Contents

1 Module `Arch` : Supported BAP architectures

```
type arch =
```

```
  | X86_32
  | X86_64
val arch_to_string : arch -> string
val arch_of_string : string -> arch
val type_of_arch : arch -> Type.typ
val bits_of_arch : arch -> int
val bytes_of_arch : arch -> int
val mode_of_arch : arch -> Disasm_i386.mode
val mem_of_arch : arch -> Var.t
val sp_of_arch : arch -> Var.t
```

2 Module `Arithmetic` : Basic integer arithmetic on N-bit integers

These are common operations which are needed for constant folding or evaluation.

Author(s): Ivan Jager

```
module D :
   Debug.Make(  sig
     val name : string
     val default : [> `NoDebug ]
  end )
exception ArithmeticEx of string
val power_of_two : int -> Z.t
val bitmask : int -> Z.t
val bits_of_width : Type.typ -> int
val to_big_int : Z.t * Type.typ -> Z.t
val to_sbig_int : Z.t * Type.typ -> Z.t
val t_div : Z.t -> Z.t -> Z.t
val t_mod : Z.t -> Z.t -> Z.t
val toshift : Type.typ -> Z.t * Type.typ -> int
val to_val : Type.typ -> Z.t -> Z.t * Type.typ
val exp_bool : bool -> Z.t * Type.typ
val binop :
  Type.binop_type -> Z.t * Type.typ -> Z.t * Type.typ -> Z.t * Type.typ
     binop operand lhs lhst rhs rhst
val unop : Type.unop_type -> Z.t * Type.typ -> Z.t * Type.typ
val cast : Type.cast_type -> Z.t * Type.typ -> Type.typ -> Z.t * Type.typ
val extract : Z.t -> Z.t -> Z.t * Type.typ -> Z.t * Type.typ
```

```
val concat : Z.t * Type.typ -> Z.t * Type.typ -> Z.t * Type.typ
val is_zero : Z.t * Type.typ -> bool
val bytes_to_int64 : [< `Big | `Little ] -> int64 list -> int64
```

3 Module Asmir : High level interface to libasmir.

The functions in this file should be used instead of calling Libasmir functions directly. These functions should be easier to use, and, unlike the Libasmir ones, will handle garbage collection.

Author(s): Ivan Jager, Ivan Jager

```
exception Memory_error
exception Disassembly_error
type asmprogram
val arch_i386 : Arch.arch
val arch_x8664 : Arch.arch
type varctx
val gamma_create : Var.t -> Var.t list -> varctx
```

gamma_create mem decls creates a new varctx for use during translation. mem is the var that should be used for memory references, and decls should be a list of variables already in scope.

```
val gamma_lookup : varctx -> string -> Var.t
val decls_for_arch : Arch.arch -> Ast.var list
val gamma_for_arch : Arch.arch -> varctx
val get_asmprogram_arch : asmprogram -> Arch.arch
val x86_mem : Var.t
val x86_regs : Var.t list
val x64_mem : Var.t
val x64_regs : Var.t list
val x86_all_regs : Var.t list
val x64_all_regs : Var.t list
val multiarch_all_regs : Var.t list
val all_regs : Arch.arch -> Var.t list
val arch_to_bfd : Arch.arch -> Libbfd.bfd_architecture * Libbfd.machine_t
```

Convert a BAP architecture to a BFD architecture and machine

```
val translate_trace_arch :
  Trace.Arch.bfd_architecture -> Trace.Arch.machine_t -> Arch.arch
```

Translate libtrace architecture to BAP architecture

```
val open_program : ?base:Type.addr -> ?target:string -> string -> asmprogram
```

Open a binary file for translation

```
val asmprogram_to_bap : ?init_ro:bool -> asmprogram -> Ast.program
```

Translate an entire Libasmir.asm_program_t into a BAP program

```
val asm_addr_to_bap : asmprogram -> Type.addr -> Ast.program * Type.addr
```

Translate only one address of a Libasmir.asm_program_t to BAP

```
val asmprogram_to_bap_range :
  ?init_ro:bool -> asmprogram -> Type.addr -> Type.addr -> Ast.program
val bap_fully_modeled : Ast.program -> bool
```

`bap_fully_modeled p` returns `true` when `p` represents a fully lifted BAP program. `bap_lift_success p` will return `false` if an unmodeled instruction or system call is encountered.

Load entire trace into memory at once. If pin is true, loads a PinTrace. If pin is false, loads an old, TEMU-based trace format.

```
val serialized_bap_from_trace_file : string -> Ast.program * Arch.arch
```

Load entire trace into memory from the new SerializedTrace format.

Open a PinTrace/TEMU-based trace in streaming format depending on the value of `pin`.

```
val serialized_bap_stream_from_trace_file :
  int64 -> string -> Ast.stmt list Stream.t * Arch.arch
```

Open a SerializedTrace trace in streaming format.

```
val get_symbols : ?all:bool -> asmprogram -> Libbfd.asymbol array
val get_dynamic_symbols : asmprogram -> Libbfd.asymbol array
val find_symbol : asmprogram -> string -> Libbfd.asymbol
val get_flavour : asmprogram -> Libbfd.bfd_flavour
val get_all_asections : asmprogram -> Libbfd.section_ptr array
val get_section_startaddr : asmprogram -> string -> Type.addr
val get_section_endaddr : asmprogram -> string -> Type.addr
val get_base_address : asmprogram -> Type.addr
```

Lowest address of program in memory

Start address of program

```
val get_start_addr : asmprogram -> Type.addr
val get_asm_instr_string : asmprogram -> Type.addr -> string
val get_asm_instr_string_range :
  asmprogram -> Type.addr -> Type.addr -> string
val is_load : Libbfd.section_ptr -> bool
```

Is section s loaded?

