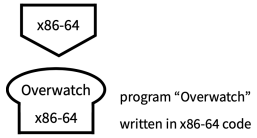
Programs are just data

T-diagrams



Bread diagram. Program name on top and the language its written in below.
Inverted diagram. Processor type
Connect bread diagram on top to inverted diagram below. Processor type **must match** language of bread of bread diagram.
Machine code can be interpreted as well, a process called hardware emulation.

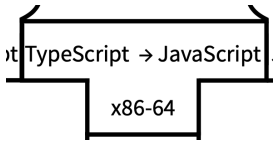
Translators

Special cases of compilers are assemblers, where the “high-level” language is assembler code and the low-level language is machine code, and disassembler, which are decompilers that translate from machine code to assembler code. Compilers translate from a high-level language into a lower level language, and decompilers translate from a low-level language into a higher one.

Interpreter

Determines the meaning of another program.
Diagram is a box, with the target language on top and source language below. **Just another program**. Interpreter with the same source and target is useless, but it we want to print out the program that we received, this is non-trivial.
Source language. Language that its written in.
Target language. Language which program input is written in for interpreter to execute.

Compiler



Translates programs from one language to another. When the source language is the same as the output language, the compiler just needs to churn out the input. Bottom value means the compiler is written in that language.
From-language. Input language.
To-language. Output language.
Compiler takes the program and produces a new output that can be reused multiple times even after the compiler is destroyed. Interpreter have to always be there to interpret the programs.

Backus-Naur Form

program	::=	*expr*;	program
expr	::=	*integer*	primitive integer expr.
	|	`true` | `false`	primitive boolean expr.
	|	*unary-operator* `[` *expr* `]`	unary operator
	|	*binary-operator* `[` *expr* `,` *expr* `]`	binary operator comb.
	|	*binary-logical* `[` *expr* `,` *expr* `]`	logical comp.
binary-operator	::=	`+` | `-` | `*` | `===` | `>` | `<`	binary operator
binary-logical	::=	`&&` | `||`	logical comp. symbol
unary-operator	::=	`!`	unary operator

Perform operator on expression.
Infix Notation. Operation between expression.
Associativity. Changing the grouping of parenthesis shouldn’t affect the result.

Dynamic Semantics

Keep processing based on operator precedence until we get the final answer. Formalised using $\gg_{S0}$. Have to constantly search for a reducible subexpression.

$$\frac{E \mapsto_{S0} E'}{p_1[E] \mapsto_{S0} p_1[E']}\text{[OpArg}_1\text{]}$$

$$\frac{E_1 \mapsto_{S0} E_1'}{p_2[E_1, E_2] \mapsto_{S0} p_2[E_1', E_2]}\text{[OpArg}_2\text{]}$$

$$\frac{E_2 \mapsto_{S0} E_2'}{p_2[E_1, E_2] \mapsto_{S0} p_2[E_1, E_2']}\text{[OpArg}_3\text{]}$$

Top is the condition for bottom to happen. Bottom is the formal definition of finding things to simplify. Basically means that E evaluates to E’ in one step if E contracts to E’. p_x refers to an operator.
Evaluation. Applying **zero** or more one step contractions. Programs are defined as a series of contractions to a value which an integer or boolean. Does not prescribe the order that evaluations should be executed in.

$$\frac{}{E \mapsto_{S0}^* E}[\mapsto_{S0}^*_R]$$

$$\frac{E_1 \mapsto_{S0} E_2}{E_1 \mapsto_{S0}^* E_2}[\mapsto_{S0}^*_I]$$

$\mapsto_{S0}^*$ is reflexive transitive closure of $\mapsto_{S0}$

$$\frac{E_1 \mapsto_{S0}^* E_2 \quad E_2 \mapsto_{S0}^* E_3}{E_1 \mapsto_{S0}^* E_3}[\mapsto_{S0}^*_T]$$

Transitive Reflexive Closure. If A evals to B and B evals to C, then A evals to C. And that A evals to A as well.
Meaning of programs. Basically taking an expression and applying 0 or more one step evaluations to obtain a value of type integer or boolean.
Reducible. Possible to carry out one step evaluation.
Irreducible. Not possible to carry out one step evaluation.

