Gathered notes from:

- Modern Compiler Implementation in C [1]
- High Performace Compilers for Parallel Computing [3]
- The implementation of Functional Programming Languages [2]

1 Parsing

Common types of parsers:

- LL(k): left to right, left-most derivation, k token lookahead
- LR(k): left to right, right-most derivation, k token lookahead

1.1 Utility helpers

- `First(x)`: set of all terminal symbols at the front of strings derivable from `x`
- `Nullable(x)`: true if empty string is derivable from `x`, false otherwise
- `Follow(x)`: set of of all terminal symbols that can immediately follow `x`

Constructing the above using iterative fixed point algorithm:

Algorithm 1: Fist/Follow/Nullable Computation

```
for i ∈ symbols do
    if terminal(i) then
        first[i] = i
    else
        first[i] = {}
    null[i] = false
    follow[i] = {}
while true do
    for (x → y_0 y_1 y_2 y_n) in productions do
        if (∀i ∈ [0..n]) null[y_i] then
            null[x] ← true
        if (∀i ∈ [0..k)) null[y_i])∧ !null[y_k] then
            first[x] ← first[x] ∪ first[y_k]
        for k ∈ [0..n] do
            if (∀i ∈ [k+1..n]) null[y_i] then
                follow[y_k] ← follow[y_k] ∪ follow[x]
        for k ∈ [0..n] do
            if (∀i ∈ [k+1..j) null[y_i] ∧ !null[y_j] then
                follow[y_k] ← follow[y_k] ∪ first[y_j]
    if no change in first/follow/null set then
        break
```

1.2 LL(1)

Implementation can use recursive descent. For LL(1): 1st terimnal symbol of subexpression has enough information to pick production rule to use.

Recursive descent: mapping a function for each production rule.

If there exists overlapping symcols in First set for different production rules, then the algorithm cannot be handled.

Solution: left factor: create auxilliary intermediate symbols to remove left recusion (convert to right recursion).

For production rules with same starting symbols, Take the different endings of the production rules and create a new symbol for them.

$$S \to if\ A\ then\ B\ else\ C$$
$$S \to if\ A\ then\ B$$

to:

$$S \to if\ A\ then\ B\ D$$
$$D \to else\ C$$
$$D \to \emptyset$$

Another example:

$$E \to T$$
$$E \to E + T$$

to:

$$E \to T\ E_2$$
$$E_2 \to +\ T\ E_2$$
$$E_2 \to \emptyset$$

1.3 LR(0)

Characterized by:
Use of stack of symbols processed, optionally use stack of states for caching (avoid scanning all elements in symbol stack to determine state). Apply DFA to the stack to determine action (shift/reduce). Shift-reduce conflict exists $\implies$ cannot be processed by parser.

Definitions:
Item $\equiv$ a grammar rule and dot (position on RHS of the rule)
State $\equiv$ a set of items

Actions using DFA applied to state stack:

- shift(n): advance 1 input symbol and push state n onto the stack
- reduce(x): RHS match grammar rule x; pop states off from stack as many times as number of symbols on RHS of rule x. LHS symbol x of the rule is the current symbol.
- goto(n): look at current state on top of the stack and the current symbol s (from previous reduction) to get transition to the next state n (push state n onto the stack)
- accept: end of successful parse
- error: Ø/empty entry transitioned

Operations on State (I):

- closure(I): adds more items when there are more matching items (dot is to the left of a non-terminal)
- goto(I, X): move dot past all X in all items in set I, where X is a grammar symbol (non-terminal)

1.3.1 Algorithms

Algorithm 2: LR(0) Parser Table Construction

```
//augment to include a starting production (S' → .S$),
  where S is the original top level symbol
let T = {Closure(S' → .S$)} //a set of states
let E = {} //set of shift or goto edges
let changed = true
while changed do
    let T1 = {}
    let E1 = {}
    for state I ∈ T do
        for item (A → α.Xβ) ∈ I do
            T1 ← T1 ∪ { Goto(I, X) }
            E1 ← E1 ∪ { I --X--> J }
    changed = T1 ≠ T ∨ E1 ≠ E
    T ← T1
    E ← E1
//calculate reduce actions:
R ← {}
for state I ∈ T do
    for item (A → γ.) ∈ I do
        R ← R ∪ {(I, A → γ)}
//construct parser table:
for (I --X--> J) ∈ E do
    X is a terminal ⟹ shift J at (I, X)
    X is a non-terminal ⟹ goto J at (I, X)
for state I ∈ T do
    for item ∈ state I do
        match item {
         (S' → S.$) ⟹ accept at (I, $)
         prod n (A → γ.) ⟹ reduce n at (I, Y) ∀ Y
        }
```

Algorithm 3: Goto

```
Input  : I(input State), X(a symbol)
Output: output State
Z: State = {}
for item (A → α.Xβ) ∈ I do
    Z ← Z ∪ {(A → αX.β)}
return Closure(Z)
```

Algorithm 4: Closure

```
Input  : I(input State)
Output: output State
let changed = true
while changed do
    let Z ← I
    for item (A → α.Xβ) ∈ I do
        for item (X → .γ) ∈ productions do
            Z ← Z ∪ {(X → .γ)}
    changed ← Z ≠ I
    I ← Z
return I
```

1.4 SLR

Modification of LR(0):
Add reduction action that takes acount of the Follow set:
(I, X, A $\rightarrow \alpha$), State I, top symbol X, reduce by $A \rightarrow \alpha$

Resulting parse table contains fewer reduction entries than tha tale for LR(0).

Algorithm 5: SLR Reduce Action Modification

//calculate reduce actions:
$R \leftarrow \{\}$
for *state* $I \in T$ **do**
  for *item* $(A \rightarrow \gamma.) \in I$ **do**
    for $X \in$ *Follow(A)* **do**
      $R \leftarrow R \cup \{(I, X, A \rightarrow \gamma)\}$

1.5 LR(1)

Idea: augment item to include a lookahead symbol x:
$(A \rightarrow \alpha.\beta, x) \iff$ sequence α is on the top of the stack, input is derivable from βx

Computation of Goto, Closure, and State also augmented by incoporating the additional lookahead symbol.

Algorithm 6: LR(1) Goto

Input : I(input State), X(a symbol)
Output: output State
Z: State = $\{\}$
for *item* $(A \rightarrow \alpha.X\beta, \gamma) \in I$ **do**
  $Z \leftarrow Z \cup \{(A \rightarrow \alpha X.\beta, \gamma)\}$
return Closure(Z)

Algorithm 7: LR(1) Closure

Input : I(input State)
Output: output State
let *changed* = true
while *changed* **do**
  let $Z \leftarrow I$
  for *item* $(A \rightarrow \alpha.X\beta, z) \in I$ **do**
    for *item* $(X \rightarrow .\gamma) \in$ *productions* **do**
      for $c \in$ *First*(βz) **do**
        $Z \leftarrow Z \cup \{(X \rightarrow .\gamma$, c$)\}$
  changed $\leftarrow Z \neq I$
  $I \leftarrow Z$
return I

Algorithm 8: LR(1) Construct Reduce Actions

Input : T(set of States)
Output: R(reduce actions)
$R \leftarrow \{\}$
for *state* $I \in T$ **do**
  for *item* $(A \rightarrow \gamma., \beta) \in I$ **do**
    //in state I with lookahead symbol β,
    // reduce by $A \rightarrow \gamma$
    $R \leftarrow R \cup \{(I, \beta, A \rightarrow \gamma)\}$

Augment the start state:
$\{(S' \rightarrow .S\$, @)\}$, where @ denotes end of file and will never be shifted.

1.5.1 Ambiguous Grammar

Shift-reduce conflicts: resolve by prioritizing one of them, or introduce intermediate non-terminals for matched statement and unmatched statement.

If it is not possible to infer type of variable during parsing, defer that until the semantic phase.

1.6 LALR(1)

Idea: reduce size of parse table by meerging LR(1) parse table with identical states, exclusing lookahead sets.

2 Concrete Syntax

todo

3 Abstract Syntax

Transformations to abstract syntax:
source $\rightarrow_{lexer}$ tokens $\rightarrow$ concrete parse tree $\rightarrow$ abstract syntax

concrete parse tree:

- represents conrete syntax of source language
- leaf for input token
- internal node for each grammar rule reduced
- may contain uninformative tokens that are not useful after parsing (contains extra non-terminals and intermediate production rules for technicality of parsing)

abstract syntax:

- isolation between parsing and semantic analysis via an interface
- parsing issues resolved when abstract syntax is obtained, even though grammar of abstract syntrax may be unfriendly to parsing
- discards some of the uninformation tokens present in conrete parse tree
- contains phrase structure of source program
- include source location info for error reporting (need to be propagated though tokenizer phase as well)
- may use symbols instead of strings for efficiency (convert it once and use symbols throughout the rest of the program)
- may need special coalescing for mutually recursive functions and types (make them into 1 type instead of separate ones)

4 Semantic Analysis, Type Checking

4.1 environments

Mapping of identifier to type or value (variable/function)

Use of predefined/base environments:

- base type environment: natively supported types
  eg: `int -> TyInt, string -> TyString`
- base value environment: predefined functions

As compilation continues, environments are:

- augmented
- queried for: type checking, intermediate code generation

Strategies for environment management:

- auxilliary stack, or
- threading nodes to achieve stack-like behaviour
- special token/node for delimiting scope under consideration

4.2 Semantic Module

Top level function: give it an abstact syntax for semantic module to process

Things that occur during semantic processing:

- use abstract syntax interface for manipulating nodes during semantic processing
- semantic checking on reserved words in language
- error message for mismatched types / undeclared identifiers

Use of mutual recursion to process different types of nodes in abstract syntax:

```
translate_var(..) -> expty
translate_exp(..) -> expty
translate_dec(..) -> expty
translate_ty(..) -> expty
```

where:

`struct expty { TrExp exp, Ty_ty ty }` is the result of type checking `TrExp` is the translated expression in intermediate code

May include intermediate code translation if type checking is combined with translation phase.

4.2.1 Type checking of different expressions/parts of abstract syntax

Declaration:
Augment env. with identifiers in initializing expression.
Optional type constraints in declarations checked with initializing expression for compatibility.

- Type Declaration
  simply augment type env. with `iden -> type mapping`
- Function Declaration
  - process formal param. list types and return type, augment type env. with:
    `iden -> func type params`
  - begin val env. scope
  - process value parameters and augment val env. with
    `param iden -> parameter type`

  - process the body of function resursively
  - exit the current val env. scope (pop items in env. until last env. scope token is reached and pop that as well)
- Variable Declaration
  - translate expression on initializing expression
  - obtain optional constraint: check against translated initializing expression's type
  - enter entry, `iden -> variable type`, into val env.
- Let Expression
  - begin val env. scope
  - begin type env. scope
  - process each declaration in let expression
  - process body of let expression
  - exit val env. scope
  - exit type env. scope
  - return translated expression of body of let expression
- Mutually Recursive Types and Functions
  - process headers on 1st pass, use placeholders and argument env.
  - process bodies on 2nd pass, using previously augmented environment
  - cycle checking for validity: cycle can only appear in fields of records or arrays

5 Stack Frame / Activation Record

- higher order function: function valued variable
- nesting of higher order functions:
  local variables need longer lifetimes then original enclosing function (need something more powerful than a stack data structure to hold prolonged variables)
- nested function is not supported or higher order function is not supported $\implies$ implementable using stack to store variables in instantiated functions on their entry
- for stackable language: use stack frame to store locals and temporaries of an instantiated function
- frame pointer: local origin of addressing for current function's stack frame
- stack pointer: boundary of current valid frame
- use of frame pointer for referencing offsets of function formal parameters / local variables
- conditions of a variable to be in a frame and not register-resident(*: condition of an escape variable):
  - pass by reference (need actual memory address) *
  - access of variable from an inner nested function *
  - large variable
  - address arithmetic on variable *
  - hardware register reserved for other purpose
  - lack of hardware register vacancy

5.1 Frame Module

Abstraction for implementation detail of frame layout depending on specific machines. Populates info related to frame that is specific to target machine) into some data structures.

Sample interface:

```
module Fr {
  fn new_frame(NameLabel, [formal_escape]) -> FrFrame;
  ..
}

trait FrFrame {
  fn get_frame_label(&self) -> NameLabel;
  fn get_formals() -> [FrAccess];
  fn alloc_local(escape: bool) -> FrAccess; //alloc var in frame
}

enum FrAccess {
  InFrame(isize), //in memory, relative offset from fp
  InReg(NameTemp), //in register
}
```

Also perform shift of view between caller and callee depending on calling convention of target machine (do this per formal parameter passed into new frame, eg when constructing a new frame):

- parameters seen from inside the function
- instruction to create shift of view to manipulate stack-pointer/framepointer, save/move of values

Concrete frame data structure contains info on:

- formal parameter locations
- view shift instructions
- locals allocated in frame

- label of where machine code for the function begins

Local variable allocation within frame:

- each declaration results in space reservation in frame (may be optimized out in later compiler phases)
- end of a scope disassociates names to allocated local variables in that scope

Calculating escape of a variable via mutual recursion depending on Abstract Syntax node type, record escape info in an environment, eg: True: escape, False: not escape.

Use this environment when processing expressions in nested scopes within the scope of the said variable. Mark escape to true for the variable when:

- encounter use of the variable
- address of variable taken explicitly or call by reference

5.2 Name Module

`NameLabel`: abstraction for location of procedure body / code address (static memory address)
`NameTemp`: abstraction for location of register associated with variable

Defer concrete assignments to later phases of compilation.

Interface:

```
module Name {
  fn new_temp() -> NameTemp; //auto-generate id
  fn new_label() -> NameLabel; //auto-generated id
  fn new_label(String) -> NameLabel;
  ..
}

trait NameLabel {
  fn get_label_str() -> String;
}
```

5.3 Layers of Abstractions

<table>
<tr><td colspan="2">Semant</td></tr>
<tr><td colspan="2">Translate</td></tr>
<tr><td>Frame</td><td>Name</td></tr>
</table>

- Semant: operates on AST level for type checking
- Translate: provides notion of scopes and static nesting levels, translation to intermediate representation for later phases
- Frame: view shift conventions depending on target machine
- Name: label and temp naming abstraction for deferred assignment

5.4 Environment

Information needed in an environment:

variable entry:

- Type
- `TrAccess`

function entry:

- Types of formal parameters
- Type of return value
- label of function
- `TrLevel`

```
enum EnvEntry {
  EntryVar((TrAccess, Type)),
  EntryFun((TrLevel, //static nesting level
            NameLabel,
            [TypeFormal],
            TypeReturn)
}
type TypeFormal = Type;
type TypeReturn = Type;
```

5.5 Translate interface

```
module Tr {
  fn new_level(parent: TrLevel,
               label: NameLabel,
               formal_escapes: [bool]) -> TrLevel;
  ..
}

impl TrLevel {
  fn get_formals() -> [TrAccess];
  fn alloc_local(escape: bool) -> TrAccess;
  ..
}

impl TrAccess {
  //static nesting level of declaration
  //used in query of variable in its frame of declaration
  TrLevel level,
  FrAccess access,
}
```

`Tr::new_level(..)`:

- augment formal parameter list with extra entry for static link (set to escape)
- calls `Fr::new_frame(..)` with label and formal parameter list, get back a `FrFrame` abstracted object (use it to access formals/allocated locals/etc.)

`Tr::alloc_local(TrLevel, escape: bool) -> TrAccess`:

- create `FrAccess`, via `Frame::alloc_local(..)` provided by Frame module
- return `(TrLevel, FrAccess)` wrapped in `TrAccess`

6 Translation to Intermediate Representation

Goals:

- lower to abstracted machine operation without committing too much to specific concrete machine detail
- agnostic to source languages
- allow different transformations and analysis to be performed on it before lowering to lower level machine language

Function bodies translated to IR.

Defer entry and exit of function to glue code that will take care of bookkeeping and respect stack/register conventions.

3 categories of expressions in IR:

```
enum TrExp {
  TrEx(TExp), //expression
  TrNx(TStm), //no result
  TrCx(Cx), //conditional with jumps
}

enum TExp {
  BinOp((TBinOp, TExp left, TExp right)),
  Mem(..),
  Temp(..),
  Call(..),
  ..
}

enum TStm {
  Seq((TStm left, TStm right)),
  Label(..),
  Jump(..),
  Exp(..),
  ..
}

struct Cx {
  PatchList trues, //defer filling in
  PatchList falses,
  TStm stm, //sequence of cjumps and labels
}
```

Casting between the above types of expressions may need insertion of additional instructions and temporaries. Helper functions:

```
tr_ex(TExp) -> TrExp;
tr_nx(TStm) -> TrExp;
tr_cx(Cx) -> TrExp;

un_cx(TrExp) -> Cx;
un_nx(TrExp) -> T_Stm;
un_ex(TrExp) -> TExp;
```

Translate module need to convert Abstract Syntax into IR types. Eg for a variable declared in stack: `Mem(BinOp(Plus, Temp fp, Const k))`

Conditional

- Composition of statements and labels.
- Use of patchlist to fill in jump destinations (places where labels need to be filled in) later.

TrExp structure in IR: union of expression, statement, conditional.

6.1 Translation of Types to IR

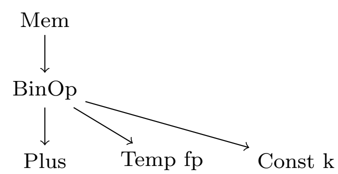
6.1.1 Translate Module Interface

- manipulation of IR nodes handled by Translate module
- Semant module is agnostic to IR nodes
- handles static nesting level for escaped variables
- has access to Frame module, which contains machine dependent definitions

6.1.2 Example for SimpleVar (variable declared in stack frame)

Semant module's `trans_var(..)` type-checks variable using type env. and variable env., returns `ExpTy` with `TrExp` and `TyTy`

IR translation phase adds additional info. into the `TrExp` structure:



Use Semant module to provide access of variable, level of function where variable is used to translate module function, and get back a `TrExp`

`TrSimpleVar(TrAccess, TrLevel) -> TrExp`

where:
`TrAccess` contains level of variable declaration
`TrLevel` is the level of variable use

Turning `FrAccess` (formals and local variables allocated in frame/register) into IR expression:

`FrExp(FrAccess, TrExp) -> TrExp`

where:
`TrExp` is the frame pointer where `FrAccess` lives, calculated via static links (variable may be in different static nesting level); ignored if variable is in register. `TrExp` is the generated IR with pointer and offset calculations

Calculating `TrExp` using static links for SimpleVar:

tr_exp = Mem(+(Const k_n, Mem(+(Const k_{n-1}, ..
  Mem(+(Const k_1, Temp fp))..)

where:
$k_1, .., k_n$ are static link offsets in nested functions
`fp` is the current frame pointer of where the variable is used

let l_f be the level of function f where variable is used
let l_g be the level of function g where variable is declared
then $l_f - l_g$ static link offsets are followed to calculate variable location where it is originally declared

6.2 Large Variable

Eg: arrays and records

Implementation variations:

- pointers: assignment means pointer assignment
- content as value: assignment means copying entire value

6.3 Structured l-value

large values that don't fit in 1 word
data structure may need additional info for size
eg: `Mem` operator needs size parameter:
`TrMem(TrExp, size) -> TrExp`
`Mem(+(Temp fp, Const k), S)`

6.4 Subscript/Field Selection

Compute offset from a base to get a component of interest

6.5 Array subscript expression

- l-value for base and smaller subranges
- l-value coerced to r-value via `Mem` operator in cases of: 1) pass by value, 2) assignment to another array variable

6.6 Language without structured l-value

Pointers are passed instead where base address of l-value object is the value stored in a pointer variable $\implies$ an additional `Mem` operator is required to "deref" the pointer.

accessing element of width w at index i of an array with base address stored in variable e:
`Mem(Plus((Mem e), BinOp(Mul, i, Const w)))`

analogous to `*(*e + i*w))` in C code

l-value is technically an address without `Mem` operator.

6.7 Error detection for out of bounds component access

- compile time checks desirable
- nullptr checks before deref

6.8 Convert if else to conditional jump

convert `if a then b else c` to conditional jump:

- use of temporary and true/false labels
- move instruction to temporary in branches
- use of `un_cx(a), un_ex(b), un_ex(c)`
- 1 final join label for branches
- eg: nested conditionals:
  `Seq(cx(s_1,z,f), Seq(z, cx(s_2,t,f)))`

6.9 String comparison

use of external runtime routine via `Call` node

6.10 Long lived objects allocated on the heap

1. call a runtime function to allocate space and return pointer to a temporary `r`
2. do a series of `Move`s using returned pointer and offsets to initialize large values
3. reading the expression is `Temp(r)`

6.11 Calling external runtime function

```
Call(Name(NameLabel("init_array")),
  TrExpList(a, TrExpList(b, null)))
```
$\iff$ `init_array(a,b);`

Possibly wrap in a helper in Frame module:
`Fr::external_call(f: String, args: TrExpList) -> TrExp`

6.12 Function call

`Call(l: NameLabel, args: [sl, e_1, e_2, ..])`
where:
`l`: label of function to call
`sl`: static link (address of frame of enclosing function)
`e_1, e_2, ...`: other normal arguments

6.13 Declarations

`trans_dec(..)` also side-effects the frame data structure:

- per variable declaration: additional space will be reserved in current frame
- per function declaration: new "fragment" of code will be kept for function body
- variable initialization / definition:
  - translates to an expression, put before the ody of `let` expression
  - return `TrExp` containing assignment expressions that accomplish initializations of variables
- function definition: translation to a segment of assembly language with:
  - prologue: setup instructions for function call
  - body: containing body expression translation
  - epilogue: teardown instructions for function call

6.13.1 prologue

- pseudo instruction for an assembly language
- label definition for function name
- instructions to adjust stack pointer in allocating new frame
- instructions to save escaping arguments into frame (including staic link)
- instructions to move non-escaping arguments into fresh temporary registers
- store instructions to save callee-save registers (eg: return address register)

6.13.2 Epilogue

- instruction to move return value to register reserved for the purpose
- load instructions to restore callee-save registers
- instruction to reset stack pointer (frame deallocation)
- return instructions (jump to return address)
- announce end of function to assembler

Many of these info (eg: exact frame size) are unknown until later, so these instructions need to be generated late in the compilation process

6.14 Procedure Fragments

Using:

- already translated function body expression, and
- function definition with a static nesting level

Translate phase should:

- produce a descriptor for function containing necessary information:
  - frame descriptor with machine specific info. about local variables and parameters
  - result returned from a helper function that does restore of registers and moving incoming formal parameters
- define an interface within Translate module via a frag datatype (this gets accumlated into a list during translation of procedures)

7 Basic Blocks, Traces

Goal: transform structure of tree nodes (IR) so that resulting trees are in a canonical form that can be further linearized and reordered without unsoundness

Strategy:

- pull ESEQs out of list of expressions/statements and lift them to the top of the tree; these get transformed into SEQs; linearize them into sequential instructions
- restrict CALLs to occur only under specific types of node: `EXP(..)`, `MOVE(TEMP(t, ..))`
- parent of a `SEQ` node can only be a `SEQ` node

7.1 Lifting ESEQ out of expressions

ESEQ(s, e) = e_{original}
where by definition of ESEQ, s is executed before e

use equivalence identities for transformations of nodes (TODO)

use of commutativity simplifies transformations but hard to know if it's applicable to an expression in general

7.2 Trace construction

Goal: generate a set of traces that cover the entire program

Sample algo:

- grow a trace from a selected basic block: when encountering a cjump/jump (at the end of the block), continue trace by selecting a possible path and adding the selected block to the current trace
- delete jump instruction afterwards
- trace stops growing when successors to current frontier block are already covered by existing trace(s)

7.3 General Procedure

1. linearize: list of statements (T_stm) → canonical trees (T_stmList): apply rewrite rules: ESEQs removed, CALLs restricted to be under nodes of certain types
2. group into basic blocks: canonical trees → basic blocks (these now can be reordered arbitrarily)
3. trace schedule: basic blocks → T_stmList: a set of traces covering entire program

7.4 Rewrite Rules

7.4.1 Reorder

`reorder([expr]) -> stm`

Goal

- pull ESEQs out of given list of expressions
- combine statement parts of ESEQs together into 1 sequence and return as statement
- rewire nodes with ESEQ child node to point to new expression as child instead

Delegate to `do_exp(..)`, `do_stm(..)`

7.4.2 Expr Rewrite

`do_exp(expr) -> ESEQ(stm',expr')`

Goal: pull out statements out of given expression and return ESEQ(s,e) where it is equivalent to original expression

Recursively call reorder when necessary depending on expression type.

