**1. Run-Time Environments**

• What are Activation Records (AR) and what are they used for?

* Activation Records are Run-Time structures to hold State regarding the execution of a procedure.
* Activation Records are run-time data structures used to manage the execution of functions or procedures in a program. Each time a function or procedure is called, a new activation record is created and added to the call stack. **(ChatGPT)**
* ARs are used so run-time environments can manage the execution of functions and procedures efficiently, keeping track of necessary information for each active function call. **(ChatGPT)**

• What information does an AR capture and where are they allocated and why?

* Each activation record contains:
  + parameters – space for parameters to the called/current routine
  + register save area – saved register contents
  + return value – if function, space for return value
  + return address – address to resume caller
  + access link – help with non-local access
  + caller’s ARP – to restore caller’s ARP on a return
  + local variables – space for local variables & variables (including spills)
* Where are they allocated?
  + “In general, the ARs cannot be allocated statically whenever they involve procedures that are either directly recursive or mutually recursive”
  + if lifetime of AR matches lifetime of procedure AND code normally executes a “return”:
    - keep ARs on a stack
  + if procedure can outlive its caller OR it can return an object that can reference its execution state
    - ARs MUST be kept in the Heap
  + if a procedure makes no calls
    - ARs can be allocated statically
  + Efficiency prefers Static, Stack, then Heap
* ***and why?***

• Support for Object-oriented languages: class structure and inheritance.

* If A is a superclass of B, then B is a subclass of A and any method defined in A must operate correctly on B, but not the other way round.
* Static Class Structure
  + can map name to code at compile time
  + lead to 1-level jump vector
  + A screen shot of a computer program

    Description automatically generated with low confidencecopy superclass methods
  + fixed offsets & indirect calls
  + less flexible & expressive
  + In this static class structure, the relationships between the classes (Circle and Rectangle inheriting from Shape) are determined during compilation. The compiler can generate code based on this static class structure, allowing for efficient method dispatch and memory layout. **(ChatGPT)**

A screen shot of a computer

Description automatically generated with medium confidence

* Dynamic Class Structure
  + cannot map name to code at compile time
  + multiple jump vector (1/class)
  + must search for method
  + run-time lookups caching
  + much more expensive to run
  + The dynamic nature of the class structure allows for flexibility and adaptability. Objects can be created based on these dynamic classes, and method dispatch is resolved dynamically at runtime. **(ChatGPT)**
* Single Inheritance & Dynamic Dispatch
  + Use **prefixing** of tables (image)
* A diagram of a table

  Description automatically generated with low confidenceMultiple Inheritance
  + The idea:
    - allow more flexible sharing of methods and attributes
    - relax the inclusion requirement: “If B is a subclass of A, it need not implement all of A’s methods”
  + Problems with C inherits from both A & B:
    - C’s method table can extend A or B, not both
    - Other classes, ex. D, can inherit from both C and B
    - Both A and B might provide fum(), which is seen in C?
  + A diagram of a table

    Description automatically generated with low confidenceSolution:
    - Use **prefixing** of storage (image)
    - casting:
      * Usage of Point:
        + no extra action (prefixing does everything)
      * Usage of CThing:
        + increment self by 12
      * Usage of CPoint:
        + lay out data from Cthing and self + 12
        + when calling the “rev” method:

call in table points to a “trampoline” function that adds 12 to self, then calls rev

ensures that “rev”, which assumes that self points to a CThing data area, gets the right data (and the correct layout offsets)

**2. Register Allocation and Instruction Scheduling**

• The significance and importance of register allocation (and assignment)

