COMP3012/G53CMP: Lecture 8

Contextual Analysis: Types and Type Systems I

Henrik Nilsson

University of Nottingham, UK

This Lecture

- Types and Type Systems
- Language safety
- Achieving safety through types
  - Introduction: relation between static and dynamic semantics
  - Operational dynamic semantics of a small example language.

Much of this lecture follows parts of the first few chapters of B. C. Pierce 2002 *Types and Programming Languages* closely.

Types and Type Systems (1)

Type systems are an example of ***lightweight formal methods***:

- highly automated
- but with limited expressive power.

A plausible definition (Pierce):

> *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*

Types and Type Systems (2)

Notes on the definition:

- ***Static checking*** implied as the goal is to ***prove*** absence of certain errors.
- Done by ***classifying*** syntactic phrases (or ***terms***) according to the ***kinds*** of value they compute: a type system computes a ***static approximation*** of the run-time behaviour.

Types and Type Systems (3)

Example: if known that two program fragments exp_1 and exp_2 compute integers (***classification***), then it is safe to add those numbers together (***absence of errors***):

$$exp_1 + exp_2$$

Also known that the result is an integer. While not known exactly which integers are involved, at least known they are integers and nothing else (***static approximation***).

Types and Type Systems (4)

- "Dynamically typed" languages do not have a type system according to this definition; they should really be called ***dynamically checked***.

Example. In a dynamically checked language, $exp_1 + exp_2$ would be evaluated as follows:

- Evaluate exp_1 and exp_2
- Add results together in a manner depending on their types (integer addition, floating point addition, . . .), or signal error if not possible.

Types and Type Systems (5)

- A type system is necessarily ***conservative***: some well-behaved programs will be rejected.

For example, typically

$$\texttt{if } \textit{complex test} \texttt{ then } S \texttt{ else } \textit{type error}$$

will be rejected as ill-typed, even if $\textit{complex test}$ actually always evaluates to true because that cannot be automatically proved in general.

Types and Type Systems (6)

- A type system checks for ***certain*** kinds of bad program behaviour, or ***run-time errors***. Exactly which depends on the type system and the language design.

For example: current main-stream type systems typically

> ***do check*** that arithmetic operations only are done on numbers
> ***do not check*** that the second operand of division is not zero, that array indices are within bounds.

Types and Type Systems (7)

- The ***safety*** or ***soundness*** of a type system must be judged with respect to its own set of run-time errors.

Language Safety (1)

Language safety is a contentious notion. A possible definition (Pierce):

A safe language is one that protects its own abstractions.

For example: a Java object should behave as an object; e.g. it would be bad if it was destroyed by creation of some *other* object.

Other examples: lexical scope rules, visibility attributes (`public`, `protected`, ...).

Language Safety (2)

- Language safety *not* the same as static typing: safety can be *achieved* through static typing and/or dynamic run-time checks.
- Scheme is a dynamically checked safe language.
- Even statically typed languages usually use some dynamic checks; e.g.:
  - checking of array bounds
  - down-casting (e.g. Java)
  - checking for division by zero
  - pattern-matching failure

Language Safety (3)

Some examples of statically and dynamically checked safe and unsafe high-level languages:

	Statically chkd	**Dynamically chkd**
Safe	ML, Haskell, Java	Lisp, Scheme, Perl, Python, Postscript
Unsafe	C, C++	Certain Basic dialects

Advantages of Typing (1)

- Detecting errors early
  Programs in richly typed languages often "just work". Why?
  - Simple, common mistakes very often lead to type inconsistencies.
  - Programmers forced to think a bit harder.
- Enforcing disciplined programming
  Type systems are the backbone of
  - Modules
  - Classes

Advantages of Typing (2)

- Documentation
  - Unlike comments, type signatures will always remain current.
- Efficiency
  - First use of types in computing was to distinguish between integer and floating point numbers.
  - Elimination of many of the dynamic checks that otherwise would have been needed to guarantee safety.

Disadvantages of Typing

Type systems sometimes do get in the way:

- Simple things can become quite complicated if have to work around the type system. (Example: heterogeneous lists in Haskell)
- Sometimes it becomes impossible to do what one wants to do, at least not without loss of efficiency.

Increasingly sophisticated type systems, which keep track of more invariants, can help.

But that can make the type systems harder to understand and less automatic.

Static and Dynamic Semantics

In summary:

- A type system *statically* proves properties about the *dynamic* behaviour of a programs.
- To make precise exactly what these properties are, and formally *prove* that a type system achieves its goals, both the
  - *static semantics*
  - *dynamic semantics*

  must first be formalized.

Example Language: Abstract Syntax

Example language. (Will be extended later.)

$t \rightarrow$		*terms:*
	`true`	*constant true*
$\mid$	`false`	*constant false*
$\mid$	`if` t `then` t `else` t	*conditional*
$\mid$	`0`	*constant zero*
$\mid$	`succ` t	*successor*
$\mid$	`pred` t	*predecessor*
$\mid$	`iszero` t	*zero test*

Values (1)

The *values* of a language are a subset of the terms that are *possible results of evaluation*.