7.4.3 Statement Rewrite

`do_stm(stm) -> stm`

Goal: pull out ESEQs out of a statement and reorder to return ordered statement

Recursively call reorder when necessary to apply rewrite rule on sub-expressions of the statement.

7.4.4 Linearize

`linearize(stm) -> [stm]`

Goal

- flatten parts of the tree previously processed by rewrite rules where all SEQ nodes are at the top
- return a list of statements where SEQs are eliminated

Apply `linearize(do_stm(..))` on body of target function to get back linearized statements.

Pass the resulting list of statements to constructor of basic blocks.

7.5 Basic Block Properties

Construct basic block(s) from list of linearized statements with the following invariants by adding labels, modifying jumps, starting new blocks as necessary.

- a label is at the start of the block
- a cjump/jump is at the end of the block
- no other jumps exist in any other part of the block

7.6 Trace Schedule

Goal

- reduce amount of jump overhead by combining basic blocks
- placement of false label to fall through for conditional jump

Strategies

- Reorder blocks so unconditional jumps are followed immediately by destination blocks and these blocks can be added to a same trace, so that jumps can be eliminated later.
- Reorder blocks so condition jumps are followed immediately by blocks with false label, invert conditional logic if necessary. This allows better mapping to branching instruction on most hardware where false label jumps can be eliminated later.

Sample iterative algo.: grow current trace with a work list of basic blocks and looking at uncovered successor blocks of current block.

8 Instruction Selection

Goal: cover the IR tree with a minimal set of instruction patterns that are non-overlapping, typically using an idealized cost model for selection

8.1 Sample Tiling Selection Algos

8.1.1 Maximal Munch

- Greedy, locally optimal
- Top down growth of tiles starting from the root
- sample strategy: select a tile of the largest size that is feasible and rooted at the current node (looking at node type)
- if there exists a tile for every node type of the tree, then algo will not get stuck
- process sub-expressions and statements recursively: `munch_exp(..)`, `munch_stm(..)`

8.1.2 Dynamic Programming

- Globally optimum
- bottom up traversal and compute minimal accumulated cost at current node: cost of a feasible tile rooted at current node plus subtree costs of its child tiles
- 2nd pass to emit instructions based on selected tiles in post order (emit instr for rooted nodes of child tiles, emit instr for the tile rooted at current node)

8.2 Register Allocation Pass

schedule register allocation pass after instruction selection means instruction selection needs to work without exact registers $\implies$ use an abstraction for instructions without register (Abstract Assembly Language Instructions), which is independent of target machine assembly language.

8.3 Abstract Assembly Language Instructions

```
enum AsInstr {
  Oper(
    asm: string,
    dst: [NameTemp],
    src: [NameTemp],
    jumps: [NameLabel]),
  Label(
    asm: string,
    label: NameLabel),
  Move(
    asm: string,
    dst: [NameTemp],
    src: [NameTemp]),
}
```

- `[NameTemp]`: list of registers assigned by register allocator
- Oprations that fall through $\implies$ `jump` is empty
- `Label`: point in program that `jumps` can go
- `asm` instruction does not know about register assignments; use generic enumerations `s<n>, d<n>, j<n>`, eg: `LOAD d0 <- M[s0 + 0]`
- `asm` instruction may have registers that are both used for src and dst
- After choosing temporaries by instruction selector: `LOAD d0 <- M[s0 + 0]; dst:[t909], src: [t92]`

- After register allocation, sample asm:
  `LOAD r2 <- M[r12+0]`

Use of `AsInstr` during IR tiling selection phase:

- `munch_exp(..)`, `munch_stm(..)` to emit instructions (accumulate instructions into a sequence)
- `munch_exp(..)` returns temporary of the expression result
- sample `munch_exp`:

```
fn munch_exp(e) {
  match e {
    MEM(BINOP(PLUS, e1, CONST(i))) => {
      let r: NameTemp = NameTemp::new();
      emit(AsOper(
        asm=format("LOAD d0 <- M[s0 + {}]", i),
        dest=[r],
        src=munch_exp(e1):[],
        jumps=[]));
      return r;
    },
    CONST(i) => {
      let r: NameTemp = NameTemp::new();
      emit(AsOper(
        asm=format("ADD d0 <- r0 + {}", i),
        dest=[r],
        src=[],
        jumps=[]));
      return r;
    },
    ..
  }
}
```

  note special `r0` register for zero value

- sample `munch_stm`:

```
fn munch_stm(s) {
  match s {
    MOVE(TEMP(i), e2) => {
      emit(AsMove(
        asm="ADD d0 <- s0 + r0",
        dest=[i],
        src=munch_exp(e2):[]));
    },
    MOVE(MEM(CONST(i)), e2) => {
      emit(AsOper(
        asm=format("STORE M[r0+{}] <- s0",i),
        dest=[],
        src=munch_exp(e2):[],
        jumps=[]));
    },
    ..
  }
}
```

8.3.1 NameMap utility

new() → NameMap
query(NameMap, NameTemp) → string
update(NameMap, NameTemp, string)
layer(over: NameMap, under: NameMap) → NameMap

Use cases for NameMap:
register allocator: temporary → register name
frame module: preallocated register → register name
debugging: temporary → stringified name

8.3.2 Procedure calls

- ```
  EXP(CALL(e, args)) => {
    let r: NameTemp = munch_exp(e);
    let l: [NameTemp] = munch_args(args);
    emit(AsOper(
      asm="CALL s0",
      dest=calldefs, //mutated registers from call
      src=r:l,
      jumps=[]));
  }
  ```
- Use of utility function `munch_args(..) -> [NameTemp]` to generate code to move all arguments to correct positions in registers and/or memory. Returned temporaries pass in as sources to the instruction (may not be explicitly written in assembly language) in order for liveness analysis to work correctly.
- Call may side-effect registers (caller-save, return-address, return-value) $\implies$ list involved registers as destinations in order for later analysis to know they get affected.

entry point, passing rewritten/reordered statements (of a function body) that are processed by earlier phases:

```
fn codegen(stm_list: [T_stm]) -> [AsInstr] {
  for s in stm_list {
    munch_stm(s);
  }
  INSTR_LIST
}
fn emit(instr: AsInstr) -> () {
  INSTR_LIST = INSTR_LIST ++ (instr: [])
}
```
```

9 Liveness Analysis

9.1 Definitions

$succ[n] = \{x :$ n has an arrow to x$\}$ (nodes connected by outgoing edges)
$pred[n] = \{x :$ x has an arrow to n$\}$ (nodes connected by incoming edges)

assignment to variable/temporary defines that variable:
$def[n] = \{v :$ node n defines variable v$\}$

variable on right hand side of assignment uses the variable:
$use[n] = \{v :$ node n uses variable v$\}$

$def(var) = \{n :$ node n defines variable var$\}$

$use(var) = \{n :$ node n uses variable var$\}$

liveness: variable is live on an edge if there exists a directed path from the edge to use of that variable that does not go through a def

live-in ($in[n]$): variable is live on ≥ 1 in-edge(s) of that node

live-out ($out[n]$): variable is live on ≥ 1 out-edge(s) of that node

9.2 Dataflow Equations for Solving Liveness Range

$in[n] = use[n] \cup (out[n] \setminus def[n])$
$out[n] = \cup_{s \in succ[n]} in[s]$

Solve via:

- fixed point iteration, or
- per variable search backwards (starting at use and ending on a definition of the variable)

Liveness flows backwards, so perform iteration of dataflow equations in reverse order of control flow graph).

Merging of nodes to basic blocks $\implies$ allows faster perfomance due to reduced number of graph elements.

9.3 Complexity

Worst case: outer loop of fixed point iteration $O(N^2)$ due to bounding of in-set/out-set to N elements.
Set operation between nodes: $O(N^2)$. Then overall worst case is $O(N^4)$
Reordering of nodes $\implies$ outer loop usually needs 2-3 iterations.
Live-sets sparse $\implies$ $O(N)$ to $O(N^2)$ in most cases.

9.4 Solutions to Dataflow Equations

may be approximations:
live variables present in approximation
presence of variable in approximation that may not need to be live

9.5 Static vs. Dynamic Liveness

Special case algo. can improve liveness analysis in some cases

Dynamic liveness:
variable a is dynamically live at node n if some execution of the program goes from n to use of a without going through a definition of a.

Static liveness:
variable a is statically live at node n if there exists a path of control flow edge from n to some use of a that does not go through a definition of a.

dynamically live $\implies$ statically live

In general, optimize using static liveness for approximation

9.6 Liveness for Register Allocation

overlapping live ranges of temporaries $\implies$ use separate register at that point of execution

interference graph: expressing non-overlapping live range constraints between pairs of variables using edges

9.7 Move Instruction Optimization

no need to create an interference edge for `t <- s`

however if a later non-`Move` definition of `t` happens, interference still needs to be accounted at that point

algo:

- definition of `a` at non-move instruction $\implies$: add interference edge between sources and destination (`a`) variables.
  interference edges: $\{(s, a)\}, \forall s \in$ sources
- definition of `a` at move instruction `a <- c`: if variable `b` is live-out and `b` $\neq$ `c` $\implies$ add interference edge (a, b); eg: `b` will be used later

9.8 Control Flow Graph Module

Construct a control flow graph. Use this later to perform liveness analysis of variables and produce an interference graph.

Use a module, FlowGraph, to manage nodes:

- node represents instruction/ basic block
- instruction `m` can be followed by instruction `n` $\implies$ edge (m, n) exists in graph
- internal data of FlowGraph hidden from outside behind an interface
- let nodes represent abstract instructions; take in a list of instructions and return flow graph where info of node is abstract assembly instruction
- jump fields of instruction used in creating control flow edges
- source and destination fields of instructions used to obtain use and def information

Info associated with each node:
```
FlowGraph::def(n) -> {t: temporary t defined at node n}
FlowGraph::use(n) -> {t: temporary t used at node n}
FlowGraph::isMoveInstruction(n) -> bool
```

Separate association of node to extra info possible by an external mapping function.

9.9 Liveness Module

Takes in a flow graph and produces:

- interference graph
- list of node(variable)-pairs representing Move instructions that may be elided in later phases of compilation

Interference graph node `n`:
`Live::temp(n) = temporary variable represented by node n`
$(n \mapsto NameTemp)$

Maintain data structure for remembering what is live at exit of each flow graph node (live-out)

Calculation of `live_map` (set of live temporaries at current location) used to construct interference graph:
$(\forall$ flow node $n)(\forall d \in def[n])(\forall i)$ add interference (d, t_i)
where:
$\{t_1, t_2, ..\} \equiv$ temporaries in `live_map`
$def[n] \equiv$ newly defined temporaries at node n

9.9.1 Zero Length Live Range

Definition may include side effects (eg: write to regsiter, even if variable is not used after)

May interfere with any overlapping live ranges.

Therefore 0-length live range needs to be taken into account.

10 Register Allocation

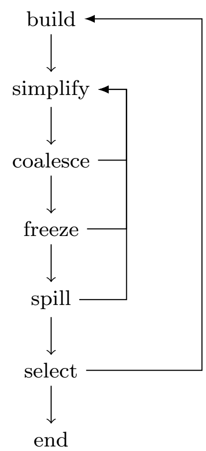
Goal: colour nodes of interference graph:

- no 2 adjacent ndoes are coloured the same
- minimize number of colours used
- a colour corresponds to a specific register

General phases:
1. define an order of nodes
2. colour nodes in defined order while preserving colour invariant

10.1 Common Algorithm

Phases: build, simplify, coalesce, freeze, potential spill, select



10.1.1 Build

Construct interference graph for register resident variables (create edges for temporaries that have overlapping live ranges). Mark node as move / non-move.

10.1.2 Simplify

Put nodes in some stack order.

For non-move nodes, if there exists a node from graph that has $< k$ (where k is the number of available registers) degree $\implies$ add it to stack (and remove from graph), otherwise go to potential spill phase.

10.1.3 Coalesce

Tentatively merge nodes (assigned same colour) based on some safe algoithm (eg: Briggs / George), and go to simplify step. Repeat until all nodes remain have $\geq k$ degree or are only move-related nodes.

Potentially coalesce move-related nodes and thus delete move instructions. If no edge exists between src and dest. of move instruction $\implies$ edges are coalesced when src and dest. nodes merge.

Coalescing makes nodes with more edges in general, safe strategies used (if original graph is colourable, and if applying safe strategy for coalescing is successful then the result is also colourable).

Briggs:
Nodes a and b merged has $< k$ neighbours of significant degree ($\geq k$ edges) $\implies$ can be merged

George:
all neighbours of a either interferes with b or $< k$ degree $\implies$ can be merged

interleaving of simplify and coalescing phases is effective at removing redundant move instructions while not introducing spills.

10.1.4 Freeze

At this stage, all nodes are $\geq k$ degree or move-related.

If possible, select move node of low degree and mark it as non-move (give up coalescing), and go to simplify step.

10.1.5 Potential Spill

If there exists no $< k$ degree node, select a node (eg: of highest degree) to be potentially represented in memory instead of register (edges in graph removed) and push onto the stack, and go to simplify step. Repeat until no spill occurs (remaining nodes have $< k$ degree) and all nodes have been simplified and added to the stack. Optimistic colouring heuristic: select node with high degree to remove, continue process and hope that node will eventually be colourable after more removal.

10.1.6 Select

Pop all nodes from stack and try assigning colours.
Either assignment is successful, or if not assignable: rebuild the graph by inserting instructions to relocate variable to memory (actual spill occurs), discard coalesces found in this round, and go to build step.

If all assignments are succesful, then the algorithm ends successfully.

10.2 Coalescing of Spilled Nodes

- spill pairs are ususally never live simultaneously since number of memory locations aren't practically bounded
- use liveness information to construct interference graph for spilled nodes
- if there is a pair of non-interfering spilled nodes connected by move instruction, then coalesce them
- use simplify and select to colour the graph; no spilling occurs here
  simplify: picks lowest degree node
  select: picks 1st available colour (no limit on number of colours here due to plentiful memory addresses)
- colours correspond to activation record locations of spilled variables
- performed before generating spill instructions and interference graph of register-resident temporary (register allocation for remaining nodes) to avoid fetch-store sequences for coalesced moves of spilled nodes.

10.3 Precoloured Nodes

These don't simplify (no freedom of assigning an arbitrary colour) and spill (can't spill to memory since these are specific to register)

Generated for certain calling conventions where particular temporaries are permanently bound to certain registers (eg: frame pointer, standard argument 1 register)

Ordinary temporaries may be assigned same colour as precoloured register as long as they don't interfere.

Desirable: Keep live ranges of precoloured nodes short. Use move instructions to assign to fresh temporaries (these can be spilled) before moving it back to precoloured register when needed.
Eg: callee-save registers
May be coalesced and move eliminated back to original if there isn't register pressure.

10.4 Optimization

- variable not live across procedure call $\implies$ allocate to caller-save register to save extra instructions
- variable live across procedure call $\implies$ keep in callee-save register
- specialized allocation algo. on expression trees is efficient

10.5 Kempe Graph Colouring

heuristic for phase 1 of graph colouring (defining an order of nodes)

Solve subproblem recursively until base case:

- there are $\leq k$ nodes remaining in graph $\implies$ graph is colourable, or
- remove a node with $< k$ degree and push onto stack for later colour assignment

10.6 Algo

```
fn reg_alloc(){
  liveness_analysis();
  build();
  make_worklists();
  while !all_worklists_empty() {
    if !simplify_worklist.empty() { simplify(); }
    if !move_worklist.empty() { coalesce(); }
    if !freeze_worklist.empty() { freeze(); }
    if !spill_worklist.empty() { select_spill(); }
  }
  assign_colours();
  if !spilled_nodes.empty() {
    rewrite_program(spilled_nodes);
    reg_alloc();
  }
}
```

11 GC

goal: minimize number of reachable record that are not live

11.1 Mark and Sweep

traverse directed graph to mark all reachable nodes

unmarked nodes at end of sweep are garbage and can be reclaimed (via freelist)

reset marks of all nodes for next sweep

use of freed memory to use as stack vs. explicit stack (threading pointers in free nodes)

sweeping phase: put unmarked records into freelist (can use different freelists for different sized records)

issues: external vs. internal fragmentation

11.2 Reference Counts

record contains additional info on the nummber of pointers pointing to the record

increment number when pointer is stored to somewhere

decrement when pointer is removed from somewhere

when number is 0 $\implies$ put record into freelist; recursively decrement internal pointers in record during allocation (removal from freelist) for shother GC pauses

issues: cycles cannot be reclaimed (count never reaches 0) even if not reachable from program

incrementing ref counts is costly (due to instructions for putting things into freelists and possibly decrementing previous pointer records)

may be optimized by aggregating count updates via dataflow analysis

11.3 Copying Collection

division of space into to-space and from-space

compacting copying

easy to allocate

pointer forwarding: operations to copy a pointer in from-space to to-space correctly

```
fn forward(p) {
  if p.is_in_from_space(){
    if p.f1.is_in_to_space(){
      return p.f1
    }
    for i in p.fields() {
      next.field(i) = p.field(i);
    }
    p.f1 = next;
    next = next + sizeof(p);
    return p.f1
  }
  p //already in to-space
}
```

BFS copying GC (Cheney's algo.):

- scan and next cursors
- 1st forwarding of root objects

```
scan = next = start_of_to_space
for r in roots {
  r = forward(r);
}
while scan < next {
  record = get_record_at(scan);
  for i of record.fields(){
    scan.field(i) = forward(scan.field(i));
  }
  scan = scan + sizeof(record);
}
```

start of to-space $\leq$ scan $\leq$ next

area between scan and next may contain pointers not yet forwarded (still points to from-space)

issues: locality of reference

variant of algo: depth first copying

cost: O(number of nodes marked)

ammortized cost (per word allocated): $\frac{cR}{\frac{H}{2}-R}$

11.4 Generational Collection

effective when old objects rarely update to point to new objects

objects promoted to older generation area after surviving a few collections

collect by: mark and sweep, or copying collection

in addition of root nodes, we keep a set of objects from older generations (remembered set) that have updated to point to objects in newer generations; these are scanned to update the pointers when objects in newer generations are copy collected to new to-space

- insert extra instructions when updating pointer field: put object into a set of updated objects that gets scanned for pointers back to G_0
- runtime GC gets this set to run its algo.

from-space $\equiv$ G_0 arena hemisphere

roots $\equiv$ program variables + remembered set

to-space $\equiv$ G_0 arena new hemisphere

pointers to older generations not changed

marking algo.: not mark objects in older generations

copying algo.: copies these verbatim without forwarding

after several collection of G_0, run collection for G_0 and G_1:

- remembered set for roots contained in $G_1, G_2, ..., G_k$
- collect G_0 nad G_1 together

11.5 Incremental Collection

interleave collection work in order to avoid long interruptions

types:

- incremental: collector operates only when mutator (program changing graph oof reachable data) requests it
- concurrent: collector can operate between any instructions executed by mutator

classes of records:

- white: objects not yet visited by DFS/BFS

- grey: objects that have been marked/copied, but childrens have not been examined
- black: objects that have themselves and their children marked

generalization of mark-and-sweep and cpoying collection algo: tricolour marking algo.

