$\mathcal{L}_1$

a formal, minimal, imperative, class based object oriented language *without* inheritance

$\mathcal{L}_1$-a formal, minimal, imperative, class based object oriented language without inheritance

$\mathcal{L}_1$ is a minimal object oriented language, in the style of commercially available languages. It supports

- classes which describe the behaviour of objects ($\mathcal{L}_1$ is *class based*),
- fields,
- methods - no overloading,
- assignment and aliasing,
- expressions as the bodies of methods ($\mathcal{L}_1$ is *expression oriented*, as opposed to *statement oriented*).

As for any programming language, the formal description of $\mathcal{L}_1$ consists of

- the syntax,
- the operational semantics,
- the type system.

Type soundness is demonstrated through a theorem.

$\mathcal{L}_1$ will serve as a platform for the introduction of further features. We study $\mathcal{L}_1$ in some detail in order to learn (or remind ourselves of) the formal treatment of programming languages.

The structure of $\mathcal{L}_1$, and the syntax of $\mathcal{L}_1$ expressions

$$
\mathit{Progr} \;=\; \mathit{ClassId} \longrightarrow \Big(\; (\mathit{FieldId} \longrightarrow \mathit{type}) \;\times\; (\mathit{MethId} \longrightarrow \mathit{meth}) \;\Big)
$$

where

$$
\begin{array}{lcl}
\mathit{meth} & ::= & \mathit{type}\ m\ \ \textbf{(}\ \mathit{type}\ \mathsf{x}\ \textbf{)}\ \ \textbf{\{}\ e\ \textbf{\}} \\
\mathit{type} & ::= & \mathsf{bool} \ \mid\ c \\
e & ::= & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e\ \mid \\
 & & e\,.f \ \mid\ e\,.f := e \ \mid\ e\,.m\,\textbf{(}\,e\,\textbf{)} \ \mid\ \mathsf{new}\ c \ \mid \\
 & & \mathsf{x} \ \mid\ \mathsf{this} \ \mid\ \mathsf{true} \ \mid\ \mathsf{false} \ \mid\ \mathsf{null}\ \textbf{.}
\end{array}
$$

with the identifier conventions

c	$\in$	*ClassId*	for class identifiers
f	$\in$	*FieldId*	for field identifiers
m	$\in$	*MethId*	for method identifiers

For the sake of simplicity, all methods have exactly one parameter, called x, and there is no syntax for sequences of expressions.

These simplifications allow a more succinct presentation of the operational semantics and the type system; the corresponding restrictions are not essential – why?

Note:

$A \times B$ denotes the set of tuples, so that

$$(a, b) \in A \times B \quad \text{if and only if} \quad a \in A, b \in B.$$

$A \longrightarrow B$ denotes the set of *partial* functions, so that

for $f \in A \longrightarrow B$, and $a \in A$, either $f(a) \in B$ or $f(a)$ is undefined.

An Example in $\mathcal{L}_1$

The following example

```
class Book{
        bool good;
        Person owner;
        bool readBy( Person  x) {  this.good:= true  }
}

class Person{
        Book  like;
        Book meet( Person  x) {  this.like:= x.like  }
}
```

is represented through

$$
\begin{array}{llll}
P_{BP} = & \mathsf{Book} \mapsto & (& (\ \mathsf{good} \mapsto \mathsf{bool},\ \mathsf{owner} \mapsto \mathsf{Person}\ ), \\
 & & & \mathsf{readBy} \mapsto \mathsf{bool\ readBy}\boldsymbol{(}\ \mathsf{Person}\ \ \mathsf{x}\boldsymbol{)}\ \boldsymbol{\{}\ \ \mathsf{this.good}{:=}\,\mathsf{true}\ \ \boldsymbol{\}}\quad)\ , \\
 & \mathsf{Person} \mapsto & (& \mathsf{like} \mapsto \mathsf{Book}\ , \\
 & & & \mathsf{meet} \mapsto \mathsf{Book\ meet}\boldsymbol{(}\ \mathsf{Person}\ \ \mathsf{x}\boldsymbol{)}\ \boldsymbol{\{}\ \ \mathsf{this.like}{:=}\,\mathsf{x.like}\ \ \boldsymbol{\}}\quad)\ .
\end{array}
$$

Questions

What are the values of

$P_{BP}(\mathsf{Book})$,

$P_{BP}(\mathsf{Book}) \downarrow_1 (\mathsf{good})$,

$P_{BP}(\mathsf{Book}) \downarrow_2 (\mathsf{readBy})$,

$P_{BP}(\mathsf{Book}) \downarrow_2 (\mathsf{meet})$?

$P_{BP}(\mathsf{readBy})$,

$P_{BP}(\mathsf{Book}) \downarrow_1 (\mathsf{like})$?

The following program:

```
class B {
        D m( E x) { true } ;
        C f
}
```

would be represented as:

$\mathsf{P}_{\mathsf{ok}} \equiv$

But the following program

```
class A {
        bool f;
        A f;
        bool m( bool x) { true }
}
```

even though syntactically correct in Java, *cannot* be represented in $\mathcal{L}_1$. Why?

Does the fact that class A cannot be represented in $\mathcal{L}_1$ reduce the applicability of $\mathcal{L}_1$?

Field and Method lookup functions

We define the field and method lookup functions:

$$\mathcal{F}(\mathsf{P}, \mathsf{c}, \mathsf{f}) \quad = \quad \mathsf{P}(\mathsf{c}) \downarrow_1 (\mathsf{f})$$

$$\mathcal{F}s(\mathsf{P}, \mathsf{c}) \quad = \quad \{\, \mathsf{f} \mid \mathcal{F}(\mathsf{P}, \mathsf{c}, \mathsf{f}) \text{is defined} \}$$

$$\mathcal{M}(\mathsf{P}, \mathsf{c}, \mathsf{m}) \quad = \quad \mathsf{P}(\mathsf{c}) \downarrow_2 (\mathsf{m})$$

For example,

$\mathcal{F}(\mathsf{P}_{\mathsf{BP}}, \mathsf{Book}, \mathsf{good})$ = bool,

$\mathcal{F}s(\mathsf{P}_{\mathsf{BP}}, \mathsf{Book})$ = { good, owner }.

$\mathcal{M}(\mathsf{P}_{\mathsf{BP}}, \mathsf{Book}, \mathsf{readBy})$ = bool readBy**(** Person x**)** **{** this. good:= true **}** .

What is the value of $\mathcal{F}(\mathsf{P}_{\mathsf{BP}}, \mathsf{Book}, \mathsf{like})$, and what is $\mathcal{M}(\mathsf{P}_{\mathsf{BP}}, \mathsf{Book}, \mathsf{meet})$?

The Operational Semantics of $\mathcal{L}_1$

Operational semantics describe execution of expressions.

Execution takes place in the context of a program (*prog*). It *rewrites* tuples of expressions (*expr*), stack frames (*stack frame*), and heaps (*heap*) into pairs of results (*res*) and heaps.
Thus, the signature of the rewriting relation $\rightsquigarrow$ is:

$$\rightsquigarrow \quad : \quad \mathit{progr} \longrightarrow \mathit{expr} \times \mathit{stack\ frame} \times \mathit{heap} \longrightarrow \mathit{res} \times \mathit{heap}$$

The stack frame is a tuple; it gives an address for this, and a value for x.
The heap maps addresses to objects.
Objects may contain fields whose value is an address - thus we can represent aliasing.