`val is_code : Libbfd.section_ptr -> bool`

Is section s code?

```
val byte_insn_to_bap :
  Arch.arch -> Type.addr -> char array -> Ast.program * Type.addr
val byte_sequence_to_bap :
  char array -> Arch.arch -> Type.addr -> Ast.program list
```

`val get_exec_mem_contents : asmprogram -> Type.addr -> char`

`get_exec_mem_contents p` returns a function `f` such that `f addr` returns the executable byte in memory at `addr` if one exists. If no such byte exists,

`val get_exec_mem_contents_list : asmprogram -> (Type.addr * char) list`

`get_exec_mem_contents_list p` returns a list of `(addr, byte)` tuples indicating the executable memory at `addr` is `byte`.

`val get_readable_mem_contents : asmprogram -> Type.addr -> char`

`get_readable_mem_contents` is like `Asmir.get_exec_mem_contents`[3] but for any readable memory.

`val get_readable_mem_contents_list : asmprogram -> (Type.addr * char) list`

`get_readable_mem_contents_list p` is like `Asmir.get_exec_mem_contents_list`[3] but for any readable memory.

4 Module `Asmir_consts` : Constants for asmir.ml.

These could go in asmir.ml, but then we'd need an entry in asmir.mli for each one, and well, I'm just lazy.

```
val (<<) : int -> int -> int
val bsf_global : int
val bsf_debugging : int
val bsf_function : int
val bsec_no_flags : int
```

no flags

`val bsec_alloc : int`

allocate space when loading

`val bsec_load : int`

load the section during loading

`val bsec_reloc : int`

section has reloc info

`val bsec_readonly : int`

read only

`val bsec_code : int`

code

`val bsec_data : int`

data

`val bsec_rom : int`

ROM

5 Module `Asmir_disasm` : Framework for incremental disassembly methods

```
type succs =
  | Addrs of Type.label list
  | Error
  | Exit
  | Indirect
module type STATE =
  sig
     type t
     val init : t
  end
module type FUNCID =
  sig
     module State :
     Asmir_disasm.STATE
     val find_calls :
       Cfg.AST.G.t ->
       Cfg.AST.G.V.t list ->
       Cfg_ast.unresolved_edge list ->
       State.t -> Cfg_ast.unresolved_edge list * State.t
     val find_rets :
       Cfg.AST.G.t ->
       Cfg.AST.G.V.t list ->
       Cfg_ast.unresolved_edge list ->
       State.t -> Cfg_ast.unresolved_edge list * State.t
```

```
  end

module type DISASM =
  sig

     module State :

     Asmir_disasm.STATE

     val get_succs :
       Asmir.asmprogram ->
       Cfg.AST.G.t ->
       Cfg_ast.unresolved_edge list ->
       State.t -> (Cfg_ast.unresolved_edge * Asmir_disasm.succs) list * State.t
```

Function that returns the successors of one or more nodes in the unresolved list.

```
     val fixpoint : bool
```

Should `get_succs` be called until a fixpoint is reached?

```
  end

module Make :
   functor (D : DISASM) -> functor (F : FUNCID) ->   sig

     val disasm_at :
       ?callsig:Var.defuse ->
       Asmir.asmprogram -> Type.addr -> Cfg.AST.G.t * D.State.t

     val disasm :
       ?callsig:Var.defuse -> Asmir.asmprogram -> Cfg.AST.G.t * D.State.t

  end

val recursive_descent :
  ?callsig:Var.defuse -> Asmir.asmprogram -> Cfg.AST.G.t

val recursive_descent_at :
  ?callsig:Var.defuse -> Asmir.asmprogram -> Type.addr -> Cfg.AST.G.t

type vsaresult = {
  origssa : Cfg.SSA.G.t ;
  optssa : Cfg.SSA.G.t ;
  vsa_in : Cfg.ssastmtloc -> Vsa_ssa.AbsEnv.t option ;
  vsa_out : Cfg.ssastmtloc -> Vsa_ssa.AbsEnv.t option ;
}

val vsa_full :
  ?callsig:Var.defuse ->
  Asmir.asmprogram -> Cfg.AST.G.t * vsaresult option

val vsa_at_full :
  ?callsig:Var.defuse ->
  Asmir.asmprogram -> Type.addr -> Cfg.AST.G.t * vsaresult option
```

```
val vsa : ?callsig:Var.defuse -> Asmir.asmprogram -> Cfg.AST.G.t
val vsa_at :
  ?callsig:Var.defuse -> Asmir.asmprogram -> Type.addr -> Cfg.AST.G.t
type algorithm =
  | Vsa
  | Rd
val recover :
  ?callsig:Var.defuse ->
  algorithm -> Asmir.asmprogram -> Cfg.AST.G.t
val recover_at :
  ?callsig:Var.defuse ->
  algorithm -> Asmir.asmprogram -> Type.addr -> Cfg.AST.G.t
```

6 Module `Asmir_rdisasm` : Lift a program using recursive disassembly.

Author(s): Ricky Zhou

```
type callback = Type.addr -> Type.addr -> Ast.stmt list -> bool
val rdisasm_at :
  ?f:callback ->
  Asmir.asmprogram -> Type.addr list -> Ast.program * string
```

Recursively disassemble `p` beginning from known addresses in `startaddrs`. If `f` is defined and `f addr stmts` returns false, raises `Asmir.Disassembly_error`[3].

```
val rdisasm : ?f:callback -> Asmir.asmprogram -> Ast.program * string
```

Recursively disassemble `p` beginning at the program's defined start address and any function symbols. `f` behaves the same as in `Asmir_rdisasm.rdisasm_at`[6].

```
val max_callback : int -> callback
```

`max_callback n` produces a callback function that stops disassembling after discovering `n` instructions. The resulting callback function can be passed to the `f` arguments in `rdisasm_at` or `rdisasm`.