```
while let Some(p) = grey_objects() {
  for i in p.fields(){
    if p.field(i).is_white() {
      p.field(i).colour = grey
    }
  }
  p.colour = black
}
```

invariants that a mutator respects:

- no black object points to a while object
- every grey object is on collector's stack/queue/data structure (grey-set)

incremental algo variants:

- Dijkstra, Lamport, et. al.
- Steele
- Boehm, Demers, Shenker
- Baker
- Appel, Ellis, Li

general types of implementation:

- write barrier algo: write/store by mutator checked to make sure invariant is maintained
- read barrier algo: read/fetch instructions checked for invariant

these must synchronize with collector:

- using software implementations of barrier: explicit synchronization instructions (may be expensive)
- use of virtual memory H/W (sync. implicit in a page fault (mutator faults on a page $\Longrightarrow$ O/S ensures no other process has access to that page before processing the fault

11.5.1 Baker's Algorithm

example of an incremental copying collection algo.

TODO

11.6 Interface to the Compiler

live analysis: derived pointer implicitly keeps its base pointer live so the GC algo. does not get confused by elimination of pointers in live analysis

use of inline expansion and merge ops from multiple allocations to save instruction related to allocation with GC

11.6.1 GC Layout of Objects

OO language contains class descriptors (compile time generation) in every object for dynamic lookup $\Longrightarrow$ no additional per-object cost if these are also used for GC

Compiler needs to signal to GC collector every pointer containing temporary and local variable (in register or activation record)

set of live temporaries different at every point in the program $\Longrightarrow$ more efficient to describe pointer map at points where new GC cycle can begin:

- calls to alloc function
- each function call (since they can laso call alloc)

example impl.: map function return address to live pointer set: for all pointer live immediately after call, map tells their register/frame location

root finding:

- collector starts at top of the stack and scans downward frame by frame
- in each frame, collector marks/forwards (if using copying collection) pointers in that frame (these pointers are obtained from address map to pointer entries)
- callee-save registers with special handling:
  function must describe which of its callee-save registers contain pointers at call to another function via additional info in pointer map so that the called function knows which callee-saved registers contain pointers

12 Extensions with OOP

Static methods: find method using ancestor chain.

Dynamic method: a list of method instances in class descriptor (from ancestor classes in the case of single inheritence tree).

Method for field initialization for class objects.

Method instance lookup for multiple inheritance can use static analysis for packing:

- Compile time hashing of fields/methods to offsets/code addresses.
- Graph colouring: add edge between fields that exist in its class and ancestor classes (these are the ones that cannot coexist). A colour corresponds to a slot. May end up with vacant slots due to colouring constraints.

12.1 Compaction

Pack fields and methods in class object instances; map uncompacted slots to compacted offsets in a separate descriptor which is shared between many class objects.

Field lookup: in class object instance, fetch descriptor pointer. Fetch field offset from descriptor, perform store/fetch on data at offset wrt. object. Further perform static analysis to elide extraneous operations (eg: common expressions).

12.2 Class Membership Test

Use class descriptor info to compare against class type tags.

Use linked list of base classes.

12.3 Type Coercion

- sub to base class coercion safe at compile time
- base to sub class coercion requires runtime type check and exception support
- static/runtime casts/tests usually needs language support
- conditional test allows compiler to apply type narrowing/coercion and type propagation optimization on relevant code paths

12.4 Private Members

Type check phase enforces it.

Implemenation can simply use an indicator information for each member in symbol table. Uses of members require checking the indicator.

12.5 Converting dynamic calls to methods to static method calls

Less execution pipeline stalls due to runtime indirections.

Determination:

- When method call is always with the same method instance, then replace dynamic call with static call
- use subclass info, if no overrides for method call exist then use static call at call site
- static dataflow analysis: type propagation from declaration/assignment to call sites may constrain the possible method instances at call sites

13 Extensions with FP

Flavours of FP:

- impure and higher order (function as argument / return value of function)
- strict and pure
- non-strict and pure

Techniques for saving/accessing non-local environment:

- lambda lifting
- closure with code pointer and static link
  Without nesting of functions: machine code address of function (and representation of function in some variable)
  With nesting of functions: machine code pointer and access to non-local variable used by function (such as through static link)

13.1 An impl of code pointer and non-local variable access via static link

Heap allocation for either:

- entire activation record of function
- only variables that escape (used by inner functions); let stack frame's variable point to heap allocated escape variable record

Escape variable record:

- heap allocated record containing local variables that is needed by inner functions
- static link to environment (escape variable record) provided by enclosing function

Cleanup / recycle via garbage collector

Static Link:

- access free/non-local variables (eg: the environment)
- passed in as an argument to function
- stored in a record on the heap

Pointer to escape variable record:

- accessible in stack frame
- not escape and therefore can be spilled

When allocating formal parameters and local variables:

- escape variables allocated to a record on heap
- calculation of offsets uses escape variables record when walking the static link chain

13.2 Pure FP Language

Immutability of variables allows equational reasoning:

- variable assignment only happens once
- side effects disallowed

COW for producing new values

GC recycles unreachable variables

Optimizations:

- control flow graph: more complex due to function variables that are not statically defined

- equational reasoning (with immutable variables): can replace variable values with constraints once value propagation has been performed
- inline expansion
  - renaming of parameter/variable to avoid name clases
  - replacement of formal parameters and their occurences in function body by new and unique variables using let declarations
  - after inlining, unreferenced functions can be removed entirely
- recursive function inlining: split function into 2 parts:
  - prelude: callable from outside; calls loop header
  - loop header: callable from prelude and body of loop header

  inlining applies to the prelude portion at call sites (prelude is expanded)
- loop invariant arguments: arguments passed to recursive calls and function are invariant, then apply hoisting transformation:
  - remove parameters from function
  - replace use of these parameters with original values from outer call site
- cascading inlining: inlining applied multiple times: function calls within inlined code may be inlined again
- control code size from inlining: heuristic candidates:
  - very frequently executed function
  - small ratio of body instruction to function call instruction
  - function is only called once
- un-nesting let declarations: put declarations in a same scope:

  ```
  let dec1
      dec2 in exp
  ```

13.2.1 Closure Conversion

Eliminate function's access to non-local variables by turning them into explicit formal parameter access. Then, there is no need to query non-local access via static links afterwards.

Rewrite:

`f(a1, a2, ..., an) = B` at nesting depth `d`, with:
escaping local variables and formal parameters `x1, ..., xn`
non-escaping variables `y1, ..., yn`

into:

`f(a0, a1, ..., an) = let var r = { a0, x1, ..., xn } = B'`
where:
`a0` $\equiv$ explicit parameter for static link
`r` $\equiv$ record with escaping variables and enclosing static link; `r` becomes static link argument when calling inner functions (depth of $d+1$)

use of non-local variable (declared in depth of $< d$) in `B` transformed into some access of offset using record's `a0` in rewritten body `B'`

13.2.2 Tail Recursion

Function called as the last thing in an enclosing expression and that expression is recursively the tail context up to the body of an outer function $\Longrightarrow$ function is a tail call

Optimize: skip returning to current function and jump to outer nested function

- move params into argument registers
- restore callee-save registers
- pop stack frame of current/caller function
- jump to callee

Static time escape analysis: closure records that do not outlive function that created them can be stack allocated instread of heap

Heap allocation / GC optimizations (todo: section 13.7)

13.3 Lazy Evaluation

β-substitution: `f(x) = B` $\Longrightarrow$ `f(E) = B[x->E]`

Equivalent if both programs halt, otherwise use lazy evaluation

13.3.1 Call-by-Name Evaluation

compute expression only when result is needed

- variable is a thunk (function that computes a value on demand): eg: `() -> int` instead of `int`
- variable creation: create a function value
- variable use: function application

13.3.2 Call-by-Need Evaluation

- modification of call-by-name: evaluate a thunk only once
- associate thunk with a cache (memo)
- on 1st creation: memo slot is empty
- on use of thunk: check memo slot first and only call thunk function when slot is empty
- example impl: record of thunk function + memo slot

13.3.3 Optimization

Benefits of lazy language over strict functional and impure language during optimization:
Properties of being side-effect free and preservation of termination of the original program after transformation allows the following to be more easily implemented:

- dead code removal
- invariant hoisting: relocate invariant computation out of loops
- deforestation: removal of intermediate lists/trees/... from function return values
- strictness analysis for optimizing thunk overhead
  - goal: determine if it's safe to eval an argument before passing it to a function (eg: even if function never uses the argument and function halts then the argument is deemed not strict)
  - thunk at places when absolutely necessary
  - eval. now if it's certain variable need to be computed

- definition of strictness:
  `f(x)` is strict $\equiv$
  $(\forall a)a$ fails to terminate $\implies f(a)$ fails to terminate

  `f(x1, ..., xn)` is strict wrt. `xi` $\equiv$
  $(\forall a)a$ value for xi parameter fails to terminate $\implies$ $f(..., a, ...)$ fails to terminate

- if `f` is strict wrt. `x`, then x can be evaluated immediately and pass its result to `f` instead of a thunk
- approximation of strictness analysis: if strictness is indeterminate, then be conservative and take a sound approach: function argument must be assumed to be non-strict: $(\exists a)a$ fails to terminate $\wedge$ $f(a)$ terminates; use a thunk

13.4 Continuation Based I/O

- special return type for I/O to make it visible to the compiler for reasoning
- rid of while and for loops, compound statements, assignment statements from language

TODO

14 Polymorphic Types

Types:

- parametric polymorphism: same algo for all types of argument
- overloading / ad hoc polymorphism: different algo depending on type of argument

14.1 Parametric Polymorphism

intermediate representation typed, even for implicit typed language

ability to run type checker after optimization passes to debug the compiler

14.2 Case Study: Explicitly Typed Polymorphic Lang.

14.2.1 parametric type constructor

```
type id tyvars = ty
```

```
eg:
```

```
type list <e> = { head: e, tail: list <e> }
```

where:

`list` is a type constructor

`list<T>` is a type

`T` is a type argument to the type constructor

`{ ... }` is an initializer

14.2.2 function call with instantiation for calling polymorphic function

```
type list< e > = { head: e, tail: list<e> }
function append< e >(a: list<e>, b: list<e>): list<e>
  = ...
```

14.2.3 Polymorphic Type Checking

idea: replace β with t in a type expression

to avoid name clash when substituting, use α-conversion before application

application: apply a type constructor to type arguments, eg:
`App(Int, [])` $\iff$ `Int<>`

arrow type constructor represent function:
$a \to b \iff$ `App(Arrow, [a, b])`

`Tyfun`$([\alpha_1, ..., \alpha_n], t) \iff$ type function / constructor
where:
α_i's: bound type variables for `t`
`t`: may use α_i's

use `App` on `Tyfun(..)` and provided type substitution variables to replace α_i's to get actual type of function

`Poly(tyvarlist, ty)` may use types in `tyvarlist`

14.2.4 Substitution Rules

$$subst(Var(\alpha), \{\beta_1 \to t_1, .., \beta_k \to t_k\})$$
$$= \begin{cases} t_i, & \text{if } \alpha \equiv \beta_i \\ Var(\alpha), & \text{otherwise} \end{cases}$$
$$subst(nil, \sigma) = nil$$
$$subst(App(Tyfun([\alpha_1, ..., \alpha_n], t), [u_1, ..., u_n]), \sigma)$$
$$= subst(subst(t, \{\alpha_1 \to u_1, ..., \alpha_n \to u_n\}), \sigma)$$
$$subst(App(tycon, [u_1, ..., u_n]), \sigma)$$
$$= App(tycon, [subst(u_1, \sigma), ..., subst(u_n, \sigma)])$$
$$subst(Poly([\alpha_1, ..., \alpha_n], u), \sigma)$$
$$= Poly([\gamma_1, ..., \gamma_n], subst(u', \sigma))$$

where :

$\gamma_1, ..., \gamma_n$ are new variables not occuring in σ or in u

$u' = subst(u, \{\alpha_1 \to Var(\gamma_1), ..., \alpha_n \to Var(\gamma_n)\})$

can avoid apply subst twice in Poly clauses by composing 2 substitutions

14.2.5 Structural Equivalence of Types

freely substitute definitions of types

eg:

```
type number = int
number → int
```

```
type transformer<e> = e → e
transformer → Tyfun([e], App(Arrow, [Var(e), Var(e)]))
```

14.2.6 Occurence Equivalence of Types

generates new type per occurence of definition, eg: via via pointer comparison

distinction of declaration of record that might have exact same fields

14.2.7 Checking Type Equivalence

expand: to see internal structure of a type

often need to expand `Tyfun` in `App(Tyfun(..), ..)`, substituting actual parameters for formal parameters

for `App(Unique(Tycon, z), ...)`, don't expand `Tycon` but test the uniqueness mark `z`

algo 16.5: Unify (check for quivalence of types), extend it to infer types for implicitly typed language; give error message if it fails

algo 16.6: Expand types for equivalence check or operations requiring info of internal structures

algo 16.7: type declarations and translation rules into internal representation of type module (via modification of environments)

algo 16.8 type checking rules for polymorphic language

14.3 Type Inference for Implicitly Typed Languages

types omitted except for polymorphic type declarations in record, etc.

Hindley-Milner type inference algo (algo. 16.10):

- conversion into explicitly typed program
- introduce a meta-variable for an unknown type: $ty \to Meta(metavar)$
  - used as placeholder for an unknown type that we need to infer
  - solve for as much meta-variables as possible
- use unify to derive relationship between meta-variables
- remaining undetermined meta type variables can be converted to variables bound by Poly types (polymorphic types)
- generalization:
  - free type variables allow different instantiations at use sites
  - type inference at declaration assignment, and not at later variable assignment
- explicitly typed representation in intermediate language is popular:
  - after type check, transform into functional intermediate form (canonicalization)
  - perform optimizations on typed IR

14.4 Representation of Polymorphic Variables

type/size of variable not known; instantiated differently at use sites, but we must prepare for instruction selection

possible solutions:

- expansion
- boxing/tagging
- coercions
- type passing

14.4.1 Expansion of Polymorphic Functions

- inline expand until everything is monomorphic: substitute actual types at every use site
- consider the case of recursion:
  - call is the same form as the function $\implies$ use monomorphic function introduced in `let`
  - recursive call has different actual type parameters (polymorphic recursion):
$\implies$ cause calls to have different types each time at different stages of recursion which cause code bloat
$\implies$ restrict its use via language design

14.4.2 Fully Boxed Translation

all values are of same size; each describe itself to GC

- use pointer (boxed value) for large data
- record contains meta-info (eg: size of item) to GC, and pointer to the record is the boxed value

alternatively, use bit tagging for small data types (less than 1 word size), to elide cost of fetching and unboxing value as well as boxing value after computation

14.4.3 Coercision Based Representation

use boxing (known size, use of extra instructions for box/unbox when value is used) only in polymorphic (one representation for multiple different types) variables

use unboxed representations for values in monomorphic (known explicit types in each instantiation) variables $\implies$

- efficient in monomorphic parts of program
- extra cost in polymorphic functions

conversion from unboxed to boxed values occur at call to a polymorphic function

wrap and unwrap instructions for different types of values that need boxing (table 16.12 for primitive types)

recursive wrapping (table 16.13)

- build from bottom up
- arguments to a function need to be wrapped
- function agumented to take in wrapped values; within it, unwraps arguments and applys unwrapped value to original function
- result of the function is wrapped

for the case of a parameter being a polymorphic type variable (table 16.14): wrapping and unwrapping for different cases of actual vs. formal type:

actual args at calsite	formal params	transformation
$y : \bullet$	$\bullet$	y
$y : \text{int}$	$\bullet$	$\text{wrap}_{\text{int}}(y)$
$y : (t_1, t_2)$	$\bullet$	$\text{wrap}_{(t_1,t_2)}(y)$
$y : (t_1, t_2)$	$(t_1, \bullet)$	$(y.1, \text{wrap}_{t_2}(y.2))$
$f : t_1 \to t_2$	$\bullet$	$\text{wrap}_{t_1 \to t_2}(f)$
$f : t_1 \to t_2$	$\bullet \to t_2$	let fun $f_w(a) = f(\text{unwrap}_{t_1}(a))$ in f_w end
$f : t_1 \to t_2$	$\bullet \to \bullet$	let fun $f_w(a) = \text{wrap}_{t_2}(f(\text{unwrap}_{t_1}(a)))$ in f_w end

polymorphic function returns result into a monomorphic context $\implies$ result must be unboxed/unwrapped

if result is fully polymorphic $\implies$ use fully recursive unwrapping/unboxing on the result

if result itself is (partially) polymorphic $\implies$ additional unwrapping steps must be applied for boxed elements before rebuilding the fully unboxed structure

performance advantage: depends on that instantiation of polymorphic variables, where extra coercion steps are inserted, does not occur often wrt. ordinary ones

14.4.4 Parsing Types as Runtime Arguments

idea:

- keep data in natural representation
- pass info of actual type of formal parameter
- do this at runtime

pass in type description as additional variable $\implies$ may need closure since it can be a free variable

runtime cost for constructing description of types

runtime cost for polymorphic function: treat variables differently depending on type parameter

descriptors of types can be also used for GC

enable typecase facility (runtime type checking / matching)

14.5 Overloading / Ad-hoc Polymorphism

map `f` to multiple/different implementations (add to bindings) when processing declarations

callsites of `f` analyzed with types of actual parameters to determine which binding should be used, eg: select one that is the most specific

15 Analysis: Dataflow

transformations using dataflow analysis:

- common subexpression elimination: use reaching expression / available expression
- constant propagation: use reaching definition
- copy propagation
- dead code elimination: use liveness
- constant folding
- register allocation

note: one optimization may enable cascade of other transforms

use a general dataflow analyzer and notify analyzer when an optimizer (out of multiple optimizers) changes program

intra procedural optimization: spans all basic blocks within a function

use a simplified tree language:

- each Exp has only a sinle MEM or BINOP node
- MOVE with MEM node only has TEMP/CONST on rhs and TEMP/CONST under MEM

quadruple representation: `(a,b,c, op)` $\iff$ `a` $\leftarrow$ `b op c`

translate trees to quadruples in 1 pass

intraprocedural optimizations:

- take quadruple from canon. phase
- transofrm them into a new set of quadruples
- modify quadruples
- feed result into instruction selection phase (maybe necessarty to turn them back into nested expression)

make control flow graph of quadruples: a directed edge from each node (eg: statement) to its successors (nodes that can immediately execute after it)

various dataflow analysis on CFG of quadruples: reaching definition, available expression

15.1 Reaching Definition

forward dataflow problem

an expression in node s of flow graph, $t \leftarrow$ `x op y`, reaches node n if there is a path from s to n that does not go through any assignment to x or y or `x op y`.

practically can be implemented via backward search from node n and stop when `x op y` is found

ambiguous definition of a variable: a statement that not assign value to the variable unambiguous definition: a statement that unconditionally modify a temporary: `x` $\leftarrow$ `a op b` or `x`$\leftarrow$ `M[a]`

assume analysis on only unambiguous definitions

express reaching definition calculation as the solution of dataflow equations

label each MOVE statement with a definition-ID

generation of definition of variable `t` by a statement `s`:

`gen[s]` = `def-ID d: t` $\leftarrow$ `x op y`

any generation of definition for a variable kills all other definitions of that variable

$kill[n] = \{d : d$ is a definition of t that gets killed by statement n$\}$

`defs(t)` $\equiv$ set of all definitions (def-IDs) of temporary `t`

sets of definition that reach beginnning/end of each node n:

$$
\begin{aligned}
in[n] &= \cup_{p \in pred[n]} out[p] \\
out[n] &= gen[n] \cup (in[n] \setminus kill[n]) \\
gen[s] &= \{d\} \\
kill[s] &= defs(t) \setminus \{d\} \\
&\quad d : t \leftarrow \text{b op c}, or \\
&\quad d : t \leftarrow M[b]
\end{aligned}
$$

solvable via dixed point iteration:

- initialize `in[n]`, `out[n]` to empty sets for all nodes
- iterate until `in[n]`, `out[n]` do not change for all nodes

use this info for optimizations, eg: constant propagation

15.2 Available Expression

forward dataflow problem

expression is available at node n $\iff$ on every path from entry node of the graph to node n, expression is computed ≥ 1 times(s), and no other definitions of involved variables in the expression exist since the most recent occurence of the expression on that path

generation at statement `s: t = b op c`
$gen[s] = \{b\ op\ c\} \setminus kill[s]$
where $kill[s]$ is any expression containing `t`

store instruction, $M[a] \leftarrow b$, might modify any memory location $\implies$ kills all fetch expression $(\forall x) M[x]$

in/out set equation for node n via dixed point iteration:

$$
\begin{aligned}
in[n] &= \cap_{p \in pred[n]} out[p] \\
out[n] &= gen[n] \cup (in[n] \setminus kill[n])
\end{aligned}
$$

initialization:

- in set of start node empty
- all other sets to full (set of all expressions)

15.3 Liveness Analysis Expressed in Terms of Gen and Kill Sets

$$
\begin{aligned}
in[n] &= gen[n] \cup (out[n] \setminus kill[n]) \\
out[n] &= \cup_{s \in succ[n]} in[s]
\end{aligned}
$$

backward flow problem

any use of variable $(.. \leftarrow var)$ generates liveness

any definition $(var \leftarrow ..)$ kills liveness

ordering of nodes matters in computation efficiency in the fixed point approach (eg: quasi-sorted worklist reduces number of iterations)

use-def chains: `use` $\mapsto$ `{definition that reaches use}`

- for each use of variable x, keep a list of definitions of x reaching the use $\implies$ allow efficient optimization algo. implementation that uses results of dataflow anlysis
- generalization is SSA form

def-use chains: `definition` $\mapsto$ `{possible use of definition}`

- another way to represent results of liveness anlysis

- for each definition, keep a list of all possible uses of that definition

worklist algo:

- detect change in a node during solving $\implies$ neighbours of that affected node are put into the worklist for future processing
- select an item from worklist using some priority (eg: ordering) and repeat processing until worklist is empty

incremental dataflow analysis:

- cascading effects
- cycles of dataflow analysis and optimiaztion passes $\implies$ may need many cycles
- use bookkeeping of info. so a change allows efficient updates to livesness info.

15.4 Alias Analysis

possibly use type info to annotate memory accesses (creation/address of a variable is taken) by creating an alias class for each of the following:

- every frame lcoation created
- every record field of every record type
- every array type

analyze it during semantic analysis phase; later phaes lose this info about types

heuristic:

`MEM_i[x], MEM_j[y]`, where i, j are types (alias classes) of MEM nodes

$i = j \implies MEM_i[x]$ may alias $MEM_j[y]$

pointer/reference can point o different alias classes depnding on conditional context

associate MEM node with a set of alias classes

merging of control flow branches $\implies$ require merging of alias class sets from the branches

use of `(t,d,k)` tuple to encode set of alias class of all instances of kth field of a record at location `d`, assignment to variable `t`

flow analysis equations:

$$in[n] = \cup_{p \in pred[n]} out[p]$$
$$out[n] = trans_n(in[n])$$

where:

$in[s_0] = A_0$, $s_0 \equiv$ start node

transfer function(in set of alias classes) $\mapsto$ out set of alias classes

$(\exists d, k)(p, d, k) \in in[s] \wedge (q, d, k) \in in[s] \implies$ p may alias q at statement s

using may alias relation: treat each alias class as a variable in dataflow analysis

eg:

statement: $M[a] \leftarrow b$

$gen[s] : \{\}$

$kill[s] : \{M[x]$: a may alias x at s$\}$

alias analysis in pure functional language:

- alias analysis not needed since there exists only 1 definition per variable
- pointers cannot mutate data
- immutable variable $\implies$ cannot be killed
- easier to optimize: constant propagation and loop invariant detection easy to see
- easier to debug

16 Analysis: Loops

reducible flow graph: any cycle of nodes has an unique header node; efficient to analyze

loop S:

- 1 header node, h
- all nodes in S $\to$ h
- h $\to$ all nodes in S
- all edges into S from external nodes go through h

finding loops in graph: use dominators:

- $D[n] = \{n\} \cup (\cap_{p \in pred[n]} D[p])$
- equation solvable by fixed point iteration
- initialize $(\forall n \neq S_0) D[n] = \{x : x \in S\}$

a dominates b $\iff$ all paths from start node s_0 to b go through a

$dom(x) \equiv \{y : y$ dominates x$\}$

a idom b $\iff$ $(\forall x)$ x dominates b $\implies$ x dominates a

$(\forall x \neq S_0)(\exists i) i = idom(x) \iff$
$i \neq x \wedge$ i dominates x $\wedge \neg$(i dominates all other dominators of x)

idom is unique for every node except the start node (empty in that case)

existence of an edge from node n to node h such that h dominates n $\implies$ there is a subgraph that is a loop

a node can be header of ≥ 1 natural loop(s)

a region $\equiv \{x :$ h dominates x$\}$ where one header node dominates all nodes in the region

natural loop: a region where there are edge(s) that all nodes can transit through back to the header node

property of natural loop: 2 natural loops are either disjoint or nested

proper nested loop:

- loops A, B
- a is header of A
- b is header of B
- $a \neq b$
- $b \in A \implies$ B is a proper subset of A and B is a nested loop in A

loop nest tree:

- tree of loop headers (merge if necessary) such that h_1 is above h_2 if h_2 is in loop of h_1
- leaves of the tree are innermost loops
- nodes not in any loop put in a pseudo loop, corresponding to the entire procedure body, sitting at the root of the loop nest

loop preheader:

- insert new node before loop header as a location for other optimizations (such as hoisting variables out of loop)
- all edges from nodes outside loop redirected to the new node
- add edge from new node to loop header node
- unique predecessor node of the header node

loop postbody: 1 unique node that connect to header node via back edge

loop invariant computations:

- constant value wrt. loop iteration $\implies$ hoist computation outside of the loop
- definition, d: $\leftarrow a_1$ op a_2, is loop invariant in loop L if:
  - $(\forall i) a_i$ is a constant, or
  - a_i's definitions that reach d are outside of the loop, or
  - only 1 definition of a_i reaches d and the definition is loop-invariant
- can use iterative algo. to find loop invariant definitions satisfying the above conditions

hoisting (after determining a definition is loop-invariant): transforms with valid result when hoisting an assignment, d: t $\leftarrow$ a op b, in a loop to the end of a preheader:

- d dominates all loop exits where t is live-out, and
- only 1 definition of t in the loop exists, and
- t is not live-out of the loop preheader (t is not live-in into the loop)

may need to transform while loop to a do while loop to satisfy constraint 1: hoist 1st iteration out and add guard to it; reorder conditional jump in the loop to the end exit node

induction variable analysis in loops:

- triplet of (ind var, offset, stride/step size):
  $(i, a, b) \equiv a + ib$, where a and b are loop-invariant
- basic linear induction variable $\iff$ induction variable changes by same amount every iteration
- derived induction variable: uses basic induction variable via affine expression and can be expressed by triplet
- detection of basic induction variable i: definitions of i in loop are only of the form $i \leftarrow i + c$ or $i \leftarrow i - c$ where c is loop-invariant

useless variable elimination: dead at all exits from loop and only use is in a definition of itself $\implies$ all definitions of this variable can be removed

rewriting comparisons (for almost useless variable)

- replace expression in comparison using coordinated induction variable and loop invariant expression $\implies$ makes almost useless variable useless
- some constraints: integer divisibility, sign is known

array bound check:

- remove bound checks that can be proved redundant in a safe language
- for the case of subscript expression with induction variable $\implies$ may be intractable to analyze
- conditions for eliminating bounds checking (todo)

loop unrolling: copy loop body, reduce number of loop iterations by some factor

strength reduction

17 Data Dependence in Loops

types of dependence:

- δ^f: flow dependence (write $\rightarrow$ read on same variable)
- δ^a: anti-dependence (read $\rightarrow$ write on same variable)
- δ^o: output dependence (write $\rightarrow$ write on same variable)
- δ^i: input dependence (read $\rightarrow$ read on same variable); usually not marked as dependence

normalized iteration index:

- consecutive index of iteration differ by 1 and lexicographically increasing
- starting index at 0

semi-normalized iteration index:

- partial index at 0; others start at an offset

dependence vector when there exists dependence from iteration i_{src} to iteration i_{dest}:

δ^*_d where $d = i_{dest} - i_{src}$

eg: $d = (0, 1, -2)$

dependence direction from dependence vector; useful when dependence distance can vary but maintain same direction:

- $\delta^*_d \rightarrow \delta^*_<$ where $d > 0$
- $\delta^*_0 \rightarrow \delta^*_=$
- $\delta^*_d \rightarrow \delta^*_>$ where $d < 0$

eg, for multiple indices: $d = (0, 1, -2) \rightarrow (=, <, >)$

other conventions include $sign(d)$, $(+, 0, -) \equiv (<, =, >)$

17.1 Control Dependence

Y is control dependent on X
$\iff$ X is in the dominance frontier of Y in the reverse cfg
$\iff$ X is in the post-dominance frontier of Y

17.2 Parallel Loops

use of access control constructs to respect presence of data dependence

different parallel loops with different semantics exist

17.2.1 Forall Loop

Equivalent to a sequence of array assignments.

For all statements, rhs expression of assignment is computed first for all values of the index variable before any writes are made.

Any data access conflicts between iterations of the loop resolved using lexically earliest access.

Note: lexical ordering defines rhs of assignment precedes lhs.

Then:

- within a single statement, there does not exist any flow dependence
- usual case is anti-dependence

17.2.2 Dopar Loop

All iterations of the loop starts with a copy of original values of variables before the loop started. Then:

- no flow dependence can exist in between different iterations of the loop, since no new updated values are accessible across iterations
- data access conflict (between writes) $\implies$ undefined behaviour (programming error)
- no dependence relation exist $\implies$ reduces to doall loop
- loop independent dependence relations still holds as in sequential code
- typically anti-dependence relation may exist across iterations

17.2.3 Dosingle Loop

Single assignment to the same variable/memory allowed. Then:

- no output dependence relation can exist
- no anti-dependence relation can exist
- only flow dependence relation can exist

17.3 Nested Loops

data access conflict carried by outermost loop with a non-zero data access conflict distance, so look at that outermost loop that carries the conflict

- data access conflict carried by sequential for loop resolved when: dependence distance for the loop is positive (using normalized iteration vectors); dependence does not go backwards wrt. iteration space ordering
- data access carried by dopar loop resolved when: anti-dependence relation exists between def and use; > 1 definition to a same variable/memory is undefined behaviour
- data access conflict carried by dosingle loop resolved when: flow dependence exists between def and use; > 1 def to same variable/memory is an error
- data cess conflict carried by forall loop resolved when: execution of list of statements are performed in lexical order

special cases of reduction operations that are commutative can ignore dependence relations and may be reordered in any way

17.4 program dependence graph

composed of data dependence graph and control dependence graphs

a node for each statement

used to prevent reordering using data and control precedence constraints

18 Scalar Analysis with Factored Use-Def Chains

insertion of control flow merge points using ϕ-node (counts as an assignment)

subsequent use of any variable has only one link in its use-def chain (either the original def statement or the inserted ϕ-node)

insertion of merge points computable using join sets equivalent computation using dominance frontier

19 Alternative Form of IR: SSA

improvement to def-use chain

1 type of intermediate representation where:

- each variable has 1 definition in program text (1 static site of definition, as opposed to dynamic single assignment in pure functional program)
- may be in a loop executed many times

x dominates $y \iff$ node x exists on every path from entry node to y

x strictly dominates $y \iff x$ dominates $y \wedge x \neq y$

advantages:

- simpler dataflow analysis
- simpler optimization algorithms
- size of SSA form is linear wrt. size of original program
- useful to dominator structure of CFG $\implies$ simpler algos such as interference graph construction
- eliminate unnecessary relationship for unrelated uses of same variable since they become different variables in SSA form
- dominance property of SSA form:
  either:
  - x is the argument of ϕ-function in block n $\implies$ definition of x dominates ith predecessor of n but not n
  - x is used in non-ϕ- statement in block n $\implies$ definition of x dominates n (definition dominates uses)

19.1 Construction of SSA Form

- add ϕ-function for variables
- renames all definitions and uses of variable with enumeration

19.1.1 Inserting ϕ-functions using Path Convergence

criteria for insertion:

- path-convergence criterion
  - there is a block x with a definition of variable a
  - there is a block $y \neq x$ with a definition of a
  - P_{xz} path exists from x to z
  - P_{yz} path exists from y to z
  - P_{xz} and P_{yz} do not have any node in common other than z
  - node z may appear at max once in either P_{xz} or P_{yz}
- ϕ-function counts as a definition of a
- consider start node to contain definition of all variables (formals, uninitialized ones)
- solve by iteration: while criterion is not satisfied and z does not contain a ϕ-function for a: insert $a \leftarrow \phi(a, ..., a)$ at node z

19.1.2 Efficient Insertion of ϕ-Function Using Dominator Tree of CFG

$$\begin{aligned} \mathit{DominanceFrontier}(x) \equiv \{y : (\exists z \in \mathit{Pred}(y)) x \text{ dominates } z \\ \wedge \neg(x \text{ strictly dominates } y)\} \end{aligned}$$

eg: a border between dominated and undominated nodes wrt. x

dominance frontier criterion:

node x contains a definition for some variable y $\implies$ all nodes in the dominance frontier of x need ϕ-function for y

19.1.3 Computing Dominance Frontier Using Dominator Tree

dominator tree definition:

a tree where there exists an edge $(x \mapsto y) \iff$ x is the immediate dominator of y
idom(y) = x $\iff$ y$\in$(idom^{-1}(x))

immediate dominator of x (eg: closest node that strictly dominates x):

$$\begin{aligned} y = \mathit{idom}(x) &\iff y \text{ strictly dominates } x \\ &\wedge (\forall z) z \text{ dominates } x \implies z \text{ dominates } y \\ \mathit{idom}^{-1}(x) &= \{y : x = \mathit{idom}(y) \text{ holds}\} \end{aligned}$$

note:

$(\forall x \neq$ entry node$)$ $\mathit{idom}(x)$ exists and is unique

$\mathit{idom}^{-1}(x)$ are children of node x in the dominator tree

dominator set:

$\mathit{dom}(x) = \{y : y \text{ dominates } x\}$

recursive definition: $\mathit{dom}(x) = \{x\} \cup (\cap_{p \in pred(x)} \mathit{dom}(p))$

$\mathit{dom}^{-1}(x) = \{y : x \text{ dominates } y\}$

$\mathit{sdom}(x) = \mathit{dom}(x) - \{x\}$

$x = \mathit{idom}(y) \iff$ x strictly dominates y $\wedge$ $((\forall z)$ z strictly dominates y $\implies$ z dominates x)

dominance frontier:

$$\begin{aligned} DF(x) &= DF_{local}(x) \cup (\cup_{c \in idom^{-1}(x)} DF_{up}(c)) \\ DF_{local}(x) &= \{y : y \in \mathit{Succ}(x) \wedge \mathit{idom}(y) \neq x\} \\ DF_{up}(x) &= \{y : y \in DF(x) \\ &\quad \wedge \neg(\mathit{idom}(x) \text{ strictly dominates } y)\} \\ &\quad \text{reduces to:} \\ DF(x) &= \{y : y \in \mathit{Succ}(x) \wedge \mathit{idom}(y) \neq x\} \\ &\quad \cup (\cup_{c \in idom^{-1}(x)} \\ &\quad \{y : y \in DF(c) \wedge (y = x \vee \neg(x \text{ dominates } y)\}) \end{aligned}$$

note:
$(y = x \vee \neg(x \text{ dominates } y))$
$\iff \neg(\mathit{idom}(c) = \mathit{sdom}(y)),\ x = \mathit{idom}(c)$

Algorithm 9: Computing Dominance Frontier

Input : x (input node)
Output: DF of x

$S \leftarrow \{\}$
$//DF_{local}$
$S \leftarrow S \cup \{y : y \in \mathit{Succ}(x) \wedge \mathit{idom}(y) \neq x\}$
//use dom. tree
for $c \in \mathit{idom}^{-1}(x)$ **do**
  $//DF_{up}$
  $DF(c) \leftarrow \mathit{computeDF}(c)$
  $S \leftarrow S \cup \{w : w \in DF(c)$
    $\wedge(w = x \vee \neg(x \text{ dominates } w))\}$
  $DF(x) \leftarrow DF(X) \cup S$
$DF(x)$

19.1.4 Post Dominator

assume there is an unique exit node that all nodes can reach, then:

y is control dependent on x

$\iff$ $x \in$ post dominance frontier of y

$\iff$ y post post dominates some successor of x and does not post dominate some other successor of x

$\iff$ $x \in$ dominance frontier of y wrt. reverse CFG

$\iff$ y dominates some predecessor of x and not dominate some other predecessor of x wrt. reverse CFG

19.1.5 Add ϕ-functions

Algorithm 10: Add ϕ Functions

Input : nodes (input nodes)
Output: nodes with $\phi - functions$ added

defsites :: $\{var\ \mathrm{a}, w :: \{Node\}\} \leftarrow \{\}$
for $n \in nodes$ **do**
  $//A_{orig}[n] \equiv set\ of\ vars\ defined\ in\ node\ n$
  for $var\ a \in A_{orig}(n)$ **do**
    $defsites[a] \leftarrow defsites[a] \cup \{n\}$
for $(a,\ w) \in defsites$ **do**
  for $node\ y \in DF(w.take())$ **do**
    $//A_\phi[y] \equiv \{z : z\ has\ \phi(..)\ at\ node\ y\}$
    if $a \notin A_\phi[y]$ **then**
      $A_\phi[y] \leftarrow A_\phi[y] \cup \{a\}$
      //note: phi function counts as a definition
      insert $a \leftarrow \phi(a, ...)$ at top of node y
      if $a \notin A_{orig}[y]$ **then**
        $w \leftarrow w \cup \{y\}$

19.1.6 Renaming Variables