The result of an execution is either a value, or an exception.

$$
\begin{array}{lcl}
\textit{stack frame} & = & \textit{addr} \times \textit{val} \\
\textit{heap} & = & \textit{addr} \longrightarrow \textit{object} \\
\textit{val} & = & \{\ \mathsf{true}\ ,\ \mathsf{false}\ ,\ \mathsf{null}\} \cup \textit{addr} \\
\textit{object} & = & \textit{ClassId} \times (\ \textit{FieldId} \longrightarrow \textit{val}\ ) \\
\textit{addr} & = & \{\ \iota_i\ \mid\ \text{i a natural number}\ \} \\
\textit{dev} & = & \{\mathsf{nullPntrExc}, \mathsf{stuckErr}\} \\
\textit{res} & = & \textit{dev} \cup \textit{val}
\end{array}
$$

Values are the source language values (*i.e.* true , false and null), or addresses.
Addresses may point to objects, but *not* to other addresses, or primitive values. Thus, in $\mathcal{L}_1$, as in Java, pointers are implicit – no pointers to pointers.

Deviations (*dev*) indicate abnormal termination due to null-pointer de-referencing, or an evaluation being stuck. As we shall show, well-typed expressions are never stuck, although they may throw null pointer exceptions.

Example stack frame and heap

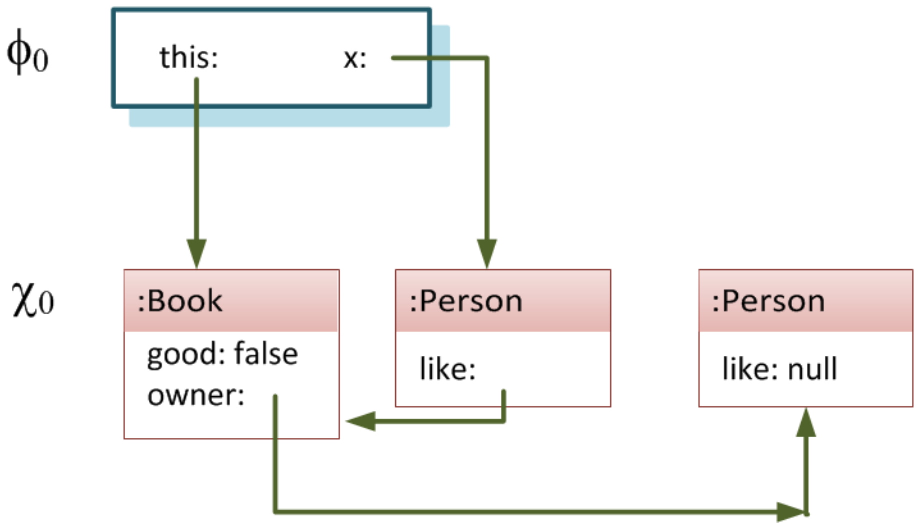
The following stack frame, ϕ_0, and heap, χ_0, correspond to some execution of $\mathsf{P_{BP}}$:

$$\phi_0 \quad = \quad (\iota_3, \iota_4)$$

$$\begin{array}{lcl}
\chi_0(\iota_3) & = & (\ \mathsf{Book},\ (\mathsf{good} \mapsto \mathsf{false}\ ,\ \ \mathsf{owner} \mapsto \iota_5)\ ) \\
\chi_0(\iota_4) & = & (\ \mathsf{Person},\ (\mathsf{like} \mapsto \iota_3)\ ) \\
\chi_0(\iota_5) & = & (\ \mathsf{Person},\ (\mathsf{like} \mapsto \mathsf{null})\ )
\end{array}$$

Visual representation of χ_0 and ϕ_0

$$
\begin{aligned}
\phi_0 &= (\iota_3, \iota_4) \\
\chi_0(\iota_3) &= (\ \mathsf{Book},\ (\mathsf{good} \mapsto \mathsf{false}\ ,\ \ \mathsf{owner} \mapsto \iota_5)\ ) \\
\chi_0(\iota_4) &= (\ \mathsf{Person},\ (\mathsf{like} \mapsto \iota_3)\ ) \\
\chi_0(\iota_5) &= (\ \mathsf{Person},\ (\mathsf{like} \mapsto \mathsf{null})\ )
\end{aligned}
$$

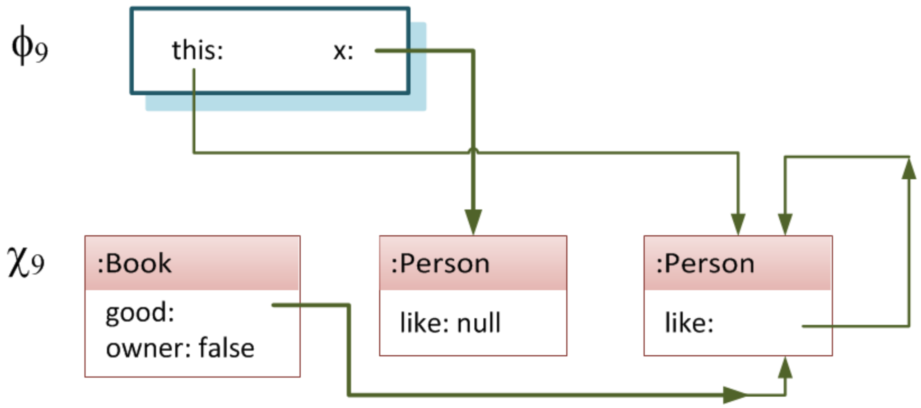


According to our definition, the tuple ($\mathtt{true}$,ι_3) is *not* a stack frame, but the tuple (ι_3,$\mathtt{true}$) is.

Also, a mapping χ^{bad}, where $\chi^{bad}(\iota_3)$=$\mathtt{true}$, is *not* a heap.

Exercise

Write out formally the heap χ_9 and frame ϕ_9 which correspond to the following diagram.



Operations on the heap

We would expect the operational semantics to give:

$$\mathsf{x.like.good}, \phi_0, \chi_0 \rightsquigarrow_{\mathsf{P_{BP}}} \mathsf{false}, \chi_0$$

In order to define the operational semantics we need to define operations on objects and heaps.

Field access, objects and heap update functions

For object o, heap χ, value v, address ι, identifier f, define:

- $\mathsf{o}(\mathsf{f}) = \mathsf{o} \downarrow_2 (\mathsf{f})$

- $\mathsf{o}[\mathsf{f}{\mapsto}\mathsf{v}]$ gives a new object, so that
  $\mathsf{o}[\mathsf{f}{\mapsto}\mathsf{v}] \downarrow_1 = \mathsf{o} \downarrow_1$, and
  $\mathsf{o}[\mathsf{f}{\mapsto}\mathsf{v}](\mathsf{f}) = \mathsf{v}$, and $\mathsf{o}[\mathsf{f}{\mapsto}\mathsf{v}](\mathsf{f}') = \mathsf{o}(\mathsf{f}')$ if $\mathsf{f}' \neq \mathsf{f}$,

- $\chi[\iota{\mapsto}\mathsf{o}]$ gives a new heap so that
  $\chi[\iota{\mapsto}\mathsf{o}](\iota) = \mathsf{o}$, and $\chi[\iota{\mapsto}\mathsf{o}](\iota') = \chi(\iota')$ if $\iota' \neq \iota$.

Field access, objects and heap update functions - 2

We also define the following shorthands

- $\chi(\iota, \mathsf{f}) \;=\; \chi(\iota)\downarrow_2(\mathsf{f})$
- $\chi[\iota, \mathsf{f}{\mapsto}\mathsf{v}] \;=\; \chi[\iota{\mapsto}(\chi(\iota)[\mathsf{f}{\mapsto}\mathsf{v}])]$.
- $\phi(\mathsf{this}) = \phi\downarrow_1$, and $\phi(\mathsf{x}) = \phi\downarrow_2$

$\chi(\iota, \mathsf{f})$, and $\chi[\iota, \mathsf{f}{\mapsto}\mathsf{v}]$ describe field access and the update of an object in the heap. And $\phi(\mathsf{this})$ and $\phi(\mathsf{x})$ give the current receiver and argument. **Note:** there is no operation which changes the class of an object.