* product correct code that uses k (or fewer) registers
* minimize added loads and stores
* minimize space used to hold spilled values
* operate efficiently
* at each point of code:
  + allocation: determine which values will reside in registers
  + assignment: select a register for each such value
  + Goal: allocaiton the “minimizes” running time
* importance of register allocation:
  + Use of one of the most critical processor resources
    - affects almost every statement of the program
    - register accesses are much faster than memory accesses
      * eliminates expensive memory instructions
      * wider gap in faster newer processors
      * number of instructions goes down due to direct manipulation of registers (no need for load and store functions)
  + what can be put in registers?
    - scalar variables
    - big constants
    - some array of elements and record fields
    - register set depending on the data-type
      * floating point in FP registers
      * fixed-point in integer registers
  + Allocation of Variables (including temporaries) up-to-now stored in Memory to Hardware Registers
    - pseudo or virtual registers
      * unlimited number of registers
      * space is typically allocated on stack with the stack frame
    - hard registers
      * set of registers available in the processor
      * usually need to obey some usage convention

• Global and Local Allocation Algorithms (top-down and bottom-up)

* top-down (ex. “Register Allocation - Introduction and Local Allocators” slide, page 16)
  + just put in register names that occur more ofter
  + estimate the benefits of putting each variable in a register in a particular basic block
    - benefit(v,b) = number of uses and defs of the var V in basic block B
  + estimate the overall benefit
    - TotBenefit(v) = Benefit(v,b)\*frequency(b) for all basic block B
    - If freq(b) not known, use 10depth where depth represents the nesting depth of B in the CFG of the code. **(n percebi)**
  + assign the r-feasible highest-payoff variables to registers
    - reserve feasible registers for basic calculations and evaluation
    - rewrite the code inserting load/store operation where appropriate
* bottom-up (ex. “Register Allocation - Introduction and Local Allocators” slide, page 38)
  + evaluate each instruction and keep track of when values are needed later on
  + basic idea:
    - focus on the needs of each instructions in a basic block
      * ensure each instruciton can execute
      * instruction operands and results in registers
    - transitions between instructions
      * observe which values are used next – in the future
  + on-demand allocation:
    - iterate through the instructions of a basic block
    - allocate the value of the operand, if not already in register
    - allocate register for result
    - when out of registers:
      * release register whose value is to be used farthest into the future
      * dirty register value requires memory operation to update storage
    - def: a maximal sequence of instructions such that:
      * only the first instruction can be reached from outside the basic block
      * all the instructions are executed consecutively iff the first instruction is executed
        + no branch or jump instructions in the basic block
        + except the last instruction
        + no labels within the basic block
        + except before the first instruction
    - basic block – algorithm (ex: “Register Allocation - Introduction and Local Allocators” slide, page 31)
      * input: sequence of three-address instructions
      * output: a list of basic blocks
      * algorithm:
        + determing the set of leader instructions – the head of each basic block – using the following:

the first statement of the program is a leader

any statement that is the target of a goto (either conditional or not) is a leader instructions

any statement that immediately follows a goto or unconditional goto statement is a leader instruction

* + - * + for each leader instruction, its basic block consists of the leader instruction and all the statements up to but not incuding the next leader instruction or the end of the program
  + details:
    - instructions in format: vrz <- vrx op vry using virtual registers or temps
    - data structures:
      * number of physical registers available
      * virtual name associated with each physical register
      * for each virtua name a distance to the next use in the basic block
      * flag indicating if the corresponding physical register is in free or in use
      * a stack of free physical registers with a stack pointer (integer index)
    - auxiliary functions:
      * ensure(vrx) allocates a physical register to ensure storage for vrx
      * free(vrx) releases the phyiscal register holding vrx
        + allocate(vrx) just sets some flags claiming register is being used
      * dist(vrx) returns the distance to the next reference to vrx in the basic block

• DU-chains and webs and their interference

* webs (ex. “Register Allocation - Global Allocators” slide, page 7)
  + divide accessess to a variable into multiple webs
    - all Definitions that reach a Use are in the same Web
    - all Uses of a Definition are in the same Web
    - divide the Variable into Live Ranges
  + implementation: use DU (Def-Use) chains
    - a DU-chain connects a definition to all uses reached by each definition
    - a web combined DU-chains containing a common use
  + in two Webs of the same Variable:
    - no use in one web will ever use a value defined by the other web
    - thus, no value need to be carried between webs
    - each web can be treated independently as values are independent
  + Web is used as the Unit of Register Allocation
    - if a web is allocated to a register, all the uses and definitions within that web don’t need to load and store from memory
    - solves the issue of cross Basic Block reigster assignment issue
    - different webs may be assigned to different registers or one to register and one to memory
* Interference (ex. “Register Allocation - Global Allocators” slide, page 18)
  + A blue and green squares