```
fn rename_for_var(n: node/basic_block, a: variable):
  for each statement S in n:
    for each use of var x in S and S is not a phi-function:
      i = stack[x].top
      change x to x_i in S
    for each definition of var a in S:
      count[a]++
      i = count[a]
      stack[a].push(i)
      change a to a_i for the definition in S
  for each successor y of n:
```

```
    let j be index where n is jth predecessor of y
    i = stack[a].top
    change a to a_i in jth operand of phi-function
  for each child x of n in dominator tree:
    rename_for_var(x, a)
  for each definition of variable a in original S:
    stack[a].pop()

for each variable a:
  count[a] = 0
  stack[a] = [0]
  rename_for_var(entry_node, a)
```

19.1.7 Computing Dominator Tree

creating a spanning tree with dfs and enumerate nodes ($dfnum(n)$) on first visit: $dfnum(a) \leq dfnum(b) \implies$ a is an ancestor of b

there exists an edge connecting node a to node b where $dfnum(a) > dfnum(b) \implies$ node a is branched off from some node that is an ancestor of b

$d = idom(n) \implies dfnum(d) < dfnum(n)$

$dfnum(x) <= dfnum(n) \wedge x \notin Dom(n) \implies$ there exists a path that branches off above x and rejoins below x and at or above n

s is the semidominator of n $\iff$ s is the highest possible ancestor of n and there is a path that departs from the tree at s and rejoins the tree at node n

Semidominator theorem

$$let\ d \equiv dfnum$$
$$semidom(n) = \underset{x \in X}{\operatorname{argmin}}\ d(x)$$
$$s.t.:$$
$$X = \bigcup \begin{cases} d(v) < d(n) & : \{v\} //proper\ ancestor \\ d(v) > d(n) & : \{semidom(u) : \\ & \quad (\forall u)\ d(u) \leq d(v)\} \end{cases}$$

Calculate dominator from semidominator

Let s be the semidominator of n. There exists a path that bypasses s by departing from spanning tree at a node above s and rejoins spanning tree at a node between s and n $\implies$ s does not dominate n

Let y be the node between s and n with smallest number semidomiator and semidom(y) is a proper ancestor of s $\implies$ idom(y) immediately dominates n

Dominator Theorem

let ys be nodes on the spanning tree path below semidom(n) and above or including n

let y be the node in ys with smallest numbered semidominator, $y = \operatorname{argmin}_{x \in ys} dfnum(semidom(x))$

$$idom(n) = \begin{cases} semidom(n) & , semidom(y) = semidom(n) \\ idom(y) & , semidom(y) \neq semidom(n) \end{cases}$$

19.1.8 Lengauer-Tarjan Algorithm

algo. for computing idoms using semidominators

```
//helper functions

//create spanning tree and enumerate dfnum
fn dfs_dfnum(p, n) {
  if dfnum[n].is_some(){
    return
  }
  dfnum[n] = Some(N);
  order[N] = n;
  parent[n] = p
  N += 1
  for w in n.successors() {
    dfs_dfnum(n, w);
  }
}
//add edge to spanning forest (p->n)
fn link(p, n){
  ancestor[n] = p
  best[n] = n
}
fn ancestor_with_lowest_semi(v){
  a = ancestor[v];
  if ancestor[a].is_some() {
    b = ancestor_with_lowest_semi(a);
    ancestor[v] = ancestor[a]; //path compression
    //best[v]:= best node in skipped over path btw.
    //ancestor[v] (non-inclusive) and v (inclusive)
    if dfnum[semi[b]] < dfnum[semi[best[v]]] {
      best[v] = b;
    }
  }
  best[v]
}

N = 0;
dfnum = nodes.iter().map(|_| None);
root = 0;
dfs_num(-1, root);

let bucket = nodes.iter().map(|_| {});
let semi = nodes.iter().map(|_| None);
let ancestor = nodes.iter().map(|_| None);
let idom = nodes.iter().map(|_| None);
let samedom = nodes.iter().map(|_| None);

for i in [1..N).rev() { //skip root
  let n = order[i];
  let p = parent[n];
  //calc. semidom of n using semidominator theorem
  let s_prime = n.predecessors().map(|v|{
    if dfnum[v] <= dfnum[n] { v }
    }else{ semi[ancestor_with_lowest_semi(v)] }})
    .min_by(|a,b| dfnum[a] < dfnum[b]);
  let s = p.min(s_prime);

  semi[n] = s; //s is semidominator of n with lowest dfnum

  //To defer calc. of n's dominator until
  //path from s to n has been linked in forest.
  //These are candidate nodes that can be idom of n
  //maps s -> {n: s maybe the idom of n}
  bucket[s] = bucket[s].union({n});

  link(p, n);

  //calculate idom of v,
  //where p may be the idom of v
  for v in bucket[p] {
    //path p to v linked in to spanning forest by now
```

```
    y = ancestor_with_lowest_semi(v);
    if semi[y] == semi[v] {
      //use Dominator Theorem's 1st clause to calc. idom[v]
      idom[v] = semi[v]; //semi[v]==p
    }else{
      samedom[v] = y; //defer until idom(y) is calculated
    }
  }
  bucket[p].clear();
}

//deferred calculation using Dominator Theorem's 2nd clause
for i in [1..N) {
  let n = order[i];
  if samedom[n].is_some(){
    idom[n] = idom[samedom[n]]
  }
}
```

19.2 Algos using SSA

data structures:

statements:
containing block
previous statement in block
next statement in block
variables defined
variabled used
statement type: assignment, ϕ-function, fetch, store, branch

variable:
definition site
list of use sites

block:
list of statements
ordered list of predecessors
successors

19.2.1 Dead Code Elimination

list of uses of a variable is empty $\implies$ variable is not live at site of definition
SSA $\implies$ definition dominates every use
when definition has no side effect $\implies$ delete defining statement, remove use site for all used variables in the statement, add these variables to the worklist if their list of uses is empty

```
w = {x: x is a variable in SSA}
while let v = w.take() {
  if !v.use_sites.empty() {
    continue;
  }
  let S = v.definition_statement
  if !S.has_no_side_effects(){
    continue;
  }
  for x in S.used_vars() {
    x.remove_use_site(S);
    if x.use_sites.empty() {
      W.add(x);
    }
  }
  program.remove(S);
}
```

19.2.2 Constant Propagation

```
w = {x: x is a statement in SSA}
while let S = w.take() {
  let if (v = phi(c,..,c)) = S where c.is_constant() {
    S.replace_with((v = c));
  }
  let if (v = c) = S where c.is_constant() {
    for each statement T where v in T.used_vars() {
      T.var(v).replace_with(c);
      w.add(T);
    }
    program.remove(S);
  }
}
```

19.2.3 Copy Propagation

```
for each statement S {
  match S {
    (x = y) | (x = phi(y)) => {
      program.remove(S);
      for i in x.use_sites() {
        i.replace_with(y);
      }
    }
    _ => {}
  }
}
```

19.2.4 Constant Folding

```
for each statement S {
  match S {
    (x = a op b) where
      a.is_constant() && b.is_constant()
    => {
      let c = x.eval_compile_time();
      S.replace_with(c);
    }
    _ => {}
  }
}
```

19.2.5 Constant Conditions

```
fn const_cond(L: block) {
  for let (if cond(a, b) goto L1 else L2) in L
    where a.is_constant() && b.is_constant() {
    let c = cond(a, b);
    let L_removed = if c { L2 } else { L1 };
    S.remove(L_removed);
    for phi_func in L {
      if let pred = phi_func.predecessor_removed() {
        phi_func.remove_argument(pred.position);
      }
    }
  }
}
```

19.2.6 Unreachable Code

```
fn try_remove_block(L: block) {
  if !L.predecessors.empty() {
     return
  }
  //block is unreachable
  for S in L.statements() {
    for x in S.vars() {
      x.remove_use_site(S);
    }
  }
  L.statements().clear();
```

```
  for succ in L.successors() {
    let l = succ.predecessors().len();
    succ.predecessors().remove(L);
    let l_after = succ.predecessors().len();
    if l > l_after && l_after == 0 {
      try_remove_block(succ);
    }
  }
}
```

19.2.7 Conditional Constant Propagation

goal: extend simple constant propagation to propagate through conditional branches

impl: use iterative algo. that removes unreachable branches and processes potential constant expressions in a worklist

block not assumed to be executed unless there is evidence that it can do so

variable assumed to be constant unless there is evidence that it can be non-constant

track tuntime value of variable through a few states, $V[var] =$

- $\perp$: not exeutable (default)
- $\langle val \rangle$: assigned with 1 value of val
- $\top$: ≥ 2 different values assigned

variables without definition (I/O, formal parameters, uninitialized var): $V[var] = \top$

$\mathcal{E}[block] =$

- false: no evidence block can be ever executed (default)
- true: there is evidence that block can be executed

block B with 1 successor block C: $\mathcal{E}[C] =$ true

run the algo. with the following observations:

- for any executable assignment $v \leftarrow x\ op\ y$ where $V[x] = c1$ and $V[y] = c2$, set $V[v] \leftarrow c1\ op\ c2$
- for any executable assignment $v \leftarrow x\ op\ y$ where $V[x] = \top$ or $V[y] = \top$, set $V[v] \leftarrow \top$
- for any executable assignment $v \leftarrow \phi(x_1, ..., x_n)$, where $V[x_i] = c1$, $V[x_j] = c2$, $c1 \neq c2$, the ith predecessor is executable, and the jth predecessor is executable, set $V[v] \leftarrow \top$
- for any executable assignment $v \leftarrow MEM()$ or $v \leftarrow CALL()$, set $V[v] \leftarrow \top$
- for any executable assignment $v \leftarrow \phi(x, ..., x_n)$ where $V[x_i] = \top$ and the ith predecessor is executable, set $V[v] \leftarrow \top$
- for any assignment $v \leftarrow \phi(x_1, ..., x_n)$ whose ith predecessor is executable and $V[x_i] = c1$; and for every j either the jth predecessor is not executable, or $V[x_j] = \perp$, or $V[x_j] = c1$, set $V[v] \leftarrow c1$
- for any executable branch if $x < y$ goto L_1 else L_2, where $V[x] = \top$ or $V[y] = \top$, set $E[L_1] \leftarrow true$ and $E[L_2] \leftarrow true$
- for any executable branch if $x < y$ goto L_1 else L_2, where $V[x] = c1$ and $V[y] = c2$, set $E[L_1] \leftarrow true$ or $E[L_2] ¸ true$ depending on $c1 < c2$

19.3 Control Dependence Graph

$x \in DF(y)$ in the reverse CFG
$\iff$ $(\exists z) x$ branches to z and y post-dominates z
$\iff$ y is control dependent on x
$\iff$ CDG has an edge $x \rightarrow y$

Using SSA graph and CDG to answer if A executes before B:

there exists a path $A \rightarrow B$ composed of SSA use-def edges and CDG edges $\implies$ A performed before B

Dead Code elimination:

assume statement is dead unless ther eis evidence that it contributes to result of the program

solve by iteration:

- mark live any statement that ferforms I/O, memory stores, side-effects
- mark live any definition of variable that is used by any live statement (reverse flow problem)
- mark live conditional statement where there exists another live statement that is control dependent on the conditional branch

finally delete all unmarked statements

19.4 Converting back from SSA form

remove ϕ-function and turn back into executable program

insert move $y \leftarrow x_i$ at the end of ith predecessor of the block containing ϕ-function $y \leftarrow \phi(x_1, x_2, ..., x_n)$

use edge split SSA form (unique successor and predecessor property) $\implies$ prevent redendant moves from being inserted

19.5 Register Allocation after SSA Transforms

live range of enumerated variables from SSA may interfere

use coalescing (copy propagation) in register allocation to eliminate move instructions

19.5.1 Live Range Analysis Using SSA

construct interference graph of SSA program prior to converting ϕ-functions to move instructions:

- walk backward from use of variable to definition of variable
- dominance property of SSA $\implies$ algo. always in region dominated by definition of variable
- live range calc. in SSA: use of live-in and live-out of blocks and statements for mutually recursive algo. to build interference graph for each original variable while walking backwards (algo 19.17)

19.6 Functional Intermediate Form

expressions broken down into oprimitive ones with order of eval specified

every intermediate result is an explicitly named temporary

every argument of operator/function is an atom (variable/constant)

every variable has single assignment (binding) only

every use of variable is in the scope of the assignment/binding $\implies$ not require calc. of dominators

relation to SSA form:

- translate from SSA form:
  control flow node with > 1 predecessor becomes a function (arguments are variables corresponding to ϕ-function at the node)
- node f dominates node g $\implies$ function for g nested inside body of function for f
- control flow edge into ϕ containing node is represented by a function call

20 Pipelining

single program instruction level parallelism (ILP)

check to ensure data dependent instructions do not exist in adjacent for enabling issuing ≥ 2 instructions in parallel

techniques:

- dynamic scheduling machine
- superscalar machine
- pipleined machine
- VLIW

optimize for instruction level parallelism with presence of constraints on instruction execution:

- data dependence
- functional unit resources
- instruciton issue unit
- register resources

partial instruction execution parallelism

paseudo-constraints (may be elided by renaming variables): write after write, write after read

loop scheduling without resource bounds

trace scheduling parallel branch execution and conditional move afterward

- optimal for branches of same exeuction cycles
- not valid for branch with side effects

static vs. dynamic H/W supported scheduling

branch preduction

- schedule long-latency, predictable operations
- low latency, short instruction programs: branch prediction is harder and instruction fetch speed is an issue
- superscalar machine:
  - can fetch multiple instr. after branch:
    branch not taken: fetched instr. used immediately
    o/w: instr. stall
  - can assume branch will be taken and fetch instr. at target:
    branch not taken: stall
    o/w: fetch instr. used
  - mix of both instr. fetch from both branches
- static (compiler predicted) vs. dynamic (H/W suiupport for recent frequently executed branch):
  - encode preduction via extra bitfield to the H/W
  - can use various heuristics in frequent branching pattern

21 FP: Lambda Calculus [2]

21.1 Denotionaal Semantics

evaluation of an expression (syntactic object according to rules of language) to a mathematical object: `eval[[expr]] = value`

context dependent variable (use of environment):