Static Semantics

Assignment of types to each subexpression.
Typing relation. Expression E : type t. E has type t.
Well-typed. Typing conditions are met, there is a type t for expression E, such that $E : t$.
Ill-typed. Typing conditions not met.

$$\frac{E_1 : t_1 \quad E_2 : t_2}{p[E_1, E_2] : t}[\textbf{Prim}_2]$$

p	t_1	t_2	t
`+`	integer	integer	integer
`-`	integer	integer	integer
`*`	integer	integer	integer
`&&`	boolean	boolean	boolean
`||`	boolean	boolean	boolean
`===`	integer	integer	boolean
`<`	integer	integer	boolean
`>`	integer	integer	boolean

Problems. Alot of rules (table for typing relation), possible to get stuck since its not able to find a value, not extendable to other programming paradigms (cannot call other models to solve problems). Limited to specific types only.

Denotational Semantics

Directly tells the value of a program, one big step evaluation. Expressible values are the results of programs that can take the type of integer or boolean.

Semantic domain	Definition	Explanation
Bool	$\{true, false\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	**Bool** + **Int**	expressible values

Semantic function

$$\cdot \rightarrowtail \cdot : \textbf{S0} \rightarrow \textbf{EV}$$

$$\frac{E_1 \rightarrowtail v_1 \quad E_2 \rightarrowtail v_2}{E_1 + E_2 \rightarrowtail v_1 + v_2} \qquad \frac{E_1 \rightarrowtail v_1 \quad E_2 \rightarrowtail v_2}{E_1 - E_2 \rightarrowtail v_1 - v_2} \qquad \frac{E_1 \rightarrowtail v_1 \quad E_2 \rightarrowtail v_2}{E_1 * E_2 \rightarrowtail v_1 \cdot v_2}$$

$$\frac{E_1 \rightarrowtail v_1 \quad E_2 \rightarrowtail v_2}{E_1 \,\&\&\, E_2 \rightarrowtail v_1 \wedge v_2} \qquad \frac{E_1 \rightarrowtail v_1 \quad E_2 \rightarrowtail v_2}{E_1 \,||\, E_2 \rightarrowtail v_1 \vee v_2} \qquad \frac{E \rightarrowtail v}{!E \rightarrowtail \neg v}$$

$$\frac{E_1 \rightarrowtail v_1 \quad E_2 \rightarrowtail v_2}{E_1 === E_2 \rightarrowtail v_1 \equiv v_2} \qquad \frac{E_1 \rightarrowtail v_1 \quad E_2 \rightarrowtail v_2}{E_1 > E_2 \rightarrowtail v_1 > v_2} \qquad \frac{E_1 \rightarrowtail v_1 \quad E_2 \rightarrowtail v_2}{E_1 < E_2 \rightarrowtail v_1 < v_2}$$

$\rightarrowtail_N$ assigns integer values to decimal numerals.

Virtual Machine

Does not describe the process to get the type. Inductive rule requires recursion and where are the intermediate results stored? Compile it to machine code so that it can be executed.

Instruction	Explanation
`DONE`	last instruction
`LDCI` i	i is integer
`LDCB` b	b is boolean
`PLUS`	addition
`MINUS`	subtraction
`TIMES`	multiplication
`AND`	conjunction
`OR`	disjunction
`NOT`	negation
`LT`	less-than
`GT`	greater-than
`EQ`	equality

Machine program is a set of instructions ending in reverse polish notation (operands then operator) with DONE.
Translation Function. $\twoheadrightarrow$ appends DONE to the result of $\hookrightarrow$.
Helper functions. $\hookrightarrow$ with rules from the vm instructions table.
PC. Program counter is the index of instruction array.
OS. Operand stack which is a stack of expressible values. **Machine.** Contains OS and PC. PC starts at 0 and terminates when DONE is reached. Store the intermediate results with OS.

Example machine program:

```
[LDCI 1, LDCI 2, PLUS, DONE]
```

Example: $[\mathit{LDCI}\ 10, \mathit{LDCI}\ 20, \mathit{PLUS}, \mathit{LDCI}\ 6, \mathit{TIMES}, \mathit{DONE}]$

$(\epsilon, 0) \rightrightarrows (10\ .\ \epsilon, 1) \rightrightarrows (20\ .\ 10\ .\ \epsilon, 2) \rightrightarrows (30\ .\ \epsilon, 3) \rightrightarrows (6\ .\ 30\ .\ \epsilon, 4) \rightrightarrows (180\ .\ \epsilon, 5)$

$\rightrightarrows$ is a transitive function between machine states. Add the results before ϵ. To evaluate, pop the operands from the stack, evaluate then push result to the stack. Behavior of function depends on the instruction at PC.

