EDA045F: Program Analysis

LECTURE 11: TYPES 2

Christoph Reichenbach

In the last lecture...

- Basic Natural Semantics

$$e \Downarrow v$$

- Basic Structural Operational Semantics

$$e \longrightarrow e'$$

- Basic Type Systems

$$e : \tau$$

Formal Properties

- Many semantics and type systems are possible
- Not all are useful
- Some key properties to look out for (or formally prove):
  1. Every execution path produces the same result
  2. Evaluating an expression doesn't change its type
  3. If a program typechecks and isn't a value, we can evaluate it further
- Usually easier to show on small-step (structural) operational semantics

Confluence

No matter which evaluation path we take, the program always produces the same result:

- If we have the following:
  - $p_1 \longrightarrow^\star p_2$
  - $p_1 \longrightarrow^\star p_3$
- Then the following holds for some p_4:

$$p_2 \longrightarrow^\star p_4 \longleftarrow^\star p_3$$

Any operational semantics with this property is *confluent*

Preservation

Evaluating (part of) program one step doesn't change result type:

- Assume:
  - p : τ
  - $p \longrightarrow^{\star} p'$
- Then we have

$$p' : \tau$$

Operational semantics have the *preservation* property if they satisfy the above

Progress

A well-typed program does not get stuck:

- Assume that p is a program.
- Then p is a value if and only if there is no p' such that

$$p \longrightarrow p'$$

- NB: This means that runtime errors (out of bounds, null pointer dereference) must be propagated 'upwards' as errors
  - Relatively easy to handle in structural operational semantics (if **fail** is a value):

$$p \longrightarrow \textbf{fail}$$

  - Trickier in Natural semantics: need special-case handling for every recursive use of $\Downarrow$

Operational semantics have the *progress* property if they satisfy the above

Type Soundness

- Wright, Felleisen 1994: *Type Soundness* is:
  - *Progress*
  - *Preservation*

Summary

- Small-step (structural) operational semantics often basis for proofs of interesting properties
- Key properties:

1. **Confluence**: (Church-Rosser): Every execution path produces the same result
2. **Preservation**: Evaluating an expression doesn't change its type
3. **Progress**: If a program typechecks and isn't a value, we can evaluate it further
4. **Type Soundness** = **Preservation** + **Progress**

The ATL Syntax

$$
\begin{array}{lcl}
\mathit{name} & ::= & \mathit{id} \\
\mathit{val} & ::= & \mathit{num} \\
 & | & \mathsf{null} \\
\mathit{expr} & ::= & \langle \mathit{val} \rangle \\
 & | & \langle \mathit{expr} \rangle + \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{name} \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{stmt} & ::= & \langle \mathit{name} \rangle \ \mathtt{=} \ \langle \mathit{expr} \rangle \\
 & | & \{ \ \langle \mathit{stmt} \rangle \ \} \\
 & | & \mathsf{if} \ \langle \mathit{expr} \rangle \ \langle \mathit{stmt} \rangle \ \mathsf{else} \ \langle \mathit{stmt} \rangle \\
 & | & \mathsf{while} \ \langle \mathit{expr} \rangle \ \langle \mathit{stmt} \rangle \\
 & | & \mathsf{skip} \\
 & | & \mathsf{return} \ \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{stmt} \rangle ; \ \langle \mathit{stmt} \rangle
\end{array}
$$

Environments and Types

- For typing ATL, we use the following types from $\mathbb{T}_{atl}$:

$$\begin{array}{lcl} 13 & : & \textsc{Int} \\ \textbf{null} & : & \textsc{Null} \\ \textit{statements} & : & \textsc{Unit} \end{array}$$

- Typing variables: same challenge as for operational semantics
  - Convention: use $\Gamma : id \nrightarrow \mathbb{T}_{atl}$ (capital Gamma) as *typing context*:

$$\frac{\Gamma(x) = \textsc{Int}}{\Gamma \vdash x : \textsc{Int}}$$

- We construct two ternary typing relations:

$$\Gamma \vdash e : \tau \subseteq (id \nrightarrow \mathbb{T}_{atl}) \times expr \times \mathbb{T}_{atl}$$
$$\Gamma \vdash_s s : \tau \subseteq (id \nrightarrow \mathbb{T}_{atl}) \times stmt \times \mathbb{T}_{atl}$$

Typing Expressions

$$\frac{v \in num}{\Gamma \vdash v : \textsc{Int}} \ (t\text{-}num)$$

$$\frac{}{\Gamma \vdash \textbf{null} : \textsc{Null}} \ (t\text{-}null)$$

$$\frac{}{\Gamma \cup \{id : \tau\} \vdash id : \tau} \ (t\text{-}id)$$

$$\frac{\Gamma \vdash e_1 : \textsc{Int} \quad \Gamma \vdash e_2 : \textsc{Int}}{\Gamma \vdash e_1 + e_2 : \textsc{Int}} \ (t\text{-}plus)$$