```
eval[[x]] p ≡ (p x)
where:
  p is an environment
  p :: variable name -> variable value

eval[[λx.E]] p a
  ≡ eval[[E]] p[x=a]
  ≡ (p[x=a] E)
  where:
  variable x is bound to value a in p
  p[x=a] x = a
  p[x=a] y = p y, x != y
```

21.2 Strictness

f is strict $\iff f\perp = \perp$, where $\perp$ does not halt

f is strict wrt. 2nd parameter $\iff (\forall\ a, b) f\ a\ \perp\ b = \perp$

lazy f as an implemenation of a non-strict f

21.3 Convertibility and Extensional Equality

E_1 and E_2 convertible via lambda calculus
$\implies$ `eval[[E1]] = eval[[E2]]`

F_1 and F_2 extensionally equal
$\iff$ `eval[[F1]] = eval[[F2]]` for all possible arguments

21.4 Enriched Lambda Calculus

enriched lambda calculus with additional constructs: $\square$, pattern matching lambda abstraction, case expression

fatbar operator:

$$\square\ a\ b = \begin{cases} b & ,\ a = Fail \\ a & ,\ otherwise (a = \perp \lor a = Not\ Fail) \end{cases}$$

equivalently:

$$\begin{aligned} \perp \square\ b &= \perp \\ Fail \square\ b &= b \\ a \square\ b &= a,\ a \neq \perp \land a \neq \text{Fail} \end{aligned}$$

let expression semantics:

```
let v = B in E
≡ ((\v.E) B)
```

multiple definitions $\iff$ nested let expressions

using Y (fixed-point) combinator to make definition non-recursive:

```
letrec v = B in E
≡ let v = Y (\v.B) in E
```

if there are multiple definitions, use pattern matching to pack definitions into a product type variable before applying Y

`Y = \h.((\f.(h(f f))) (\f.(h(f f))))`

properties:

```
X = H X
Y H = X
Y H = H X = H(H(..(X)..) = H(H(..(Y H)..)
```

TE: translation scheme for expression,

`TE[[ expr ]] = ..`

where $TE[[..]]$ is a syntactic object and RHS is also syntactic

eg:

```
TE[[ k ]] = k, k is a constant
TE[[ E_1 E_2 ]] = TE[[E_1]] TE[E_2]]
TE[[ func ]] = Func, where Func is a built-in function
```

TD: translation scheme for definition,

`TD[[ v = E ]]` $\equiv$ `v = TE[[ E ]]`

`TD[[ f v1 .. vn = E ]]` $\equiv$ f $= \lambda v_1\ ..\ \lambda v_n$. `TE[[ E ]]`

(lambda abstraction is generated around a body of definition)

recursively apply TE and TD schemes until no more conversions can occur

A program consists of a set of definitions and a top level expression to be evaluated, eg:

```
defs:
f a b = a + b
g x = x * x
---
f a (3+5)
```

translates to:

```
letrec
  TD[[ f a b = a + b ]]
  TD[[ g x = x * x ]]
in
  TE[[ f 2 (3+5) ]]
```

apply TE and TD schemes (recursively):

```
f a b = \a.\b.(+ a b)
g x = \x.(* x x)
in
  f 2 (+ 3 5)
```

translate via TD scheme:

`(\f.(f 2 (+ 3 5))) (\a.\b.(+ a b))`

22 FP: Structured Types

general structured type:

$$
\begin{aligned}
T := \quad & C_1\ T_{1,1}\ ..\ T_{1,r_1} \\
& |\ .. \\
& |\ C_n\ T_{n,1}\ ..\ T_{n,r_n}
\end{aligned}
$$

sum of products $(T_{i,1}\ ..\ T_{i,r_i})$

$n = 1 \implies$ type is a product type (1 constructor)

$n > 1 \implies$ type is a sum type (more than 1 constructor)

Eg:

```
tree * := Leaf * | Branch (tree *) (tree *)
```

type forming operator / type constructor: `tree`
constructor functions: `Leaf, Branch`

sample syntactic shortcuts for value expressions:

```
TE[[ [] ]] = Nil
TE[[ : ]] = Cons
TE[[ (E1, E2) ]] = Pair TE[[E1]] TE[[E2]]
TE[[ [E1, .., En] ]] ≡ Cons TE[[E1]] TE[[ [E2, .., En] ]]
```

where:

```
list * := Nil | Cons * (list *)
```

example type expression: `list *`

23 FP: Pattern Matching

pattern matching abstraction properties:

- overlapping pattern $\implies$ order of pattern eval matters
- nested pattern
- non-exhaustive equations in patterns
- guards on RHS of conditional equations; can be combined with use of pattern maching on LHS of equations
- use of repeated variable sin pattern matches $\iff$ variables are equal on LHS of equation

types of patterns:

- simple variable: v
- constant: k
- sum constructor: $(s\ p_1\ ..\ p_n)$
  where:
  $s \equiv$ a sum constructor
  $p_i \equiv$ patterns
- product constructor: $(t\ p_1\ ..\ p_n)$
  where:
  $t \equiv$ a product constructor
  $p_i \equiv$ patterns

refutable patterns: sum constructor pattern, constant pattern

irrefutable patterns: simple variable pattern, product constructor pattern

23.1 Lambda Abstraction Pattern Matching

binding names present in LHS pattern of definitions are available to components destructured on RHS

```
TD[[ p = R ]] ≡ TE[[ p ]] = TR[[ R ]]
```

where p is a pattern

Eg:

```
f w = x+ y
      where
      (x, y) = w
```

translates to:

```
f = \w.( letrec (Pair x y) = w
         in (+ x y)
       )
```

pattern matching with multiple arguments for function:

$$f\ p_1\ ...\ p_m = E$$

where p_is are patterns and f is a function

This translates to lambda abstraction:

$$f = \lambda v_1\ ..\ \lambda v_m.((((\lambda TE[[p_1]]\ ..\ \lambda TE[[p_m]]).TE[[E]])\ v_1\ ..\ v_m) \\ \square\ Error)$$

where v_is are new unique variables that do not occur in E

TR: translation scheme for right hand side of a definition, eg:

```
TR[[= A1, G1
    = ..
    = An, Gn
where
    D1
    ..
    Dm]]
≡
letrec TD[[D1]]
       ..
       TD[[Dm]]
in (IF TE[[G1]] TE[[A1]]
    (IF TE[[G2]] TE[[A2]]
      ..
       (IF TE[[Gn]] TE[[An]] Fail)..))
```

where A_i is an epxression, G_i is a boolean expression, D_i is a definition

note visiblity of *where* clause is extended over alternatives and guards

eg:

```
f (x:xs) = x, x < 0
  (x:[]) = x
  (x:xs) = f xs

f = \v.(((\(Cons x xs). IF (< x 0) x Fail) v)
       □((\(Cons x Nul). x) v)
       □((\(Cons x xs). f xs) v)
       □Error)
```

24 FP: Semantics of Pattern Matching Lambda Abstraction

`(\p.E)` where p is: variable / constant / sum constructor / product constructor

24.1 Simple Variable Pattern

where $p = v$ is a simple variable

then no-op:

$(\lambda p.E) \Rightarrow (\lambda v.E)$

24.2 Constant Pattern Lambda Abstraction

where k is a constant

```
Eval[[λk.E]] ⊥ = ⊥
Eval[[λk.E]] a = Eval[[E]], if Eval[[k]] = a
Eval[[λk.E]] a = Fail, if Eval[[k]] ≠ a ∧ a ≠ ⊥
```

Possible implementation, with a new lambda abstraction λx

```
λk.E ≡ λx. IF (= x k) E Fail
```

where x does not occur free in E

Example of definition using constant pattern matches:

```
f 0 = 1
f 1 = 0

f = \x.(((\0.1) x)
       □((\1.0) x)
       □Error)

f = \x.(((\y. IF (= y 0) 1 Fail) x)
       □((\y. IF (= y 1) 0 Fail) x)
       □Error)
```

24.3 Sum Constructor Pattern

s is a sum constructor of arity r

```
Eval[[ λ(s p1 .. pr).E ]] (s a1 .. ar)
  = Eval[[ λp1 .. λpr.E ]] a1 .. ar
Eval[[ λ(s p1 .. pr).E ]] (s' a1 .. ar)
  = Fail, s ≠ s'
Eval[[ λ(s p1 .. pr).E ]] ⊥ = ⊥
```

let

```
UnpackSum_s f (s a1 .. ar) = f a1 .. ar
UnpackSum_s f (s' a1 .. ar) = Fail, if s ≠ s'
UnpackSum_s f ⊥ = ⊥
```

then

```
λ(s p1 .. pr).E = UnpackSum_s (λp1 .. λpr.E)
```

eval constructor form but not its components to remain lazy

if constructor pattern matches, then apply arguments and bind names to components of structure with β-reduction rule

naturally handles nested patterns

24.4 Product Pattern Lambda Abstraction

π is a product constructor of arity n

```
Eval[[λ(π p1 .. pn).E]] a
  = Eval[[λp1 .. λpn.E]] (Sel_{π,1}a) .. (Sel_{π,n} a)
```

where $(Sel_{t,i}\ a)$ lazily projects a component:

```
Sel_{t,i}(t a1 .. ar) = ai
Sel_{t,i} ⊥ = ⊥
```

if no component is used, then the structure is not evaluated:

```
Eval[[ \(Pair x y).0 ]] ⊥
= Eval[[ \x.\y.0 ]] (Sel_{Pair,1} ⊥) (Sel_{Pair,2} ⊥)
= Eval[[ \y.0 ]] (Sel_{Pair,2} ⊥)
= 0
```

where by construction: $\text{Sel}_{Pair,i} \perp = \perp$

if struct analysis determines an argument is needed, then use strict product matching instead (eval argument at time of function application)

Possible Implementation:

```
λ(t p1 .. pn).E = UnpackProduct_t (λ p1 .. pn).E

where

UnpackProduct_t f x = f (Sel_{t,1} x) .. (Sel_{t,n} x)

Eval[[λ(t p1 .. pn).E]] a
  = UnpackProduct_t ((λ p1 .. pn).Eval[[E]]) a
```

24.5 Reducing number of Built-in Functions

use of an id/tag to discriminate objects built from different constructors:

(tag, component fields) to represent a structured object

results in a more homogeneous system for functions

loses type info afterwards, therefore apply this after type checking in a type checked language

runtime type checking not possible after these transformations

modify utility functions:

```
UnpackSum_{d,r_s}
PackSum_{d,r_s}: sum constructor
```

where d is the structure tag/id of sum type, r_s is arity of the sum structure

```
UnpackProduct_{r_t}
PackProduct_{r_t}: product constructor
Sel_{r_t,i}: projection of ith component
```

where r_t is arity of product type (no need to have a structure tag)

25 FP: Enriched Lambda Calc. to Ordinary Lambda Calc.

goal: transform enriched lambda calculus into ordinary lambda calculus by getting rid of pattern matching abstractions

- irrefutable let ⇒ simple let
- simple let ⇒ ordinary lambda calculus
- irrefutable letrec ⇒ simple letrec
- irrefutable letrec ⇒ irrefutable let
- general let(rec) ⇒ irrefutable let(rec)

refutable to irrefutable pattern: use conformality check to ensure pattern match succeeds before continuing

- strict: check when let(rec) expression begin
- lazy: check on 1st use of any components of pattern

25.1 Irrefutable Let to Simple Let

introduce new variable, use component projection $\text{Sel}_{t,i}$:

```
let (t p₁ .. pₙ) = B in E
```

⇓

```
let v = B
in let p₁ = Sel_{t,1} v
   ..
   let pₙ = Sel_{t,n} v
   in E
```

25.2 Simple Let to Ordinary Lambda Calculus

```
let v = B in E
```

⇓

```
(λv.E) B
```

25.3 Irrefutable Letrec to Simple Letrec

introduce new variable, use component projection:

```
letrec (t p₁ .. pₙ) = B
       ..  (possibly other definitions)
in E
```

⇓

```
letrec v = B
       p₁ = Sel_{t,1} v
       ..
       pₙ = Sel_{t,n} v
       ..
in E
```

25.4 Irrefutable Letrec to Irrefutable Let

pack definitions into a product type(irrefutable), apply Y combinator to make it non-recursive:

```
letrec p₁ = B₁
       ..
       pₙ = Bₙ
in E
```

⇓

```
letrec (t p₁ .. pₙ) = (t B₁ .. Bₙ) in E
p' ≡ (t p₁ .. pₙ)
B' ≡ (t B₁ .. Bₙ)
letrec p' = B' in E
let p'' = Y (λp'.B') in E
```

25.5 General Let(rec) to Irrefutable Let(rec)

apply pattern matching lambda abstractions to match on specified pattern, add Error clause if no match occurs

```
p = B
```

⇓

```
let v = B
in
(t v₁ .. vₙ) = ((λp.(t v₁ .. vₙ)) v) ▯ Error

where:

v is a new variable

vᵢs are any variables that appear in pattern p

t is a product constructor of arity n

vᵢ ∈ Var(p)

Var(p) = {v}, p is a variable
       = {}, p is a constant
       = ⋃_{i=1}^{n} Var(pᵢ), p is a structured pattern (c p₁ .. pₙ)
```

note recursive definition if p is a constructor pattern

25.6 Other

▯ (fatbar) syntactic sugar: replace it with a prefix built-in function with same semantics as ▯

26 FP: Efficient Compilation of Pattern Matching

compiling function with pattern matching on RHS into efficient case expression:

lambda abstraction → match expression → case expression

rearrange using variable/constructor/empty/mixture rule before compiling into the final case expression:

- variable rule
- constructor rule
- empty rule
- mixture rule

26.1 Preliminary

function definition represented with lambda abstraction:

```
((λp_{1,1} .. λp_{1,n}.E_1) u_1 .. u_n)
 □ ..
 □ ((λp_{m,1} .. λp_{m,n}.E_m) u_1 .. u_n)
 □ Error
```

equivalent to match expression as shorthand:

```
match [u_1 .. u_n]
      [c_1,
       ..
       c_m]
      E_default
where:
c_i is a pattern matching clause ([p_{i,1}, .., p_{i,n}], E_i)

=

match [u_1 .. u_n]
  [([p_{1,1}, .., p_{1,n}], E_1),
   ..
   ([p_{m,1}, .., p_{m,n}], E_m)]
   E_default
```

26.2 Empty Rule

```
match [] [([], E_1),
          ..
          ([], E_m)]
         E_default

⇓

E_1 □ .. □ E_m □ E_default
```

26.3 Variable Rule

all equations have list of patterns that begins with variable

```
let (u:us) = [u_1 .. u_n]

match (u:us) [((v_1:ps_1), E_1),
              ..
              ((v_m:ps_m), E_m)]
             E_default

⇓ Apply β-reduction(sub u at occurence of v_i in E_i)

match us [((ps_1, E_1[u/v_1]),
           ..
           (ps_m, E_m[u/v_m])]
         E_default
```

26.4 Constructor Rule

all equations begin with same constructor

group them if necessary: equation exchangeable iff relative order of equations with same constructor remains the same

case expression is generated for each group associated with a same type of pattern

missing constructors ⟹ insert error clause into case expression using empty rule: `match [] [] Error`

```
match (u:us) (qs_1 ++ .. ++ qs_k) E_default

where:

qs_i (grouping of same constructor) ≡
  [((((c_i ps'_{i,1}):ps_{i,1}), E_{i1}),
    ..
    (((c_i ps'_{i,m_i}):ps_{i,m_i}), E_{im_i})]

p'_{s_i,j} is a constructor pattern p_a p_b .. p_r for constructor c_i

⇓ transform to case expression

case u of
c_1 us'_1 ⇒ //group 1
  match (us'_1 ++ us)
        [((ps'_{1,1} ++ ps_{1,1}), E_{1,1}),
         ..
         ((ps'_{1,m_1} ++ ps_{1,m_1}), E_{1,m_1})]
        E_default
..
c_k us'_k ⇒ //group k
  match (us'_k ++ us)
        [((ps'_{k,1} ++ ps_{k,1}), E_{k,1}),
         ..
         ((ps'_{k,m_k} ++ ps_{k,m_k}), E_{k,m_k})]
         E_default

where:

us'_i is a list of unique variables destructured from
  constructor c_i
qs_i corresponds to grouping of clauses with same constructor
  at head of list
ps_{i,j} is the remaining patterns of a clause
```

example:

```
g f [] ys = []
g f (x:xs) [] = []
g f (x:xs) (y:ys) = (f x y):(g f xs ys)

⇓ transforms to

g = λu_1.λu_2.λu_3.
      case u_2 of
        Nil ⇒ Nil
        Cons u_4 u_5 ⇒
          case u_3 of
            Nil ⇒ Nil
            Cons u_6 u_7 ⇒ (u_1 u_4 u_6):(g u_1 u_5 u_7)
```

26.5 Mixture Rule

equation with constructor and variable at head of pattern matching list

use a cascade of matches where each section only has equations beginning with the same kind of pattern type (either variable or constructor)

```
match us qs E_default

⇓

//partition pattern clauses into groups
let (qs_1 ++ .. ++ qs_k) = qs

where:

qs_i is a list containing only equations that begins
  with variable or constructor but not both:

qs_i = [([p_1,1 , .., p_1,n], E_1),
        ..
        ([p_m,1 , .., p_m,n], E_m)]
(∃t ∈ {Var,Constructor})(∀i) p_i,1 is a pattern of type t

⇓

match us
      qs_1
      (match us
             qs_2
             ..
                      (match us
                             qs_k
                             E_default ..))
```

apply rules to transform match to case expression for each section:

```
us is empty ⇒ apply empty rule
      o/w   ⇒ all equations begin with:
                 variable ⇒ apply variablwe rule
                 constructor ⇒ apply constructor rule
                 mixture ⇒ apply mixture rule
```

26.6 Optimization with Multiway Jump / Case Expressions

constant time eval of constructor type to select a clause

```
case t of
  c_1 v_1,1 .. v_1,r_1 ⇒ E_i
  ..
  c_n v_n,1 .. v_n,r_n ⇒ E_n
```

where:

- patterns are exhaustive and not nested
- $v_{i,j}$ is a variable
- c_is are constructors that cover the structured type

this is logically equivalent to the sequence:

$$((\lambda(c_1\ v_{1,1}\ ..\ v_{1,r_1}).E_1)\ v)$$
$$\square\ ...$$
$$\square\ (\lambda(c_n\ v_{n,1}\ ..\ v_{n,r_n}).E_n)\ v)$$

eval during pattern matching possible if argument is determined to be strict

possible implementations:

- using ordinary lambda calculus:
  use UnpackProduct_t / UnpackSum_{s_i}
- avoid lambda abstraction:
  use $\text{Sel}_{t,i}$ instead of UnpackProduct
  use SelSum_{s_i,r_i} instead of UnpackSum

26.6.1 Example impl. of case expression with product type using $\text{Sel}_{t,i}$

```
case v of
  t v_1 .. v_r ⇒ E

⇓

let v_1 = Sel_1 v
    ..
    v_r = Sel_r v
in E
```

26.6.2 Example impl. of case expression with sum type using tagged enumeration

```
case v of
  s_1 v_1,1 .. v_1,r_1 ⇒ E_1
  ..
  s_n v_n,1 .. v_n,r_n ⇒ E_n

⇓

case_T v //use enumeration of v to select one of the arguments
  (let v_1,1 = Sel_r_1,1 v
       ..
       v_1,r_1 r = Sel_r_1,r_1 v
   in E_1)
  ..
  (let v_n,1 = Sel_r_n,1 v
       ..
       v_n,r_n r = Sel_r_n,r_n v
   in E_n)

where:

T is a sum type

case_T (s_i a_1 .. a_r_i) b_1 .. b_i .. b_n = b_i
case_T ⊥ b_1 .. b_n = ⊥

SelSum_n,i where n is arity of target constructor
```

case_T selects 1 of the arguments using constructor of 1st argument instead of applying 1 of its arguments to components of 1st argument; avoids lambda abstractions in favour of let expressions when transforming to ordinary lambda calculus

27 FP: Dependency analysis of definitions

SSC + dependeing sort ⇒ regroup into let and letrec groups

- replace letrec with let whenever possible for: efficient impl, easier typechecking
- separate into groups of mutually recursive definitions (SSC in graph)
- coaslesce SSC into single node
- singleton node corresponds to non-recursive definition
- perform dependecy anlaysis (topo. sort) on resulting acyclic graph and output ordering
- nodes with > 1 variables: use letrec, otherwise use let
- nesting of scopes implies dependency of definitions
- nodes are nested according to dependency analysis (group A(a1, a2) is in a nested scope of B(b) if definitions of A uses B):

