# 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:

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



## 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 ***normal forms***: they cannot be evaluated further.



## 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.

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

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

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



## Recap: One Step Evaluation Rel. (2)

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

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

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

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



## Recap: One Step Evaluation Rel. (3)

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

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

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



## Recap: One Step Evaluation Rel. (4)

Evaluation of:

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

Step 1:

$$\frac{\dfrac{\dfrac{}{\texttt{pred (succ 0)} \longrightarrow \texttt{0}}\ \text{E-PREDSUCC}}{\texttt{iszero (pred (succ 0))} \longrightarrow \texttt{iszero 0}}\ \text{E-ISZERO}}{\texttt{if (iszero (pred (succ 0))) then (pred 0) else (succ 0)} \longrightarrow \texttt{if (iszero 0) then (pred 0) else (succ 0)}}\ \text{E-IF}$$



## Recap: One Step Evaluation Rel. (5)

Step 2:

$$\frac{\dfrac{}{\texttt{iszero 0} \longrightarrow \texttt{true}}\ \text{E-ISZEROZERO}}{\texttt{if (iszero 0) then (pred 0) else (succ 0)} \longrightarrow \texttt{if true then (pred 0) else (succ 0)}}\ \text{E-IF}$$

Step 3:

$$\frac{}{\texttt{if true then (pred 0) else (succ 0)} \longrightarrow \texttt{pred 0}}\ \text{E-IFTRUE}$$

Step 4:

$$\frac{}{\texttt{pred 0} \longrightarrow \texttt{0}}\ \text{E-PREDZERO}$$

## 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.



## Stuck Terms (2)

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



## 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!***



## 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.



## Types

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

| | | |
|---|---|---|
| $T$ $\rightarrow$ | | *types:* |
| | `Bool` | *type of booleans* |
| $\vert$ | `Nat` | *type of natural numbers* |



## 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.



## Typing Rules

$$\texttt{true} : \texttt{Bool} \qquad \text{(T-TRUE)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-FALSE)}$$

$$\frac{t_1 : \texttt{Bool} \quad t_2 : T \quad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \qquad \text{(T-IF)}$$

$$\texttt{0} : \texttt{Nat} \qquad \text{(T-ZERO)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{succ } t_1 : \texttt{Nat}} \qquad \text{(T-SUCC)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{pred } t_1 : \texttt{Nat}} \qquad \text{(T-PRED)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{iszero } t_1 : \texttt{Bool}} \qquad \text{(T-ISZERO)}$$



## 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 (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.

## 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 $t \longrightarrow t'$.

  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 $t : T$.

  (Strong form: exact type $T$ preserved.)



## Progress: A Proof Fragment (1)

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

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

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

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

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



## Progress: A Proof Fragment (2)

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

Case T-IF: $t = \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$

$t_1 : \texttt{Bool} \quad t_2 : T \quad t_3 : T$

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, $t \longrightarrow \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3$.



## 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"!



## 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

$$\texttt{head []} \longrightarrow \texttt{error}$$

- typing rule: $\texttt{error} : T$



## 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.:

$$\texttt{pred error} \longrightarrow \texttt{error}$$

- 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 $t \longrightarrow t'$.



## Extension: Let-bound Variables (1)

Syntactic extension:

| | | | |
|---|---|---|---|
| $t$ | $\rightarrow$ | $\ldots$ | *terms:* |
| | $\mid$ | $x$ | *variable* |
| | $\mid$ | $\texttt{let}\ x = t\ \texttt{in}\ t$ | *let-expression* |

New evaluation rules:

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

$$\frac{t_1 \longrightarrow t_1'}{\texttt{let}\ x = t_1\ \texttt{in}\ t_2 \longrightarrow \texttt{let}\ x = t_1'\ \texttt{in}\ t_2} \quad \text{(E-LET)}$$



## 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 $\Gamma$.



## 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:

$$x : T \in \Gamma \quad \text{or} \quad \Gamma(x) = T$$

## Extension: Let-bound Variables (4)

Updated typing rules:

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

$$\Gamma \vdash \texttt{false} : \texttt{Bool} \qquad \text{(T-FALSE)}$$

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



## Extension: Let-bound Variables (5)

Updated typing rules:

$$\Gamma \vdash \texttt{0} : \texttt{Nat} \qquad \text{(T-ZERO)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Nat}}{\Gamma \vdash \texttt{succ}\ t_1 : \texttt{Nat}} \qquad \text{(T-SUCC)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Nat}}{\Gamma \vdash \texttt{pred}\ t_1 : \texttt{Nat}} \qquad \text{(T-PRED)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Nat}}{\Gamma \vdash \texttt{iszero}\ t_1 : \texttt{Bool}} \qquad \text{(T-ISZERO)}$$



## Extension: Let-bound Variables (6)

New typing rules:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-VAR)}$$

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



## Extension: Let-bound Variables (7)

Recursive bindings?

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

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

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

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

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



## Extension: Functions (1)

Syntactic extension:

| | | | |
|---|---|---|---|
| $t$ | $\rightarrow$ | $\ldots$ | *terms:* |
| | $\mid$ | $\lambda x : T\,.\,t$ | *abstraction* |
| | $\mid$ | $t\ t$ | *application* |
| $v$ | $\rightarrow$ | $\ldots$ | *values:* |
| | $\mid$ | $\lambda x : T\,.\,t$ | *abstraction value* |
| $T$ | $\rightarrow$ | $\ldots$ | *types:* |
| | $\mid$ | $T \rightarrow T$ | *type of functions* |



## Extension: Functions (2)

New evaluation rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-APP2)}$$

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

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:

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1\,.\,t_2 : T_1 \rightarrow T_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-APP)}$$