Types and Static Type Checking (Introducing Micro-Haskell)

Informatics 2A: Lecture 13

Alex Simpson

School of Informatics
University of Edinburgh
als@inf.ed.ac.uk

16 October, 2012

Thus far in the course, we have examined the machinery that, in the case of a programming language, takes us from a program text to a parse tree, via the stages of lexing and parsing.

One the program has been parsed, meaning that it is syntactically correct, the parse tree can be converted into an abstract syntax tree (AST) which contains just the information needed for further processing.

In particular, the AST is fed to an evaluator or compiler to execute the program.

Sometimes, however, additional checks are placed on the program before execution, in order to ensure that certain blatant errors are avoided. This is called static analysis.

This lecture looks at one common form of static analysis: type-checking.

Types

Consider the expression

```
3 + True
```

How is a compiler or interpreter supposed to execute this?

It does not make sense to apply the numerical addition operation to the argument `True`, which is a boolean.

This is an example of a type error.

Different programming languages take different approaches to such errors.

Approaches to type errors

Laissez faire: Even if an operation does not make sense for the data its being applied to, just go ahead and apply it to the (binary) machine representation of the data. In some cases this will do something harmful. In other cases it might even be useful. (Adopted, e.g., in C.)

Dynamic checking: At the point during execution at which a type mismatch (between operation and argument) is encountered, raise an error. This gives rise to helpful runtime errors. (Adopted, e.g., in Python.)

Static checking: Check (the AST of) the program to ensure that all operations are applied in a type-meaningful way. If not, identify the error(s), and disallow the program from being run until corrected. This allows many program errors to be identified before execution. (Adopted, e.g., in Java and Haskell.)

In this lecture we look at static stype-checking using a fragment of Haskell as the illustrative programming language.

We call the fragment of Haskell Micro-Haskell (MH for short).

MH is the basis of this year's Inf2A Assignment 1, which uses it to illustrate the full formal-language-processing pipeline.

For those who have never previously met Haskell or who could benefit from a Haskell refresher, we start with a gentle introduction to MH.

Micro-Haskell: a crash course

In mathematics, we are used to defining functions via equations, e.g. $f(x) = x^2 + 3x + 7$.

The idea in functional programming is that programs should look somewhat similar to mathematical definitions:

```
f x = x*x + 3*x + 7 ;
```

This function expects an argument `x` of integer type (let's say), and returns a result of integer type. We therefore say the type of `f` is `Integer -> Integer` ("integer to integer").

By contrast, the definition

```
g x = x*x < 3*x + 7 ;
```

returns a boolean result, so the type of g is `Integer -> Bool`.

Multi-argument functions

What about a function of two arguments, say `x :: Integer` and `y :: Bool` ? E.g.

```
h x y = if y then x else 0-x ;
```

Think of `h` as a function that accepts arguments one at a time. It accepts an integer and returns another function, which itself accepts a boolean and returns an integer.

So the type of `h` is `Integer -> (Bool -> Integer)`. By convention, we treat $->$ as right-associative, so we can write this just as `Integer -> Bool -> Integer`.

Note incidentally the use of '`if`' to create expressions rather than commands. In Java, the above if-expression could be written as

```
(y ? x : -x)
```

Typechecking in Micro-Haskell

In (Micro-)Haskell, the type of `h` is explicitly given as part of the function definition:

```
h :: Integer -> Bool -> Integer ;
h x y = if y then x else 0-x ;
```

The typechecker then checks that the expression on the RHS does indeed have type `Integer`, assuming `x` and `y` have the specified argument types `Integer` and `Bool` respectively.

Function definitions can also be recursive:

```
div :: Integer -> Integer -> Integer ;
div x y = if x<y then 0 else 1 + div (x-y) y ;
```

Here the typechecker will check that the RHS has type `Integer`, assuming that `x` and `y` have type `Integer` and also that `div` itself has the stated type.

Higher-order functions

The arguments of a function in MH can themselves be functions!

```
F :: (Integer -> Integer) -> Integer ;
F g = g 0 + g 1 + g 2 + g 3;
```

The typechecker then checks that the expression on the RHS does indeed have type `Integer`, assuming `x` and `y` have the specified argument types `Integer` and `Bool` respectively.

For an example application of `F`, consider the following MH function.

```
inc :: Integer -> Integer ;
inc x == x+1 ;
```

If we then type

```
F inc
```

into an evaluator (i.e., interpreter) for MH, the evaluator will compute that the result of the expression `F inc` is 10.

In principle, the -> constructor can be iterated to produce very complex types, e.g.

```
(((Integer->Bool)->Bool)->Integer)->Integer
```

Such monsters rarely arise in ordinary programs.

Nevertheless, MH (and full Haskell) has a precise way of checking whether the function definitions in the program correctly respect the types that have been assigned to them.