ATL Typing

- Type system again should ensure program doesn't 'get stuck'
- Places where program can get stuck:
  1. Non-INT condition to **while**
  2. Non-INT condition to **if**
  3. Non-INT parameter to **+**
  4. Use of variable before definition
  5. Inconsistent variable type
  6. **return** anywhere other than end of program
- We already know how to do 1–3.
- 4 is difficult with type-based analysis
  - Language design options:
    - require definite assignment (dataflow) analysis (Java locals)
    - default value (Java fields)
    - dynamic error (Python)
    - 'undefined' (C)
    - require initialisation as variables enter scope

    ...

Detecting Incorrect Uses of **return**

- We want **return** only at the end of a program
- One possible solution:
  - Give **return** e the type of e
  - Give all other statements the type $\textsc{Unit}$
  - Require type $\textsc{Unit}$ of all non-final statements
  - Bonus: Type τ of entire program tells us:
    - $\tau = \textsc{Unit}$: No return value
    - Otherwise return value of type τ

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash_s \textbf{return}\ e : \tau}\ (\textit{t-return})$$

$$\frac{}{\Gamma \vdash_s \textbf{skip} : \textsc{Unit}}\ (\textit{t-skip})$$

$$\frac{\Gamma \vdash s_1 : \textsc{Unit} \quad \Gamma \vdash s_2 : \tau}{\Gamma \vdash_s s_1;\ s_2 : \tau}\ (\textit{t-seq})$$

Detecting Inconsistent Variable Types

- Require assignments to match type from environment:

$$\frac{\Gamma \vdash id : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash_s id = e : \textsc{Unit}} \ (\textit{t-assign})$$

- This disallows e.g.:

```
x = null; // require x : NULL ∈ Γ
x = 1     // require x : INT ∈ Γ
```

  - Can't satisfy both requirements at once
    Γ is a partial function, so at most one type for x

Finding the Right Γ

We must find a Γ such that

$$\Gamma \vdash_s \textit{program} : \tau$$

- Γ must satisfy all constraints imposed by *program*
- Two standard approaches:
  - Require *variable declarations* with explicit types, extract Γ ahead of time and merely check
  - Collect requirements imposed by program, solve (⇒ more later)

Collecting (unordered) requirements and solving them is the core of advanced type-based analyses

Summary

- Typing with variables requires use of **contexts**
  - Traditionally written as Γ, sometimes Δ etc.
- Typing relation extended to represent context:

$$\Gamma \vdash e : \tau \subseteq (id \nrightarrow \mathbb{T}_{atl}) \times expr \times \mathbb{T}_{atl}$$

- Typing rules express constraints on context

$$\frac{\dfrac{}{\Gamma' \cup \{id : \tau\} \vdash id : \tau}\,(t\text{-}id) \qquad \Gamma \vdash e : \tau}{\Gamma \vdash_s id = e : \textsc{Unit}}\,(t\text{-}assign)$$

- Depending on type system, constraints may be trivial to solve or require more complex machinery
- Can detect many situations in which program would get stuck

The ATL Syntax

$$
\begin{array}{lcl}
\mathit{name} & ::= & \mathit{id} \\
\mathit{val} & ::= & \mathit{num} \\
 & | & \mathsf{null} \\
\mathit{expr} & ::= & \langle \mathit{val} \rangle \\
 & | & \langle \mathit{expr} \rangle + \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{name} \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{stmt} & ::= & \langle \mathit{name} \rangle \ \mathtt{=} \ \langle \mathit{expr} \rangle \\
 & | & \{ \ \langle \mathit{stmt} \rangle \ \} \\
 & | & \mathsf{if} \ \langle \mathit{expr} \rangle \ \langle \mathit{stmt} \rangle \ \mathsf{else} \ \langle \mathit{stmt} \rangle \\
 & | & \mathsf{while} \ \langle \mathit{expr} \rangle \ \langle \mathit{stmt} \rangle \\
 & | & \mathsf{skip} \\
 & | & \mathsf{return} \ \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{stmt} \rangle ; \ \langle \mathit{stmt} \rangle
\end{array}
$$