    Description automatically generated with low confidencetwo webs interfere if their live ranges overlap in execution time
    - what does this mean precisely?
    - there exists an instruciton common to both ranges where
      * their variable values of webs are operands of the instruction
      * if there is a single instruction in the overlap
        + and the variable for the web that ends at that instruction is an operand
        + and the variable for the web that starts at the instruction is the destination of the instruction
      * then the webs do not interfere
    - non-interfering webs can be assigned to the same register
  + interference graph (ex. same slides, page 20)
    - representation of Webs and their interferance
      * nodes are the webs
      * an edge exists between two nodes if they interfere

• Global Register Allocation via graph coloring

* each Web is allocated a Register
  + each node gets a register (color)
* if two webs interefere they cannot use the same register
  + if two nodes have an edge between them, they cannot have the same color
* heuristics for register coloring (ex. same slides, page 36)
  + coloring a graph with N colors
  + if degree < N (degree of node = # of edges)
    - node can always be colored
    - after coloring the rest of the nodes, you’ll have at least one color left to color the current node
  + if degree >= N
    - still may be colorable with N colors
    - exact solution is NP complete
  + remove nodes that have degree < N
    - push the removed nodes onto a stack
  + if all nodes have degree >= N
    - find a node to spilll (no color for that node)
    - remove that node
  + when empty, start the coloring step
    - pop a node from stack back
    - assign it a color that is different from its connected nodes (since degree < N, a color should exist)
* spilling and splitting (ex. same slides, page 67)
  + when the graph in non-N-colorable
  + select a web to spill
    - find the least costly web to spill
    - Use and Defs of that web are read and write to memory
  + split the web
    - split the web into multiple webs so that there will be less interferance graph making it N-colorable
    - spill the value to memory and load it back at the points where the web is split
  + splitting
    - identify a program point where the graph is non R-colorable (point where # of webs > N)
      * pick a web that is not used for the largest enclosing block around that point of the program
      * split that web
      * redo the interferance graph
      * try to re-color graph
    - cost and benefit of splitting
      * cost of splitting node
        + proportion to number of time splitted edge has to be crossed dynamically
        + estimate by its loop nesting
      * benefit
        + increase colorabiltiy of the nodes the splitted web interferes with
        + can approximate by its degree in the interferance graph
      * greedy heuristic
        + pick the live-range with the highest benefit-to-cost ratio to spill
  + more optimizations
    - register coalescing
      * find register copy instructions sj = si
      * if sj and si do not interfere, combine their webs
      * pros:
        + similar to copy propagation
        + reduce the number of instructions
      * cons:
        + may increase degree of combined node
        + colorable graph may become non-colorable
    - register targeting (pre-coloring)
      * some variables need to be in special registers at specific points in the execution
        + first 4 arguments to a function
        + return value
      * pre-color those webs and bind them to the appropriate register
      * will eliminate unnecessary copy instructions
    - pre-spilling of the webs
      * some ranges have very large “dead” regions
        + large region where the variable is unused
      * break-up the ranges
        + need to pay a small cost of spilling
        + but the graph will be very easy to color
      * can find strategic locations to break-up
        + at a call site (need to spill anyway)
        + around a large loop nest (reserve registers for values used in the loop)
    - inter-procedural register allocation
      * saving registers across procedure boundaries is expensive
        + especially for programs with many small functions
      * calling convention is too general and inneficient
      * customize calling convention per function by doing inter-procedural register allocation

• Instruction Scheduling: List scheduling algorithms **(n sai)**

• Interdependency between instruction scheduling and register allocation **(n sai)**