7 Module `Asmir_vars` : Asmir variables

```
val x86_regs : Ast.var list
val x64_regs : Ast.var list
val full_regs : Ast.var list
val x86_mem : Var.t
```

```
val x64_mem : Var.t
module X86 :
  sig
     module R32 :
     Disasm_i386.R32
     module R64 :
     Disasm_i386.R64
  end
val mem_of_type : Type.typ -> Var.t
val arm_regs : Var.t list
val subregs : (string * (string * int * Type.typ)) list
val x86_all_regs : Ast.var list
val x64_all_regs : Ast.var list
val multiarch_all_regs : Ast.var list
```

8 Module Ast : The Abstract Syntax Tree.

This IL allows nested expressions, making it closer to VEX and the concrete syntax than our SSA form. However, in most cases, this makes analysis harder, so you will generally want to convert to SSA for analysis.

Author(s): Ivan Jager

```
type var = Var.t
type exp =
  | Load of (exp * exp * exp * Type.typ)
```

Load(arr,idx,endian,t)

```
  | Store of (exp * exp * exp * exp * Type.typ)
```

Store(arr,idx,val,endian,t)

```
  | BinOp of (Type.binop_type * exp * exp)
  | UnOp of (Type.unop_type * exp)
  | Var of var
  | Lab of string
  | Int of (Big_int_Z.big_int * Type.typ)
  | Cast of (Type.cast_type * Type.typ * exp)
```

Cast to a new type.

```
  | Let of (var * exp * exp)
  | Unknown of (string * Type.typ)
  | Ite of (exp * exp * exp)
  | Extract of (Big_int_Z.big_int * Big_int_Z.big_int * exp)
```

Extract hbits to lbits of e (Reg type)

```
  | Concat of (exp * exp)
```

Concat two reg expressions together

```
type attrs = Type.attributes
type stmt =
  | Move of (var * exp * attrs)
```

Assign the value on the right to the var on the left

```
  | Jmp of (exp * attrs)
```

Jump to a label/address

```
  | CJmp of (exp * exp * exp * attrs)
```

Conditional jump. If e1 is true, jumps to e2, otherwise jumps to e3

```
  | Label of (Type.label * attrs)
```

A label we can jump to

```
  | Halt of (exp * attrs)
  | Assert of (exp * attrs)
  | Assume of (exp * attrs)
  | Comment of (string * attrs)
```

A comment to be ignored

```
  | Special of (string * Var.defuse option * attrs)
```

A "special" statement. (does magic)

```
type program = stmt list
val exp_of_lab : Type.label -> exp
```

Make an expression corresponding to the given label, for use as the target of a `Jmp`.

```
val lab_of_exp : exp -> Type.label option
```

If possible, make a label that would be refered to by the given expression.

```
val exp_false : exp
```

False constant. (If convenient, refer to this rather than building your own.)

```
val exp_true : exp
```

True constant.

```
val little_endian : exp
val big_endian : exp
val newlab : ?pref:string -> unit -> Type.label
val num_exp : exp -> int
val getargs :
  exp ->
```

```
  exp list * Type.typ list * Type.binop_type list * Type.unop_type list *
  var list * string list * Big_int_Z.big_int list * Type.cast_type list
val subexps : exp -> exp list
val quick_exp_eq : exp -> exp -> bool
val full_exp_eq : 'a -> 'a -> bool
```

full_exp_eq e1 e2 returns true if and only if e1 and e2 are structurally equivalent.

```
val (===) : 'a -> 'a -> bool
val compare_exp : Var.t -> Var.t -> int
val num_stmt : stmt -> int
val getargs_stmt :
  stmt ->
  exp list * var list * Type.label list * attrs list * string list
val quick_stmt_eq : stmt -> stmt -> bool
```

quick_stmt_eq returns true if and only if the subexpressions in e1 and e2 are *physically* equal.

```
val full_stmt_eq : 'a -> 'a -> bool
```

full_stmt_eq returns true if and only if e1 and e2 are structurally equivalent.

```
val is_indirectjump : stmt -> bool
val is_syscall : stmt -> bool
val full_stmts_eq : 'a list -> 'a list -> bool
val is_true : exp -> bool
val is_false : exp -> bool
val get_attrs : stmt -> attrs
```

9 Module Ast_cond_simplify : Simplify predicates so that VSA and other abstract interpretations can use them.

```
val simplifycond_cfg : Cfg.AST.G.t -> Cfg.AST.G.t
```

Simplify conditions used in edge labels

10 Module Ast_convenience : Utility functions for ASTs.

It's useful to have these in a separate file so it can use functions from Typecheck and elsewhere.

```
val cjmp : Ast.exp -> Ast.exp -> Ast.stmt list
```

Create a single target cjmp. Uses a hopefully-unique label for the other.

```
val ncjmp : Ast.exp -> Ast.exp -> Ast.stmt list
```