Recursive Interpreters

To interpret a complex program is essentially the interpretation of its parts.
Environment. Allow to account for names to denote values. In other words, allow assignment of values to variables. Uses recursion and defines many of the constructs of the target language by direct reference to the same construct of the source language. Functions from names to values.

Semantic domain	Definition	Explanation
DV	**Bool** + **Int**	denotable values
Id	alphanumeric strings	identifiers
Env	**Id** $\rightharpoonup$ **DV**	environments

$\rightharpoonup$ means that environments are partial functions, since an environment may be undefined for a given identifier. If an identifier does not have a value and its called in an environment, it would be undefined.

$$\cdot \Vdash \cdot \rightarrowtail \cdot : \textbf{Env} \times \mathit{expr} \rightarrow \textbf{EV}$$

Example: $\emptyset \Vdash 1 + 2 \rightarrowtail 3$

Expressions take an environment as an additional argument since it needs to check whether a particular variable has a value or not.

$$\frac{\Delta(x) \neq v \text{ for any } v}{\Delta \Vdash x \rightarrowtail \perp}$$

below, v indicates denotable value

$$\frac{\Delta \Vdash E_1 \rightarrowtail \perp}{\Delta \Vdash E_1 + E_2 \rightarrowtail \perp} \qquad \frac{\Delta \Vdash E_1 \rightarrowtail v \quad \Delta \Vdash E_2 \rightarrowtail \perp}{\Delta \Vdash E_1 + E_2 \rightarrowtail \perp}$$

Evaluation fails when it includes an identifier that has not been introduced. This means that the expressible value could now be an error as well, represented by $\perp$. As long as one of the expression evaluates to a failure, the entire expression would evalaute to failure.
Program Values. The value after the program exits, could be standard expressible values, error or undefined. double headed $\rightarrowtail$ returns a pair that has the program value and an env. A pair is returned for a program because the program statements could modify the environment. So we need to return the modified environment as well, on top of the program value. Program contains statements.

Environment extension: $\cdot[\cdot \leftarrow \cdot] : \textbf{Env} \times \textbf{Id} \times \textbf{DV} \rightarrow \textbf{Env}$

Examples: $\emptyset$ $\quad \emptyset\ [x \leftarrow 3] \quad \emptyset\ [x \leftarrow 3]\ [y \leftarrow 4]$

Assignment appends the variables from left to right. To read the values, go from right to left.

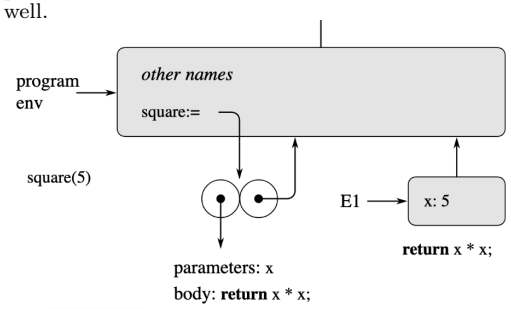
- result of evaluating an empty program gives undefined

- result of evaluating a non-empty program is the result of evaluating its last statement, except if an error occurs, then it will be propogated.
- if evaluation does not succeed, give back the original environment
- reassignment gives the latest value but does not remove the old value.

Takeaways. Use error handling of source language to handle target language errors, this greatly simplies the interpreter. Never update the same environment twice. Updates sets the environment to a new array, destroy the previous array.

Source 3

Contains functions, blocks, returns and internal declarations. Cannot operate on a single environment since some values are function specific only and not meant to be accessed globally.
Eval blocks. Create some unassigned values that if they were accessed, throw a value.
Inner functions. Func env is an extension of outer environment with its own parameters and function parameters. But it can also access outer env stuff as well.



Function assignment in environment points to 2 balls. First ball contains parameters and body, second ball points to the environment that calls it so that it could return there. When its called with a value, creates a new environment E1.
Eval returns. Return the value if it has one, otherwise return undefined.
Interpretative Overhead. Evaluation is carried out multiple times. Originally applies to both env and component multiple times. Instead, apply to component multiple times then one shot apply to env.