The ATL Syntax

$$
\begin{array}{lcl}
\mathit{name} & ::= & \mathit{id} \\
\mathit{val} & ::= & \mathit{num} \\
 & | & \mathtt{null} \\
\mathit{expr} & ::= & \langle \mathit{val} \rangle \\
 & | & \langle \mathit{expr} \rangle \mathtt{+} \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{name} \rangle \\
 & | & \mathit{id}\ \mathtt{(}\ \langle \mathit{expr} \rangle \star\ \mathtt{)} \\
\mathit{ty} & ::= & \mathtt{nat} \\
 & | & \mathtt{null}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{stmt} & ::= & \langle \mathit{name} \rangle\ \mathtt{=}\ \langle \mathit{expr} \rangle \\
 & | & \mathtt{\{}\ \langle \mathit{stmt} \rangle\ \mathtt{\}} \\
 & | & \mathtt{if}\ \langle \mathit{expr} \rangle\ \langle \mathit{stmt} \rangle\ \mathtt{else}\ \langle \mathit{stmt} \rangle \\
 & | & \mathtt{while}\ \langle \mathit{expr} \rangle\ \langle \mathit{stmt} \rangle \\
 & | & \mathtt{skip} \\
 & | & \mathtt{return}\ \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{stmt} \rangle\mathtt{;}\ \langle \mathit{stmt} \rangle \\
\mathit{decl} & ::= & \mathtt{proc}\ \mathit{id}\ \mathtt{(}\ (\mathit{id} : \langle \mathit{ty} \rangle) \star\ \mathtt{)}\ \langle \mathit{stmt} \rangle \\
\mathit{prog} & ::= & \langle \mathit{decl} \rangle\ \langle \mathit{prog} \rangle \\
 & | & \langle \mathit{stmt} \rangle
\end{array}
$$

Subroutines

```
proc mult(x:int, y:int) {
  if y {
    return x + mult(x, y + -1)
  } else {
    return 0
} }
return mult(3, 7)
```

Must collect procedure definitions in order to execute

Subroutines: Representation

- Must represent subroutines in model
  Minimal information:
  - Subroutine body (*stmt*)
  - ordered sequence of parameter names ($id\star$)

$$\textsc{Sub} = stmt \times id\star$$

- Disallow multiple definitions of same subroutine name?

Collect from program p e.g. as $\langle \emptyset, p \rangle \blacktriangleright E_p$:

$$\frac{}{\langle E_p, s \rangle \blacktriangleright E_p}$$

$$\frac{\langle E_p, p \rangle \blacktriangleright E'_p \quad id \notin \mathrm{dom}\ E'_p \quad E''_p = [id \mapsto \langle s, [id_1, \ldots, id_k] \rangle] E'_p}{\langle E_p, \textbf{proc}\ id(id_1 : t_1, \ldots, id_k : t_k)\ s\ p \rangle \blacktriangleright E''_p}$$

Subroutines: Two Semantics

Separate Environments

- E stores variables, E_p subroutines

$$\frac{E_p(id) = \langle s, [id_1, \ldots, id_k] \rangle \qquad \begin{array}{c} \langle E, e_1 \rangle \Downarrow v_1 \cdots \langle E, e_k \rangle \Downarrow v_k \\ \langle \{id_1 \mapsto v_1, \cdots, id_k \mapsto v_k\}, s \rangle \Downarrow v \end{array}}{\langle E, id(e_1 \ \ldots \ e_k) \rangle \Downarrow v} \ (call)$$

Subroutines as Values

- E stores variables and subroutines
  funcvar = mult;
  funcvar(1, 2)
- E_p acts as 'parent scope' without local variables, still needed

$$\frac{E(id) = \langle s, [id_1, \ldots, id_k] \rangle \qquad \begin{array}{c} \langle E, e_1 \rangle \Downarrow v_1 \cdots \langle E, e_k \rangle \Downarrow v_k \\ \langle E_p \cup \{id_1 \mapsto v_1, \cdots, id_k \mapsto v_k\}, s \rangle \Downarrow v \end{array}}{\langle E, id(e_1 \ \ldots \ e_k) \rangle \Downarrow v} \ (call)$$

Subroutines: Types

$$\mathcal{T}(t) = \begin{cases} \textsc{Int} & \iff \quad t = \text{int} \\ \textsc{Null} & \iff \quad t = \text{null} \end{cases}$$

- We encode the type of functions that take types $\tau_1, \ldots, \tau_k$ as parameters and return type τ_0 as:

$$\tau_1 \times \cdots \times \tau_k \to \tau_0$$

$$\frac{\begin{array}{l} \tau_1 = \mathcal{T}(t_1) \quad \cdots \quad \tau_k = \mathcal{T}(t_k) \\ F_{id} : \tau_1 \times \cdots \times \tau_k \to \tau_0 \in \Gamma \end{array} \qquad \begin{array}{r} \Gamma \cup \{id_1 : \tau_1, \ldots, id_k : \tau_k\} \vdash_s s : \tau_0 \\ \Gamma \vdash_p p : \tau \end{array}}{\Gamma \vdash_s \textbf{proc}\ id(id_1 : t_1, \ldots, id_k : t_k)\ s\ p : \tau} \ (t\text{-}proc)$$

$$\frac{\Gamma \vdash F_{id} : \tau_1 \times \cdots \times \tau_k \to \tau_0 \quad \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_k : \tau_k}{\Gamma \vdash id(e_1, \ldots, e_k) : \tau_0} \ (t\text{-}call)$$