Create a single target cjmp with inverted condition. Uses a hopefully-unique label for the other.

```
val unknown : Type.typ -> string -> Ast.exp
val binop : Type.binop_type -> Ast.exp -> Ast.exp -> Ast.exp
val unop : Type.unop_type -> Ast.exp -> Ast.exp
val concat : Ast.exp -> Ast.exp -> Ast.exp
val extract : int -> int -> Ast.exp -> Ast.exp
val exp_and : Ast.exp -> Ast.exp -> Ast.exp
val exp_or : Ast.exp -> Ast.exp -> Ast.exp
val exp_eq : Ast.exp -> Ast.exp -> Ast.exp
val exp_not : Ast.exp -> Ast.exp
val exp_implies : Ast.exp -> Ast.exp -> Ast.exp
val ( +* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( -* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( ** ) : Ast.exp -> Ast.exp -> Ast.exp
val ( <<* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( >>* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( >>>* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( &* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( |* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( ^* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( ==* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( <>* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( <* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( >* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( <=* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( >=* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( =* ) : Ast.exp -> Ast.exp -> Ast.exp
```

bitwise equality

```
val ( ++* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( %* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( $%* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( /* ) : Ast.exp -> Ast.exp -> Ast.exp
val ( $/* ) : Ast.exp -> Ast.exp -> Ast.exp
val cast : Type.cast_type -> Type.typ -> Ast.exp -> Ast.exp
val cast_low : Type.typ -> Ast.exp -> Ast.exp
```

```
val cast_high : Type.typ -> Ast.exp -> Ast.exp
val cast_signed : Type.typ -> Ast.exp -> Ast.exp
val cast_unsigned : Type.typ -> Ast.exp -> Ast.exp
val exp_int : Big_int_Z.big_int -> int -> Ast.exp
val it : int -> Type.typ -> Ast.exp
val exp_ite : ?t:Type.typ -> Ast.exp -> Ast.exp -> Ast.exp -> Ast.exp
val parse_ite : Ast.exp -> (Ast.exp * Ast.exp * Ast.exp) option
val parse_implies : Ast.exp -> (Ast.exp * Ast.exp) option
val rm_duplicates : Ast.exp -> Ast.exp
```

Duplicate any shared nodes. Useful for using physical location as a unique identity.

XXX: I think this would be much faster if we only duplicated things that actually occur more than once.

```
val parse_extract : Ast.exp -> (Z.t * Big_int_Z.big_int) option
val parse_concat : Ast.exp -> (Ast.exp * Ast.exp) option
val rm_ite : Ast.exp -> Ast.exp
val rm_extract : Ast.exp -> Ast.exp
val rm_concat : Ast.exp -> Ast.exp
val last_meaningful_stmt : Ast.stmt list -> Ast.stmt
val min_symbolic : signed:bool -> Ast.exp -> Ast.exp -> Ast.exp
val max_symbolic : signed:bool -> Ast.exp -> Ast.exp -> Ast.exp
val extract_element : Type.typ -> Ast.exp -> int -> Ast.exp
val extract_byte : Ast.exp -> int -> Ast.exp
val extract_element_symbolic : Type.typ -> Ast.exp -> Ast.exp -> Ast.exp
val extract_byte_symbolic : Ast.exp -> Ast.exp -> Ast.exp
val reverse_bytes : Ast.exp -> Ast.exp
val concat_explist : Ast.exp BatEnum.t -> Ast.exp
```

11 Module Ast_mapper : Apply a mapping to all expressions in a program.

Author(s): Thanassis

```
class type map =
  object
    method exp : Ast.exp -> Ast.exp
    method stmt : Ast.stmt -> Ast.stmt
    method prog : Ast.program -> Ast.program
    method cfg : Cfg.AST.G.t -> Cfg.AST.G.t
```

```
  end
```

A mapping object.

```
val map_e : (Ast.exp -> Ast.exp Type.visit_action) -> map
```

`map_e f` returns a map object for `f`.

For instance, to map all expressions in a cfg, use `(map_e f)#cfg cfg`.

12 Module `Ast_piqi` : Convert AST programs to the Piqi serialization format, which can convert to protobuffers, xml, and json.

Author(s): Edward J. Schwartz

```
val to_pb : Ast.program -> string
```

`to_pb p` converts `p` to protobuffer format.

```
val to_json : Ast.program -> string
```

`to_json p` converts `p` to JSON format.

```
val to_xml : Ast.program -> string
```

`to_xml p` converts `p` to XML format.

13 Module `Ast_slice` : A module to perform chopping on ASTs

```
module CHOP_AST :
  sig
    module Traverse :
    Graph.Traverse.Dfs(Cfg.AST.G)
    module SG :
      sig
        type t = Cfg.AST.G.t
        module V :
        Cfg.AST.G.V
        val iter_vertex : (Cfg.AST.G.vertex -> unit) -> Cfg.AST.G.t -> unit
        val iter_succ :
          (Cfg.AST.G.vertex -> unit) -> Cfg.AST.G.t -> Cfg.AST.G.vertex -> unit
```

```
    end
  module Comp :
  Graph.Components.Make(SG)
  val rewrite_missing_labels : Cfg.AST.G.t -> Cfg.AST.G.t
  val get_scc : SG.t ->
    Cfg.AST.G.vertex -> Cfg.AST.G.vertex -> SG.t
  val compute_cds :
    Depgraphs.CDG_AST.G.t ->
    (Cfg.AST.G.vertex * int, Cfg.AST.G.vertex * int) Hashtbl.t -> unit
  val add_jmp_stmts :
    Cfg.AST.G.t ->
    (Cfg.AST.G.vertex * int, Cfg.AST.G.vertex * int) Hashtbl.t -> unit
  val get_dds :
    Depgraphs.CDG_AST.G.t ->
    (Depgraphs.DDG_AST.location, Depgraphs.DDG_AST.location) Hashtbl.t
  val slice :
    Depgraphs.CDG_AST.G.t -> Cfg.AST.G.V.t -> int -> Depgraphs.CDG_AST.G.t
  val chop : SG.t -> int -> 'a -> int -> int -> Depgraphs.CDG_AST.G.t
end
```

14 Module `Ast_visitor` : Visitor for AST programs and expressions.

Visitors are a systematic method of exploring and modifying programs and expressions.

