rhd \tau'}{\Gamma \vdash \texttt{let } x = t_1 \texttt{ in } t_2 \rhd \tau'} \textsc{Let}$$

$$\frac{\Gamma[f \mapsto \tau \longrightarrow \tau''; x \mapsto \tau] \vdash t_1 \rhd \tau'' \quad \Gamma[f \mapsto \tau \longrightarrow \tau''] \vdash t_2 \rhd \tau'}{\Gamma \vdash \texttt{letfun } f\ x \texttt{ = } t_1 \texttt{ in } t_2 \rhd \tau'} \textsc{LetFun}$$

Typing Terms

A term t has type τ if

$$\varnothing \vdash t \triangleright \tau$$

Guessing the type of parameters

- The idea of our typing system is to compute and propagate constraints.
- What if there are not enough constraints?

```
fun x => x
```

Which is the type of `x`?

Is it *int*?

$$\dfrac{\dfrac{}{\varnothing[x \mapsto \mathit{int}] \vdash x \rhd \mathit{int}}\ \textsc{Var}}{\varnothing \vdash \texttt{fun}\ x\ \texttt{=>}\ x \rhd \mathit{int} \longrightarrow \mathit{int}}\ \textsc{Fun}$$

Or maybe *bool* $\longrightarrow$ *bool*?

$$\dfrac{\dfrac{}{\varnothing[x \mapsto (\mathit{bool} \longrightarrow \mathit{bool})] \vdash x \rhd (\mathit{bool} \longrightarrow \mathit{bool})}\ \textsc{Var}}{\varnothing \vdash \texttt{fun}\ x\ \texttt{=>}\ x \rhd (\mathit{bool} \longrightarrow \mathit{bool}) \longrightarrow (\mathit{bool} \longrightarrow \mathit{bool})}\ \textsc{Fun}$$

Guessing the types of parameters

- This is not a problem in theory
- The term is typed if there exists a guess that works
- We could derive the most adequate type, if a guess works we stick with it, otherwise we try a new one (but notice that they are not finite)

Guessing the types of parameters

Sometimes constraints are there, so you can guess the right type

$$\dfrac{\dfrac{\dfrac{}{\varnothing[x \mapsto \mathit{int}] \vdash x \rhd \mathit{int}}\ \textsc{Var}}{\varnothing \vdash \texttt{fun}\ x\ \texttt{=>}\ x \rhd \mathit{int} \longrightarrow \mathit{int}}\ \textsc{Fun} \qquad \dfrac{}{\varnothing \vdash 3 \rhd \mathit{int}}\ \textsc{Num}}{\varnothing \vdash (\texttt{fun}\ x\ \texttt{=>}\ x)\ 3 \rhd \mathit{int}}\ \textsc{FunApp}$$

still the deduction system does not help you to guess

MiniTyFun

The simplest solution is to update the syntax of the language and to ask the programmer to specify the types of parameters

$$t ::= n \mid v \mid x \mid \texttt{let}\ x = t\ \texttt{in}\ t \mid \texttt{letfun}\ f\ x\texttt{:}\tau\ \texttt{=}\ t\ \texttt{in}\ t$$
$$\mid t\ t \mid t\ op\ t \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \mid \texttt{fun}\ x\texttt{:}\tau\ \texttt{=>}\ t$$
$$\tau ::= \texttt{int} \mid \texttt{bool} \mid \tau\ \texttt{->}\ \tau$$

Notice

- the symbol `:` is the syntactic counterpart of $\triangleright$
- the symbol `->` is the syntactic counterpart of $\longrightarrow$
- for recursive functions $\texttt{letfun}\ f\ x\texttt{:}\tau\ \texttt{=}\ t\ \texttt{in}\ t$, τ must be the (functional) type of f, i.e. some type $\tau'\texttt{->}\tau''$ where τ' is the type of the parameter x and τ'' is the type of the value returned by the function

Project Fragment

1. Complete the definition of the type system of MiniTyFun extending the syntax of MiniFun with type annotations;
2. Write it down in the report (just the missing rules)
3. Produce an OCaml module for MiniTyFun, with an OCaml type for the abstract syntax tree, and a type check function that given a MiniTyFun term returns *Some* τ if τ is its type or *None* if it cannot be typed.