**3. Control-Flow Analysis**

• Definition and rationale of basic block construction

* def: a maximal sequence of instrucitons such that:
  + onlythe first instruction can be reached from outside the basic block
  + all the instructions are executed consecutively iff the first instruction is executed
    - no branch or jump instructions in the basic block
    - except the last instruction
    - no labels within the basic block
    - except before the first instruction
* algorithm: (ex. “Introduction to Optimization - Control-Flow Analysis”, page 53)
  + input: sequence of three-adddress instructions
  + output: a list of basic blocks
  + algorithm:
    - determine the set of leader instructions – the head of each basic block – using the following:
      * the first statement of the program is a “leader”
      * any statement that is the target of a goto (conditional or not) is a “leader instructions”
      * any statement that immediately follow a goto or unconditional goto statemente is a “leader instruction”
    - for each leader instruction, its basic block consists of the leader intruction and all the statements up to but not incuding the next leader instruction or the end of the program
* control flow grah (cfg)
  + control-flow graph G = <N, E>
  + nodes(N): basic blocks
  + edges(E): (x,y) ∈ E iff first instruciton in the basic block y follows the last instruction in the basic block x
    - first instruction in y is the target of branch or jump instruction (last instruction) in basic block x
    - first instruction of y is next after the last instruction of x in memory and the last instruction of x is not a jump instruction
  + block with first instruction of the procedure is the entry node (block with the procedure label)
  + the block with the return instructions are exit nodes
    - can make a single exit node by adding a special node

• Dominators and an iterative algorithm to compute them

* Node x Dominates Node y (x dom y) if every possible execution path from entry to node y includes node x (ex. same slide, page 65)
* way to find dominators of all nodes (same slide, page 78)
* computing dominators:
  + a dom b iff
    - a = b
    - or, a is the unique immediate predecessor of b
    - or, a is a dominator of all immediate predecessor of b
  + algorithm: (ex. same slide, page 85)
    - make dominator set of the entry node itself
    - make dominator set of the remained node to be all graph nodes
    - visit the nodes in any order
    - make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
    - repeat until no change
* back edges (ex. same slide, page 117)
  + an edge(x,y) is a back edge if “y dom x”
    - if y is in dominator set of x, then it’s a back edge
* natural loop
  + in a CFG, a back edge induces a natural loop
  + finding the nodes of a loop
    - given back edge (s,d)
    - traverse backwards (against flow) from d
    - until reaching s
    - collected nodes in CFG form natural loop

• Introduction to concepts of program optimization: constant propagation and the

use of dominance for correctness reasoning (tou a sentir q faltam cenas)

* implementing constant propagation (ex. same slide, page 41 (?))
  + find an RHS expression that is a constant
  + replace the use of the LHS variable with the RHS constant given that:
    - all paths to the use(s) of LHS variable pass through the assignment to the LHS with the constant
    - there are no intervening definition of the RHS variable
  + need to know the “control-flow” of the program

**4. Data-Flow Analysis**

• What is Data-Flow Analysis and its general iterative algorithmic framework

* iterative data-flow analysis algorithm in action (“Data-Flow Analysis - Introduction and Examples” slide, page 39 (?))
  + initialize all the nodes to a given value
  + visit nodes in some order
  + calculate the node’s value
  + repeat until no value changes (fixed-point computation)
* what is data-flow analysis
  + “a collection of techniques for compile-time reasoning about the runtime flow of avalues in a program”
* local analysis
  + analyze the “effect” of eahc instruction in each basic block
  + compose “effects” of instructions to derive information from beginning of basic block to eahc instruction
* data-flow analysis
  + iteratively propagate basic block information over the control-flow graph until no changes
  + calculate the final value(s) at the beginning/end of the basic block
* local propagation
  + propagate the infromation from the beginning/end of the basic block to each instruction

• Basic examples of available expressions and algorithmic implementation using

the concepts of Gen and Kill sets as well as Transfer functions and control-flow

merging functions (ns onde ta isso?)