Users create visitors that describe what actions to take when various constructs are encountered. For instance, the `visit_exp` method is called whenever an expression is encountered. This visitor is passed to `accept` functions, which takes an object to visit, and calls the visitor's methods are the proper time.

```
class type t =
  object
```

`method visit_exp : Ast.exp -> Ast.exp Type.visit_action`

Called when visiting an expression

`method visit_stmt : Ast.stmt -> Ast.stmt Type.visit_action`

Called when visiting a statement.

Note that in a Move() or Let(), referenced variables will be visited first, so that this can be used to add the assigned variable to your context.

FIXME: would be nice to be able to add stmts... We may change this.

`method visit_label : Type.label -> Type.label Type.visit_action`

Called when visiting a label. (IE: inside a statment like Cjmp)

`method visit_rvar : Ast.var -> Ast.var Type.visit_action`

Called when visiting a referenced variable. (IE: inside an expression)

`method visit_avar : Ast.var -> Ast.var Type.visit_action`

Called when visiting an assigned variable. (IE: On the LHS of a Move

```
method visit_lbinding :
  Ast.var * Ast.exp -> (Ast.var * Ast.exp) Type.visit_action
```

Called on the binding when recursing into a Let. This allows doing stuff between the first and second expressions in a Let.

`method visit_ulbinding : Ast.var -> Ast.var Type.visit_action`

Called when visiting a variable being unbound. For instance, variable x after visiting let x = y in z.

`end`

The type for a visitor

`class nop : t`

A nop visitor that visits all children, but does not change anything. This visitor can be inherited from to build a new one.

14.0.1 Accept functions

`val label_accept : #t -> Type.label -> Type.label`

Visit a label definition

`val rvar_accept : #t -> Ast.var -> Ast.var`

Visit a referenced variable

`val avar_accept : #t -> Ast.var -> Ast.var`

Visit an assigned variable

`val lbinding_accept : #t -> Ast.var * Ast.exp -> Ast.var * Ast.exp`

Visit a let variable being bound

`val ulbinding_accept : #t -> Ast.var -> Ast.var`

Visit a let variable being unbound

`val exp_accept : #t -> Ast.exp -> Ast.exp`

Visit an expression

`val stmt_accept : #t -> Ast.stmt -> Ast.stmt`

Visit a statement

`val prog_accept : #t -> Ast.program -> Ast.program`

Visit an AST program

`val cfg_accept : #t -> Cfg.AST.G.t -> Cfg.AST.G.t`

Visit an AST program in CFG form

15 Module `BatListFull` : Opening this module will put a more tail-recursive List module into scope.

A complete list module that uses Batteries functions when possible.
Why is this not in batteries itself?

```
module List :
  sig
    include List
    include BatList
  end
```

16 Module `Big_int_convenience` : Convenience functions for Big_ints

```
module HashType :
  sig
    type t = Big_int_Z.big_int
    val equal : Z.t -> Z.t -> bool
    val hash : 'a -> int
  end
module OrderedType :
  sig
    type t = Big_int_Z.big_int
    val compare : Z.t -> Z.t -> int
```

```
  end
module BIH :
   Hashtbl.Make(HashType)
module BIS :
   Set.Make(OrderedType)
module BIM :
   Map.Make(OrderedType)
val bi0 : Z.t
val bi1 : Z.t
val bi2 : Z.t
val bi3 : Z.t
val bi4 : Z.t
val bi5 : Z.t
val bi6 : Z.t
val bi7 : Z.t
val bi8 : Z.t
val bi9 : Z.t
val bia : Z.t
val bib : Z.t
val bic : Z.t
val bid : Z.t
val bie : Z.t
val bif : Z.t
val bim1 : Z.t
val bim2 : Z.t
val bim3 : Z.t
val bim4 : Z.t
val bim5 : Z.t
val bim6 : Z.t
val bim7 : Z.t
val bim8 : Z.t
val bim9 : Z.t
val bima : Z.t
val bimb : Z.t
val bimc : Z.t
val bimd : Z.t
val bime : Z.t
val bimf : Z.t
```

`val biconst : int -> Z.t`

`val bi : int -> Z.t`

`val biconst32 : int32 -> Z.t`

`val bi32 : int32 -> Z.t`

`val biconst64 : int64 -> Z.t`

`val bi64 : int64 -> Z.t`

`val big_int_of_bool : bool -> Z.t`

`val (==%) : Z.t -> Z.t -> bool`

Infix operator to test if two big ints are equal.

`val (<>%) : Z.t -> Z.t -> bool`

Infix operator to test for non-equality

`val (<%) : Z.t -> Z.t -> bool`

Infix operator for $<$

`val (<=%) : Z.t -> Z.t -> bool`

Infix operator for $\leq$

`val (>%) : Z.t -> Z.t -> bool`

Infix operator for $>$

`val (>=%) : Z.t -> Z.t -> bool`

Infix operator for $\geq$

`val (<<%) : Z.t -> int -> Z.t`

Infix operator for $<<$

`val (>>%) : Z.t -> int -> Z.t`

Infix operator for $>>$

`val ($>>%) : Z.t -> int -> Z.t`

Infix operator for $\$>>$

`val (+%) : Z.t -> Z.t -> Z.t`

Infix operator for $+$

`val ( *% ) : Z.t -> Z.t -> Z.t`

Infix operator for *

`val (++%) : Z.t -> Z.t`

Operator for incremeneting

`val (-%) : Z.t -> Z.t -> Z.t`

Infix operator for -

`val (|%) : Z.t -> Z.t -> Z.t`

Infix operator for |

`val (&%) : Z.t -> Z.t -> Z.t`

Infix operator for &

`val (/%) : Z.t -> Z.t -> Z.t`

`val (%%) : Z.t -> Z.t -> Z.t`

Infix operator for mod (%)

`val (~%) : Z.t -> string`

Operator for printing as string

`val bi_is_zero : Z.t -> bool`

bi_is_zero bi returns true iff bi = 0

`val bi_is_one : Z.t -> bool`

bi_is_one bi returns true iff bi = 1

`val bi_is_minusone : Z.t -> bool`

bi_is_minusone bi returns true iff bi = -1

`val uintmax64 : Z.t`

For big_int addresses

`val sintmax64 : Z.t`

`val addr_to_int64 : Z.t -> int64`

`val addr_of_int64 : int64 -> Z.t`

`type address = Big_int_Z.big_int`

`val address_to_int64 : address -> int64`

`val address_of_int64 : int64 -> address`

17 Module `Cfg` : Control flow graphs.

Control flow graphs contain nodes for each basic block of code, with edges between nodes representing a possible control flow.