Explicit Control Evaluators

Every statement produces a value and declarations produces undefined. While and for loops produce undefined as well. Exceptions are included here as well.

Structural Operational Semantics

Modify the machine to contain a pair of OS and an agenda (stack of expressions to be evaluated). Expand everything and evaluate them by pushing into OS.

Result of evaluating an expression E is a value v where: $(\epsilon, E) \Rightarrow_s^* (v, \epsilon)$

Example: evaluation of (10 + 20) * 6:

$$
\begin{aligned}
&(\epsilon, \texttt{(10 + 20) * 6}) \Rightarrow_s \\
&(\epsilon, \texttt{10 + 20 . 6 . TIMES}) \Rightarrow_s \\
&(\epsilon, \texttt{10 . 20 . PLUS . 6 . TIMES}) \Rightarrow_s \\
&(10, \texttt{20 . PLUS . 6 . TIMES}) \Rightarrow_s \\
&(\texttt{20 . 10, PLUS . 6 . TIMES}) \Rightarrow_s \\
&(\texttt{30, 6 . TIMES}) \Rightarrow_s \\
&(\texttt{6 . 30, TIMES}) \Rightarrow_s \\
&(180, \epsilon)
\end{aligned}
$$

Include a third part to the configuration which is the environment since now we deal with function calls as well.
Sequences. Statements are executed in order and only the last value remains on the stash. After every statement except the last, push a pop instruction. This instruction is to perform a POP from the stash.
Conditional Expression. Convert to conditional and the branches. Pop and push predicate to stash. When evaluated, push the right branch onto the agenda and evaluate accordingly.
Blocks. Extend the current environment by creating a copy of it and a link to it as well. Pop the block from agenda and push the body. So everything in the subsequent statements would all be for this current environmental frame. Value look ups are in current environment, otherwise keep going up until we find it.
Functions.

- Wrap the function in a block. Since functions will have their own environments anyway.
- Function declarations can be written as lambda expressions which is an assignment.
- Since its an assignment, value of the lambda would be a function expression, which points to the current environment. The assignment would allow the function name to be pointing to the function bubbles.
- Pop the assignment which is just a pointer to the function, since now the function name is pointing to the function.
- For a function call, split into the function name, the parameters, and a CALL instruction that takes number of parameters. Popping each of these from the agenda pushes values (parameters) and pointers (function name) to the stash.
- When CALL is executed, pop function related stuff from the stash and extend the function environment by including the parameter. Previous environment is meant to enclose the function in a block only.
- Each CALL of the same function would create a new environment from the one with the function declaration.

Leads to closure on the stash that contains a triple, lambda expression parameters, body and environment at time when lambda was evaluated. Functions should make a reference to the environment that has them.

Restoring Environments

After executing a block, the need to return to the previous environment. To do this, have a marker that restores the previous environment. This marker is meant for instructions that modifies the environment. After each block/call instruction, put a RESTORE instruction after the body.
RESTORE. Pointer to the environment we should go to after this block is done executing.

Problems

General return statements. Multiple return statements could lead to problems on the agenda since there will be remaining items that belong to the function on the agenda. After each CALL instruction, push a MARKER to the agenda. When **return** is executed, push a RESET instruction onto the agenda.
RESET instruction. Pop everything from the agenda including and up to the next MARKER.
Tail Recursion. Run time checks are enough. Some markers and resets not needed. Since tail recursion returns one of the parameters as the result. Cannot give rise to a new frame. Mark the tail call and don't do reset, rely on the previous reset.
Exception Handling. Place exception marker on agenda. When exception is thrown, clear agenda until exception marker, then execute the catch.