I.e. the values are the *meaning* of terms according to the *dynamic semantics* of the language.

- The evaluation rules are going to be such that no evaluation is possible for values.
- A term to which no evaluation rule applies is a *normal form*.
- All values are normal forms.

Values (2)

$$
\begin{array}{llll}
v & \rightarrow & & \textit{values:} \\
 & & \texttt{true} & \textit{true value} \\
 & | & \texttt{false} & \textit{false value} \\
 & | & nv & \textit{numeric value} \\
\\
nv & \rightarrow & & \textit{numeric values:} \\
 & & \texttt{0} & \textit{zero value} \\
 & | & \texttt{succ}\ nv & \textit{successor value}
\end{array}
$$

One Step Evaluation Relation (1)

$t \longrightarrow t'$ is an ***evaluation relation*** on terms. Read:

t evaluates to t' in one step.

The evaluation relation constitutes an ***operational*** (dynamic) ***semantics*** for the example language.

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \qquad \text{(E-IFTRUE)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \qquad \text{(E-IFFALSE)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \qquad \text{(E-IF)}$$

Evaluation Relation??? (1)

- Recall that a ***mathematical relation*** can be understood as a (possibly infinite) set of pairs of “related things”. For example:

$$\{(1,1),(1,2),(1,3),(2,2),(2,3)\} \subseteq (\leq)$$

- The idea of our “one step evaluation relation” is that the “related things” are ***terms*** and that one term is related to another iff the first evaluates to the second in one step.
- For example:

$$(\texttt{if true then succ 0 else 0}, \texttt{succ 0}) \in (\longrightarrow)$$

Evaluation Relation??? (2)

- But the evaluation relation is infinite . . . so we can’t enumerate all pairs.
- Instead, (schematic) ***inference rules*** are used to specify relations:

$$\frac{\textit{Premise}_1 \ \textit{Premise}_2 \ \ldots \ \textit{Premise}_n}{\textit{Conclusion}}$$

(`en.wikipedia.org/wiki/Rule_of_inference`)

- If there are no premises, the line is often omitted:

$$\frac{}{\textit{Conclusion}} \quad \text{or} \quad \textit{Conclusion}$$

Evaluation Relation??? (3)

- ***Schematic*** means that universally quantified variables are allowed in the rules. For example:

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2$$

- Such a ***rule schema*** actually stands for an ***infinite set*** of rules:

$$\begin{array}{c}
\ldots \\
\texttt{if true then 0 else 0} \longrightarrow \texttt{0} \\
\texttt{if true then succ 0 else 0} \longrightarrow \texttt{succ 0} \\
\texttt{if true then true else false} \longrightarrow \texttt{true} \\
\ldots
\end{array}$$

Evaluation Relation??? (4)

- The ***domain*** of a variable is often specified by ***naming conventions***. E.g. the name of a variable may indicate some specific ***syntactic category*** such as t, v, or nv:

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}$$

$$\texttt{pred}\ (\texttt{succ}\ nv_1) \longrightarrow nv_1$$

One Step Evaluation Relation (2)

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ } t_1 \longrightarrow \texttt{succ } t_1'} \qquad \text{(E-SUCC)}$$

$$\texttt{pred 0} \longrightarrow \texttt{0} \qquad \text{(E-PREDZERO)}$$

$$\texttt{pred}\ (\texttt{succ}\ nv_1) \longrightarrow nv_1 \qquad \text{(E-PREDSUCC)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred } t_1 \longrightarrow \texttt{pred } t_1'} \qquad \text{(E-PRED)}$$

One Step Evaluation Relation (3)

$$\texttt{iszero 0} \longrightarrow \texttt{true} \qquad \text{(E-ISZEROZERO)}$$

$$\texttt{iszero}\ (\texttt{succ}\ nv_1) \longrightarrow \texttt{false} \qquad \text{(E-ISZEROSUCC)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{iszero } t_1 \longrightarrow \texttt{iszero } t_1'} \qquad \text{(E-ISZERO)}$$

One Step Evaluation Relation (4)

Let’s evaluate some terms according to $\longrightarrow$:

- `pred (pred 0)`
- `if (iszero (pred (succ 0)))`
  `then (pred 0) else (succ 0)`
- `if 0 then 0 else 0`

(On the whiteboard.)

Values and Stuck Terms

Note that:

- ***Values*** cannot be evaluated further. E.g.:
  - `true`
  - `0`
  - `succ` (`succ 0`)
- Certain "obviously nonsensical" states are ***stuck***: the term cannot be evaluated further, but it is not a value. For example:

  `if 0 then pred 0 else 0`

Stuckness and Run-Time Errors

- We let the notion of getting stuck ***model run-time errors***.
- The ***goal*** of a type system is thus to ***guarantee*** that a program ***never gets stuck***!

These ideas will be made more precise next time.