Author(s): Ivan Jager

17.0.2 Basic block identifiers

```
type bbid =
```

```
  | BB_Entry
```

Entry node

```
  | BB_Exit
```

Return/exit node

```
  | BB_Indirect
```

Indirect jump to/from a node

```
  | BB_Error
```

Error node

```
  | BB of int
```

Regular basic blocks

A basic block identifier.

```
val bbid_to_string : bbid -> string
```

Convert a bbid to a string.

```
val edge_direction : (bool option * 'a) option -> bool option
```

Extract the direction taken from an edge

```
module BBid :
  sig
    type t = Cfg.bbid
    val compare : 'a -> 'a -> int
    val hash : Cfg.bbid -> int
    val equal : 'a -> 'a -> bool
  end
module BS :
   Set.S  with type elt = BBid.t
module BH :
   Hashtbl.S  with type key = BBid.t
module BM :
   Map.S  with type key = BBid.t
```

17.0.3 Control flow graphs

```
module type CFG =
  sig
```

```
type stmt
type lang = stmt list
type exp
include Graph.Builder.S
```

Edge labels: None → unconditional edge Some(None, e) → indirect edge Some(true, e) → conditional true edge Some(false, e) → conditional false edge

```
val find_vertex : G.t -> G.V.label -> G.V.t
```

Finds a vertex by a bbid

```
val find_label : G.t -> Type.label -> G.V.t
```

Finds a vertex by a label in its stmts

```
val get_stmts : G.t -> G.V.t -> lang
```

Gets the statements from a basic block

```
val default : lang
```

Get an empty list of statements

```
val set_stmts : G.t -> G.V.t -> lang -> G.t
```

Sets the statements for a basic block

```
val join_stmts : lang -> lang -> lang
```

Joins two statement lists

```
val lang_to_string : lang -> string
```

Convert lang to string

Generate a new ID that wasn't previously generated for the given graph

```
val create_vertex : G.t -> lang -> G.t * G.V.t
```

Creates a new vertex with new ID and adds it to the graph with the given statements.

```
val copy_map : G.t -> G.t
```

Copy the metadata of a CFG without copying the vertices

```
val remove_vertex : G.t -> G.V.t -> G.t
val remove_edge : G.t -> G.V.t -> G.V.t -> G.t
val remove_edge_e : G.t -> G.E.t -> G.t
val v2s : G.V.t -> string
```

Convert a vertex's label to a string

```
  end
```

The type of a control flow graph

```
module AST :
   CFG  with type stmt = Ast.stmt and type exp = Ast.exp
```

Control flow graph in which statements are in `Ast.stmt`[8] form.

```
module SSA :
   CFG  with type stmt = Ssa.stmt and type exp = Ssa.exp
```

Control flow graph in which statements are in `Ssa.stmt`[61] form. All variables are assigned at most one time in the program, and expressions do not contain subexpressions.

17.0.4 Location of statements in CFGs

```
type aststmtloc = AST.G.V.t * int
```

Unique identifier for an AST CFG statement. `(v,n)` means the `n`th (zero-indexed) statement in basic block `v`.

```
type ssastmtloc = SSA.G.V.t * int
```

Unique identifier for an SSA CFG statement. `(v,n)` means the `n`th (zero-indexed) statement in basic block `v`.

17.0.5 Helper functions for CFG conversions

```
val map_ast2ssa :
  (Ast.stmt list -> Ssa.stmt list) ->
  (AST.exp -> SSA.exp) ->
  AST.G.t ->
  SSA.G.t * (SSA.G.V.t -> AST.G.V.t) *
  (SSA.G.E.t -> AST.G.E.t)
val map_ssa2ast :
  (Ssa.stmt list -> Ast.stmt list) ->
  (SSA.exp -> AST.exp) ->
  SSA.G.t ->
  AST.G.t * (AST.G.V.t -> SSA.G.V.t) *
  (AST.G.E.t -> SSA.G.E.t)
```

18 Module `CfgDataflow` : Dataflow module for use with Control Flow Graphs.

Use this module to write dataflow analyses in BAP instead of `GraphDataflow`[40].

```
module type CFG =
  sig
```

```
    type exp
    type stmt
    type lang = stmt list
    module G :
      sig
        type t
        module V :
        Graph.Sig.COMPARABLE
        module E :
        Graph.Sig.EDGE  with type vertex = V.t and type label = (bool option * exp)
        option
        val pred_e : t -> V.t -> E.t list
        val succ_e : t -> V.t -> E.t list
        val fold_vertex : (V.t -> 'a -> 'a) -> t -> 'a -> 'a
      end
    val get_stmts : G.t -> G.V.t -> lang
    val v2s : G.V.t -> string
  end
```

Types of control flow graph that data flow is defined on

```
module type DATAFLOW =
  sig
    module L :
    GraphDataflow.BOUNDED_MEET_SEMILATTICE
    module CFG :
    CfgDataflow.CFG
    module O :
    GraphDataflow.OPTIONS
    val stmt_transfer_function :
      O.t ->
      CfgDataflow.CFG.G.t -> CFG.G.V.t * int -> CfgDataflow.CFG.stmt -> L.t -> L.t
```