Remember our heap χ_0 from earlier, where

$$\begin{array}{lcl} \chi_0(\iota_3) & = & (\ \mathsf{Book},\ (\mathsf{good} \mapsto \mathsf{false}\ ,\ \ \mathsf{owner} \mapsto \iota_5)\ ) \\ \chi_0(\iota_4) & = & (\ \mathsf{Person},\ (\mathsf{like} \mapsto \iota_3)\ ) \\ \chi_0(\iota_5) & = & (\ \mathsf{Person},\ (\mathsf{like} \mapsto \mathsf{null})\ ) \end{array}$$

Then, application of the operations defined earlier give following results:

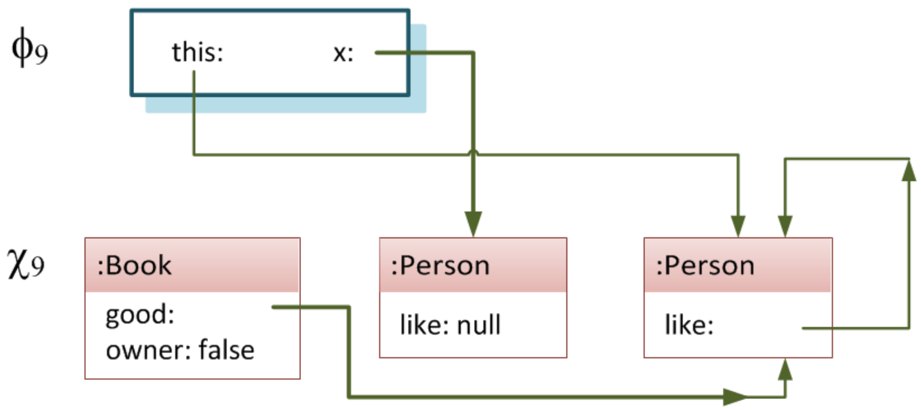
- $\chi_0(\iota_3, \mathsf{good}) = \mathsf{false}$,
- $\chi_0(\iota_3, \mathsf{like})$ is undefined
- $(\ \mathsf{Book},\ (\mathsf{good} \mapsto \mathsf{false}\ ,\ \ \mathsf{owner} \mapsto \iota_5)\ )[\mathsf{good} \mapsto \mathsf{true}\ ] =$
  $(\ \mathsf{Book},\ (\mathsf{good} \mapsto \mathsf{true}\ ,\ \ \mathsf{owner} \mapsto \iota_5)\ )$
- $\chi_0[\iota_3, \mathsf{good} \mapsto \mathsf{true}\ ] = \chi_1$, where

$$\begin{array}{lcl} \chi_1(\iota_3) & = & (\ \mathsf{Book},\ (\mathsf{good} \mapsto \mathsf{true}\ ,\ \ \mathsf{owner} \mapsto \iota_5)\ ) \\ \chi_1(\iota_4) & = & (\ \mathsf{Person},\ (\mathsf{like} \mapsto \iota_3)\ ) \\ \chi_1(\iota_5) & = & (\ \mathsf{Person},\ (\mathsf{like} \mapsto \mathsf{null})\ ) \end{array}$$

Exercise -1

The diagram below describes heap χ_9 and frame ϕ_9. Modify the heap in the diagram so that it describes

$$\chi_{10} = \chi_9[\phi_9(x), \mathsf{good} \mapsto \phi_9(\mathsf{this})],$$



Write some expression, whose execution in the context of ϕ_9, χ_9 would modify the heap so that it becomes χ_{10}.

Exercise -2

How would we express the requirement that all persons should own any books they like?

Inference Rules

We will describe the type system and the operational semantics in terms of *inference rules*. Inference rules have a name, a (possibly empty) set of premisses, and one conclusion:

$$\frac{\begin{array}{l} premiss_1 \\ \ldots \\ premiss_n \end{array}}{conclusion}\ \textsf{rule\_name}$$

Which can be read as "if you can establish $premiss_1$, ... $premiss_n$, then you can establish $conclucion$".

Such inference rules can be used to give an *axiomatic definition* for sets, or relations. For example, the following gives and axiomatic definition of the set of natural numbers, and the function add:

$$\frac{}{\text{Z is a natural number}}\ \text{zero} \qquad \frac{\text{n is a natural number}}{\text{S(n) is a natural number}}\ \text{succ}$$

$$\frac{}{\text{add(Z,n)=n}}\ \text{addZero} \qquad \frac{\text{add(n,m) = k}}{\text{add(S(n),m)= S(k)}}\ \text{addSucc}$$

We can *prove* that

$\forall$n, m : n, m are natural numbers $\Longrightarrow$ add(n,m) $=$ add(m,n).

Note: we do *not* write the above as an inference rule! The above is a theorem – not an axiom.

Exercise-1

Write out the derivation which demonstrates that $S(S(Z))$ is a natural number.

Exercise-2

Write out the derivation which describes the calculation of $add(S(S(Z)), S(Z))$.

Exercise-3

Write inference rules which describe $fib(n)$, where
$fib(Z) = S(Z)$, and
$fib(S(Z)) = S(Z)$, and
$fib(S(S(n))) = add(fib(S(n)), fib(n))$

Exercise-4

Write out the calculation of $fib(S(S(Z))$, using the inference rules from Exercise-3.

Operational Semantics - Rules

val

$$\frac{\mathsf{v} \in val}{\mathsf{v},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{v},\chi}$$

this

$$\frac{}{\mathsf{this},(\iota,\mathsf{v}),\chi \leadsto_{\mathsf{P}} \iota,\chi}$$

par

$$\frac{}{\mathsf{x},(\iota,\mathsf{v}),\chi \ \leadsto_{\mathsf{P}} \ \mathsf{v},\chi}$$

cond_1

$$\frac{\mathsf{e},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{true},\chi'' \qquad \mathsf{e}_1,\phi,\chi'' \ \leadsto_{\mathsf{P}} \ \mathsf{v},\chi'}{\mathsf{if\ e\ then\ e}_1\ \mathsf{else\ e}_2,\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{v},\chi'}$$

cond_2

$$\frac{\mathsf{e},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{false},\chi'' \qquad \mathsf{e}_2,\phi,\chi'' \ \leadsto_{\mathsf{P}} \ \mathsf{v},\chi'}{\mathsf{if\ e\ then\ e}_1\ \mathsf{else\ e}_2,\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{v},\chi'}$$

fld

$$\frac{\mathsf{e},\phi,\chi \rightsquigarrow_{\mathsf{P}} \iota,\chi'}{\mathsf{e.f},\phi,\chi \rightsquigarrow_{\mathsf{P}} \chi'(\iota,\mathsf{f}),\chi'}$$

fldAss

$$\frac{\begin{array}{l}\mathsf{e},\phi,\chi \rightsquigarrow_{\mathsf{P}} \iota,\chi'' \\ \mathsf{e}',\phi,\chi'' \rightsquigarrow_{\mathsf{P}} \mathsf{v},\chi''' \\ \chi' = \chi'''[\iota,\mathsf{f}\mapsto\mathsf{v}]\end{array}}{\mathsf{e.f}{:=}\mathsf{e}',\phi,\chi \rightsquigarrow_{\mathsf{P}} \mathsf{v},\chi'}$$