```
let b = ..
in ..
   let a1 = ..
       a2 = ..
   in ..
```

28 FP: List Comprehension

```
[ expr | qualifier₁; ..; qualifierₙ ]
```

where qualitfier can be of one of the following types:

- p ← generator where p is a refutable pattern
- boolean predicate
- empty

shorthand: $x, y \leftarrow z \iff x \leftarrow z; y \leftarrow z$

28.1 reduction rules assuming generator qualifier pattern is a variable

```
//generator qualifier
[ E | v ← []; QS ] ⇒ [] //early termination

[ E | v ← H:TS; QS ] ⇒
  [ E | QS ][H/v] ++ [ E | v ← TS; QS ]

//boolean qualifier
[ E | False; QS ] ⇒ [] //early termination

[ E | True; QS ] ⇒ [ E | QS ]

//empty qualifier
[ E | ] ⇒ [ E ]

where:

: ≡ Cons

QS is a list of qualifiers

[ E | QS ][H/v] processes current selected element from
  current generator qualifier by substituting H for v
  in [ E | QS ]

[ E | v ← TS; QS ] selects another element from current
  generator qualifier to process
```

example with lazy evaluator implementation:

```
[ square x | x ← [1 2 3]; odd x ]
→ [ square x | odd x ][1/x] ++ [ square x | x ← [2 3]; odd x ]
→ [ square 1 | odd 1 ] ++ [ square x | x ← [2 3]; odd x ]
→ [ square 1 | True ] ++ [ square x | x ← [2 3]; odd x ]
→ [ square 1 | ] ++ [ square x | x ← [2 3]; odd x ]
→ [ square 1 ] ++ [ square x | x ← [2 3]; odd x ]
→ [ 1 ] ++ [ square x | x ← [2 3]; odd x ]
..
→ [ 1 ] ++ [ 2 ] ++ [ square x | x ← [3]; odd x ]
→ [ 1 2 ] ++ [ square x | x ← [3]; odd x ]
..
→ [ 1 2 3 ]
```

28.2 Translation to Lambda Calculus

```
TE[[ [ E | v ← L; QS ] ]] =
  flatmap (λv.TE[[ [ E | QS ] ]]) TE[[ L ]]

TE[[ [ E | B; QS ] ]] = IF TE[[ B ]] TE[[ [ E | QS ] ]] Nil

TE[[ [ E | ] ]] = Cons TE[[ E ]] Nil

where:

flatmap f [] = []
flatmap f x:xs = (f x) ++ (flatmap f xs)

E: expression
B: boolean expression
L: list valued expression
Q: sequence of qualifiers
v: a variable
```

28.3 Efficiency Improvements with Inplace Expansion

replace with enriched lambda calculus:

```
flatmap (λv.E) L
```

⇓

```
letrec h = λus.case us of
  Nil ⇒ Nil
  Cons v us' ⇒ Append E (h us')
in (h L)
```

then,

```
TE[[ [ E | v ← L; Q ] ]] =
  flatmap (λ.TE[[ [ E | Q ] ]]) TE[[ L ]]
```

⇓

```
TE[[ [ E | v ← L; Q ] ]] =
  letrec h = λus.case us of
    Nil ⇒ Nil
    Cons v us' ⇒ Append TE[[ [ E | Q ] ]] (h us')
  in (h TE[[ L ]])
```

28.4 Efficiency Improvements by Reducing Number of Cons operations

let

```
TQ[[ [ E | v ← L1; QS ] ++ L2 ]] ≡
  letrec h = λus.case us of
    Nil ⇒ TE[[ L2 ]]
    Cons v us' ⇒ TQ[[ [ E | QS ] ++ (h us') ]]
  in (h TE[[ L1 ]])

TQ[[ [ E | B; QS ]] ++ L ]] ≡
  IF TE[[ B ]] TQ[[ [ E | QS ]] ++ L ]] TE[[ L ]]

TQ[[ [ E | ]] ++ L ]] ≡ Cons TE[[ E ]] TE[[ L ]]
```

28.5 Pattern Matching in List Comprehension

apply pattern matching lambda abstraction

if the pattern does not match in generator qualifier, then skip the element

reduction rules:

```
[ E | p ← []; QS ] → []
[ E | p ← H:T; QS ] →
  (((λp.[ E | QS ]) H) □ []) ++ [ E | p ← T; QS ]
```

in the case of pattern being a simple variable then it will be irrefutable and lambda pattern matching abstraction can be simplified:

```
(((λv.[ E | QS ]) H) □ [])
⇒ (([ E | QS ][H/v]) □ []) //β-reduction
⇒ ([ E | QS ][H/v])
```

28.6 Modifications to Translation Scheme

```
TE[[ [ E | p ← L; QS ] ]] =
  flatmap
  (λu.(((λTE[[ p ]].TE[[ [ E | QS ] ]]) u) □ Nil))
  TE[[ L ]]

TE[[ [ E | v ← L; QS ] ]] =
  flatmap (λv.TE[[ [ E | QS ] ]]) TE[[ L ]]

TE[[ [ E | B; QS ] ]] =
  IF TE[[ B ]] TE[[ [ E | QS ] ]] Nil

TE[[ [ E | ] ]] = Cons TE[[ E ]] Nil
```

where v is a simple variable

note u is a new unique variable introduced

28.7 Modifications of TQ scheme

```
TQ[[ [ E | q ← L1; QS ] ++ L2 ]] ≡
  letrec h = λus.case us of
    Nil ⇒ TE[[ L2 ]]
    Cons v us' ⇒
      ((λTE[[ q ]].TQ[[ [ E | QS ] ++ (h us') ]]) u) □ (h us')
  in (h TE[[ L1 ]])

TQ[[ [ E | B; QS ]] ++ L ]] ≡
  IF TE[[ B ]] TQ[[ [ E | QS ]] ++ L ]] TE[[ L ]]

TQ[[ [ E | ]] ++ L ]] ≡ Cons TE[[ E ]] TE[[ L ]]
```

note h, u, us, us' are new unique variables introduced

29 FP: Type Checking

goals of type inference:

- check if program is well typed
- determine type of any expression in the program

kinds of polymorphism:

- ad-hoc: impl. conditioned on input types
- parametric, eg: `forall types a, [a] -> num`

object kinds wrt. expressions with schematic (generic) type variables:

- polymorphic: may take on different types at different occurences
- monomorphic: no schematic/generic type variables

intermediate language for constructing a type checker: lambda calculus; do this before transform to supercombinator program or lambda lifting; do dependency analysis before this

- variables
- lambda abstractions
- applications
- simultaneous definitions / let-expressions
- mutual recursion / letrec-expressions

note: pattern matching function definitions replaced by case functions with type-forming operations; alternatively impl: typchecking can still be done when pattern matches are present

deduction of form of types that an expression can take via 2 routes:

- type required by the surrounding context (deduction from outside)
- type object can take (deduction from inside)

two type expressions from the above deductions must match for the expression to be well typed

29.1 Finding Types

analyze type structure of an expression using an upsidedown tree:

- variables and constants at the top
- internal nodes: constructors applied to form expressions
- each node corresponds to a subexpression and has a type

eg:

```
\x\y\z.x z (y z)
```

$\Updownarrow$

```
x::T0 z::T1   y::T2 z::T3
-----------@  -----------@
    T4            T5
    ---------------@
           T6
           -- \x.\y.\z.
           T7

T0 = T1 -> T4
T2 = T3 -> T5
T4 = T5 -> T6
T7 = T0 -> T2 -> T1 -> T6
T1 = T3
---
T0 = T1 -> T5 -> T6
T2 = T1 -> T5
T4 = T5 -> T6
T7 = (T1 -> T5 -> T6) -> (T1 -> T5) -> T1 -> T6
```

rules:

- all occurences of a λ-bound variable have the same type
- 2 compound types are equivalent $\Longrightarrow$ their constituent parts are equivalent and formed wtih same construction:

  `T1 -> T2 = T1' -> T2'` $\Longrightarrow$ `T1 = T1' and T2 = T2'`

- `(f a) :: V where a::A, f::A -> V`
- in order for expression to be well typed, it has to work in any surrounding context

occurences of defined names in expression body are instances of the type of the corresponding RHS of the defined names. Eg, instantiate variables in types of those RHS with new distinct variables during inference.

an approach for letrec (mutually recursive expressions):

- during definitions of defined variables, all occurences of defined variables must share same type of RHS of their definitions
- after definitions (during use), each occurence of defined variables are polymorphic and can be instantiated differently to fit constraints of local contexts

30 FP: Type Checker Construction

30.1 Definitions

```
subst := tvname -> type_exp

scomp :: subst -> subst -> subst
scomp sub2 sub1 = \tvn -> sub_type sub2 (sub1 tvn)
scomp sub2 sub1 tvn = sub_type sub2 (sub1 tvn)
property:
sub_type (scomp phi psi) =
  (sub_type phi) . (sub_type psi)

type_exp := TVAR tvname | TCONS [char] [type_exp]

sub_type :: subst -> type_exp -> type_exp
sub_type phi (TVAR tvn) = phi tvn
sub_type phi (TCONS tcn ts) =
  TCONS tcn (map (sub_type phi) ts)

id_subst :: subst
id_subst tvn = TVAR tvn
property:
sub_type id_subst t = t
```

30.1.1 Delta Substitution

delta substitution: a substitution that affects only one variable:

```
delta :: tvname -> type_exp -> subst
delta tvn t = \tvn' -> = t, if tvn = tvn'
                       = TVAR tvn', otherwise

delta tvn t tvn' = t, if tvn = tvn'
                 = TVAR tvn'
```

30.1.2 Fixed Point

a type expression t is a fixed point of a substitution phi if:

`sub_type phi t = t`

`t := (TVAR x)` is a fixed point of substitution phi $\Longrightarrow$
x is unmoved by phi

30.1.3 Idempotent Substitution

a substitution, phi, is idempotent if: `scomp phi phi = phi` or `(sub_type phi) . (sub_type phi) = sub_type phi`

fully worked out substitution is idempotent (unless there is circularity which is an error)

phi is idempotent and phi moves tvn $\Longrightarrow$
`sub_type phi (VAR tvn)` is a fixed point of phi and cannot contain tvn

30.2 Unification

given an equation: `t1 = t2`, solve the equation by finding a subsitution phi such that: `sub_type phi t1 = sub_type phi t2`

then, phi is a unifier of t1 and t2; extend this to a set of equations

idea: find a maximally general solution substitution phi, then find `psi = rho scomp phi` where psi is an extension of phi (phi is no less general than psi)

30.3 Robinson's Unification Algo.

given a substitution, phi, that already solves a set of equations, $t_1 = t'_1, .., t_n = t'_n$, extend the system to add one additional equation $t_{n+1} = t'_{n+1}$. Extend idempotent substituion phi by phi' where phi' is an idempotent unifier of $t_{n+1} = t'_{n+1}$ by returning (Ok phi'). If there is no extension of phi which unifies $t_{n+1} = t'_{n+1}$, return Failure.

```
unify :: subst -> (fype_expr, type_exp)
  -> Reply subst
unify phi ((TVAR tvn),t)
  = extend phi tvn phit, if phitvn = TVAR tvn
  = unify phi (phitvn,phit) //guaranteed tvn cannot
                            //occur in phitvn or phit
  where
    phitvn = phi tvn
    phit = sub_type phi t

unify phi ((TCONS tcn ts),(TVAR tvn))
  = unify phi ((TVAR tvn),(TCNS tcn ts))

unify phi ((TCONS tcn ts),(TCONS tcn' ts'))
  = unifyi phi (ts zip ts'), if tcn = tcn'
  = Failure

Reply a = Ok a | Failure

unifyi :: subst -> [(fype_exp, fype_exp)]
  -> Reply subst
unifyi phi eqns = foldr unify' (Ok phi) eqns
  where
    unify' eqn (Ok phi) = unify phi eqn
    unify' eqn Failure = Failure

type_exp := TVAR tvname | TCONS [char] [type_exp]

extend :: subst -> tvname -> type_expr
  -> Reply subst
extend phi tvn t = Ok phi, if t = TVAR tvn
                 = Failure, if tvn in tvars_in t
                 = Ok ((delta tvn t) scomp phi)

subst := tvname -> type_exp

delta :: tvname -> type_exp -> subst
delta tvn t tvn' = t, if tvn = tvn'
                 = TVAR tvn'

scomp :: subst -> subst -> subst
scomp sub2 sub1 tvn = sub_type sub2 (sub1 tvn)

sub_type :: subst -> type_exp -> type_exp
sub_type phi (TVAR tvn) = phi tvn
sub_type phi (TCONS tcn ts) =
  TCONS tcn (map (sub_type phi) ts)

tvars_in ::type_exp -> [tvname]
tvars_in t = tvars_in' t []
  where
    tvars_in' (TVAR x) l = x:l
    tvars_in' (TCONS y ts) l =
      foldr tvars_in' l ts
```

30.4 Tracking Types During Type Checking

when type checking free variables in expressions

30.4.1 Occurences

look at either occurences of free variables in the expression:

free variables are either builtin functions or functions associated with the type definitions

types deduced for each occurence of free variables can be instances of supplied by the constraint system

associate a type expression with each occurence of a free variable in an expression

30.4.2 Variables

type check definition variables, then check body expression

at each occurence of a defined variable, construct an instance of its associated type (using a type scheme / template) and substitude newly (uniquely) generated type variables for schematic type variables while copying non-schematic type variables

schematic type variables of the type scheme of a variable are those that are freely instantiated in order to satisfy the type constraint of the variable in its surrounding context at each occurence of variable

non-schematic type variables are constrained and not newly instantiated, thus only copied

possible representation of type scheme in a type checker:

```
type_scheme ::= SCHEME [tvname] type_exp

// type variable for a type scheme is unknown if
// it does not occur in [tvname] of SCHEME
unknowns_scheme :: type_scheme -> [tvname]
unknowns_scheme (SCHEME scvs t) = tvars_in t bar scvs

bar :: [*] -> [+] -> [+]
bar xs ys = [ x <- xs | !(x in ys) ]
in :: * -> [*] -> bool
in x' [] = False
in x' (x:xs) = True, x=x'
             = x' in xs, otherwise
```

during type checking, substitution occurs on type scheme to further constraint its unknown type variables when more info. is gathered

```
sub_scheme :: subst -> type_scheme -> type_scheme
sub_scheme phi (SCHEME scvs t)
  = SCHEME scvs (sub_type (exclude phi scvs) t)

where

exclude phi scvs tvn = TVAR tvn, tvn in scvs
                     = phi tvn, otherwise
```

associate a type scheme with each free variable in an expression

use of a suitable container to:

- map free variables of expression to type schemes
- determine range of mapping

build a solution (a substitution phi) for type equations that are implied by an expression, eg for let expression:

(let x = E in E')

with type environment:
x_1 :: ts_1
..
x_n :: ts_n

find phi such that:
E :: t
x_1 :: ts'_1
..
x_n :: ts'_n

where ts'_i is the image ts_i under substitution phi

then, in the extended environment, form the type scheme ts associated with x when type checking E':

x_1::ts'_1, .., x_n::ts'_n, x::ts

use of an association list as a container for storing types of free variables in an expression

passing info. to type checker via `type_env` object

```
type_env == assoc_list vname type_scheme

assoc_list a b == [(a,b)]

unknowns_te :: type_env -> [tvname]
unknowns_te gamma = appendlist (map unknowns_scheme (rng gamma))
appendlist :: [ [a] ] -> [a]
appendlist lls = foldr (++) [] lls
sub_te :: subst -> type_env -> type_env
sub_te phi gamma
  = [ (x,sub_scheme phi st) | (x,st) <- gamma ]
```

30.5 An Impl.

30.5.1 type checking lists of expr

```
tcl :: type_env -> name_supply -> [vexp]
  -> reply (subst, [type_exp])
tcl gamma ns [] = OK (id_subst,[])
tcl gamma ns (e:es)
  = tcl1 gamma ns0 es (tc gamma ns1 e)
    where (ns0,ns1) = split ns
tcl1 gamma ns es FAILURE = FAILURE
tcl1 gamma ns es (OK (phi,t))
  = tcl2 phi t (tcl gamma' ns es)
    where gamma' = sub_te phi gamma
tcl2 phi t FAILURE = FAILURE
tcl2 phi t (OK (psi,ts))
  = OK (psi scomp phi, (sub_type psi t) : ts)
```

30.5.2 type checking variables

type checking variable x

construct phi substitution that solves e1 and e2

if t1 and t2 are derived for e1 and e2, then create an extension of phi with additional constraint:

`t1 = t2 -> t'` where t' is a new type variable

apply unification to `t1` with `t2 -> t'` to obtain the extension

```
tcvar :: type_env -> name_supply -> vname
  -> reply (subst,type_exp)
tcvar gamma ns x
  = OK (id_subst, newinstance ns scheme)
    where scheme = val gamma x

where

newinstance :: name_supply -> type_scheme -> type_exp
newinstance ns (SCHEME scvs t)
  = sub_type phi t
  where al = sevs $zip (name_sequence ns)
        phi = alto_subst al
```

30.5.3 type checking application

`(AP e1 e2)`

```
tcap :: type_env -> name_supply -> vexp -> vexp
  -> reply (subst,type_exp)
tcap gamma ns e1 e2
  = tcap1 tvn (tcl gamma ns' [e1,e2])
    where tvn = next.name ns
          ns' = deplete ns
tcap1 tvn FAILURE = FAILURE
tcap1 tvn (OK (phi,{t1,t2]))
  = tcap2 tvn (unity phi (t1,t2 $arrow (TVAR tvn)))
tcap2 tvn FAILURE = FAILURE
tcap2 tvn (OK phi) = OK (phi, phi tvn)
```

30.5.4 type checking lambda abstraction

`(LAMBDA x e)`

```
tclambda :: type_env -> name_supply -> vname
  -> vexp -> reply (subst,type_exp)
tclambda gamma ns x e
  = tclambda1 tvn (tc gamma' ns' e)
    where ns' = deplete ns
          gamma' = new_bvar (x,tvn) : gamma
          tvn = nextmame ns
tclambda1 tvn FAILURE = FAILURE
tclambda1 tvn (OK (phi,Q)
  = OK (phi, (phi tvn) Sarrow t)
new_bvar (x,tvn) = (x,SCHEME [] (TVAR tvn))
```

30.5.5 type checking let-expressions

`(LET xs es e)`

general steps:

1. type check es
2. update environment in xs (associates name with appropriate schematic types)
3. type check e

```
tclet :: type_env -> name_supply
  -> [vname] -> [vexp] -> vexp
  -> reply (subst, type_exp)
tclet gamma ns xs es e
  = tcleti gamma ns0 xs e (tcl gamma ns1 es)
    where (ns0,ns1) = split ns
tclet1 gamma ns xs e FAILURE = FAILURE
tclet1 gamma ns xs e (OK (phi,ts))
  = tclet2 phi (tc gamma' ns1 e)
    where gamma'' = add_decls gamma' ns0 xs ts
          gamma' = sub_te phi gamma
          (ns0,ns1) = split ns
tclet2 phi FAILURE = FAILURE
tclet2 phi (OK (phi',t))
  = OK (phi' scomp phi, t)

add_decls :: type_env -> name_supply
  -> [vname] -> [type_exp] -> type_env
add_decls gamma ns xs ts
  = (xs zip schemes) ++ gamma
     where schemes = map (genbar unknowns ns) ts
           unknowns = unknowns_te gamma
genbar unknowns ns t = SCHEME (map snd al) t'
  where al = scvs $zip (name_sequence ns)
        scvs = (nodups (tvars_in t)) $bar unknowns
        t' = subtype (alto_subst al) t

nodups :: [a] -> [a]
nodups xs = f [] xs
            where
            f ace [] = acc
            f acc (x:xs) = f acc xs, x in acc
                         = f (x:acc) xs
```

30.5.6 type checking letrec-expressions

`(LETREC xs es e)`

general steps:

1. Associate new type schemes with the variables xs. These schemeswill have no schematic variables, in accordance with our decision to insist that all occurrences of a defined name in the right-hand sides of a recursive definition should have the sametype.
2. Type-check the right-hand sides. If successful, this will yield a substitution and a list of types which may be derived for the right-hand sides if the type environmentis constrained by the substitution.
3. Unify the types derived for the right-handsides with the types associated with the correspondingvariables, in the context of that substitution. This is in accordance with our decision that the right-hand sides of recursive definitions must receive the same types as occurrences of the

corres- ponding variables. Should the unification succeed, that constraint can be met.

4. We are now in muchthe samesituation as we were in with expressions of LET form, when the definitions had been processed, and it remained to type-check the body e, after updating the type environment with appropriate schematic types.