Summary

- Handling subroutines requires a lookup mechanism that retrieves subroutine bodies from identifiers
  - Also retrieves parameter names in order
- Subroutine environment should contain:
  - global names (subroutine definitions)
  - parameters
  - nothing else (if we use static scoping)
- Our semantics permit us to treat subroutines as values

Limitations

- Requiring explicit type annotations for subroutine arguments is limiting:
  - Analysis depends on user knowledge
  - Type system is needlessly restrictive:

```
proc idfun(x:?) return x

a = idfun(1);
b = idfun(null)
```

Try to avoid explicit types, *infer* them instead

Type Inference

- Idea:
  - Look at constraints between program pieces
  - Write down constraints
  - Try to solve them
  - Unsolvable? $\Rightarrow$ Type error

Let's investigate this in the context of functional languages

Type Inference: Simple Example

```
fun f ( x ) = (x, x)

val y = f ( 2 )
```

- $\beta = \alpha$
- $\gamma = \alpha \times \alpha$
- f: $\alpha \to \alpha \times \alpha$
- y: $\delta = \alpha \times \alpha$
- $\alpha = \texttt{int}$
- y: $\texttt{int} \times \texttt{int}$

$$\frac{f : \tau_1 \to \tau_2 \quad a : \tau_1}{f(a) : \tau_2} \; (funapp)$$

Type Inference: ideas

- Bind each program variable (including functions) to a fresh type variable
- Systematically apply typing rules, derive equations
- Solve equations
- Unsolvable $\Rightarrow$ Type error

Let's make it more abstract

Type Constructors

- Recall Parametric Polymorphism:
  - `Set[`α`]`
  - `Map[`α`, `β`]`
- Type constructors: things like `Set`, `Map`
  - Take type parameters α, β
  - Build new type
- Other type constructors:
  - $\times$: constructs product types
  - $\rightarrow$: constructs function types
- General notation: $C_i^k(\tau_1, \ldots, \tau_k)$
  - E.g.: $\texttt{int} \rightarrow \texttt{string} = C_{\rightarrow}^2(\texttt{int}, \texttt{string})$
  - E.g.: $Set[Set[\texttt{int}]] = C_{\texttt{Set}}^1(C_{\texttt{Set}}^1(\texttt{int}))$

Type Inference: Solving Equations

- Each equation has one of these forms:
  - $\alpha = \alpha$ (trivial)
  - $\alpha = \beta$
    - Solution: Replace β with α everywhere
  - $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
    - $\boxed{\textit{Type Error}}$ if $i \neq j$ or $k \neq l$
    - Otherwise: Generate equations:

$$\begin{array}{rcl} \tau_1^a & = & \tau_1^b \\ \ldots & & \ldots \\ \tau_k^a & = & \tau_k^b \end{array}$$

  - $\alpha = C_i^k(\tau_1, \ldots, \tau_k)$
    - Solution: Substitute $C_i^k(\alpha_1, \ldots, \alpha_k)$ for α everywhere
    - **Except:** $\alpha = C_i^k(\ldots, \alpha, \ldots) \Rightarrow \boxed{\textit{Type Error}}$

Not fully universal yet

Polymorphic Type Inference

$\forall \alpha.\alpha \to \alpha \times \alpha$ $\quad \alpha \quad \alpha \times \alpha$

```
fun f ( x ) = (x, x)
```

$\alpha_1 \to \alpha_1 \times \alpha_1$

```
val y = f ( 2 )
```

$\alpha_2 \to \alpha_2 \times \alpha_2$

```
val z = f ( "foo" )
```

- Once resolved: make type of `f` polymorphic over all free type variables
- Can instantiate type of `f` more than once:
  - $\alpha_1 = \texttt{int}$
  - $\alpha_2 = \texttt{string}$

(Mutual) Recursion

- If `f`, `g` call each other recursively:
  - Can't resolve the type of one before the other
  - Treat both *monomorphically* (no $\forall$) at first
  - Generalisation (via $\forall$) only after both are considered
- When/where to introduce $\forall$:
  - SML: Language syntax dictates abstraction boundaries
  - Haskell: Cluster functions by mutual recursion, resolve

