#### **G54FOP: Lecture 12** *Types and Type Systems I*

#### Henrik Nilsson

University of Nottingham, UK

#### **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.

#### **This Lecture**

- Types and type systems
- Language safety
- Achieving safety through types:
  - relating static and dynamic semantics
  - Safety = Progress + Preservation

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 (2)**

Notes on the definition:

- Static (= compile time) checking implied since 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.

G54FOP: Lecture 12 - p.3/34

G54FOP: Lecture 12 - p.1/34

G54FOP: Lecture 12 - p.2/34

## **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*).

G54FOP: Lecture 12 - p.5/34

G54FOP: Lecture 12 - p.7/34

## **Types and Type Systems (5)**

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

#### For example, typically

if  $complex \ test$  then S else  $type \ error$ 

will be rejected as ill-typed, even if *complex test* actually always evaluates to true, since that cannot be known statically in general.

## **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.

G54FOP: Lecture 12 - p.6/34

G54FOP: Lecture 12 - p.8/34

#### **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 (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 bt zero
  - pattern-matching failure

#### 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 (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

G54FOP: Lecture 12 - p.11/34

G54FOP: Lecture 12 - p.9/34

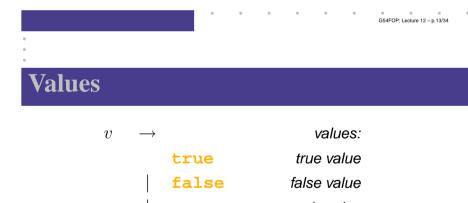
G54FOP: Lecture 12 - p.10/34

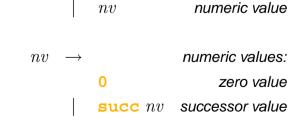
#### **Static and Dynamic Semantics**

#### 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.



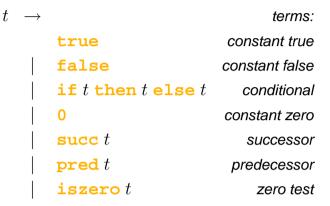


G54FOP: Lecture 12 - p 15/34

Recall: all values are *normal forms*.

## **Example Language: Abstract Syntax**

#### Example language. (Will be extended later.)



## **Dynamic Semantics (1)**

We will define the dynamic semantics *operationally* by giving a (small step) evaluation relation:

 $t \longrightarrow t'$  Read: t evaluates to t' in one step

G54FOP: Lecture 12 - p.14/34

G54FOP: Lecture 12 - p.16/34

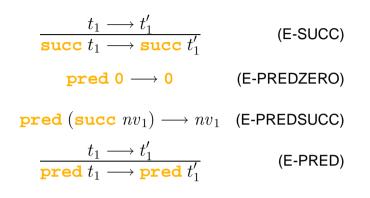
**if true then**  $t_2$  **else**  $t_3 \longrightarrow t_2$  (E-IFTRUE)

**if false then**  $t_2$  **else**  $t_3 \longrightarrow t_3$  (E-IFFALSE)

$$\frac{t_1 \longrightarrow t'_1}{\texttt{if } t_1 \texttt{then } t_2 \texttt{else } t_3} \tag{E-IF}$$

$$\longrightarrow \texttt{if } t'_1 \texttt{then } t_2 \texttt{else } t_3$$

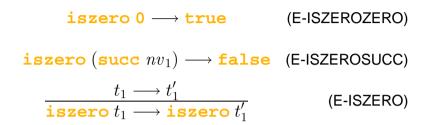
#### **Dynamic Semantics (2)**



6 654EOP Lecture 12 – p 17/34

654FOP: Lecture 12 – p. 19/34

#### **Dynamic Semantics (3)**



G54EOP: Lecture 12 – p 18/34

654EOP: Lecture 12 – p.20/34

#### **Stuck Terms**

- Recall that values are normal forms and cannot be evaluated further; for example:
  - true
  - succ (succ 0)
- However, all normal forms are not values! Can you find an example?
  - if 0 then pred 0 else 0

Normal forms that are not values are called *stuck terms*.

#### **Stuckness and Run-Time Errors**

- Why stuck?
  - A stuck term is *nonsensical* according to the dynamic semantics.
  - We are attempting to *break the abstractions* of the language.
- We let the notion of getting stuck *model run-time errors*.
- The goal of a type system is to rule out all ill-defined programs, thus guaranteeing that a "good', i.e., well-typed, program never gets stuck!

#### Aside: Curry vs. Church Style

This is the "Curry-style" approach: the dynamic semantics comes before the static semantics.

Alternatively, one can start with the static semantics, and then only consider the dynamic semantics of well-typed terms: the "Church-style" approach.

G54FOP: Lecture 12 - p.21/34

G54FOP: Lecture 12 - p.23/34

#### Types

At this point, there are only two types, booleans and the natural numbers:

T	$\rightarrow$		types:
		Bool	type of booleans
		Nat	type of natural numbers

# Typing Rules

true:Bool	(T-TRUE)	
false:Bool	(T-FALSE)	
$\frac{t_1:\texttt{Bool} t_2:T t_3:T}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3:T}$	(T-IF)	
0:Nat	(T-ZERO)	
$rac{t_1:  extsf{Nat}}{ extsf{succ} \ t_1:  extsf{Nat}}$	(T-SUCC)	
$rac{t_1: \mathtt{Nat}}{\mathtt{pred} \ t_1: \mathtt{Nat}}$	(T-PRED)	
$\displaystyle rac{t_1:  extsf{Nat}}{ extsf{iszero}  t_1:  extsf{Bool}}$	(T-ISZERO)	

**Safety = Progress + Preservation (1)** 

G54FOP: Lecture 12 - p.22/34

G54FOP: Lecture 12 - p.24/34

The most basic property of a type system: *safety*, or *"well typed programs do not go wrong"*, where "wrong" means entering a "stuck state".