Virtual Machines

Translate language to a sequence of instructions prior to execution. Then hand over the instruction sequence to a virtual machine.
Proposition. Let E be a well-typed expression. There exists a value, v, and an SVML instruction sequence s, st $E \rightarrowtail v, E \hookrightarrow s$ and for any instruction sequence s' and any operand stack os, we have $(os, 0) \Rightarrow_{s,s'}^* (v.os, |s|)$ This means that any sequence puts a value on the operand stack.
Theorem. For every well-typed expression, E, if $E \rightarrowtail v, E \twoheadrightarrow s$, then $R(M_s) = v$. When running a machine with sequence s, the machine will output the value v.
Corollary. For a well-typed expression E, the corresponding machine M_s where $E \twoheadrightarrow s$, will never get stuck. Guarenteed to always make progress but does not guarenetee that the program will terminate.
Lemma. The runtime of a program obtained by compiling a well-typed program is $|s| - 1$. At each step, the PC increases by 1, so the DONE instruction will be reached at $|s| - 1$.
Compiler Implemetation. Write counter to denote where to write instruction to. Look at the tag of the component, and compile it. Add to instruction array and increase the counter. Still uses postfix notation. Use individual components to reduce binary operator overhead.
VM Implementation. Run through the instruction by looking at the tag, then look at the microcode to get the result. When done, just peek at the operand stack to get the result.
Sequences. When reched the end of a sequence, just pop from operand stack by having a pop instruction after the sequence.
Conditionals. Have to execute the boolean expression. If the boolean is true, execute the body and have a GOTO instruction after the body. GOTO basically goes to outside the conditional statements. If the boolean is false, set the next executing instruction to the right line by JUMP ON FALSE to go to the second statement and execute as per normal.

$$\frac{s(pc) = \texttt{GOTOR } i}{(os, pc) \rightrightarrows (os, pc + i)} \qquad \frac{s(pc) = \texttt{JOFR } i}{(true.os, pc) \rightrightarrows (os, pc + 1)}$$

$$\frac{s(pc) = \texttt{JOFR } i}{(false.os, pc) \rightrightarrows (os, pc + i)}$$

Names. Add another register for the environment. Executing names pushes the value which the name refers to onto the operand stack.
Function Compilation.

```
instrs[wc++] = {tag: 'LDF', prms: comp.prms, addr: wc + 1};
// jump over the body of the lambda expression
const goto_instruction = {tag: 'GOTO'}
instrs[wc++] = goto_instruction
compile(comp.body)
instrs[wc++] = {tag: 'LDC', val: undefined}
instrs[wc++] = {tag: 'RESET'}
goto_instruction.addr = wc;
```

Application is compiled by adding a CALL instruction that remembers the number of arguments of the application. Push the entire function and jump to the code after the function body (not readily executed). Purpose of GOTO is to jump to function body when called. RESET instruction is to bring us back to the original frame.
Function Execution. Pop argument into vector. Pop closure into operand stack. If the function is a builtin, can handle directly, otherwise push a new call frame for the function.
Iterative Process. Does not consume extra space on the RTS. While recursive processes will have their frames left behind even after returning. Changing from a conditional expression to conditional statements will change the process to an iterative one because of the return statements.
Environments are a list of frames, length of the frames determine the efficiency of the LOAD instruction. The length of the environment is bounded by the maximal nesting depth of blocks (including functions) in a program. The number of property in each frame is not determined by the nesting depth of blocks in the given program. The size of the runtime stack is bounded by the number of function calls that get executed during the run of a program.
SourceA. Conversion to load instructions followed by operation. Operation pops everything and push the result onto the **operand stack**. Evaluation can get stuck if non of the rules apply. DONE instruction breaks the loop by setting RUNNING to false.
SourceB. Add error handling by using another register that breaks out of the loop when error occurred.
SourceC. Conditionals allowing jumping around. Based on the evaluated expression, perform JUMP ON FALSE RELATIVE if the expression is false, or GOTO if its true. GOTO basically goes to the end of the conditional, after the false block.
SourceD. Load instructions to update environments. Function declarations loads everything then jump to the end of the function body. Runtime stack to facilitate returning. Each entry in the runtime stack contains the address of the instruction to return to, and the operand stack os and environment e to be reinstalled after the function call. Such a triplet (address, os, e) is called runtime stack frame, or simply stack frame. Function calls pushes things on the runtime stack since it needs to be returned.

More realistic VM

Faster memory allocation. Currently just a linked list of environments which is expensive to go through.
Lexical Scoping. Instead, have a compile time environment that contains mappings of values to frame-index pair. For lambda expression, extend the environment by including parameters in the compile time environment.

The runtime of the LD instruction is constant. The runtime of the CALL instruction depends on the number of frames in the environment of the function being called.

Heap Allocation

Attempt to have random memory access by using an array buffer of bytes. Heap environment extension is a shallow copy, copying addresses, environment represented by addresses. Cannot be a deep copy otherwise previous environments would change as well.
Nan boxing. Modify the remaining bits of the NaN bit string. First 13 bits is default for the NaN tag. Next 4 bits are possible tags to indicate the type of value. Then fill the rest with the value.
String Pooling. Strings can be dynamic (concatenation) or static (declaration). Possible store multiple copies of the same string will be a problem. Accessing string should be done in constant time and creating a new string should not be more expensive than creating an object. Have a set of strings that have been created in the String Pool and then just add to the pool.