Summary

- Type Inference allows static type-checking without explicit type annotations
- Algorithm:
  - Introduce type variables
  - Generate equations by systematically applying typing rules
  - Resolve equations recursively, disallowing infinite recursion
  - Abstract over free type variables ($\forall$) to introduce polymorphism
- Mutual recursion:
  - Defer polymorphic abstraction until all mutually recursive functions are resolved
- Standard Type Inference algorithm:
  - Damas-Hindley-Milner (Algorithm $\mathcal{W}$)
  - Formulated for functional languages

The ATL Syntax

$$
\begin{array}{lcl}
\mathit{name} & ::= & \mathit{id} \\
\mathit{val} & ::= & \mathit{num} \\
 & | & \mathtt{null} \\
\mathit{expr} & ::= & \langle \mathit{val} \rangle \\
 & | & \langle \mathit{expr} \rangle \mathtt{+} \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{name} \rangle \\
 & | & \mathit{id}\ \mathtt{(}\ \langle \mathit{expr} \rangle \star\ \mathtt{)} \\
\mathit{ty} & ::= & \mathtt{nat} \\
 & | & \mathtt{null}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{stmt} & ::= & \langle \mathit{name} \rangle\ \mathtt{=}\ \langle \mathit{expr} \rangle \\
 & | & \mathtt{\{}\ \langle \mathit{stmt} \rangle\ \mathtt{\}} \\
 & | & \mathtt{if}\ \langle \mathit{expr} \rangle\ \langle \mathit{stmt} \rangle\ \mathtt{else}\ \langle \mathit{stmt} \rangle \\
 & | & \mathtt{while}\ \langle \mathit{expr} \rangle\ \langle \mathit{stmt} \rangle \\
 & | & \mathtt{skip} \\
 & | & \mathtt{return}\ \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{stmt} \rangle\mathtt{;}\ \langle \mathit{stmt} \rangle \\
\mathit{decl} & ::= & \mathtt{proc}\ \mathit{id}\ \mathtt{(}\ (\mathit{id} : \langle \mathit{ty} \rangle) \star\ \mathtt{)}\ \langle \mathit{stmt} \rangle \\
\mathit{prog} & ::= & \langle \mathit{decl} \rangle\ \langle \mathit{prog} \rangle \\
 & | & \langle \mathit{stmt} \rangle
\end{array}
$$

The ATL Syntax

$$
\begin{array}{lcl}
\mathit{name} & ::= & \mathit{id} \\
 & | & \mathit{name}\ .\ \mathit{id} \\
 & & \\
\mathit{val} & ::= & \mathit{num} \\
 & | & \mathsf{null} \\
 & & \\
\mathit{expr} & ::= & \langle \mathit{val} \rangle \\
 & | & \langle \mathit{expr} \rangle + \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{name} \rangle \\
 & | & \mathit{id}\ (\ \langle \mathit{expr} \rangle \star\ ) \\
 & & \\
 & | & \mathsf{new}() \\
 & & \\
\mathit{ty} & ::= & \mathsf{nat} \\
 & | & \mathsf{null}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{stmt} & ::= & \langle \mathit{name} \rangle\ \mathtt{=}\ \langle \mathit{expr} \rangle \\
 & | & \{\ \langle \mathit{stmt} \rangle\ \} \\
 & | & \mathsf{if}\ \langle \mathit{expr} \rangle\ \langle \mathit{stmt} \rangle\ \mathsf{else}\ \langle \mathit{stmt} \rangle \\
 & | & \mathsf{while}\ \langle \mathit{expr} \rangle\ \langle \mathit{stmt} \rangle \\
 & | & \mathsf{skip} \\
 & | & \mathsf{return}\ \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{stmt} \rangle;\ \langle \mathit{stmt} \rangle \\
 & & \\
\mathit{decl} & ::= & \mathsf{proc}\ \mathit{id}\ (\ (\mathit{id} : \langle \mathit{ty} \rangle) \star\ )\ \langle \mathit{stmt} \rangle \\
 & & \\
\mathit{prog} & ::= & \langle \mathit{decl} \rangle\ \langle \mathit{prog} \rangle \\
 & | & \langle \mathit{stmt} \rangle
\end{array}
$$

Modelling the Heap (1/2)

- Heap-allocated objects allow *sharing*:

```
a = new();
b = a;
a.v = 1;
return b.v
```

- We cannot express this with just environments
- Need indirection via *references*
  - Represent *concrete heap graph* (cf. points-to analysis)
  - Unbounded set REF contains all references
  - **new**() produces fresh references
  - Heap-allocated objects accessed indirectly via references

Modelling the Heap (2/2)