* available expressions (ex. same slide, page 46)
  + an expression if available at point p if and only if
    - all paths of the execution reaching the current point p pass through the point where the expression was defined (the definition point)
    - no variable used in the expression was modified between the definition point and the current point p
  + in other words, expression is still current at p
* gen and kill sets (ex. same slide, page 69)
  + gen set
    - if a basic block (or instructioon) defined the expression then the expression number is in the gen set for that basic block (or instruction)
  + kill set
    - if a basic block (or instruction) (re)defined a variable in the expression then that expression number is in the kill set for that basic block (or instruction)
    - expression is thus not valid after that basic block (or instruction)
* algorithm for available expression
  + assign a number to each expression in the program (ex. same slide page 65)
  + compute gen set and kill set for each basic block (or instruction)
    - compute gen set and kill set for each instruciton in basic block
    - compose them to create basic block gen and kill sets
  + compute aggregate gen and kill sets for each basic block (ex of aggregate gen and kill sets, same slides, page 74)
  + initialize available set at each basic block as follows: (ex. same slide, page 105)
    - IN and OUT as the entire set (universe of the set and expressions)
    - exception: IN != 0 for first basic block
  + iteratively propagate available expression set over the CFG (ex. same slide, page 107)
  + propagate solution within basic block (same slide, page 139)
* transfer functions (ex. “Data-Flow Analysis: Iterative Frameworks and Examples” slide, page 27)
  + how each instruction (statement) and control-flow construct affects the abstract quantities V
  + ex. the OUT equation for each statement
  + F: V -> V properties:
    - F has an identity function I, such that I(x) for all x ∈ V
    - F is closed under composition
      * ex. for any two functions f and g ∈ F, the function h(x) = g(f(x)) ∈ F

• Basic properties of transfer functions: commutativity, associativity and the semilattice of values. Termination and speed.

* semilattice (faltam cenas) (ex. “Data-Flow Analysis: Iterative Frameworks and Examples”, page 7)
  + is a set L and a meet operation ^ such that,
    -  a, b and c € L:
      * a ^ a = a
      * a ^ b = b ^ a
      * a ^ (b ^ c) = (a ^ b) ^ c
    - ^ imposes an order on L, a, b and c € L:
      * a >= b <=> a ^ b = b
      * a > b <=> a >= b and a != b
    - a semilattice has a bottom element, denoted ꓕ :
      *  a € L, ꓕ ^ a = ꓕ
      * a € L, a >= ꓕ



* termination
  + if the semilattice of the framework is monotone and of finite height, then the algorithm is guaranteed to converge (termination)
* speed
  + if a data-flow framework meets these admissibility conditions then it has a unique fixed-point solution
    - the iterative algorithm find the (best) answer
    - the solution does not depend on the order of computation
    - algorithm can chose an order that converges quickly
  + intuition:
    - choose an order so that changes propagate as far as possible on each “sweep” or “pass” over the CFG
      * process a node’s predecessors before the node
    - cycles pose problem, naturally
      * ignore back edges when computing evaluation order
      * easily done with “visit” flag

• Live-Variable analysis and Copy-Propagation formulations as examples of Backward and Forward problem formulations