new

$$\frac{\begin{array}{l}\mathcal{F}s(\mathsf{P},\mathsf{c}) = \{\mathsf{f}_1,...,\mathsf{f}_r\} \\ \forall \mathsf{l} \in 1,...,\mathsf{r}: \quad \mathsf{v}_l \text{ initial for } \mathcal{F}(\mathsf{P},\mathsf{c},\mathsf{f}_\mathsf{l}) \\ \iota \text{ is new in } \chi \\ \chi' = \chi[\iota\mapsto(\mathsf{c}, (\mathsf{f}_1\mapsto\mathsf{v}_1,...,\mathsf{f}_r\mapsto\mathsf{v}_r))]\end{array}}{\mathsf{new}\ \mathsf{c},\phi,\chi \rightsquigarrow_{\mathsf{P}} \iota,\chi'}$$

null

$$\frac{\mathsf{e},\phi,\chi \rightsquigarrow_{\mathsf{P}} \mathsf{null},\chi'}{\begin{array}{l}\mathsf{e.f}{:=}\mathsf{e}',\phi,\chi \rightsquigarrow_{\mathsf{P}} \mathsf{nullPntrExc},\chi' \\ \mathsf{e.f},\phi,\chi \rightsquigarrow_{\mathsf{P}} \mathsf{nullPntrExc},\chi' \\ \mathsf{e.m}(\mathsf{e}_1),\phi,\chi \rightsquigarrow_{\mathsf{P}} \mathsf{nullPntrExc},\chi'\end{array}}$$

The initial value for bool is false , the initial value for any class c is null.

methCall

$$\frac{\begin{array}{l} e_0, \phi, \chi \leadsto_P \iota, \chi_0 \\ e_1, \phi, \chi_0 \leadsto_P v_1, \chi_1 \\ \chi_1(\iota)\downarrow_1 = c \\ \mathcal{M}(P, c, m) = t\ m\textbf{(}\ t_1\ x\textbf{)}\ \ \textbf{\{}\ e\ \textbf{\}} \\ e, (\iota, v_1), \chi_1 \leadsto_P v, \chi' \end{array}}{e_0.m(e_1), \phi, \chi \leadsto_P v, \chi'}$$

- Notes:
  The operational semantics is not affected by static types.
  There is no rule that changes a complete object; rules can only update fields.

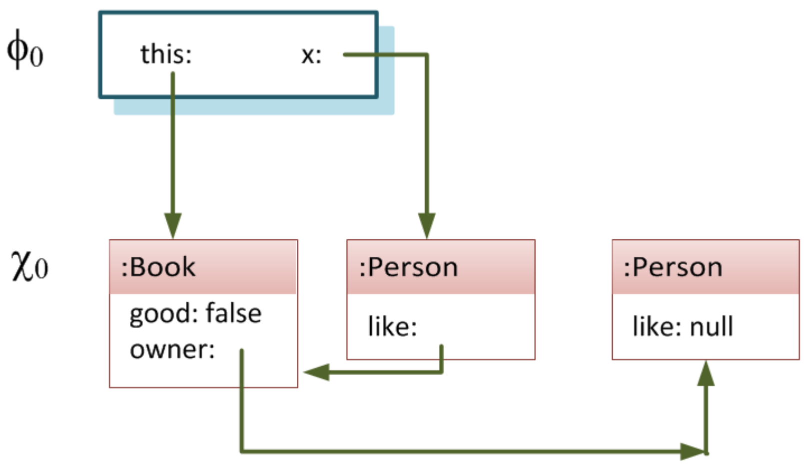
- Questions:
  Why do we need rule val?
  What determines the order of execution?

Exercise-5:

Write out the derivation for executing $\mathsf{this.like{:=}x.like}, \phi_3, \chi_0$, where $\phi_3 = (\iota_5, \iota_4)$

Exercise-6:

Write out the derivation for executing $\texttt{this.owner.meet(x)}, \phi_0, \chi_0$.



Questions

- Does the following proposition hold?

$\forall$ P, χ, e, ϕ: $\quad \exists\, \chi'$, r with e,ϕ,$\chi \leadsto_{\mathsf{P}} r$,$\chi'$

- What about program start, i.e., the `main` method?

- What happens to unreachable objects?

Questions - 2

What about expressions like:

$$\text{this.meet}(\text{ x}) ,\phi_0,\chi_0 \rightsquigarrow_{\mathsf{P}_{\mathsf{BP}}} \quad ????$$

or, like

$$\text{this.owner.like.owner.good},\phi_0,\chi_0 \rightsquigarrow_{\mathsf{P}_{\mathsf{BP}}} \quad ????$$

We need to add operational semantics rules in order to describe "stuck execution" (corresponding to the Smalltalk "object does not understand message" exception), and exception propagation:

Stuck Execution

$$\frac{\begin{array}{l}\mathsf{e},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{v},\chi' \\ \mathsf{v} \neq \mathsf{true} \text{ and } \mathsf{v} \neq \mathsf{false}\end{array}}{\mathsf{if\ e\ then\ e_1\ else\ e_2},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{stuckErr},\chi'}$$

$$\frac{\begin{array}{l}\mathsf{e},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{v},\chi' \\ \chi'(\mathsf{v}) \text{ is undefined}\end{array}}{\begin{array}{l}\mathsf{e.f},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{stuckErr},\chi' \\ \mathsf{e.f{:=}e'},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{stuckErr},\chi'\end{array}}$$

$$\frac{\begin{array}{l}\mathsf{e_0},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{v},\chi_0 \\ \mathsf{v} \neq \mathsf{null} \\ \chi_0(\mathsf{v}) \text{ is undefined}\end{array}}{\mathsf{e_0.m(e_1)},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{stuckErr},\chi_0}$$

$$\frac{\begin{array}{l}\mathsf{e_0},\phi,\chi \ \leadsto_{\mathsf{P}} \ \iota,\chi_0 \\ \mathsf{e_1},\phi,\chi_0 \ \leadsto_{\mathsf{P}} \ \mathsf{v_1},\chi_1 \\ \chi_1(\iota)\downarrow_1 = \mathsf{c} \\ \mathcal{M}(\mathsf{P},\mathsf{c},\mathsf{m}) \text{ is undefined}\end{array}}{\mathsf{e_0.m(e_1)},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{stuckErr},\chi_1}$$

Exception Propagation

$$\frac{\begin{array}{l}\mathsf{e},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{dv},\chi' \\ \quad \text{or} \quad (\mathsf{e},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{true}\ ,\chi'' \text{ and } \mathsf{e}_1,\phi,\chi'' \leadsto_{\mathsf{P}} \ \mathsf{dv},\chi') \\ \quad \text{or} \quad (\mathsf{e},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{false}\ ,\chi'' \text{ and } \mathsf{e}_2,\phi,\chi'' \leadsto_{\mathsf{P}} \ \mathsf{dv},\chi')\end{array}}{\mathsf{if}\ \mathsf{e}\ \ \mathsf{then}\ \ \mathsf{e}_1\ \ \mathsf{else}\ \mathsf{e}_2,\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{dv},\chi'}$$

$$\frac{\mathsf{e},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{dv},\chi'}{\begin{array}{l}\mathsf{e.f},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{dv},\chi' \\ \mathsf{e.m(e_1)},\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{dv},\chi' \\ \mathsf{e.f}{:=}\,\mathsf{e}',\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{dv},\chi'\end{array}} \qquad \frac{\begin{array}{l}\mathsf{e},\phi,\chi \ \leadsto_{\mathsf{P}} \ \iota,\chi'' \\ \mathsf{e}',\phi,\chi'' \ \leadsto_{\mathsf{P}} \ \mathsf{dv},\chi'\end{array}}{\mathsf{e.f}\ := \mathsf{e}',\phi,\chi \ \leadsto_{\mathsf{P}} \ \mathsf{dv},\chi'}$$