- Indirection via a *heap* H that points to *heap objects* $\textsc{HObj}$:

$$H : \textsc{Ref} \nrightarrow \textsc{HObj}$$

- Heap objects $\textsc{HObj}$ are *records*:

$$[\mathsf{id}_1 \mapsto v_1, \ldots, \mathsf{id}_k \mapsto v_k]$$

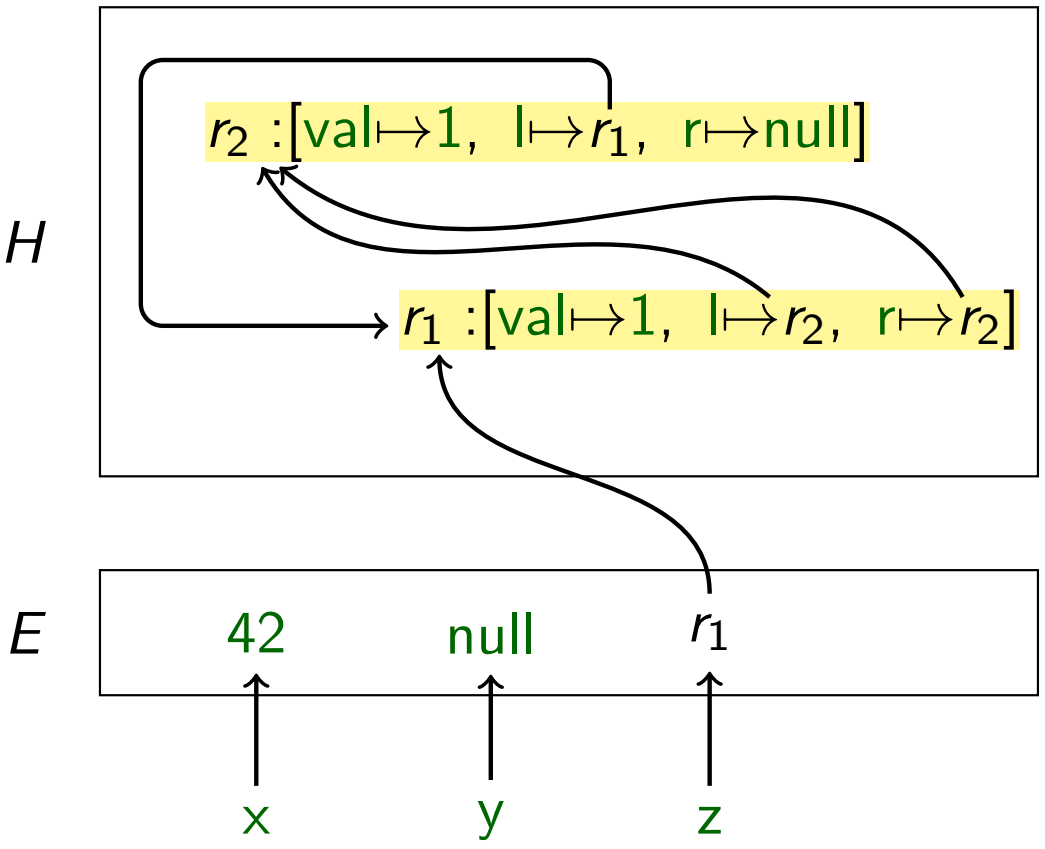
- We write them with square brackets but map semantics (same as environments and heaps)

Example:

```
a = new();
a.self = a;
a.num = 17;
```

$$H = \{r_1 \mapsto [\mathsf{self} \mapsto r_1, \mathsf{num} \mapsto 17]\}$$

References and Heap Objects



- $\textsc{Ref}$: Heap references
- *val*: primitive valus (1, 2, 3, null)
- $\mathbb{V} = \mathit{val} \uplus \textsc{Ref}$
- $\textsc{HObj} : \mathit{id} \nrightarrow \mathbb{V}$: Heap objects

$\textsc{HObj}$

$\uparrow$

$\textsc{Ref}$

$\mathbb{V}$

$\uparrow$

id

Updating Existing Evaluation Rules

- Heap H now part of program state

Call may update heap

$$\frac{E(id) = \langle s, [id_1, \ldots, id_k] \rangle \qquad \begin{array}{c} \langle H_0, E, e_1 \rangle \Downarrow v_1 \cdots \langle H_0, E, e_k \rangle \Downarrow v_1 \\ \langle H_0, E_p \cup \{id_1 \mapsto v_1, \cdots, id_k \mapsto v_k\}, s \rangle \Downarrow \langle H', v \rangle v_1 \end{array}}{\langle H_0, E, id(e_1 \ldots e_k) \rangle \Downarrow v \langle H', v \rangle}$$

Updating Existing Evaluation Rules

- Heap H now part of program state
- Must be 'threaded through' $\Downarrow$ and $\Downarrow_s$