The transfer function over statements. The second argument contains the location of the statement being processed,and is generally unused.

```
    val edge_transfer_function :
      O.t ->
      CfgDataflow.CFG.G.t ->
      CfgDataflow.CFG.G.E.t -> CfgDataflow.CFG.exp option -> L.t -> L.t
```

The transfer function over edge elements, e.g., conditions. The second argument is generally unused.

```
val s0 : O.t -> CfgDataflow.CFG.G.t -> CFG.G.V.t
```

The starting node for the analysis.

```
val init : O.t -> CfgDataflow.CFG.G.t -> L.t
```

The initial lattice value given to node `s0`. All other nodes start out with `Top`.

```
val dir : O.t -> GraphDataflow.direction
```

The dataflow direction.

```
  end
```

A dataflow problem is defined by a lattice over a CFG.

```
module type DATAFLOW_WITH_WIDENING =
  sig
    module L :
    GraphDataflow.BOUNDED_MEET_SEMILATTICE_WITH_WIDENING
    module CFG :
    CfgDataflow.CFG
    module O :
    GraphDataflow.OPTIONS
    val stmt_transfer_function :
      O.t ->
      CfgDataflow.CFG.G.t -> CFG.G.V.t * int -> CfgDataflow.CFG.stmt -> L.t -> L.t
```

The transfer function over statements. The second argument contains the location of the statement being processed,and is generally unused.

```
    val edge_transfer_function :
      O.t ->
      CfgDataflow.CFG.G.t ->
      CfgDataflow.CFG.G.E.t -> CfgDataflow.CFG.exp option -> L.t -> L.t
```

The transfer function over edge elements, e.g., conditions. The second argument is generally unused.

```
    val s0 : O.t -> CfgDataflow.CFG.G.t -> CFG.G.V.t
```

The starting node for the analysis.

```
    val init : O.t -> CfgDataflow.CFG.G.t -> L.t
```

The initial lattice value given to node `s0`. All other nodes start out with `Top`.

```
val dir : O.t -> GraphDataflow.direction
```

The dataflow direction.

```
end
```

A dataflow problem with widening.

```
module Make :
   functor (D : DATAFLOW) ->   sig
```

```
val worklist_iterate :
  ?init:(D.O.t -> D.CFG.G.t -> D.L.t) ->
  ?opts:D.O.t -> D.CFG.G.t -> (D.CFG.G.V.t -> D.L.t) * (D.CFG.G.V.t -> D.L.t)
```

`worklist_iterate g` returns a worklist algorithm for graph `g` as a pair of functions `in,out`. `in`, when given a node `v`, computes the lattice value going in to that node, `v`. `out`, when given a node `v`, computes the lattice value exiting `v`.

```
val worklist_iterate_stmt :
  ?init:(D.O.t -> D.CFG.G.t -> D.L.t) ->
  ?opts:D.O.t ->
  D.CFG.G.t -> (D.CFG.G.V.t * int -> D.L.t) * (D.CFG.G.V.t * int -> D.L.t)
```

Like `worklist_iterate`, except the dataflow is done on the statement level. A statement `bb,n` is the `n`th stmt (zero-indexed) in `bb`.

```
val last_loc : D.CFG.G.t -> D.CFG.G.V.t -> D.CFG.G.V.t * int
```

Returns the location corresponding to the last statement in `bb`.

```
end
```

Build a custom dataflow algorithm for the given dataflow problem `D`.

```
module MakeWide :
   functor (D : DATAFLOW_WITH_WIDENING) ->   sig
```

```
val worklist_iterate_widen :
  ?init:(D.O.t -> D.CFG.G.t -> D.L.t) ->
  ?nmeets:int ->
  ?opts:D.O.t -> D.CFG.G.t -> (D.CFG.G.V.t -> D.L.t) * (D.CFG.G.V.t -> D.L.t)
```

Same as `worklist_iterate`, but additionally employs the widening operator as lattice values propagate over backedges in the CFG. Backedges are identified by observing when lattices values flow in cycles.

```
val worklist_iterate_widen_stmt :
  ?init:(D.O.t -> D.CFG.G.t -> D.L.t) ->
  ?nmeets:int ->
  ?opts:D.O.t ->
  D.CFG.G.t -> (D.CFG.G.V.t * int -> D.L.t) * (D.CFG.G.V.t * int -> D.L.t)
```

Like `worklist_iterate_widen`, except the dataflow is done on the statement level. A statement `bb,n` is the `n`th stmt (zero-indexed) in `bb`.

```
val last_loc : D.CFG.G.t -> D.CFG.G.V.t -> D.CFG.G.V.t * int
```

Returns the location corresponding to the last statement in `bb`.

```
end
```

Build a custom dataflow algorithm for the given dataflow problem with widening operator `D`.

19 Module `Cfg_ast` : Conversions from AST programs to AST CFGs and vice versa.

```
type unresolved_edge = Cfg.AST.G.V.t * Cfg.AST.G.E.label * Ast.exp
val of_prog : ?special_error:bool -> Ast.program -> Cfg.AST.G.t
```

`of_prog p` converts `p` to an AST CFG.

```
val to_prog : Cfg.AST.G.t -> Ast.program
```

`to_prog cfg` converts `cfg` to an AST program.