Before discussing this process, we summarize the types of MH.

MH Types

The official grammar of MH types (in Assignment 1 handout) is

$$\begin{aligned} \textit{Type} &\rightarrow \textit{Type1 TypeOps} \\ \textit{Type1} &\rightarrow \texttt{Integer} \mid \texttt{Bool} \mid \texttt{(}\ \textit{Type}\ \texttt{)} \\ \textit{TypeOps} &\rightarrow \epsilon \mid \texttt{->}\ \texttt{Type} \end{aligned}$$

This is an LL(1) grammar for convenient parsing.

However, a parse tree for this grammar contains more detail than is required for understanding a type expression.

The following conceptually simpler grammar implements the abstract syntax of types

$$\textit{Type} \rightarrow \texttt{Integer} \mid \texttt{Bool} \mid \textit{Type}\ \texttt{->}\ \textit{Type}$$

Abstract Syntax Trees

The abstract syntax grammar is not appropriate for parsing:

- It is ambiguous
- It does not include all aspects of the concrete syntax. In particular, there are no brackets.

However parse trees for the abstract syntax grammar unambiguously correspond to types.

Instead of working with parse trees for the concrete LL(1) grammar, we convert such parse trees to parse trees for the abstract syntax grammar. Such parse trees are called abstract syntax trees (AST).

Concrete versus abstract syntax

The distinction between concrete and abstract syntax is not specific to types, but applies generally to formal and natural languages.

In the case of an LL(1)-predictively parsed formal languages, we have the following parsing pipeline:

Lexed language phrase
(sequence of lexemes)
$\Downarrow$ (LL(1) predictive parsing)
LL(1)-grammar parse tree
(uniquely determined)
$\Downarrow$ (Conversion of parse trees)
AST

Type checking

Main ideas.

1. Type checking is done compositionally by breaking down expressions into their subexpressions, type-checking the subexpressions, and ensuring that the top-level compound expression can then be given a type itself.

2. Throughout the process, a type environment is maintained which records the types of all variables in the expression.

Illustrative example

```
h :: Integer -> Bool -> Integer ;
h x y = if y then x else 0-x ;
```

First the type environment Γ is set according to the the type declaration.

$$\Gamma := \texttt{h :: Integer -> Bool -> Integer}$$

Next, the type environment is extended to assign types to the argument variables `x and y`.

$$\Gamma := \begin{array}{l}\texttt{h :: Integer -> Bool -> Integer,}\\ \texttt{x :: Integer,}\\ \texttt{y :: Bool}\end{array}$$

Illustrative example (continued)

This is done in order to implement the general rule

- In any expression $e_1 e_2$ (a function application) we need e_1 to have a function type t_1 -> t_2 with e_2 having the correct type t_1 for its argument. The resulting type of $e_1 e_2$ is then t_2.

Thus, in our example, we have types

```
h x :: Bool -> Integer
```

and

```
h x y :: Integer
```

Illustrative example (continued)

```
h :: Integer -> Bool -> Integer ;
h x y = if y then x else 0-x ;
```

We have

```
h x y  :: Integer
```

with the type environment

$$\Gamma := \begin{array}{l} \texttt{h :: Integer -> Bool -> Integer,} \\ \texttt{x :: Integer,} \\ \texttt{y :: Bool} \end{array}$$

Our remaining task is to type-check (relative to Γ) the expression:

```
if y then x else 0-x  ::  Integer
```

Illustrative example (continued)

General rule:

- In any expression `if` e_1 `then` e_2 `else` e_3 we need e_1 to have type `Bool`, and e_2 and e_3 to have the same type t. The resulting type of `if` e_1 `then` e_2 `else` e_3 is then t.

In our example, we need to type-check

```
if y then x else 0-x  ::  Integer
```

we have `y :: Bool` and `x :: Integer` declared in Γ, so it remains only to type-check

```
0-x :: Integer
```

Illustrative example (completed)

General rule:

- In any expression e_1 - e_2 we need e_1 and e_2 to have type `Integer`. The resulting type of e_1 - e_2 is then `Integer`.

In our example, we need to type-check

```
0-x  ::  Integer
```

we have `x :: Integer` declared in Γ, also the numeral `0` is (of course) given type `Integer`.

Thus indeed we have verified

```
0-x  ::  Integer
```

whence, putting everything together,

```
if y then x else 0-x  ::  Integer
```

as required.

Static type checking — summary

The program is type-checked purely by looking at the AST of the program.

Thus type errors are picked up before the program is executed. Indeed, execution is disallowed for programs that do not type check.

Static type checking gives us a guarantee: no type errors will occur during execution.

This guarantee can be rigorously established as a mathematical theorem, using a mathematical model of program execution called operational semantics. We shall meet operational semantics later in the course (Lecture 27).