COMP3012/G53CMP: Lecture 9 *Contextual Analysis: Types and Type Systems II*

Henrik Nilsson

University of Nottingham, UK

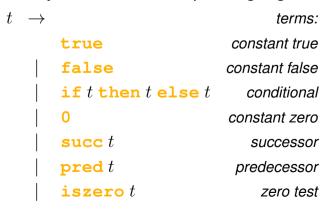
This Lecture

- Recapitulation: our example language, stuck terms, type systems.
- · Basic typing rules
- Safety = Progress + Preservation
- Extensions: typing let-expressions and functions

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

Recap: Example Language

Abstract syntax for the example language:



COMP3012/G53CMP: Lecture 9 – p.3/34

COMP3012/G53CMP: Lecture 9 - p.1/34

Recap: Values

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

v	\rightarrow		values:	
		true	true value	
		false	false value	
		nv	numeric value	
nv	\rightarrow		numeric values:	
		0	zero value	
		succ nv	successor value	
Values are <i>normal forms</i> : they cannot be				

evaluated further.

COMP3012/G53CMP: Lecture 9 – p.4/34

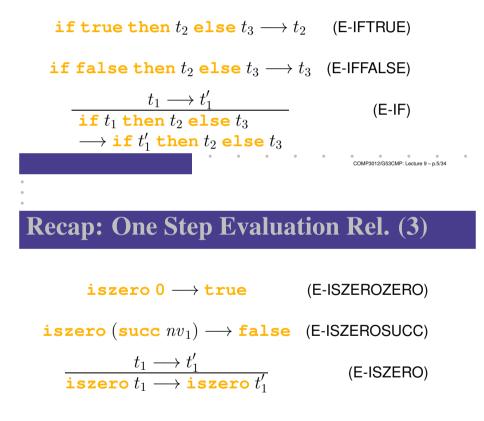
COMP3012/G53CMP: Lecture 9 - p.2/34

Recap: One Step Evaluation Rel. (1)

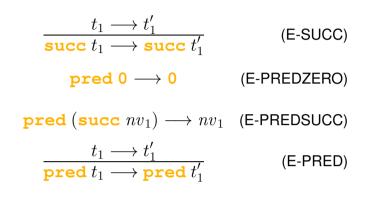
 $t \longrightarrow t'$ is an *evaluation relation* on terms. Read:

t evaluates to t' in one step.

The evaluation relation constitute an *operational semantics* for the example language.



Recap: One Step Evaluation Rel. (2)



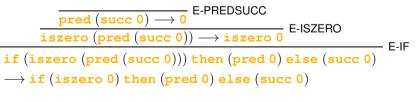
COMP3012/G53CMP: Lecture 9 – p.6/34

Recap: One Step Evaluation Rel. (4)

Evaluation of:

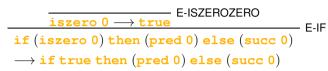
if (iszero (pred (succ 0))) then (pred 0) else (succ 0)

Step 1:



Recap: One Step Evaluation Rel. (5)

Step 2:



Step 3:

if true then (pred 0) else (succ 0) \rightarrow pred 0 E-IFTRUE

pred 0 \longrightarrow 0

E-PREDZERO

Step 4:

Stuck Terms (2)

 We let the notion of getting stuck model run-time errors.

Stuck Terms (1)

 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

- Definition: A term is *stuck* if it is a normal form but not a value.
- Why stuck???
 - The program is *not well-defined* according to the dynamic semantics of the language.
 - We are attempting to *break the abstractions* of the language.

COMP3012/G53CMP: Lecture 9 - p.10/34

Recap: Type Systems

Definitions (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.
- A safe language is one that protects its abstractions.

Our goal is thus a type system that rules out semantically ill-defined programs, i.e. that guarantees that a program never gets stuck!

COMP3012/G53CMP: Lecture 9 - p.11/34

COMP3012/G53CMP: Lecture 9 - n 9/34

COMP3012/G53CMP: Lecture 9 – p.12/34

Why Should We Care About Safety?

- One reason: security.
- C/C++ is unsafe: buffer overruns possible.
- Buffer overruns allows input data to be executed as code.
- One of the most common security holes: Had a safe variant of C been used, one might speculate that billions of dollars would have been saved.

Today, we're going to see how to go about proving that the *design* of a language is safe.

Typing Relation

We will define a *typing relation* between terms and types:

t:T

Read:

t has type T

A term that has a type, i.e., is related to a type by such a typing relation, is said to be *well-typed*.

The typing relation will be defined by (schematic) typing rules, in the same way we defined the evaluation relation.

COMP3012/G53CMP: Lecture 9 – p.15/34

COMP3012/G53CMP: Lecture 9 - p.13/34

Types

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

Т	\rightarrow		types:
		Bool	type of booleans
		Nat	type of natural numbers

Typing Rules

(T-TRUE)
(T-FALSE)
(T-IF)
(T-ZERO)
(T-SUCC)
(T-PRED)
(T-ISZERO)

COMP3012/G53CMP: Lecture 9 - p.16/34

COMP3012/G53CMP: Lecture 9 - p.14/34

Exercise

What (if any) is the type of the following terms?

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

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' such that t → 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.

(Strong form: exact type T preserved.)

COMP3012/G53CMP: Lecture 9 – p 19/34

COMP3012/G53CMP: Lecture 9 - p 17/34

Safety = Progress + Preservation (1)

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.

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

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)

$$\begin{array}{c} t_1 \longrightarrow t_1' \\ \hline \texttt{if} \ t_1 \,\texttt{then} \ t_2 \,\texttt{else} \ t_3 \\ \longrightarrow \texttt{if} \ t_1' \,\texttt{then} \ t_2 \,\texttt{else} \ t_3 \end{array}$$

(E-IF)

COMP3012/G53CMP: Lecture 9 - p 18/34

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 .