Memory Management

Static Allocation. Assign fixed memory location for every identifier. Limitations the size of each data structure only known at compile-time. Recursive functions are not possible too since they need their own copy of parameters and local variables. Data structures like closures cannot be created dynamically. Optimises for speed and safety (programs cannot run of space at run-time).
Stack Allocation. Keeps track of info on function invocations on runtime stack. Recusion is possible. Size of locals can depend on the arguments. But still difficult to manipulate recursive data structures like trees. Only objects with known compile time size can be returned from functions.
Heap Allocation. Allocation and deallocation can be done in any order. Pointer structures evolves at runtime. Management of allocated memory is the problem.

Heap Memory Model

A graph modelled by nodes (representing stack frames, operand stacks, envs and closures, primitive values) and edges(references). Labelled edges with symbols/numbers. Every edge has 1 label, 1 source and target. Space consumption of a program is measured as number of nodes and edges **created by it**.
New node. Construct a heap when a new node is added. Returns a pair containing a new node and a new heap.
Update. Updating the target of a node and the label.
Children. Returns all children.
NodeChildren. Return the children that are nodes.
Labels. Return all labels.
Deref. Specific child of a given node. Its a partial function (some nodes may not have children).
Copy. Creates a new node which has the same children under the same labels as a given node.
Stack nodes. Create a new stack by creating a node with 1 child that contains the size of it. When pushing, add another child to the node and update the size. When popping, remove the child and update the size.

Memory-aware VM

Load instruction uses the push onto the operand stack. For referencing/assignment of variables, push the derefenced value of the node onto the operand stack. For LDFS instruction, closure will have one node and each of its children would have a node as well. Closure will be on the oeprand stack so there is an edge there as well. Formals will be the parameters of the closure which would be the children of the closure. **Space consumption is number of nodes and edges**.

Implementation

Heap as an array, where each nodesize indicates the number of rows that a node can take, children start and end addresses allocates addresses to some children.
Freelist. Linked list of nodes with a pointer to the next free space we can use.

Garbage Collection

Reference Counting. Useless nodes are nodes that are not targeted by others. Keep track of number of edges that end in it. If number becomes 0, then node is unreachable and therefore its useless, so its freed. Updating an edge can modify this count, then it performs garbage collection. Exploits incrementality (reclaimation of memory on the fly, doesn't affect performance), locality (doesn't affect others), immediate reuse(quickly reuse the memory). Cannot free cycles since nodes in cycles always have a reference count that is non-zero.
Mark-Sweep. Mark nodes that are reachable from some register of the machine. Then visit every node and check if its marked. If its not marked, then free it. Amount of memory reclaimed per unit time is $\frac{m_{MS}}{t_{MS}}$. On invocation, size of the heap is the number of nodes. The residency of the program is the ratio of live to total nodes. Time taken for mark-sweep is the sum of time to visit every reachable node and the time taken to visit every node. $e_{MS} = \frac{1-r}{ar+b}$, where r is the residency.
Copying. Solves the issue of efficiency (better than linear) and fragmentation of memory. Effective memory is halved. When filled, copy live nodes to empty half,m perform BFS to find children of live nodes then free the other half altogether. The amount of memory recovered is $\frac{M}{2} - R$ since the effective memory is halved. Since BFS is done only on lived nodes, so the time for copying is just the number of live nodes. then the resulting efficiency is $\frac{1}{2cr} - \frac{1}{c}$, where c is the constant for BFS. Live memory gets compacted since its copying everything. Improving the performance of programs.
Any node that is considered "dead" in reference counting is also considered "dead" in copy garbage collection. A node is considered "dead" in mark-sweep garbage collection if and only if it is considered dead in copy garbage collection.
Comparison. How to choose which technique

- Available size of memory (mark-sweep)
- Responsiveness (reference counting)
- Available processor speed (tracing garbage collection, marksweep and copying)
- Memory architecture (compacting garbage collector)
- Typiacl residency (mark sweep)
- Presence and frequency of cyclic structure (tracing algos)
- Range of size of heap nodes (copying)

Type Safety

Declare the function input type, use $>$, then declare the output type.
Ill-typed. Typing condition is not met.
Well-typed. Typing conditions met.