Mustn't forget previous heap updates!
Need order:

- Left-to-right (Java, ...)
- Undefined (C, ...)

$$\frac{E(id) = \langle s, [id_1, \ldots, id_k]\rangle \qquad \begin{array}{c}\langle H_0, E, e_1\rangle \Downarrow \langle H_1, v_1\rangle \cdots \langle H_{k-1}, E, e_k\rangle \Downarrow \langle H_k, v_k\rangle \\ \langle H_k, E_p \cup \{id_1 \mapsto v_1, \cdots, id_k \mapsto v_k\}, s\rangle \Downarrow \langle H', v\rangle v_1\end{array}}{\langle H_0, E, id(e_1 \ldots e_k)\rangle \Downarrow v\langle H', v\rangle}$$

Storing and Loading

- $\Downarrow$ and $\Downarrow_s$ may now update heap

$$\frac{r \ \ fresh}{\langle H, E, \textbf{new}() \rangle \Downarrow \langle H \cup \{r \mapsto []\}, r \rangle} \ (new)$$

$$\frac{\langle H, E, e_1 \rangle \Downarrow \langle H, r \rangle \quad \langle H, E, e_2 \rangle \Downarrow \langle H', v \rangle \quad H'(r) = R \quad R' = [id \mapsto v]R}{\langle H, E, e_1.id = e_2 \rangle \Downarrow_s \langle [r \mapsto R']H', E \rangle} \ (store)$$

$$\frac{\langle H, E, e \rangle \Downarrow \langle H, r \rangle \quad H(r) = [id \mapsto v, \ldots]}{\langle H, E, e.id \rangle \Downarrow \langle H, v \rangle} \ (load)$$

$$\frac{\langle H, E, e \rangle \Downarrow \langle H, v \rangle}{\langle H, E, id = e \rangle \Downarrow_s \langle H, [id \mapsto v]E \rangle} \ (assign)$$

Summary

- Modelling heap interaction requires us to model heaps

$$H : \textsc{Ref} \nrightarrow \textsc{HObj}$$

  - $\textsc{Ref}$: Set of heap references ('pointers')
  - $\mathbb{V}$: Extended set of (irreducible) values that consists of:
    - *val*: Primitive values
    - $\textsc{Ref}$: Heap references
  - $\textsc{HObj}$: Heap objects, where $\textsc{HObj} : \mathit{id} \nrightarrow \mathbb{V}$
- Evaluation rules suitably access / update heap

Heap vs. Type Checking

$$\frac{\langle H, E, e\rangle \Downarrow \langle H, r\rangle \quad H(r) = [id \mapsto v, \ldots]}{\langle H, E, e.id\rangle \Downarrow \langle H, v\rangle} \ (load)$$

- Program can get stuck in new ways:
  1. Dereferencing non-REF (number, null)
  2. Dereferencing through missing field label
  3. Moreover, heap may hide type errors:

```
a = new();
b = a;
b.v = 2;
a.v = null;
return b.v + 1
```

- Due to heap indirection, it is nontrivial to detect the above precisely
- We will try an easier approach with user annotations

The ATL Syntax

$$
\begin{array}{lcl}
\mathit{name} & ::= & \mathit{id} \\
 & | & \mathit{name}\ .\ \mathit{id} \\
 & & \\
\mathit{val} & ::= & \mathit{num} \\
 & | & \mathsf{null} \\
 & & \\
\mathit{expr} & ::= & \langle \mathit{val} \rangle \\
 & | & \langle \mathit{expr} \rangle + \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{name} \rangle \\
 & | & \mathit{id}\ (\ \langle \mathit{expr} \rangle \star\ ) \\
 & & \\
 & | & \mathsf{new}() \\
 & & \\
\mathit{ty} & ::= & \mathsf{nat} \\
 & | & \mathsf{null}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{stmt} & ::= & \langle \mathit{name} \rangle\ \mathtt{=}\ \langle \mathit{expr} \rangle \\
 & | & \{\ \langle \mathit{stmt} \rangle\ \} \\
 & | & \mathsf{if}\ \langle \mathit{expr} \rangle\ \langle \mathit{stmt} \rangle\ \mathsf{else}\ \langle \mathit{stmt} \rangle \\
 & | & \mathsf{while}\ \langle \mathit{expr} \rangle\ \langle \mathit{stmt} \rangle \\
 & | & \mathsf{skip} \\
 & | & \mathsf{return}\ \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{stmt} \rangle;\ \langle \mathit{stmt} \rangle \\
 & & \\
\mathit{decl} & ::= & \mathsf{proc}\ \mathit{id}\ (\ (\mathit{id} : \langle \mathit{ty} \rangle) \star\ )\ \langle \mathit{stmt} \rangle \\
 & & \\
\mathit{prog} & ::= & \langle \mathit{decl} \rangle\ \langle \mathit{prog} \rangle \\
 & | & \langle \mathit{stmt} \rangle
\end{array}
$$