COMP3012/G53CMP: Lecture 9 - p.21/34

COMP3012/G53CMP: Lecture 9 - n 23/34

Exceptions (2)

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

For example:

- introduce a term error
- introduce evaluation rules like

head [] \rightarrow error

• typing rule: **error** : *T*

Exceptions (1)

What about terms like

- division by zero
- head of empty list
- array indexing out of bounds (like buffer overrun) 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"!

COMP3012/G53CMP: Lecture 9 - p 22/34

COMP3012/G53CMP: Lecture 9 - p 24/34

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.:

pred error \longrightarrow 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 $t \rightarrow t'$.

Extension: Let-bound Variables (1)

Syntactic extension:

New evaluation rules:

$$let x = v_1 in t_2 \longrightarrow [x \mapsto v_1]t_2 \quad (E-LETV)$$

$$\frac{t_1 \longrightarrow t'_1}{let x = t_1 in t_2 \longrightarrow let x = t'_1 in t_2} \quad (E-LET)$$

Extension: Let-bound Variables (3)

Environment-related notation:

• Extending an environment:

 $\Gamma, x: T$

The new declaration is understood to replace any earlier declaration for a variable with the same name.

 Stating that the type of a variable is given by an environment:

 $x: T \in \Gamma$ or $\Gamma(x) = T$

COMP3012/G53CMP: Lecture 9 – p.27/34

COMP3012/G53CMP: Lecture 9 - n 25/34

Extension: Let-bound Variables (2)

We now need a *typing context* or *type environment* to keep track of types of variables (an abstract version of a "symbol table").

The typing relation thus becomes a *ternary relation*:

$\Gamma \vdash t:T$

Read: term t has type T in type environment Γ .

Extension: Let-bound Variables (4)

Updated typing rules:

 $\Gamma \vdash \texttt{true} : \texttt{Bool}$ (T-TRUE)

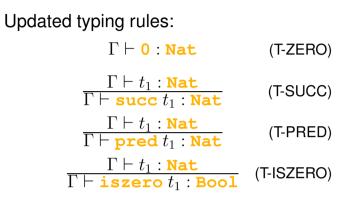
 $\Gamma \vdash \texttt{false}: \texttt{Bool} \tag{T-FALSE}$

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

COMP3012/G53CMP: Lecture 9 – p.28/34

COMP3012/G53CMP: Lecture 9 - p 26/34

Extension: Let-bound Variables (5)



Extension: Let-bound Variables (6)

New typing rules:

$$\frac{x:T \in \Gamma}{\Gamma \vdash x:T}$$
(T-VAR)
$$\frac{\Gamma \vdash t_1:T_1 \quad \Gamma, x:T_1 \vdash t_2:T_2}{\Gamma \vdash \textbf{let} \ x \ = \ t_1 \ \textbf{in} \ t_2:T_2}$$
(T-LET)

Extension: Let-bound Variables (7)

COMP3012/G53CMP: Lecture 9 - p.29/34

COMP3012/G53CMP: Lecture 9 - p.31/34

Recursive bindings?

Typing is straightforward if the recursively-defined entity is *explicitly* typed:

$$\frac{\Gamma, x: T_1 \vdash t_1: T_1}{\Gamma \vdash \texttt{le} \ x: T_1} \stackrel{\Gamma, x: T_1 \vdash t_2: T_2}{= t_1 \texttt{ in } t_2: T_2} \quad (\texttt{T-LET})$$

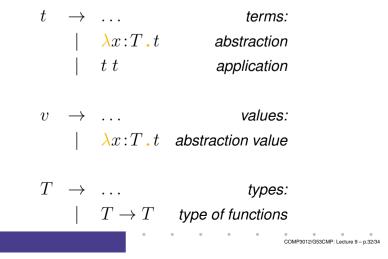
If not, the question is if T_1 is uniquely defined (and in a type checker how to compute this type):

$$\frac{\Gamma, x: T_1 \vdash t_1: T_1 \quad \Gamma, x: T_1 \vdash t_2: T_2}{\Gamma \vdash \textbf{let} \ x \ = \ t_1 \ \textbf{in} \ t_2: T_2} \quad (\textbf{T-LET})$$

(*Evaluation* is more involved: we leave that for now.)

Extension: Functions (1)

Syntactic extension:



COMP3012/G53CMP: Lecture 9 - p.30/34

Extension: Functions (2)

New evaluation rules:

$$\frac{t_1 \longrightarrow t'_1}{t_1 \ t_2 \longrightarrow t'_1 \ t_2}$$
(E-APP1)
$$\frac{t_2 \longrightarrow t'_2}{v_1 \ t_2 \longrightarrow v_1 \ t'_2}$$
(E-APP2)

COMP3012/G53CMP: Lecture 9 - p.33/34

 $(\lambda x : T_{11} \cdot t_{12})v_2 \longrightarrow [x \mapsto v_2]t_{12}$ (E-APPABS)

Note:

- left to right evaluation order: first the function (E-APP1), then the argument (E-APP2)
- call-by-value: the argument fully evaluated before function "invoked" (E-APPABS).

Extension: Functions (3)

New typing rules:

$$\begin{array}{c} \frac{\Gamma, x: T_1 \vdash t_2: T_2}{\Gamma \vdash \lambda x: T_1 \cdot t_2: T_1 \rightarrow T_2} & (\text{T-ABS}) \\ \frac{\Gamma \vdash t_1: T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2: T_{11}}{\Gamma \vdash t_1 \ t_2: T_{12}} & (\text{T-APP}) \end{array}$$

COMP3012/G53CMP: Lecture 9 - p.34/34