where $\mathsf{dv} \in dev$

$$\frac{\begin{array}{l} e_0, \phi, \chi \ \leadsto_P \ \iota, \chi_0 \\ \chi_0(\iota) \text{ is defined} \\ e_1, \phi, \chi_0 \ \leadsto_P \ dv, \chi_1 \end{array}}{e_0.m(e_1), \phi, \chi \ \leadsto_P \ dv, \chi_1}$$

$$\frac{\begin{array}{l} e_0, \phi, \chi \ \leadsto_P \ \iota, \chi_0 \\ e_1, \phi_0, \chi \ \leadsto_P \ v_1, \chi_1 \\ \chi_1(\iota) \downarrow_1 = c \\ \mathcal{M}(P, c, m) = t\ m\mathbf{(}\ t_1\ x\mathbf{)}\ \ \mathbf{\{}\ e\ \mathbf{\}} \\ e, (\iota, v), \chi_1 \ \leadsto_P \ dv, \chi' \end{array}}{e_o.m(e_1), \phi, \chi \ \leadsto_P \ dv, \chi'}$$

Thus, we obtain the execution:

$$\mathsf{this.meet(\,x)}\,,\phi_0,\chi_0 \leadsto_{\mathsf{P_{BP}}} \mathsf{stuckErr},\chi_0$$

and, also:

$$\mathsf{this.owner.like.owner.good},\phi_0,\chi_0 \leadsto_{\mathsf{P_{BP}}} \mathsf{nullPntrExc},\chi_0$$

Questions

- Does the following proposition hold?

  $\forall$ P, χ, e, ϕ: $\exists\ \chi'$, r with $\mathsf{e},\phi,\chi \leadsto_{\mathsf{P}} r,\chi'$

Note

- We will often ignore rules for stuck execution, null-pointer exception, or error propagation

Some properties of the operational semantics

Execution of expressions has the following properties

- it preserves the classes of all objects,
- it preserves the existence of any fields in an object.

Lemma For any program P, and any expression e, if

$$\mathsf{e},\phi,\chi \rightsquigarrow_{\mathsf{P}} \mathsf{r}',\chi'$$

then

- $\chi(\iota)$ is defined $\quad\Longrightarrow\quad \chi(\iota)\downarrow_1 = \chi'(\iota)\downarrow_1$
- $\chi(\iota)(\mathsf{f})$ is defined $\quad\Longrightarrow\quad \chi'(\iota)(\mathsf{f})$ is defined

Proof by structural induction over the derivation $\mathsf{e},\phi,\chi \rightsquigarrow_{\mathsf{P}} \mathsf{r}',\chi'$.

Determinism of the operational semantics

Lemma For any program P, expression e, if

$$\mathsf{e},\phi,\chi \leadsto_{\mathsf{P}} \mathsf{r}',\chi' \quad \text{and} \quad \mathsf{e},\phi,\chi \leadsto_{\mathsf{P}} \mathsf{r}'',\chi''$$

then

$$\mathsf{r}' = \mathsf{r}'', \text{ and } \chi' = \chi''$$

up to renaming of addresses.

Proof Idea First define what "equality up to renaming of addresses" means. The rest by structural induction over the derivation $\mathsf{e},\phi,\chi \leadsto_{\mathsf{P}} \mathsf{r}',\chi'$.

Renaming of Addresses - example

Consider execution of $\texttt{new Book.readBy(this)}$:

Possibly:

$$\texttt{new Book.readBy(this)}, \phi_0, \chi_0 \rightsquigarrow_{P_{BP}} \texttt{true}, \chi_1$$

and also:

$$\texttt{new Book.readBy(this)}, \phi_0, \chi_0 \rightsquigarrow_{P_{BP}} \texttt{true}, \chi_2$$

where

$$\chi_1 = \chi_0[\iota_{10} \mapsto (\ \mathsf{Book},\ (\mathsf{good} \mapsto \mathsf{true}\ , \mathsf{owner} \mapsto \mathsf{null})\ )]$$

and

$$\chi_2 = \chi_0[\iota_{20} \mapsto (\ \mathsf{Book},\ (\mathsf{good} \mapsto \mathsf{true}\ , \mathsf{owner} \mapsto \mathsf{null})\ )]$$

Here, χ_1 and χ_2 are "the same up to renaming of addresses".

Renaming of Addresses - definition

Define "equality up to renaming addresses", through *renaming* function: $\alpha : addr \longrightarrow addr$. α is partial, and bijective "up to undefinedness".

- Equality of values up to renaming defined by

$$\frac{}{\iota =_\alpha \alpha(\iota)} \qquad \frac{}{\mathsf{true} =_\alpha \mathsf{true}} \qquad \frac{}{\mathsf{false} =_\alpha \mathsf{false}}$$

- extended onto objects,

$$\frac{\mathsf{v}_i =_\alpha \mathsf{v}'_i \qquad \mathsf{i} \in 1, ..., \mathsf{n}}{(\ \mathsf{c},\ (\mathsf{f}_1 \mapsto \mathsf{v}_1, ...\mathsf{f}_\mathsf{n} \mapsto \mathsf{v}_n)\ ) =_\alpha (\ \mathsf{c},\ (\mathsf{f}_1 \mapsto \mathsf{v}'_1, ...\mathsf{f}_\mathsf{n} \mapsto \mathsf{v}'_n)\ )}$$

- and extended onto stack frames and heaps.

$$\frac{\chi(\iota) =_\alpha \chi'(\alpha(\iota)) \quad \forall \iota}{\chi =_\alpha \chi'} \qquad \frac{\iota =_\alpha \iota',\ \text{and}\ \mathsf{v} =_\alpha \mathsf{v}'}{(\iota, \mathsf{v}) =_\alpha (\iota', \mathsf{v}')}$$

- α' *extends* α, iff $\alpha(\iota)$ defined $\Longrightarrow$ $\alpha(\iota) = \alpha'(\iota)$

For the earlier example, where

$$\chi_1 = \chi_0[\iota_{10} \mapsto (\ \mathsf{Book},\ (\mathsf{good} \mapsto \mathsf{true}\ , \mathsf{owner} \mapsto \mathsf{null})\ )]$$
$$\chi_2 = \chi_0[\iota_{20} \mapsto (\ \mathsf{Book},\ (\mathsf{good} \mapsto \mathsf{true}\ , \mathsf{owner} \mapsto \mathsf{null})\ )]$$

we have that

$$\chi_1 =_{\alpha_0} \chi_2 \quad ,$$

where $\alpha_0(\iota_{10}) = \iota_{20}$, and for all $\iota \neq \iota_{10}$ with $\chi_1(\iota)$ is defined : $\alpha_0(\iota) = \iota$.

We can now reformulate the previous lemma as follows:

Lemma - 2nd version For any program P, expression e,
If

$$\mathsf{e},\phi,\chi \rightsquigarrow_{\mathsf{P}} \mathsf{r}',\chi' \quad \text{and} \quad \mathsf{e},\phi,\chi \rightsquigarrow_{\mathsf{P}} \mathsf{r}'',\chi''$$

Then

$$\mathsf{r}' =_{\alpha} \mathsf{r}'' \text{ and } \chi' =_{\alpha} \chi'' \quad \text{for some } \alpha.$$

Lemma - 3rd version For any program P, expression e, and bijection α
If

$$\mathsf{e},\phi,\chi \rightsquigarrow_{\mathsf{P}} \mathsf{r}',\chi', \text{ and } \quad \mathsf{e},\phi'',\chi'' \rightsquigarrow_{\mathsf{P}} \mathsf{r}''',\chi''', \qquad \text{and}$$
$$\phi =_{\alpha} \phi'', \text{ and } \chi =_{\alpha} \chi'',$$

Then

$$\mathsf{r}' =_{\alpha'} \mathsf{r}''', \text{ and } \quad \chi' =_{\alpha'} \chi''', \quad \text{for some } \alpha', \text{ which extends } \alpha.$$

Discussion: Compare the 2nd and 3rd version of lemma.