Type environment

Γ keeps track of the name of the type of names appearing in the statement. The partial function from names to types expresses a context in which a name x is associated with type $\Gamma(x)$.
Type Relation. $\Gamma[\ x \leftarrow t]\ \Gamma'$, names are x, t is types, contructs a type environment that specifies the type of x is t. So if $y = x$, then $\Gamma'(y) = t$ and $\Gamma(y)$ otherwise. Uniquely identifies Γ' for a given type environment, name and type. Hence the type environment relation is functional in its first 3 arguments. Applying the relation results in a new type environment that can call the names and return the specific type.
Domain. Set of names on which a type environment is defined, *dom*. Domain of the last type environment includes the domains of all previous type environments.
Free Names. Names that are not bounded by an expression or statement.
Typing Relation on Expression. The expression E has type t, under the assumption that its free names have the types given by Γ'' When E has no free names, we can write $E\ :\ t$. When E has free names, then we write $\Gamma \vdash E\ :\ t$. For every expression E and every type assignment Γ, there exists at most one type t such that $\Gamma \vdash E\ :\ t$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ [\mathbf{VarT}]$$

Theorems:

- If $\Gamma \vdash E\ :\ t$, then $\Gamma(x) = t$
- If $\Gamma \vdash n\ :\ t$, then $t =$ number for any integer n, and similarly for true and false
- If $\Gamma \vdash E?E_1 : E_2\ :\ t$ then $\Gamma \vdash E\ :$ boolean, $\Gamma \vdash E_1\ :\ t$ and $\Gamma \vdash E_2\ :\ t$
- If $\Gamma \vdash E(E_1, \ldots, E_n)\ :\ t$ then there exists types $t_1, \ldots, t_n$ such that $\Gamma \vdash E\ :\ t_1 * \cdots * t_n > t$ and $\Gamma \vdash E_1\ :\ t, \ldots, \Gamma \vdash E_n\ :\ t$

Above means we can infer the type of a given expression by looking at the form of it. Some languages exploit this like python. It carries out type inference and calculates the required type declarations. Substituting a name by an expression of the same type does not affect typing.
Typing Relation on Statements. Handle all type declarations before the typing of the other statements can be performed. Type of the body needs to coincide with the declared return type of the function.

Type safety of Typed Source

Statement S is well-typed if there is a type t such that $S : t$. Well-typedness requires that a well-typed statement has no free names.
Typing is not affected by jumk in the type assignment. If $\Gamma \vdash T\ :\ S : t$, and $\Gamma \subset \Gamma'$, then $\Gamma' \vdash S : t$

Guarentees

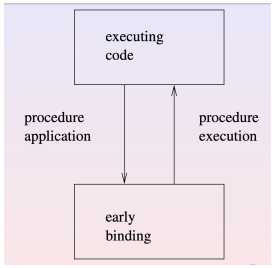
A PL with a given typing relation and one step-evaluation is type-safe if the both conditions hold:

- Preservation - if S is a well typed program and evaluates to S' in one step, then S' is also well typed with respect to the relation. Declared type will be guarenteed at the end.
- Progress - if S is a well-typed program, then either S is a value or there is a program S', which is a one step evaluation of S'. Guarenteed not to get stuck because of the return types.

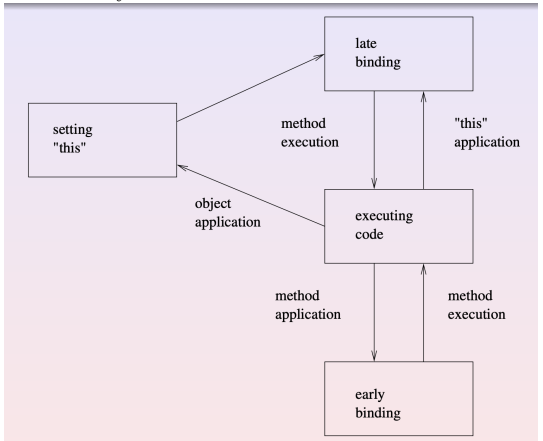
Representing Objects

Aggregation. Object within an object.
Classification. Creation of classes.
Specialisation. Inheritance.
Control Flow. Give control to other methods and objects when they are invoked.