This breaks down into two parts:

- Progress: A well-typed term is not stuck.
- Preservation: If a well-typed term takes a step of evaluation, then the resulting term is also well-typed. (Aka Subject Reduction)

Together, these properties say that a well-typed term can never reach a stuck state during evaluation.

#### **Safety = Progress + Preservation (2)**

Formally:

• THEOREM [PROGRESS]: Suppose that t is a well-typed term (i.e., t : T), then either t is a value or else there is some t' with  $t \longrightarrow t'$ .

**PROOF:** By induction on a derivation of t : T.

• THEOREM [PRESERVATION]: If t : T and  $t \longrightarrow t'$  then t' : T.

**PROOF:** By induction on a derivation of t : T.

G54FOP: Lecture 12 - p.25/34

G54FOP: Lecture 12 - p.27/34

(Strong form: exact type T preserved.)

#### **Progress: A Proof Fragment** (2)

A typical case when proving Progress by induction on a derivation of t : T.

Case T-IF:  $t = if t_1 then t_2 else t_3$  $t_1 : Bool t_2 : T t_3 : T$ 

By ind. hyp, either  $t_1$  is a value, or else there is some  $t'_1$  such that  $t_1 \longrightarrow t'_1$ . If  $t_1$  is a value, then it must be either **true** or **false**, in which case either E-IFTRUE or E-IFFALSE applies to t. On the other hand, if  $t_1 \longrightarrow t'_1$ , then by E-IF,  $t \longrightarrow if t'_1$  then  $t_2$  else  $t_3$ .

#### **Progress: A Proof Fragment (1)**

The relevant *typing* and *evaluation* rules for the case T-IF:

$$\frac{t_1: \texttt{Bool} \quad t_2: T \quad t_3: T}{\texttt{if} \ t_1 \ \texttt{then} \ t_2 \ \texttt{else} \ t_3: T} \tag{T-IF}$$

**if true then**  $t_2$  **else**  $t_3 \longrightarrow t_2$  (E-IFTRUE)

if false then  $t_2$  else  $t_3 \longrightarrow t_3$  (E-IFFALSE)

$$\begin{array}{c} t_1 \longrightarrow t'_1 \\ \hline \texttt{if} t_1 \texttt{then} t_2 \texttt{else} t_3 \\ \longrightarrow \texttt{if} t'_1 \texttt{then} t_2 \texttt{else} t_3 \end{array} \tag{E-IF}$$

G54FOP: Lecture 12 - p.26/34

G54EOP: Lecture 12 - p 28/34

#### **Preservation:** A **Proof** Fragment (1)

A typical case when proving Preservation by induction on a derivation of t : T.

Case T-IF:  $t = if t_1 then t_2 else t_3$  $t_1 : Bool t_2 : T t_3 : T$ 

Evaluation can be made by one of the rules E-IFTRUE, E-IFFALSE, E-IF.

If evaluation is by any of the two former, then the result is either  $t_2$  or  $t_3$ . But both have type T, just like t, so the type is manifestly preserved.

#### **Preservation: A Proof Fragment (2)**

Case T-IF:  $t = if t_1 then t_2 else t_3$  $t_1 : Bool t_2 : T t_3 : T$ 

If evaluation is by rule E-IF, then we know  $t_1 \longrightarrow t'_1$ . Thus, by the induction hypothesis, we know  $t'_1$ : Bool. And then we can conclude by T-IF that if  $t'_1$  then  $t_2$  else  $t_3$ : T, so the type is preserved also in this case.

G54FOP: Lecture 12 - p.29/34

G54FOP: Lecture 12 - p.31/34

#### Homework

- 1. Prove Progress for the case T-TRUE.
- 2. Prove Preservation for the case T-TRUE.
- 3. Prove Progress for the case T-ISZERO.
- 4. Prove Preservation for the case T-ISZERO.

## **Exceptions** (1)

What about terms like

- division by zero
- head of empty list

that usually are considered well-typed?

If the type system does not rule them out, we need to differentiate those from stuck terms, or we can no longer claim that "well-typed programs do not go wrong"!

## **Exceptions** (2)

Idea: allow *exceptions* to be raised, and make it well-defined what happens when exceptions are raised.

G54FOP: Lecture 12 - p.30/34

G54FOP: Lecture 12 - p.32/34

For example:

- introduce a term error
- introduce evaluation rules like

#### head [] $\longrightarrow$ error

• typing rule: **error** : T

## **Exceptions (3)**

 introduce propagation rules to ensure that the entire program evaluates to error once the exception has been raised (unless there is some exception handling mechanism), e.g.:

#### $\mathbf{pred}\;\mathbf{error}\longrightarrow\mathbf{error}$

 change the Progress theorem slightly to allow for exceptions:

THEOREM [PROGRESS]: Suppose that t is a well-typed term (i.e., t : T), then either t is a value **or error**, or else there is some t' with  $t \longrightarrow t'$ .

G54FOP: Lecture 12 - p.33/34

## Aside: error is not a value

For technical reasons, to avoid overlap between evaluation rules that propagate error and the normal ones, it can be preferable to not consider error a value.

E.g., for function application we might have

$$\lambda x \, {\scriptstyle \bullet} \, t) \; v \longrightarrow [x \mapsto v] t$$

which would overlap with

#### $v \operatorname{error} \longrightarrow \operatorname{error}$

G54FOP: Lecture 12 - p.34/34

if error were considered a value.