The Type System of $\mathcal{L}_1$

The type system assigns types to $\mathcal{L}_1$-expressions.

In the context of P_{BP}, we expect new Person.meet(new Person) to have type Book, and new Person.meet(new Book) to be type incorrect.
Therefore, typing takes a program P into account.

Also, we expect this.like to have type Book in the body of meet in class Person, and x.like to have type Book in the body of readBy in class Book.
Therefore, typing takes the type of this and x into account, expressed through environment Γ.

Thus, typing is a judgement of the form:

$$P, \Gamma \vdash e : t$$

i.e. , in context of P and Γ, expression e has type t.

Environments

Environments, Γ, map the parameter, x, to a type, and the receiver, this, to a class. We define lookup, $\Gamma(\mathsf{id})$:

for $\Gamma = \mathsf{t}\ \mathsf{x}, \mathsf{c}\ \mathsf{this}$: $\quad \Gamma(\mathsf{id}) = \begin{cases} \mathsf{t} & \text{if } \mathsf{id} = \mathsf{x} \\ \mathsf{c} & \text{if } \mathsf{id} = \mathsf{this} \\ \text{undefined} & \text{otherwise.} \end{cases}$

For example, consider environments Γ_1, Γ_2, Γ_3, Γ_4, and Γ_5, where

$\Gamma_1 = \mathsf{bool}\ \mathsf{x}, \mathsf{Person}\ \mathsf{this}$,
$\Gamma_2 = \mathsf{Person}\ \mathsf{x}, \mathsf{bool}\ \mathsf{this}$,
$\Gamma_3 = \mathsf{Person}\ \mathsf{x}, \mathsf{Person}\ \mathsf{this}$,
$\Gamma_4 = \mathsf{bool}\ \mathsf{x}, \mathsf{B}\ \mathsf{this}$,
$\Gamma_5 = \mathsf{C}\ \mathsf{x}, \mathsf{B}\ \mathsf{this}$.

Then, $\Gamma_1(\mathsf{x}) = \mathsf{bool}$, $\Gamma_2(\mathsf{x}) = \mathsf{Person}$, and $\Gamma_1(\mathsf{like})$ is undefined .

Before defining the inference rules for typing, we introduce the predicates $IsCls(\mathsf{P}, \mathsf{c})$ asserting that c is a class, and $IsTyp(\mathsf{P}, \mathsf{t})$ asserting that t is a type.

$$
\begin{array}{rcl}
IsCls(\mathsf{P}, \mathsf{t}) & \equiv & \mathsf{P}(\mathsf{t}) \text{ is defined .} \\
IsTyp(\mathsf{P}, \mathsf{t}) & \equiv & IsCls(\mathsf{P}, \mathsf{t}) \text{ or } \mathsf{t} = \mathsf{bool}.
\end{array}
$$

Then, for the programs P_{BP} and P_{ok} from slides 6 and 7, we have:

$$
IsCls(\mathsf{P}_{\mathsf{BP}}, \mathsf{Book}), \quad \neg IsCls(\mathsf{P}_{\mathsf{ok}}, \mathsf{Book}), \quad IsCls(\mathsf{P}_{\mathsf{ok}}, \mathsf{B}), \quad \neg IsCls(\mathsf{P}_{\mathsf{BP}}, \mathsf{B}).
$$

Types of Expressions

litVarThis

$$\frac{}{\begin{array}{l} P, \Gamma \vdash \mathsf{true} \ : \ \mathsf{bool} \\ P, \Gamma \vdash \mathsf{false} \ : \ \mathsf{bool} \\ P, \Gamma \vdash x : \ \Gamma(x) \\ P, \Gamma \vdash \mathsf{this} : \ \Gamma(\mathsf{this}) \end{array}}$$

newNull

$$\frac{IsCls(P, c)}{\begin{array}{l} P, \Gamma \vdash \mathsf{null} : \ c \\ P, \Gamma \vdash \mathsf{new} \ c : \ c \end{array}}$$

fld

$$\frac{\begin{array}{l} P, \Gamma \vdash e : \ c \\ \mathcal{F}(P, c, f) = t \end{array}}{P, \Gamma \vdash e.f : \ t}$$

fldAss

$$\frac{\begin{array}{l} P, \Gamma \vdash e.f : \ t \\ P, \Gamma \vdash e' : \ t \end{array}}{P, \Gamma \vdash e.f{:=}e' : \ t}$$

$$\frac{P, \Gamma \vdash e : \text{bool} \quad P, \Gamma \vdash e_1 : t \quad P, \Gamma \vdash e_2 : t}{P, \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : t} \; \text{cond}$$

$$\frac{P, \Gamma \vdash e_0 : c \quad P, \Gamma \vdash e_1 : t_1 \quad \mathcal{M}(P, c, m) = t\ m\textbf{(}\ t_1\ x\textbf{)}\ \ \textbf{\{}\ e\ \textbf{\}}}{P, \Gamma \vdash e_0.m\textbf{(}\ e_1\textbf{)} : t} \; \text{methCall}$$

Examples

$P_{BP}, \Gamma_3 \vdash \text{this.like:=x.like} : \text{Book}$
$P_{BP}, \Gamma_1 \not\vdash \text{this.like:=x.like} : \text{Book}$
$P_{ok}, \Gamma_3 \not\vdash \text{this.like:=x.like} : \text{Book}$

$P_{ok}, \Gamma_{\dots} \vdash \text{new B.m(new E)} : ???$

Write out the typing of $\text{this.good := x.like.good}$ in P_{BP} and Γ_6, where Γ_6=Book this, Person x.

Properties of the Type System

- Do the two properties from below hold? If not, can you weaken the properties so that they hold?
  - $P, \Gamma \vdash e : t$ and $P, \Gamma \vdash e : t' \quad \Longrightarrow \quad t = t'$
  - $P, \Gamma \vdash e : t$ and $P, \Gamma \vdash e' : t \quad \Longrightarrow \quad e = e'$

- What does it mean for an expression e, and P, Γ, if: $\nexists t$ with $P, \Gamma \vdash e : t$?

- Do we need to require that $\Gamma(\mathsf{this})$ is a class, or that $\Gamma(x)$ is a type?

- The type system has fewer rules than the operational semantics (6 vs 9+9). Why are the fewer rules sufficient?