Bindings



Early Binding. Program knows what to run before actually running it. Functions are known at compile time already.



The use of `this` keyword means the compiler does not know what the object is referring to at compile time. Only at runtime, then the object will be resolved. `new` keyword creates late binding. `lookup`

basically looks up whether a method is present for a particular class and gets the implementation of it.
Inheritance. Just a pointer to the parent. If `lookup` fails at the current class, go up to parent and return that one instead.

- Target function only determined at runtime
- Function `lookup` follows line of the class of the given object until a class is found that contains a method given a particular property
- Looking up can be a bottleneck if the inheritance list is long.

Inline caching. Classes unlikely to change but the objects themself do. Cache the classes with its associated address using machine instructions. Check the cache if present before performing lookup operation.

Syntax

Classes. A variable that contains a set of methods.
Inheritance. Add a field to the object which contains the parent **Object application.** Perform a lookup and apply the parameters

Logic Programming

`salary(list("name")), 100` The first symbols are the kind of informatin, the arguments are lists and the literals are the data. These are **assertions**.

```
// Query input:
job($x, list("computer", "programmer"))

// Query results:
job(list("Hacker", "Alyssa", "P"),
    list("computer", "programmer"))

job(list("Fect", "Cy", "D"),
    list("computer", "programmer"))
```

The query means list all the personnel who has a job as a computer programmer. `$x` is known as a logic variables. Some query inputs:

- `address($x, $y)` - find everyone whos name dont match their address.
- `address($x, $y)` - find everyone whos name is their address.
- `job($x, list('computer', $type))` - match everyone who has the word computer as the first word in their job.

Matching. Assertion matches a query if we can instantiate the query's logic variables with data and obtain the assertion. The matching must match in terms of length, one parameter to one word only.
Pair. This keyword allows the one parameter to match to multiple.
Query processing steps:

- Find all assignments to variables in the query pattern that satisfy the pattern. (the kind of info specified in the pattern needs to match the kind of info in an assertion, the assertion must result from the pattern by instantiating the pattern variables with values)
- system responds to query by listing all instantiations of the query pattern with the variable assignments that satisfy it

If the pattern has no variables, the query reduces to a determination of whether that pattern is in the database. The empty assignment satisfies that pattern for that data base.
And queries. All queries must be satisfied by the result.
Or queries. One query must be satisfied by the result.
Not queries. Negation, results that don't satisfy the query.
`javascript_predicate.` Queries that satisfy the predicate, something like a `where`.

Rules

`rule(conclusion, body)`, `conclusion` is a pattern and `body` defines what the pattern does. The query result would be `rule(conclusion)`. Can be used in compound queries and recursively as well.

Logic as Programs

`append($x, $y, $z)` Append x to y to form z, original lists are unchanged. Rules can be used interestingly as well.

- `append(` $1, 2$ `, $y,` $1, 2, 3$ `)` - returns possible values of y that forms the resulting list
- `append($x, $y,` $1, 2, 3$ `)` - returns possible values of x and y that can form the resulting list.

Query system Implementation

Maintain a frame that keep track of bindings to logic variables. Underlying implementation quite similar to logic gates. Each query is a block and the database response to each query individually, combine based on the type of query.
Unification. For 2 patterns is to find a frame that makes the patterns equal. Results in **an extension** of the original frame. Wrt extended frame, evaluate the query.

Interoperability

Programs communicate with each other through interfaces. Messages need to pass from one language to anther. Data structures need to be communicated between languages too.
Message-passing interfaces. Messages to target system are represented as data structures in source system. Basically communication via some API. Agree on message format. **Advantage:** minimal/no programming language support required. **Disadvantage** Cumbersome on remote system, receiver may not know how to handle message (possible solution is to accomodate to simplest format, or just call it unsupported), low effiency due to marshalling.
Interpretation of Target Language. Source system implements interpreter of target language. **Advantage:** any data structure can be communicated, easy to program. **Disadvantage:** effort to implement interpreter and interpreter overhead.
Language-level Integration. Use source language implementation to get to target language, possible when source language is implemented in target language. Link executables by using property of VM at runtime. Java implemented in C, can make use of C functions. When VM implemented in target language. **Advantage:** As efficient as source language, full support of data structure visibility (both ways), easy to implement. **Disadvantage:** Only possible when source language is implemented in target language.