```
tcletrec :: type_eny -> name_supply
  -> [vname] -> [vexp] -> vexp
  -> teply (subst, type_exp)
tcletrec gamma ns xs es e
  = tcletrec1 gamma ns0 nbvs e
      (tc] (nbvs ++ gamma) ns1 es)
    where (ns0,ns') = split ns
          (ns1,ns2) = split ns'
          nbvs = new_bvars xs ns2

new_bvars xs ns
  = map new_bvar (xs zip (name_sequence ns))

tcletrec1 gamma ns nbvs e FAILURE = FAILURE
tcletrec1 gamma ns nbvs e (OK (phi, ts))
  = tcletrec2 gamma' ns nbvs' e (unifyl phi (ts zip ts'))
  where ts' = map old_bvar nbvs'
        nbvs' = sub_te phi nbvs
        gamma' = sub_te phi gamma

old_bvar (x,SCHEME [] t) = t

tcletrec2 gamma ns nbvs e FAILURE = FAILURE
tcletrec2 gamma ns nbvs e (OK phi)
  = tclet2 phi (tc gamma'' ns1 e)
    where ts = map old_bvar nbvs'
          nbvs' = sub_te phi nbvs
          gamma' = sub_te phi gamma
          gamma'' = add_decls gamma' ns0 (map fst nbvs) ts
          (ns0,ns1) = split ns
```

31 FP: Program representatoin

give a representation of lambda expression in memory

graph reduction: transforms syntax tree (expression to be evaluated) to a graph (possibly cyclic)

data structure for discriminating different types via tag fields:

- structure tags for data objects
- system tags for system objects (application, lambda abstraction ,builtin ops, ..

boxed vs. unboxed representation: possibly implmented using pointer bit in field to discriminate these

tag bits in field to support runtime type info

32 FP: Selecting Next Redex to be Reduced

an evaluator performs reduction on the graph representation of the program to reduce the graph to a normal form

reduction needs to select which redex to be reduced

call by value (eager/strict) vs. call by need (lazy)

call by need:

- arguments to function evaluated only if value is needed, not at time of function application
- any evaluation performed only once maximum and used afterwards
- implementable via normal order reduction (order: leftmost outermost redex first), eg: select function application to reduce first before reducing arguments to the application

strict sementics uses applicative order reduction: reducing argument to lambda expression before reducing application of lambda expression to the argument

32.1 Weak Head Normal Form

sufficient to eval all top level variables (via normal order reductions) such that there is no more top level redexes; inner nested components of data structures may have redexes remaining

a lambda expression is in WHNF $\iff$

```
F E₁ .. Eₙ, n ≥ 0 and
( F is a variable / data object, or
  F is a lambda abstraction / builtin function and
    (∀m ≤ n)(F E₁ .. Eₘ) is not a redex)
```

property of WHNF: expression has no top level redex

normal form: inner redexes also need to be reduced; implies WHNF

apply normal order reduction to top level redexes to get WHNF

continue apply normal order reduction of inner redexes to get normal form

by construction, top level reduction can bever involve free variables:

- arguments of redex have no free variables
- name capture problem can never arise

32.2 Find Next Redex at Top Level

builtin function or lambda abstraction with too few arguments or no argument $\implies$ in WHNF

data object $\implies$ in WHNF

builtin function with enough arguments $\implies$ $(f\ E_1\ ..\ E_n)$ selected as outermost redex

lambda abstraction with ≥ 1 arguments available $\implies$ $(f\ E_1)$ is next redex

32.3 Normal Order Reduction Using Tree

unwinding the spine (traverse leftmost branch downward)

vertebrae: node (eg: application node)

rib: argument

get arguments during the process of unwinding, eg: explicit stack or pointer reversing impl.

spine stack: save pointers to vertebrae; number of arguments = depth of stack

rewinding the spine

33 FP: Reducing a Redex

as a local transformation of the graph representing the expression to be reduced

assuming the redex is selected for reduction, then a lambda abstraction or builtin function is located at the tip of the spine

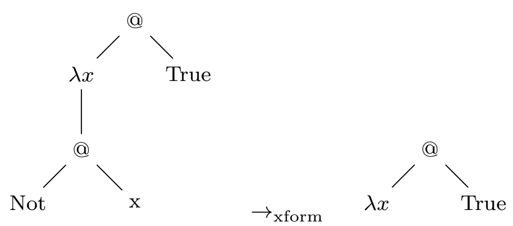
33.1 Reduction of Application with Lambda Abstraction

lambda abstraction applied to an argument

use β-reduction rule

create an instance of lambda abstraction's body

substitute argument for free occurences of the target formal parameter in the body of the lambda abstraction



$\Longleftrightarrow$

(λx.Not x) True → (Not True)

sharing of redex and lambda abstraction: create new instances and do modification on those

large arguments: use pointers instead when doing substitution to formal parameters

rewrite root of redex with result ensures expressions shared are reduced only once

subcomponents of redex may be detatched afterward from the graph for garbage collection

33.2 Preserving Original Lambda Abstraction for Reuse by Other Parts of the Program

create a new instance of its body when the abstraction is used

detatch the template lambda abstraction from local graph after instantiation

eg, use a helper utility:

```
instantiate(Body, Var, Value) ≡ Body[Value/Var]
```

recursive function: apply case analysis for how to apply substitution to lambda abstraction

33.3 Lazy Graph Reduction

eval by need: use normal order evaluation to get WHNF $\implies$ arguments to function eval'd only if they are needed

eval same expression only once:

- substitute pointers instead of raw arguments to formal parameters $\implies$ deduplicate unevaluated arugment
- update redex root with result $\implies$ evaluated redex cached for future uses to deduplicate works of reevaluating

33.4 Reduction of Application with Builtin Function

1. recursively eval arguments by evaluator and reset root of redex to current redex under consideration
2. eval function
3. replace root of redex with result

33.5 Indirection Nodes

- unboxed objects update
- update where body of lambda abstraction is a single variable

a solution to indirection such that no sharing is lost: eval result to WHNF before updating root of redex

using indirection node to result vs. copying root of result of root of redex in the case of the root of result is not constructed during reduction

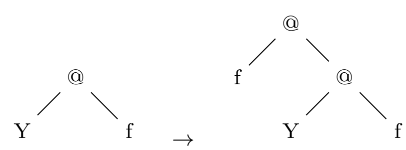
- needed if result if unboxed object
- no issue if root of result is bigger than root of redex
- but need additional tests for indirectionality and deref as they are encountered leading to slowness

33.6 Impl. of Y Combinator

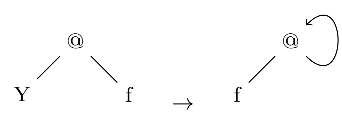
by definition:

```
T f →reduces to f ( Y f)
```

possible approaches:



or



2nd approach wuth cyclic graph:

- con: cycle is an potential issue for GC with ref counting
- pro: use finite representation in storage to represent infinite object (recursive function, infinite data structure)

34 FP: Supercombinator

goal: transform lambda expression into a form such that lambda abstractions are easier to instantiate

a possible approach via lambda lifting where lambda abstractions are transformed into supercombinators

34.1 Combinator

definition: a lambda expression which contains no occurences of free variable (eg: a pure function)

supercombinators(further restrictions on top of being combinators) $\in$ combinators $\in$ lambda expressions

previously, instantiation done through recursive tree walk over over the lambda body

alternative: compilation to a fixed sequence of instructions to construct instance of lambda body $\implies$ instantiation of lambda body becomes following instruction sequence associated with it

dealing with free variables in lambda body:

- access values of free variables via environment by code sequence associated with the lambda abstraction
- alternative: transform program where lambda abstractions can be easily compiled $\implies$ does not require additional environment

modified β-reduction by perform serveral β-reductions at once:

- less intermediate structure generated
- following algo. of performing normal order reduction until WHNF is reached $\implies$ no work can be done on inner lambda abstraction until it is given another argument

34.2 Supercombinator

lambda expression of the form:

```
λx₁. .. λxₙ.E
```

where

- $n \geq 0$
- E is not a lambda abstrction
- no free variables exist
- any lambda abstraction in E is a supercombinator

amenable for multi-argument reduction

34.3 Supercombinator of Arity $n > 0$

unit of compilation

no free variables $\implies$ can compile a fixed code sequence

any lambda abstraction in body have no free variables $\implies$ no copying when instantiating supercombinator body

34.4 Supercombinator of Arity 0 / Constant Applicative Form

known as CAF

can still be a function

no free variables and no λs at front $\implies$

- never instantiated
- no code need to be compiled
- 1 instance of its graph can be shared

34.5 Supercombinator Creation

represent a program with:

- a set of supercombinator definitions
- an expression to be evaluated

supercombinator reduction takes place when all required arguments are present

implementation aspects:

- algo. to translate all lambda abstractions into supercombinators, eg: lambda lifting
- an implemenation of supercombinator reduction

34.6 Lambda Lifting

make each free variable into an extra parameter (abstracting free variable) $\iff$ β-abstraction (inverse of β-reduction)

eg:

```
(λy. + y x) ⇒ (λw.λy. + y w) x
```

supercombinator notation:

```
$F = λW.λY. + Y W
⇔ $F W Y = + Y W (shorthand form)
```

where `$F` is an arbitrary unique name given to a supercombinator

select a lambda abstraction, where there is no inner lambda abstractions in its body, to transform to supercombinator

apply this algo. until there is no more lambda abstractions

in the final form of the transformed program, the expression has no free variables since it is a top level expression $\implies$ make it a CAF (0-parameter supercombinator)

eg:

```
original program:
(λx.λy. - y x) 3 4

supercombinator definitions:
$XY x y = - y x
..
$Prog = $XY 3 4
---
top level expression:
$Prog
```

η-reduction of supercombinator difinitions may eliminate redundant definitions and this leads to smaller supercombinator definition set in replacements in expression

order parameters such that parameters corresponding to deeper nested lambda abstraction are put towards the back of parameter list $\implies$ makes η-reduction possible

- compute level number of lambda abstraction using textual nesting level: current level is number of surrounding lambda abstractions plus one
- top level constants including supercombinators are defined to have a level of 0

impl. of compiling supercombinator body can use techniques from previous sections with graphs; jno free variables $\implies$ never need to be copied; substitution with multiple variables at once alternatively represent body of supercombinator with a sequence of code only (no environment necessary) $\Longrightarrow$ instantiation of supercombinator body corresponds to running the code sequence

35 FP: Recursive Supercombinators

extend body of supercombinator to have general graph

- `letrec` expressions for representing cyclic body and infinite data structures
- `let` expression for acyclic body

35.1 Transforming recursive program into supercombinator with graphical bodies

assign lexical level numbers to variables bound in `letrec`

- variable instantiated when immediate textually enclosing lambda abstraction is applied to an argument (when we construct instance of letrec, substituting for all free variables)
  $\Longrightarrow$ variable bound in a letrec given lexical level number of immediately enclosing lambda abstraction
- if no enclosing lambda abstraction, assign level number of 0
- no free variables $\Longrightarrow$ use lambda lifting to remove inner lambdas and transform into supercombinator

36 FP: Fully Lazy Lambda Lifting

- share dynamically created constant epressions
- each expression evaluated at most once after variables in it have been bound

36.1 Maximal Free Expressions

36.2 Formal Definition

maximal free expression of a lmabda abstraction L ≡

- all variables in expression are free
- the expression is not a proper subexpression of another free expression of L

issue: laziness lost if too much of body of lambda abstraction is instantiated

determine what parts should not be instantiated: parts of the body that contain no free occurences of formal parameter $\implies$ that subexpression is invariant accross all instantiations, therefore can use 1 instance and share it

duing β-reduction, do not instantiate maximal free expressions of lambda abstractions; point to a single shared instance in the body of lambda abstraction

36.3 Fully Lazy Lambda Lifting

modify lambda lifting algo. such that impl. of the resulting supercombinator program is automatically fully lazy: abstract maximal free expressions when doing lambda lifting of a lambda abstraction instead of abstracting only free variables of lambda abstraction as extra parameters

when maximal free expression has no free variables in it (CAF) $\implies$ give it a name and make it into a supercombinaor instead of abstracting it as an extra parameter; given name used instead of the expression

2 phases:

- float `letrec` and `let` definitions out as far as possible
- perform fully lazy lambda lifting

use a variable's set of free variables that the variable depends on
$\implies$ float the variable outwards until the next enclosing lambda abstraction binds 1 of the variables in the variable's free variable set
$\implies$ if no free variables at all, then variable/definition floated out to top level and turned into supercombinator

36.4 Implementing Fully Lazy Lambda Lifting

use lexical level number of expressions in addition to that of variables

lexical level of expression $\equiv$ $max_i(\text{lexical level of } i), \forall i \in$ free variables in expression

when lambda lifting a lambda abstraction at level n, all expressions within the body, that have levels less than n, are taken out as extra parameters

level number of any constant = 0 by definition

level number of a variable is the textual nesting depth of lambda which binds it

level number of an application `(f x)` is the maximum of the level numbers of f and x

expression's native lambda abstraction ≡ the 1st lambda abstraction which binds any variable in the expression; the enclosing lambda abstraction of the expression where level number is the same as that of the expression

implementation in a single tree walk over the expression:

1. traverse down the tree, record the level number of each lambda abstraction
2. on the way back up of the traversal, compute the level number of each expression by using the environment and level number of the expression's subexpressions
   - in application of expression, if the level numbers are the same then two expressions are merged, if not they are given new unique names (they will be maximal free expressions of distinct lambda abstractions)
   - merging mechanism $\implies$ forming maximal free expressions
3. on way back up the traversal, from smaller free expressions encountered, lambda is transformed into supercombinator; lambda abstraction is replaced by supercombinator applied to maximal free expressions (subexpressions with level number less than that of the lambda abstraction after merging)

lifting CAFs alternative:

- define a new supercombinator of 0 arguments
- use the defined supercombinator in place of the oringal expression
- constant expression with a single constant $\implies$ leave it since there is no additional benefit of liting it

reordering parameters of supercombinator in increasing level number

- maximize η-reduction opportunities
- useful for maximal free expression parameters as well (take out smaller number of larger free expressions)

36.5 float definitions given in let and letrec outward

in order to have full laziness

assumes that dependency analysis of let(recs) have already been performed earlier, or else some definitions may not be floated out as outward as possible

impl. may possibly combined this phase with dependency analysis step

an algo:

- immediately enclosing lambda abstraction has the same level number as that of the variables bound in let(rec)
- let(rec) is not present in function portion of an applicatoni
- let(rec) floated out as little as possible subject to above constraints $\implies$ enable optimization of unreachable code

more concrete algo:

1. compute level numbers of each definition body $\implies$ needed for computing level number of variables that are bound to it
2. for letrec, assume level number of variables defined in letrec is 0 (level # of recursive definition depends only in

its free variables and not on level # of recursive definition)

3. for letrec, compute maximum of level numbers of definitions' bodies $\Longrightarrow$ this is the level # for variables bound in letrec
   for let, level # is computed in previous step (level # of definition body)
   $\Longrightarrow$ use this computed # for variables bound in let(rec)
4. float out definitions up until the next enclosing lambda abstraction has the same level # as that of variables defined in let(rec) computed in previous step
5. let(rec) appears in function position of an application $\Longrightarrow$ continue to float it to next outer level

note: letrec rebinds a variable already in scope $\Longrightarrow$ cannot be floated outward unless renaming of variables is done (to avoid capturing occurences of outer variables)

situations where fully lazy lambda lifting does not gain improvements: selectively apply ordinary lambda lifting instead of fully lazy version

- builtin operator or supercombinator applied with too few arguments $\Longrightarrow$ no eval takes place anyways so extra work to abstract out expressions is not worth it
- arguments of function may be considered for abstraction
- constant expressions (candidates for new supercombinator definition) do not gain much from abstracting them out since they are irreducible anyways

36.6 general rules

lambda abstraction `\x.E` in a context that cannot be shared
$\Longrightarrow$ do not abstract free expressions from E because they will not be shared
$\Longrightarrow$ only abstract free variables

justifications:
free expressions in E are not shared outside of E by definition
free expressions in E are not shared inside E since they are abstracted from a single place in E

in the above way, lambda lifting algo. becomes context dependent

figuring out if a lambda abstraction may be shared is difficult in general, but we may give up at any time and assume partial applicatoin may be shared

37 FP: SK Combinators

```
S f g x = f x g(x)
K x y = x
I x = x
```

extensional equality

compile-time transformations with S, K, I: I-transformation:

λ`x.x` $\Longrightarrow$ `I`

K-transformation

λ`x.c` $\Longrightarrow$ `K c`

S-transformation:

$\lambda x.e_1\ e_2 \Longrightarrow S\ (\lambda x.e_1)(\lambda x.e_2)$

use of S, K, I to compile any lambda abstraction to expression with S, K, I terms and constants; other variables disappear

basic algo.:

```
while expr. contains a lambda abstraction, then:
  choose an innermost lambda abstraction in expr.
  body of the lambda abstraction is:
    application ⇒ apply S-transformation
    variable/constant ⇒ apply K or I transformation
```

recursion $\Longrightarrow$ use Y-combinator; Y is treated as a builtin function by the combinator compilation algorithm

37.1 SK Compilation Algo.

```
C[[ e1 e2 ]] = C[[ e1 ]] C[[ e2 ]]
C[[ λx.e ]] = A x [[ C[[ e ]] ]]
C[[ cv ]] = cv
A x [[ x ]] = I
A x [[ cv ]] = K cv
A x [[ f1 f2 ]] = S (A x [[ f1 ]]) (A x [[ f2 ]])

where:

cv ≡ a constant / builtin function
x ≡ a variable
fi ≡ expr. without any inner λs
ei ≡ expression
```

note: C applied to body of lambda before applying A $\Longrightarrow$ all inner lambdas are dealt with; A only has to deal with atoms and applications

37.2 K Optimization

fewer reductions, enable more laziness

```
S (K p) (K q) ⇒ K (p q)
```

when body of lambda abstraction does not use parameter of lambda:

```
A x [[ e ]] = K e ⟺ x not used in e
```

37.3 B Combinator

`S (K p) q` *Rightarrow* `B p q ≡ λx.p (q x)`

```
where:

B f g x = f (g x) //runtime reduction
```

useful when lambda abstraction only uses parameter in the right branch

37.4 C Combinator

```
S p (K q) ⇒ C p q ≡ λx.(p x) q
```

```
where:
```

```
C f g x = f x g
```

useful when lambda abstraction only uses parameter in the left branch

special case for `S (K p) I` ⇒ `p`

37.5 modification to SK compilation algo. using the above

```
A x [[ f ]] ≡ abstracts x from f
---
A x [[ x ]] = I
A x [[ cv ]] = K cv
A x [[ f1 f2 ]] = S (A x [[ f1 ]]) (A x [[ f2 ]])
  ⇒ Opt[[ S (A x [[ f1 ]]) (A x [[ f2 ]])
```

```
where:
```

```
Opt[[ S (K p) (K q) ]] = K (p q)
Opt[[ S (K p) I ]] = p
Opt[[ S (K p) q ]] = B p q
Opt[[ S p (K q) ]] = C p q
Opt[[ S p q ]] = S p q
```

37.6 S' Combinator Optimization

for compiling

```
λxn .. λx1.p q where p and q both use xn .. x1
```

let:

```
S' c f g x = S (B c f) g x
  → (B c f) x (g x)
  → B c f x (g x)
  → c (f x) (g x)
```

then:

```
S' c f g x = c (f x) (g x)
```

$\Longrightarrow$

```
p q
= S ¹p ¹q
= S' S ²p ²q
= S' (S' S) ³p ³q
...
= S' (S' (..(S' S)..) ) ⁿp ⁿq //n-1 times of S'
```

```
where:
```

```
¹p = A x1 [[ p ]]
²p = A x2 [[ ¹p ]]
...
```

`O(n)` expansion complexity

analogous B' and C' combinators for abstracting many variables used only in p or q but not both

```
B' c f g x → c f (g x)
C' c f g x → c (f x) g
```

```
compilation rules:
```

```
B (c f) g ⇒ B' c f g
C (B c f) g ⇒ C' c f g
```

eg:

```
C (B c f) g x = (B c f x) g
              = c (f x) g
```

further optimization for B':

```
B c (B f g) ⇒ B* c f g
```

```
where:
```

```
B* c f g x → c(f(g x))
```

37.7 Final SK Compilation Algo.

```
Opt[[ e ]] optimizes e
---
Opt[[ S (K p) (K q) ]] = K (p q)
Opt[[ S (K p) I ]] = p
Opt[[ S (K p) (B q r) ]] = B* p q r = λx.p(q(r x))
Opt[[ S (K p) q ]] = B p q
Opt[[S (B p q) (K r) ]] = C' p q r = λx.p (q x) r
Opt[[ S p (K q) ]] = C p q
Opt[[ S (B p q) r ]] = S' p q r = λx.p (q x) (r x)
```

```
where:
```

```
I x → x
K c x → c
S f g x → f x (g x)
B f g x → f (g x)
C f g x → (f x) g = f x g
S' c f g x → c (f x) (g x)
B* c f g x → c(f(g x))
C' c f g x → c(f x) g
```

38 FP: G Machine

compiling supercombinators to G-code

compilation scheme F:

```
F[[ $G x_1 .. x_n = E ]] = G-code for $G
```

use of graph and stack for evaluation

auxilliary R compilation scheme to compile body, E, of supercombinator:

```
R[[ E ]] p d

where:

p: maps identifier to offset of the argument from
   the base of the current context
d: depth of current context - 1
R: create instance of body of supercombinator using
     parameters on the stack
   update root of redex with copy of root of result
   remove parameters from the stack
   initiate next reduction
```

C compilation scheme:

constructs code/instructions for generating an instance of a target expression

args: expression for compilation; p and d: where arguments of supercombinator are found on the stack

result:

- code sequence that can construct instance of the expression
- substitution of found parameters with supercombinator arguments that are referenced
- a pointer to the instance on top of the stack

define `C[[ E ]] p d` for each case of expression E

```
C[[ i ]] p d = PushInt i
C[[ f ]] p d = PushBlobal f
C[[ x ]] p d = Push (d - p x) //copy to top of stack
C[[ E_1 E_2 ]] p d =
  C[[ E_2 ]] p d;
  C[[ E_1 ]] p (d+1);
  MakeAp
C[[ let x = E_x in E_b ]] p d =
  C[[ E_x]] p d;//create Ex instance on top of stack
  C[[ E_b]] p[x=d+1] (d+1);//create Eb on stack using
    instantiated pointer to Ex already on stack
  Slide 1;//discard associated item with Ex on stack
C[[ letrec D = E_b ]] p d =
  CLetrec[[ D ]] p' d';
  C[[ E_b ]] p' d';
  Slide (d'-d);//discard item associated with D on stack
    since it is referenced by Eb, while keeping item
    associated with Eb instance on top of stack

where:

i is a constant value or pointer to the value
f is a function
x is a variable; value is in the stack at offset
  (d - p x) from the top
MakeAp is an application cell that takes and pops
  2 items on stack and forms application node in
  the heap, puts a pointer to the node on top of
  the stack
p[x=y] a = y,   x = a
         = p a, x ≠ a
(p', d') = Xr[[ D ]] p d
CLetrec[[ x_1 = E_1
          ..
          x_n = E_n ]] p d =
  Alloc n;//create n holes in heap and pushes n
    pointers onto stack
  C[[ E_1 ]] p d; Update n;//construction of
    definition body instance; updates associated hole
    in heap with root of created instance of
    definition body
  ..
  C[[ E_n ]] p d; Update 1
Xr[[ x_1 = E_1
     ..
     x_n = E_n]] p d =
  (p[x_1 = d + 1
     ..
     x_n = d + n], d + n) //augmentation of p and d
```

note: if definition body is a single varialb ename fou din the same letrec:

```
letrec x = y
       ..
       y = Cons 1 y
```

then update will perform update 1 hole with another; avoid this by removing definition of x and replaceing with y at an earlier stage of compiler (elimination of redundant lets where rhs of let is a single variable by replacing occurences of lhs variable with rhs variable in body of let)

38.1 GC of CAFSs

compiled version of code:

- need explicit list of references of directly/indirectly used CAFs since it is not apparent in compiled instructions
- use of expicit list of CAFs (info given to GC implementation) that each supercombinator uses for marking (signal object is reachable from a root and cannot be eligeable for garbage collection) in GC sweeps

supercombinator with $\geq$ 1 argument(2) can't grow in size $\Longrightarrow$ no need for GC

unreduced CAF marking $\Longrightarrow$ mark its associated list of CAFs

reduced CAF marking $\Longrightarrow$ like any other heap data structure for marking

supercombinator with $\geq$ 1 argument(s) marking $\Longrightarrow$ mark all CAFs in its associated list

38.2 Options for Compiling CAFs

not compile:

- leave in graph form
- never copied
- shared
- program may be a mixture of compiled code and graph

compile:

- supercombinator with 0 arguments
- compile to G-code: code that will instantiate their graph when executed
- instantiation will need to overwrite some code for sharing so that further use will not require repeated copied of it; possible impl:
  - code alloc. a graph node representing a function (a pointer to compiled code)

  - execution of compiled code updates the graph node (1 per supercombinator; not in read only memory since it needs to be updated when code is executed) associated with it with the result; the result is sharable with other users through the shared graph node

38.3 Builtin Function

translation to G-code sequence:

- eval to WHNF whenever possible to eliminate duplicate work
- UNWIND vs. RETURN instruction
- PUSH
- EVAL
- UPDATE
- POP
- SELSUM$_{r,i}$
- SELPRODUCT$_{r,i}$
- JUMP
- JFALSE
- CASEJUMP$_{StructTag,n}$

39 FP: Abstract Machine Model Describing G-Code

machine state: `<S,G,C,D>`
S: stack
G: graph
C: code sequence (G-code)
D: dump / stack of `(stack, code sequence)`

operation of G-machine $\Longleftrightarrow$ state transition, eg:

```
<S, G, PushInt i: C, D ⇒
  <n:S, G[n=Int i], C, D>

<n: S, G[n = Ap n₁ n₂], Eval: C, D> ⇒
  <n:[], G[n = Ap n₁ n₂], Unwind:[], (S,C): D>

//will eventually update the CAF function node with
//  result of reduction
<n: S, G[n = Fun 0 C'], Eval: C, D> ⇒
  <n:[], G[n = Fun 0 C'], C':[], (S,c): D>

<n: S, G[n = Int i], Eval: C, D> ⇒
  <n: S, G[n = Int i], C, D>

where:

n: a unique new name
```

non-existent transition $\Longrightarrow$ runtime error

cases for transition of Unwind:

item on top of stack:

- Cons or Int node $\Longrightarrow$ WHNF; only item in current stack; restore saved stack and code from dump; put result on top of the restored stack
- pointer to application node $\Longrightarrow$ push head of application on stack; repeat Unwind instruction
- a function with enough arguments on stack $\Longrightarrow$ rearrange stack; execute code for the function
- a function without enough arguments $\Longrightarrow$ expression under eval is in WHNF; restore saved stack and code from dump; put result of eval on top of restored stack

39.1 Printing Mechanism

need an evaluator invocation mechanism that reduce expressions to WHNF and also print them

add an additional compone tin G-machine state, O

add a Print instruction: print element on top of the stack

eg:

```
<O, n:S, G[n = Int i], Print:C, D> ⇒
  <O:i, S, G[n = Int i], C, D>

<O, n:S, G[n = Cons n₁ n₂], Print:C, D> ⇒
  <O, n₁:n₂ :S, G, Eval:Print:Eval:Print:C, D>

<O, n:S, G[expr], i:C, D> where i ≠ Print ⇒
  O is unchanged
```

`Begin` instruction initializes the G-machine:

```
<O, S, G, Begin:C, D> ⇒ <O, [], [], C, []>
//note: empty stack, graph, dump
//and runs rest of the code C
```

39.2 Impl. of G-code

compute memory space required by each supercombinator and insert instructions to check for sufficient memory (heap) at beginning of function (do at compile with a simulated model of stack)

simulated stack expected to be empty at the end of the execution of a supercombinator

eval may cause arbitrary amount of computation to occur

issue: possible GC interruption disturb simulated stack and heap

possible solution for when flow of control can be broken

- segments of code between Evals has its own code to check for heap exhaustion $\implies$ simulated stack and heap pointers/offsets calculated relative to EP(stack pointer) and HP(heap pointer) at beginning of each code segment (not necessarily start of each supercombinator)
- before Eval is called, simulated stack is flushed oto the real stack
- also apply this separate code segment technique to code with JUmp instructions:
  - different routes due to different branches may have different amounts of heap allocated and simulated stacks are different
  - simulated stack flushed to real stack before Jump

dealing with graphs via an interface to be used from G-code execution mechanism; types of graph operation:

- node specific operation, when node being evaluated is known to be of that type
- generic operation: eval on unknown node; usually need case analysis on node type $\implies$ more expensive

tag case analysis:

- tag in a cell points to a table of code entry points (one entry point per generic operation)
- different node type associates with different entry table
- possible use of cell/boxed values to achieve a unifrom case analysis for different typed nodes
- use of indireciton node instead of copying root of result of reduction into root of the redex
- Unwind G-code instruction:
  - WHNF: return to caller
  - function with enough args: rearrange stack to prep for actual invocation of function's code
- Unwind on application call: push head of cell on stack and Unwind again
- Eval code:
  - code generated for Eval G-code instr., then
  - call eval code in an entry of an entry table associte with the node tpye via a subroutine jump instr.
  - cell type:
    * WHNF (eg: primitive/function cell/ Cons cell): return from subroutine
    * App cell/node: push current stack on dump (system stack) and call Unwind on application

per supercombinator:

- Globstart: Unwind code, GC code, entry table, function node via instr. to alloc. it at start of function code, overflow checking code preceding target code for function body
- target functinon code

Begin: initialize stack and heap for entire system and other system specific tasks

End: terminate execution of whole program

40 FP: Optimizations to G Machine

avoid heap allocation / avoid huilding graphs

try use cheaper stack to cut allocation

laziness preservation for R compilation scheme (code generation for: applying a supercombinator to its arguments)

case analysis for each kind of expression for R compilation scheme:

```
R[[ i ]] p d =
  PushInt i;
  Update (d+1);
  Pop d;
  Return //not Unwind since Int can't be applied

R[[ f ]] p d =
  PushGlobal f;
  Eval; //needed for the case of reduction of CAF
  Update (d+1);
  Pop d;
  Unwind

//for the case of body being a single variable
//  load value onto stack; eval it;
//  update result of reduction into root of redex
R[[ x ]] p d =
  Push (d - p x);
  Eval;
  Update (d+1);
  Pop d;
  Unwind

R[[ E1 E2 ]] p d =
  C[[ E1 E2 ]] p d;
  Update (d+1);
  Pop d;
  Unwind

R[[ let x = Ex in E ]] p d =
  C[[ Ex ]] p d;
  R[[ E ]] p[x = d+1] (d+1)

R[[ letrec D in E ]] p d =
  CLetrec[[ D ]] p' d';
  R[[ E ]] p' d'
  where (p'd') = Xr[[ D ]] p d
```

direction execution of builtin functions instead of building a graph and immediately unwinding it

optimize code for special cases for `C[[ E ]] p d; Eval` sequence:

`E[[ E ]] p d` generate G-code which eval.(equivalent C-eval sequence) E to WHNF and leave result on top of the stack (replace occurences of C-eval sequence in R scheme with E compilation scheme)

RS sceheme, ES scheme: enable E scheme optimization

40.1 η-reduction and lambda lifting

- may incur cost during code generation
- perform only if it eliminates an entire function definition
- eg: `(f E1 E2 ..)` if `f` is unknown such as ones passed into function as argument, then optimzed code generation is less likely

40.2 Fatbar and Fail

optimize by never producitng a Failure value and replace it with instructions

40.3 Evaluating arguments

if function is strict in an argument, then it's safe to use E scheme and avoid building a graph before eval it

partial application / supercombinator that return a function: eval provided arguments straight away instead of building graphs

40.4 Global Structness Info

using strictness analysis:

- annotate application of functions that are strict wrt. parameters
- annotate definitions and application nodes

eg: an infix op `!` denoting strict applciation and strict let expression:

`$F ! x = ..` where F is strict in x

acheives similar effect as call by value

avoid Eval to reduce cost (in G-machine instr. set):

- avoid re-evaluation in function body:
  track context info about whether item on a stack has been evaluated or not (eg: stack position -¿ Bool)
  also modify compilation schemes to return modified env. in addition to the generated code as before
- supercombinator begins with code to Eval for each argument that the function is strict on
- no-eval version of the function called after strict arguments are eval'd in the supercombinator

avoid repeated unwinding:

- add info on arugments' evaluation status (WHNF or not) eval function if it is passed as a strict argument instead of deferring when function is applied
- add a tag indicating WHNF for application node $\Longrightarrow$ Eval entry for AP-WHNF's entry table is an immediate return instead of Unwind and finding function at tip of spine graph has too few arguments (eg: in WHNF) before returning;
  modify C scheme for compiling application of a function to too few arguments
- all vertebrae below the root of redex at completion of Unwind instruction are then in WHNF because each represents application of a function to too few number of arguments;
  tags on these vertebrae can be changed to AP-WHNF as they are rearranged / removed from the stack as Unwind completes

40.5 Eager Evaluation

- avoid creating cells when evaluating expressions
- specialize C scheme to directly use code instead of creating cells
- if arguments is known to be eval'd already, then avoid extr cell creation and use direct code for eager eval $\Longrightarrow$ performs operations that may not be used (graph created with C scheme may be discarded)
  $\Longrightarrow$ this is still safe since the arguments are for sure already evaluated so it will terminate
  $\Longrightarrow$ still usually faster than allocating cells on the heap
- propagate this info. upward via environment which records items on stack being already evaluated or not

40.6 Boxing/Unboxing on Primitives

temporary/transient cells and bitmasking instr. may be used in boxing/unboxing in between other operations that are costly

instructions to:

- support operations on naked values
- fox and unbox values on top of stack (cells in heap)
- modify associated compilation scheme to support this

need modification for GC system to distinguish box vs. unboxed items on stack

- eg: can use an additional stack for only unboxed values and instructions are aware which stacks to use based on unboxed/voxed form of tiem

40.7 Peephole Optimization

sits in between G-code compiler and code generator

replace short consecutive sequence of G-code instr. with more optimal sequence (instr. compression)

eg:

`MKAP n`, `SLIDE n`

if f is a builtin/supercombinator with ≥ 1 arguments: eliminate Eval instruction:

```
// if f is not a CAF
PushGlobal f; Eval ⇒ PushGlobal F
```

build root of result directly on top of root of redex (to avoid allocating cell for result and copying result to root of redex)

use an instruction to use items on stack to build app node on top of root of redex $\implies$ modify some compilation schemes for this:

- `RS[[ f ]] p d n`
- `RS[[ x ]]`
- `CLetrec Schemes`

40.8 Unpacking Structred Objects

eg: present in lets compiled from case expressions

use a unpack instruction, eg: `UnpackSum k`: unpacks top element on stack into k components (place them on top of the stack)

eg:

```
let v_1 = SelSUm_{k,1} v
    ..
    v_k = SelSUm_{k,k} v
in E
```

$\Updownarrow$

```
Push (d - p v); SelSum_{k,1};
..
Push (d + k - 1 - p v); SelSum_{k,k};
```

$\Downarrow$

```
Push (d - p v);
UnpackSum k;
```

Pattern Matching in a function is not compiled to:

1. code sequence to eval arg.
2. multiway jump based on structure tag of the argument
3. unpack instruction (take strucure apart and puts components on the stack)
4. code sequence to eval rhs. of the function (eg: free variables and components of structure in the stack)

41 FP: Generalized Tail Call Optimization

squeeze out items on the stack associated with previous function before calling the tail function $\implies$ enable eliding extra instruction and space on stack needed for the function call

eg:

```
C[[ E3 ]] p d;
C[[ E2 ]] p (d+1);
C[[ E1 ]] p (d+2);
PushGLobal W;
Squeeze 4 2; //rid of F's 2nd and 1st args on stack
//at runtime, do case analysis on
//function on top of the stack
Dispatch 3
```

Dispatch instruction is behaviourly equivalent as doing:

1. construct spine in heap from ribs on the stack
2. update root of the redex (bottom of current context) using the constructed spine
3. Unwind W

case analysis of tail call:

- W is an Application node: in general, number of args W takes is unknown
  optimize construction of spine: do it on the stack instead of heap
  perform 1st part of Unwind as we construct the spine

  transition rule for Dispatch:

  ```
  <f:n1:n2:..:nk:r:S,
    G[f = Ap m1 m2], Dispatch k:[], D>
  ⇒
  <f:v1:v2:..:v(k-1):r:S,
    G[v1=Ap f n1;
      vi = Ap v(i-1) ni, i ∈ (1,k);
      r = Ap v(k-1) nk],
    Unwind:[], D>
  ```

  then, proceed to do as usual:

  - construct spine of body of `$F`
  - update root of `$F_redex`
  - Unwind

- W is a supercombinator and a CAP (0 args): Dispatch's behaviour same as if W is an application node (previous case)

- W has exactly the number of args as the function requires: enter the code for W:

  ```
  <f: S, G[f = Fun k C], Dispatch k:[], D> ⇒
    <S, G, C, D>
  ```

  entry into code for the function after arity check

- W: number of provided args $\geq$ the arity of function parameters

  partial reduction of body of `$F`: part of body of `$F` becomes next redex

  construct a new context where W will execute in:

  - W's args placed on top of the stack
  - root of W-redex (placeholder hole to be later filled with the result of reduction of W) pointed to by an item in the stack below W's args
  - construct top partial spine of the body of `$F`

- W: number of available args $\leq$ number of parameters function requires

  body of `$F` is then in WHNF

  need to update root of `$F-redex`:

  - construct spine in heap as in the case of an application node
  - Dispatch should test depth of the stack to see if there are enough arguments after the spine is constructed and root of `$F-redex` is updated:

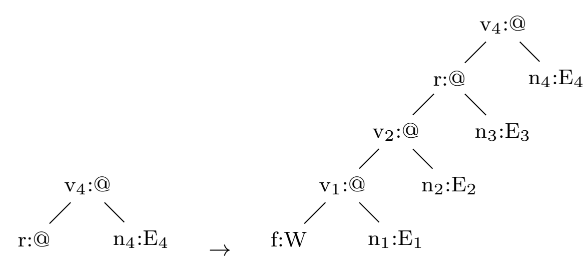
    not enough args for W to reduce $\implies$ eval is complete; Dispatch initiates a Return

    enough args for W to reduce $\implies$ Dispatch rearranges the stack to prepare the call to W and finally enter W

  ```
  //eg, assuming W takes 4 arguments
  stack:
  ---
  root of W-redex -> v4
  root of F-redex -> r
  n3:E3
  n2:E3
  n1:E1
  f: W
  ..

  //after arrangement by Dispatch
  root of W-redex

  stack:
  ---
  root of W-redex -> v4
  n4:E4 -> n4
  n3:E3 -> n3
  n2:E2 -> n2
  n1:E1 -> n1
  ```



41.1 transition rule for Dispatch when not enough arguments for function to reduce

```
<f:n1:n2:..:nk:r:v(k+1):..:vd:[],
  G[f = Fun a C], Dispatch k:[], (S, C'):D>
{k < d < a }
⇒
<vd:S,
  G[v1 = AP-WHNF f n1
    vi = AP-WHNF v(i-1) ni (1<i<k)
    r = AP-WHNF v(k-1) nk],
  C',
  D>

where:

k: argument to dispatch
d: # of args available
a: arity of function on top of the stack
```

note:

- eval complete by making a return to the caller

- vertebrae v_is are in WHNF, thus construct them as type AP-WHNF nodes

41.2 transition rule for the case of enough args. for the function to reduce

rearrangement of stack happens

followed by jump to code of the function

```
<f:n₁:..:nₖ:r:vₖ₊₁:..:vₐ:S,
  G[f = Fun a C
    vₖ₊₁ = Ap r nₖ₊₁
    vᵢ = Ap vᵢ₋₁ nᵢ, (k+1 < i ≤ a)],
  Dispatch k: [],
  D>
{k < a}
⇒
<n₁:..:nₖ₊₁:..:nₐ:vₐ:S,
  G[v₁ = AP-WHNF f n₁
    vᵢ = AP-WHNF vᵢ₋₁ nᵢ, (1<i<k)
    r = AP-WHNF vₖ₋₁ nₖ],
  C,
  D>
```

41.3 Compilation using Dispatch

modifications to RS scheme:

```
RS[[ x ]] p d n =
  Push (d - p x);
  Squeeze (n+1) (d-n);
  Dispatch n

RS[[ f ]] p d n =
  PushGlobal f;
  Squeeze (n+1) (d-n);
  Dispatch n
```

41.3.1 function is known at compile time

perform case analysis at compile time to generate code instead of analysis at runtime

eg:

```
PushGlobal $H;
Squeeze p q;
Dispatch k

where $H takes exactly k args

⇒

tail call case and generate code to jump to $H's code
(after arity check and stack rearrangement)
```

eg:

```
PushGlobal $Cons;
Squeeze 3 q;
Dispatch 2;
⇒
Cons;
Update (q+1);
Pop q;
Return;
```

41.3.2 other case

```
Push n;
Squeeze p q;
Dispatch k;
```

perform case analysis at runtime using G-code code generation using a Dispatch entry to each tag's entry table

eg:

```
moval 3, r2 //k parsed to Dispatch code via r2
movl (%EP)+, r0 //pop function into r0
movl r0, r1 //move tag into r1
//do case analysis jump using offset to
//Dispatch entry of entry table
jmp * O_Dispatch(r1)
```

41.4 Optimizing E scheme

apply similar optimization in RS scheme to ES scheme

allocate a hole to receive the result, then build ribs using ES scheme and push them on the stack

```
E[[ E₁ E₂ ]] p d =
  Alloc 1;
  ES[[ E₁ E₂ ]] p d 0;

ES[[ x ]] p d n =
  Push (d - p x);
  Call n;

ES[[ f ]] p d n -
  PushGlobal f;
  Call n;

<f:n₁:..:nₖ:r:S, G, Call k: C, D>
⇒
<f:n₁:..:nₖ:r:[], G, Dispatch k: [], (S,C): D>
```

Call is like Dispatch except stack and code pointer are saved to the dump

41.5 Comparison to Other Environment Based Implementations

typical env. based impl:

- eval body of function in an environment (container that captures variable values in scope) where formal parameters are bound to their values
- if the result of eval is a function-like object, then we return a closure (function's code pointer + environment in which the function will be executed in)
- environment implementation: linked list or tuple consisting of only used values of variables that occur free in the body of the function

similarity to eng.cased impl:

- equivalent in G-machine: a graph with supercombinator aplpied to too few args.
- extra args. produced from lambda lifting algo. to a function
- execution in G-machine is based on stack (args found on stack) whereas env.-based impl. may access free variables (not present in supercombinator) in its environment
- args passed to a function placed on stack before calling

differences to env.-based impl.:

- enable laziness in G-machine by writing the spine into the heap at ttimes vs. always keeping it in stack in some env.-based impl.
- G-machine as an effective impl. of graph reduction: support parallel execution naturally via graph reduction model

42 FP: Parallel Execution

parallel execution of functional program via concurrent graph reductions

good parallel performance: need algorithmic parallelism by construction

parallelism in functional language can be dynamic; no static division and manual sync. required by programmer; smaller unit of parallelism; dynamically adaptable

synchronization between > 1 reductions done via the shared graph: update/reduction make known by overwriting root of redex with result of reduction (no special synchronization needed)

no extra language constructs needed for parallel programs: result independent of scheduling of reductions (may vary in efficiency)

graph reduction:

- reduction is a local transformation of the graph (distributed activity)
- graph as medium of communication
- state of computation $\iff$ state of the graph

42.1 model for reduction

- sequential impl:

  pointer to root of the graph to be evaluated by evaluator

  keep reducing until graph is in WHNF, then terminate

- parallel version:

  > 1 evaluator tasks work on the graph

  evaluator may require value of subgraph $\implies$ generate new task to eval subgraph (sparking root node of the subgraph) autonomously; subgraph eventually expected to be in WHNF

  parent blocked if there exists child subgraph that hasn't finished computation

  sibling blocked if other sibling is accessing a subgraph that is also needed by the current sibling

  atomic write on root of redex on graph, thus no direct communication between tasks

  each virtual task executed by an agent (physical processor)

  task pool serviced by idling agents

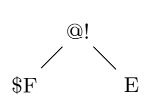
generation of new tasks:

- if subgraph is known to be eventually needed (conservative parallelism)
- speculative that subgraph is possible eventually needed

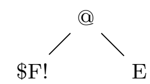
builtin function applied to argument, `f E` $\implies$ safe to parallel evaluate `E` if `f` will need value of the argument (f is strict in E)

annotate the graph with strictness info. from strict analysis

- annotate application node that arg. is needed



- annotate supercombinator that arg. is strict



need both of the above to enable maximal parallelism

42.2 speculative parallelism

may waste resource

prioritize vital tasks over speculative ones

speculative tasks may cahgen to be vital when its results are discovered to be needed (some of its subgraph tasks may also change priority to vital)

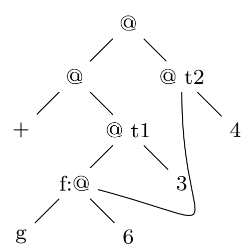
speculative tasks may be stopped and discarded when result is not needed: all subtasks/sparks need also be killed

42.3 blocking mechanism

mark node during unwinding of spine (when finding a function at the tip of the spine) to indicate nodes are currently being worked on

other tasks blocked when encountering marked nodes

rewinding of spine (popping of vertebrae from its stack) also removes the mark from vertebrae (after reduction overwrites its result) $\implies$ task blocked by affected nodes can proceed (and will see updated redex performed by earlier task)



if task t1 unwinds and reaches f first, then task t2 may block when unwinding to f

optimization:

- once a node (subgraph) is in WHNF, it will never need to be updated again $\implies$ give read-only access to > 1 tasks concurrently (discriminate with regular application node by WHNF application node type)
- unwinding to a WHNF application node $\implies$ node does not need to get marked (no blocking to other tasks)
- example nodes in WHNF: supercombinator, number, cons cell
- otherwise, nodes not in WHNF $\implies$ subgraph may contain redexes and be altered; only 1 task accesses it at any time

runtime WHNF optimization possible even if compile-time WHNF optimization is not possible

eg: ($G E1 E2 E3)

after 1st reduction at top node, the lower 2 application nodes are in WHNF $\implies$ mark these 2 nodes as WHNF when finishing reduction (as these nodes are popped off the stack)

representation of task:

- store info. to restore execution of it during suspended state typically in a task control block:

  - task stack pointer
  - task program counter
  - state of task's registers
- vs. graph representation / parellel reduction model
  - a task corresponds to a subgraph that it is evaluating (state of partially completed task encoded in the graph):
  - using a pointer to the root of the subgraph is theoretically sufficient to represent the task at any stage of its lifetime
  - suspension of a task: stop reduction; save pointer to task's root node to the task pool
  - resuming a task: take its pointer to root and continue reduction

42.4 optimization to suspending and resuming task

uses pointer reversal to avoid unwinding spine when resuming a blocked task (avoids repeatedly unwinding spine every time a task is blocked):

- reverse only pointers in vertebrae (marked nodes) $\Longrightarrow$ no other tasks will access a pointer reversed node
- state of a task represented by 2 pointers (fwd and backward pointers) $\Longrightarrow$ suspended task resumed: forward and backward pointers point to area of graph of interest
- re-reversing pointers when rewinding the spine and marking vertebrae nodes as WHNF at the same time

stack based G-machine for parallel machine

- as a sequence of optimizations to graph reduction
- part of state held in stack
- at any point, info. in stack can be used to flush and update teh graph that repr. its current state
- keep some state of the graph in stack, over some sequence of reduction steps, as optimization

when a task is blocked:

- return it to the task pool for future exeuction, or
- asynchronously reawaken (put into task pool) the task once the blocked node has its mark removed (for efficiency)

  possible impl.:

  - add a list of tasks to be reawakened to each application node
  - when application node is unmarked, move reawakening taks to task pool

42.5 locality and distributed memory scheme

- heuristics: execute tasks and allocations in some memory unit
- parallelism vs. concurrency consideration
- program annotation
- heuristics to migrate tasks to memory units that are referenced by the tasks
- granularity of task: worthwhile to migrate large computations in a task

References

[1] Andrew W Appel. *Modern Compiler Implementation in C.* Cambridge University Press, 2004.

[2] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[3] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, Inc., 1996.