Well-formed programs

We define the following two predicates:

$$
\mathit{ClssWF}(\mathsf{P},\mathsf{c}) \;\equiv\; \left\{
\begin{array}{l}
\forall \mathsf{f}:\; \mathcal{F}(\mathsf{P},\mathsf{c},\mathsf{f}) = \mathsf{t} \;\Longrightarrow\; \quad \mathit{IsTyp}(\mathsf{P},\mathsf{t}) \\
\text{and} \\
\forall \mathsf{m}:\; \mathcal{M}(\mathsf{P},\mathsf{c},\mathsf{m}) = \mathsf{t}\ \mathsf{m}\mathbf{(}\ \mathsf{t}_1\ \mathsf{x}\mathbf{)}\ \ \mathbf{\{}\ \mathsf{e}\ \mathbf{\}} \;\Longrightarrow \\
\qquad\qquad (\ \mathit{IsTyp}(\mathsf{P},\mathsf{t}),\ \text{and} \\
\qquad\qquad\ \ \mathit{IsTyp}(\mathsf{P},\mathsf{t}_1),\ \text{and} \\
\qquad\qquad\ \ \mathsf{P},\mathsf{t}_1\ \mathsf{x},\mathsf{c}\ \mathsf{this} \vdash \mathsf{e}:\ \mathsf{t}\ ).
\end{array}
\right.
$$

$$
\mathit{ProgWF}(\mathsf{P}) \;\equiv\; \forall \mathsf{c} \in \mathit{dom}(\mathsf{P}) : \mathit{ClssWF}(\mathsf{P},\mathsf{c}).
$$

Examples

1. For fields,

   (a) $\mathcal{F}(\mathsf{P}_{\mathsf{BP}}, \mathsf{Book}, \mathsf{good}) = \mathsf{bool}$, and $IsTyp(\mathsf{P}_{\mathsf{BP}}, \mathsf{bool})$.

   (b) $\mathcal{F}(\mathsf{P}_{\mathsf{BP}}, \mathsf{Book}, \mathsf{owner}) = \mathsf{Person}$, and $IsTyp(\mathsf{P}_{\mathsf{BP}}, \mathsf{Person})$.

   Therefore $\forall \mathsf{f} : \mathcal{F}(\mathsf{P}_{\mathsf{BP}}, \mathsf{Book}, \mathsf{f}) = \mathsf{t}_0 \implies IsTyp(\mathsf{P}, \mathsf{t}_0)$

2. For methods,

   (a) $\mathcal{M}(\mathsf{P}_{\mathsf{BP}}, \mathsf{Book}, \mathsf{readBy}) =$ bool readBy**(** Person x**)** **{** this.good:= true **}**

   (b) $IsTyp(\mathsf{P}_{\mathsf{BP}}, \mathsf{bool}), \quad IsTyp(\mathsf{P}_{\mathsf{BP}}, \mathsf{Person})$

(c) $\mathsf{P_{BP}, Person\ x, Book\ this} \vdash \mathsf{this.good{:=}true} \ : \ \mathsf{bool}$

Therefore, $\forall \mathsf{m} : \ \mathcal{M}(\mathsf{P_{BP}, Book, m}) = \mathsf{t\ m(\ t_1\ x)\ \ \{\ e\ \}}$
$\Longrightarrow \quad \mathit{IsTyp}(\mathsf{P, t}), \ \mathit{IsTyp}(\mathsf{P, t_1}), \ \mathsf{P, t_1\ x, Book\ this} \vdash \mathsf{e} :$

3. Therefore, from 1, 2, and def on $\mathit{ClssWF}(\mathsf{P_{\_}}, \_)$ we obtain $\mathit{ClssWF}(\mathsf{P_{BP}, Book})$.

4. In a similar manner to 1-3,

   we can also establish that $\mathit{ClssWF}(\mathsf{P_{BP}, Person})$.

5. With 3 and 4, we obtain $\mathit{ProgWF}(\mathsf{P_{BP}})$.

On the other hand, $\mathit{ProgWF}(\mathsf{P_{ok}})$ does *not* hold.

Soundness of the Type System of $\mathcal{L}_1$

The type system is sound in the sense that a converging well-typed expression returns either a value of the same type as the expression, or a nullPntrExc, but does not get stuck.

Furthermore, in both cases, the resulting heap "agrees" with the program and the environment, *i.e.* its consistency is preserved.

Agreement

We introduce agreement notions between programs, heaps, and values:

The judgment $P, \chi \vdash v \lhd t$ expresses that the value v from heap χ “agrees” with the type t as defined in program P:

We first define weak agreement:

$$\frac{}{\mathsf{P},\chi\vdash\mathsf{true} <:\mathsf{bool}} \qquad \frac{}{\mathsf{P},\chi\vdash\mathsf{false} <:\mathsf{bool}} \qquad \frac{IsCls(\mathsf{P},\mathsf{t})}{\mathsf{P},\chi\vdash\mathsf{null}<:\mathsf{t}} \qquad \frac{\begin{array}{c}IsCls(\mathsf{P},\mathsf{c})\\ \chi(\iota)\downarrow_1 = \mathsf{c}\end{array}}{\mathsf{P},\chi\vdash\iota<:\mathsf{c}}$$

Based on the weak agreement, we define strong agreement:

$$\mathsf{P},\chi\vdash\mathsf{v}\lhd\mathsf{t} \;\equiv\; \begin{cases} \mathsf{P},\chi\vdash\mathsf{v}<:\mathsf{t}, & \text{if } \mathsf{v}\in addr \\ \text{and} & \\ \forall\mathsf{f}:\;(\mathcal{F}(\mathsf{P},\mathsf{t},\mathsf{f})=\mathsf{t}' \implies \mathsf{P},\chi\vdash\chi(\mathsf{v},\mathsf{f})<:\mathsf{t}'\,) & \\ & \\ \mathsf{P},\chi\vdash\mathsf{v}<:\mathsf{t}, & \text{if } \mathsf{v}\notin addr \end{cases}$$

Note: The requirements $\mathsf{P},\chi\vdash\iota<:\mathsf{t}$ and $\iota\in addr$ imply $IsCls(\mathsf{P},\mathsf{t})$.

For example, remember our earlier program with:

$\mathsf{class\ Book}\{\ \mathsf{bool\ good},\ \mathsf{Person\ owner}\ \ \ldots\ \}\qquad \mathsf{class\ Person}\{\ \mathsf{Book\ \ like}\ \ \ldots\ \}$

Take a heap χ_8 defined as

$$
\begin{array}{lll}
\chi_8(\iota_3) & = & (\ \mathsf{Book},\ (\mathsf{good} \mapsto \mathsf{false}\ ,\ \ \mathsf{owner} \mapsto \iota_6)\ ) \\
\chi_8(\iota_4) & = & (\ \mathsf{Book},\ (\mathsf{good} \mapsto \mathsf{false}\ ,\ \ \mathsf{owner} \mapsto \iota_3)\ ) \\
\chi_8(\iota_5) & = & (\ \mathsf{Book},\ (\mathsf{good} \mapsto \mathsf{false}\ ,\ \ \mathsf{owner} \mapsto \iota_6,\ \ \mathsf{like} \mapsto \mathsf{true})\ ) \\
\chi_8(\iota_6) & = & (\ \mathsf{Person},\ (\mathsf{good} \mapsto \mathsf{true})\ )
\end{array}
$$

Which of the following assertions hold?

$$
\begin{array}{ll}
\mathsf{P}_{\mathsf{BP}}, \chi_8 \vdash \mathsf{null} <: \mathsf{Person} & \mathsf{P}_{\mathsf{BP}}, \chi_8 \vdash \mathsf{null} \lhd \mathsf{Person} \\
\mathsf{P}_{\mathsf{BP}}, \chi_8 \vdash \iota_3 <: \mathsf{Book} & \mathsf{P}_{\mathsf{BP}}, \chi_8 \vdash \iota_3 \lhd \mathsf{Book} \\
\mathsf{P}_{\mathsf{BP}}, \chi_8 \vdash \iota_4 <: \mathsf{Book} & \mathsf{P}_{\mathsf{BP}}, \chi_8 \vdash \iota_4 \lhd \mathsf{Book} \\
\mathsf{P}_{\mathsf{BP}}, \chi_8 \vdash \iota_5 <: \mathsf{Book} & \mathsf{P}_{\mathsf{BP}}, \chi_8 \vdash \iota_5 \lhd \mathsf{Book} \\
\mathsf{P}_{\mathsf{BP}}, \chi_8 \vdash \iota_6 <: \mathsf{Person} & \mathsf{P}_{\mathsf{BP}}, \chi_8 \vdash \iota_6 \lhd \mathsf{Person}
\end{array}
$$