The ATL Syntax

$$
\begin{array}{lcl}
\mathit{name} & ::= & \mathit{id} \\
 & | & \mathit{name}\ .\ \mathit{id} \\
 & & \\
\mathit{val} & ::= & \mathit{num} \\
 & | & \mathsf{null} \\
 & & \\
\mathit{expr} & ::= & \langle \mathit{val} \rangle \\
 & | & \langle \mathit{expr} \rangle + \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{name} \rangle \\
 & | & \mathit{id}\ (\ \langle \mathit{expr} \rangle \star\ ) \\
 & & \\
 & | & \mathsf{new}(\mathit{id}) \\
 & & \\
\mathit{ty} & ::= & \mathsf{nat} \\
 & | & \mathit{id}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{stmt} & ::= & \langle \mathit{name} \rangle\ \mathtt{=}\ \langle \mathit{expr} \rangle \\
 & | & \{\ \langle \mathit{stmt} \rangle\ \} \\
 & | & \mathsf{if}\ \langle \mathit{expr} \rangle\ \langle \mathit{stmt} \rangle\ \mathsf{else}\ \langle \mathit{stmt} \rangle \\
 & | & \mathsf{while}\ \langle \mathit{expr} \rangle\ \langle \mathit{stmt} \rangle \\
 & | & \mathsf{skip} \\
 & | & \mathsf{return}\ \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{stmt} \rangle;\ \langle \mathit{stmt} \rangle \\
 & & \\
\mathit{decl} & ::= & \mathsf{proc}\ \mathit{id}\ (\ (\mathit{id} : \langle \mathit{ty} \rangle) \star\ )\ \langle \mathit{stmt} \rangle \\
 & | & \mathsf{datatype}\ \mathit{id}\ (\ (\mathit{id} : \langle \mathit{ty} \rangle) \star\ ) \\
 & & \\
\mathit{prog} & ::= & \langle \mathit{decl} \rangle\ \langle \mathit{prog} \rangle \\
 & | & \langle \mathit{stmt} \rangle
\end{array}
$$

Updated Operational Semantics

- $\mathcal{D}_0 : id \nrightarrow (id \nrightarrow \mathbb{V})$
  Fields and default values for datatypes (obtained from type), e.g. 0

$$\frac{r \ \mathit{fresh} \quad \mathcal{D}_0(id) = [id_1 = v_1, \ldots, id_k = v_k]}{\langle H, E, \textbf{new}\ (id)\rangle \Downarrow \langle H \cup \{r \mapsto [id_1 \mapsto v_1, \ldots, id_k \mapsto v_k]\}, r\rangle} \ (\mathit{new})$$

$$\frac{\langle H, E, e\rangle \Downarrow \langle H, r\rangle \quad H(r) = [id \mapsto v, \ldots]}{\langle H, E, e.id\rangle \Downarrow \langle H, v\rangle} \ (\mathit{load})$$

$$\frac{\langle H, E, e_1\rangle \Downarrow \langle H', r\rangle \quad \langle H', E, e_2\rangle \Downarrow \langle H'', v\rangle \quad H''(r) = R \quad R' = [id \mapsto v]R}{\langle H, E, e_1.id = e_2\rangle \Downarrow_s \langle [r \mapsto R']H'', E\rangle} \ (\mathit{store})$$

Updated Operational Semantics

- $\mathcal{D}_T : id \nrightarrow (id \nrightarrow \mathbb{T}_{atl})$
  Field types for datatypes (obtained from type)
- Introduce T_{id} to represent id as type
  - Include $T_{id} \in \mathbb{T}_{atl}$ for all $id \in id$

$$\frac{id \in \mathrm{dom}\ \mathcal{D}_T}{\Gamma \vdash \textbf{new}(id) : T_{id}}\ (t\text{-}new)$$

$$\frac{\Gamma \vdash e : T_{id} \quad \mathcal{D}_T(id) = [id_1 : \tau, \ldots]}{\Gamma \vdash e.id_1 : \tau}\ (t\text{-}load)$$

$$\frac{\Gamma \vdash e_1 : T_{id} \quad \mathcal{D}_T(id) = [id_1 : \tau, \ldots] \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash_s e_1.id_1 = e_2 : \textsc{Unit}}\ (t\text{-}store)$$

Review

- Typing Environments
- Type Inference
- Operational Semantics with Stores

To be continued...

- More types and semantics