* copy propagation (ex. “Data-Flow Analysis: Constant PropagationFicheiro”, page 23)
  + bypass multiple copying
    - propagate a value directly to its use
  + advantages
    - leads to further algebraic simplification (page 33)
    - reduces instructions by eliminating copy operations (page 36)
  + how to perform copy propagation? (ex. page 42, page 77)
    - at each RHS expression and for each variable “v” used in the RHS expression
      * if the variable “v” is defined by a statement of the form “v = u”
      * repllace the variable “v” by “u”
    - at each point of te program need to know
      * the variables that are equal
      * track equal variables by keeping tuples of the form <u, v> in a set iff v = u at that point in the program (u, v are variables)
    - an assignment of v = u is still valid at a given point of the exectuion if and only if
      * a statement of v = u occurs in every execution path that reaches the current point
      * the variable “v” is not redefined in any of these execution paths between the assign statement and the current point
      * the variable "u” is not redefined in any of these execution paths between the assign statement and the current point
* live-variable analysis
  + what is it?
    - for each variable x where is the last program point p where the specific value of x is used
    - ini other words, for x and program point p determine if the value of x at p can still be used along some path starting at p
      * if so, x is live at p
      * if not, x is dead at p
  + live-variable analysis: illustration (backwards)
    - at point p0 the x variable is live
      * there is a path to p1 where value at p0 is used
      * beyond px towards p2 the value of x is no longer needed and is dead
    - need to observe for each variable and for each program point:
      * where is the last program point beyond which the value is not used
      * trace back from uses to definitions and observe the first definition (backwards) that reaches the sue
      * that definition kills all uses backwards of it
    - data-flow analysis formulation (ex. “Data-Flow Analysis: Live-Variable Analysis”, page 4)
      * Variable is live at a point p if its value is used along at least one Path
        + a use of x prior to any definition in basic block mean x must be alive
        + a definition of x in B prior to any subsequent use means previous uses must be dead
      * gen set: set of variables used in B
        + upward exposed reads of B
      * kill set: set of variables defined in B
      * initialize IN(B) to empty set
      * compute Gen/Use and Kill/Def for each basic block
        + tracing backwards from the end of block to beginning of block
        + initialize last instruciton’s Out(i) to empty
        + use IN(i) = use(i) ∪ (OUT(i) – def(i))
      * iteratively apply relations to basic block until convergence
        + OUT(B) = ∪(for each S a successor of B) IN(s)
        + IN(B) = Use(B) ∪ (OUT(B) – Def(B))
      * given OUT(B) use relations at instruction level to determine the live variables after each instruction
  + Use and Def function for a basic block (forward)
    - can be accomplished by a forward scanning of the block
      * keep track of which variables are read before they are written thus computing the Upwards Exposed Reads (UpExp) or Use functioon
      * track variables that are written or killed (VarKill) or Def function
    - algorithm: page 21
      * ex. page 22

**5. Global Loop-Level Optimizations**

• Loop invariant code motion (ex. “Combining it All - Loop Optimizations”, page 25)

* if a computation produces the same value in every loop iteration, move it out of the loop (ex. “Combining it All - Loop Optimizations”, page 4)
* Opportunities for LICM
  + in user code
    - complex expressions
    - easily readable code, reduce number of variables
  + after compiler optimizations
    - copy propagation, algebraic simplification
* usefulness of LICM
  + many programs spend most of their execution time in loops
  + reducing work inside loop nest is very beneficial
    - CSE of expression => x instructions become x/2
    - LICM of expressions => x instructions become x/N
* implementing LICM
  + If a computation produces the same value in every loop iteration, move it out of the loop
  + an expression can be moved out of the loop if all its operands are invariant in the loop
* invariant operands
  + constant values
  + variables whose definitions are outside the loop
  + operand has only one reaching definition and that definition is loop invariant

• Induction variable recognition

* what is an indution variable?
  + for a given loop variable v is an induction variable iff
    - its value changes at every iteration
    - is either incremented or decremented by a constant amount
      * either compile-time known or symbolically constant
* classification:
  + basic induction variables
    - a single assignment in the loop of the form x = x + constant
    - ex. variable i in “for i = 1 to 10”
  + derived induction variables
    - a linear function of a basic induction variable
    - ex. variable j in the loop assigned “j = c1 \* i + c2”
* detection of induction variables algorithm: (ex. same slide, page 51)
  + inputs: loop L with reaching definition and loop invariant
  + output: for each unduction variable j the triple (i,c,d) so that the value of j = i \* c + d
  + find the basic induction variables by scanning the loop L such that each basic induction variable has (i,1,0)
  + search for variables k with a single assignment to k of the form:
    - k = j \* b, k = b \* j, k = j / b, k += j with b a constant and j a basic induction variable
  + check if the assignment dominates the definition points for j

• The Role of dominators in code movement

* statement can be moved only if
  + all the uses are dominated by the statement
  + the exit of the loop is dominated by the statement

• Putting it all together (??)