The judgment $\mathsf{P}, \Gamma \vdash \phi, \chi \diamond$ expresses that all addresses in the heap χ point to objects that agree with their class, and that the receiver and argument on the stack frame ϕ agree with their type as declared in Γ:

$$\mathsf{P}, \Gamma \vdash (\iota, \mathsf{v}), \chi \diamond \quad \equiv \quad \begin{cases} \mathsf{P}, \chi \vdash \iota \lhd \Gamma(\mathsf{this}), \quad \text{and} \\ \mathsf{P}, \chi \vdash \mathsf{v} \lhd \Gamma(\mathsf{x}), \quad \text{and} \\ \forall \iota' : \ \chi(\iota') \downarrow_1 = \mathsf{c} \implies \mathsf{P}, \chi \vdash \iota' \lhd \mathsf{c} \end{cases}$$

Lemma

$\mathsf{P}, \Gamma \vdash \phi, \chi \diamond$ and $\chi(\iota) \downarrow_1 = \mathsf{c}$ and $\mathsf{f} \in \mathcal{F}s(\mathsf{P}, \mathsf{c})$

$\implies$

$\mathsf{P}, \chi \vdash \chi(\iota, \mathsf{f}) \lhd \mathcal{F}(\mathsf{P}, \mathsf{c}, \mathsf{f})$

Again, remember our earlier program with:

class Book{ bool good, Person owner ... }   class Person{ Book like ... }

and take

$$\begin{array}{lcl} \phi_0 & = & (\iota_3, \iota_4) \\ \chi_0(\iota_3) & = & (\text{ Book}, \ (\text{good} \mapsto \text{false} \ , \ \ \text{owner} \mapsto \iota_5) \ ) \\ \chi_0(\iota_4) & = & (\text{ Person}, \ (\text{like} \mapsto \iota_3) \ ) \\ \chi_0(\iota_5) & = & (\text{ Person}, \ (\text{like} \mapsto \text{null}) \ ) \end{array}$$

Then, we have:

$$P_{BP}, \chi_0 \vdash \iota_3 \triangleleft \text{Book} \qquad P_{BP}, \chi_0 \vdash \iota_4 \triangleleft \text{Person} \qquad P_{BP}, \chi_0 \vdash \iota_5 \triangleleft \text{Person}$$

and therefore,

$$\begin{array}{l} P_{BP}, \text{Person x, Book this} \vdash \phi_0, \chi_0 \diamond \\ P_{BP}, \text{Person x, Person this} \not\vdash \phi_0, \chi_0 \diamond \end{array}$$

On the other hand, the heap χ_8 is so “broken”, that for all possible Γs, and ϕs, $P_{BP}, \Gamma \not\vdash \phi, \chi_8 \diamond$

Type Soundness

Theorem For a program P, an environment Γ, an expression e, a heap χ, a stack frame ϕ, a type t, and a result r, if

$$\mathit{ProgWF}(\mathsf{P}), \quad \text{and} \quad \mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{t}, \quad \text{and}$$

$$\mathsf{P}, \Gamma \vdash \phi, \chi \diamond, \quad \text{and} \quad \mathsf{e}, \phi, \chi \rightsquigarrow_{\mathsf{P}} \mathsf{r}, \chi',$$

then

- $\mathsf{r} \in \mathit{val}$, and $\mathsf{P}, \chi' \vdash \mathsf{r} \triangleleft \mathsf{t}$, and $\mathsf{P}, \Gamma \vdash \phi, \chi' \diamond$,

or

- $\mathsf{r} = \mathsf{nullPntrExc}$, and $\mathsf{P}, \Gamma \vdash \phi, \chi' \diamond$.

Remarks What about non-terminating execution? Why do we care about $\mathsf{P}, \Gamma \vdash \phi, \chi' \diamond$?

Proof by structural induction over the derivation $\mathsf{e}, \phi, \chi \rightsquigarrow_{\mathsf{P}} \mathsf{r}, \chi'$.

Discussion: Object Initialization revisited - after Michal Srb

Propose an operational semantics rule for object initialization, `new c`, which does not take the definition of the class `c` into account.

What are the implications of this?

Can we change some more of the language design to mitigate these implications? What do current programming languages propose?

Object Initialization revisited - after Michal Srb - part II

space for student’s deliberations

Strong, static, typing is conservative

The theorem provides a sufficient but not necessary condition.

Consider, for example, the execution of the expressions:

$$\mathsf{x.like} := \mathsf{x};\ \ \mathsf{x.like.like}$$

(we use the semicolon, because we now know how to encode it in $\mathcal{L}_1$).

Execution of the above expression in heap χ_0 will not be "stuck", and will return the address ι_4. It could be run on Smalltalk, or Python, etc.

Nevertheless, the above expression is ill-typed in the context of $\mathsf{P_{BP}}$, and for all possible Γ's.

Strong, static typing does not attempt to anticipate such situations.

Summary

We have defined the following:

- the syntax of $\mathcal{L}_1$ expressions, and the structure of $\mathcal{L}_1$programs,
- execution, $\mathsf{e},\phi,\chi \rightsquigarrow_{\mathsf{P}} \mathsf{r},\chi'$,
- typing, $\mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{t}$,
- well-formed program, $\mathit{ProgWF}(\mathsf{P})$,
- value conforming to type, $\mathsf{P},\chi \vdash \mathsf{v} \lhd \mathsf{t}$,
- frame and heap conforming to program and environment $\mathsf{P}, \Gamma \vdash \phi, \chi \diamond$,

and have demonstrated soundness of the type system.

Many languages/features can be understood in terms of such concepts.

$\mathcal{L}_1$ is a minimal oo language reflecting common intuitions about programming and execution. There are smaller, more abstract, object-based calculi (Abadi-Cardelli, and Mitchell-Fischer, Igarashi-Pierce-Wadler).

Extensions of $\mathcal{L}_1$-like languages have been used, *e.g.* in:

- proving soundness of Java (Drossopoulou, Eisenbach, Khurshid, Valkeyvych (ECOOP'97, TAPOS'99), Syme (1998), vonOheimB, Nipkow, (POPL'98))
- mixins (Felleisen, Flatt, Khrishnamurthi POPL'98, and Ancona, Zucca ECOOP'99), traits (Smith, Drossopoulou, ECOOP'05)
- dependent classes (Jolly, Drossopoulou, Anderson, Ostermann, FTfJP 2004), Tribe (Clarke, Drossopoulou, Wrigstad, Noble, AOSD'07)
- objects changing class (Drossopoulou, Damiani, Dezani, Giannini, ECOOP'01)
- binary compatibility (Drossopoulou, TIC'00, Drossopoulou, Eisenbach, Wragg LICS'99, OOPSLA'98)
- mapping oo languages onto typed intermediate languages (League, Shao, Trifonov ICFP'99, TIC'00)
- alias restrictions (Clarke, Noble, Potter, OOPSLA'98, Clarke, Drossopoulou OOPSLA'02, Cameron, Drossopoulou, Noble, Smith, OOPSLA'07)
- aspect oriented programming (Mark Skipper'01)
- Garbage collecting Actors (Clebsch & Drossopoulou OOPSLA'13)
- Garbage collecting Objects in Actor-based Laguaages (Clebsch et al OOPSLA'17, Franco et al ESOP'18)