19.0.6 CFG Manipulation Functions

```
val create_entry : Cfg.AST.G.t -> Cfg.AST.G.t * Cfg.AST.G.V.t
```

Add `BB_Entry` in a graph.

```
val find_entry : Cfg.AST.G.t -> Cfg.AST.G.t * Cfg.AST.G.V.t
```

Find `BB_Entry` in a graph.

```
val find_error : Cfg.AST.G.t -> Cfg.AST.G.t * Cfg.AST.G.V.t
```

Find `BB_Error` in a graph, or add it if not already present.

```
val find_exit : Cfg.AST.G.t -> Cfg.AST.G.t * Cfg.AST.G.V.t
```

Find `BB_Exit` in a graph, or add it if not already present.

```
val find_indirect : Cfg.AST.G.t -> Cfg.AST.G.t * Cfg.AST.G.V.t
```

Find `BB_Indirect` in a graph, or add it if not already present.

19.0.7 Convenience Functions

```
val v2s : Cfg.AST.G.V.t -> string
```

`v2s v` returns the basic blocker identifier string associated with `v`.

19.0.8 Internal Functions

```
val add_prog :
  ?special_error:bool ->
  Cfg.AST.G.t ->
  Ast.program ->
  Cfg.AST.G.t * unresolved_edge list * Cfg.AST.G.V.t list *
  Cfg.AST.G.V.t option
```

`add_prog cfg p` adds the AST program `p` into the existing AST CFG `cfg`. It returns a tuple consisting of the updated CFG, new postponed edges, newly added BBs (the first node is the entry), and the node through which `p` can fall-through (fall off the end of the program), if one exists.

This function exists to enable incremental lifting, and should not be used for other applications.

20 Module `Cfg_pp` : Pretty printing for CFGs.

```
module CS :
   Cfg.SSA
module CA :
   Cfg.AST
module type DOTTYG =
  sig
     type t
     module V :
     Graph.Sig.COMPARABLE
     module E :
       sig
         type t
         type label
         val label : t -> label
         val src : t -> V.t
         val dst : t -> V.t
```

```
      end
    val iter_vertex : (V.t -> unit) -> t -> unit
    val iter_edges_e : (E.t -> unit) -> t -> unit
    val graph_attributes : t -> Graph.Graphviz.DotAttributes.graph list
    val default_vertex_attributes : t -> Graph.Graphviz.DotAttributes.vertex list
    val vertex_name : V.t -> string
    val vertex_attributes : V.t -> Graph.Graphviz.DotAttributes.vertex list
    val get_subgraph : V.t -> Graph.Graphviz.DotAttributes.subgraph option
    val default_edge_attributes : t -> Graph.Graphviz.DotAttributes.edge list
    val edge_attributes : E.t -> Graph.Graphviz.DotAttributes.edge list
  end
val red : int
val green : int
val blue : int
val fill : int
val linenum_color : string
module DefAttrs :
  sig
    val graph_attributes : 'a -> 'b list
    val default_vertex_attributes :
      'a ->
      [> `Fillcolor of int | `Shape of [> `Box ] | `Style of [> `Filled ] ] list
    val vertex_attributes : 'a -> 'b list
    val get_subgraph : 'a -> 'b option
    val default_edge_attributes : 'a -> [> `Color of int ] list
    val edge_attributes : 'a -> 'b list
  end
module DefAttributor :
   functor (G : Cfg.CFG) ->   sig
    val vertex_attributes : 'a -> 'b -> 'c list
    val edge_attributes : 'a -> 'b -> 'c list
  end
module EdgeColorAttributor :
   functor (G : Cfg.CFG) ->   sig
    val vertex_attributes : 'a -> 'b -> 'c list
    val edge_attributes : 'a -> G.G.E.t -> [> `Color of int ] list
```

```
  end

module type Attributor =
   functor (G : Cfg.CFG) ->   sig

     val vertex_attributes :
       G.G.t -> G.G.V.t -> Graph.Graphviz.DotAttributes.vertex list
     val edge_attributes :
       G.G.t -> G.G.E.t -> Graph.Graphviz.DotAttributes.edge list

  end

module MakeCfgPrinter :
   functor (G : Cfg.CFG) -> functor (Printer :    sig

     val print : G.G.t -> (G.G.V.t -> string) * (G.G.E.t -> string)

  end ) -> functor (Attributor :  Attributor) ->   sig


     include struct ...  end

  end
```

Makes a module suitable for use with Graph.Graphviz.Dot for writting out a CFG.

```
module PrintSsaStmts :
  sig

     val print :
       Cfg_pp.CS.G.t -> (Cfg_pp.CS.G.V.t -> string) * (Cfg_pp.CS.G.E.t -> string)

  end

module PrintAstStmts :
  sig

     val print :
       Cfg_pp.CA.G.t -> (Cfg_pp.CA.G.V.t -> string) * (Cfg_pp.CA.G.E.t -> string)

  end

module PrintAstAsms :
  sig

     exception Found of string
     val append : string -> string -> string
     val print :
       Cfg_pp.CA.G.t -> (Cfg_pp.CA.G.V.t -> string) * (Cfg_pp.CA.G.E.t -> string)

  end

module SsaStmtsDot :
   MakeCfgPrinter(CS)(PrintSsaStmts)(EdgeColorAttributor)
module AstStmtsDot :
```

```
   MakeCfgPrinter(CA)(PrintAstStmts)(EdgeColorAttributor)
module AstAsmsDot :
   MakeCfgPrinter(CA)(PrintAstAsms)(EdgeColorAttributor)
module SsaBBidPrinter :
  sig
     include CS.G
     include DefAttrs
     val vertex_name : Cfg_pp.CS.G.V.t -> string
  end
module SsaBBidDot :
   Graph.Graphviz.Dot(SsaBBidPrinter)
module AstBBidPrinter :
  sig
     include CA.G
     include DefAttrs
     val vertex_name : Cfg_pp.CA.G.V.t -> string
  end
module AstBBidDot :
   Graph.Graphviz.Dot(AstBBidPrinter)
module SsaStmtsAttDot :
   MakeCfgPrinter(CS)(PrintSsaStmts)(EdgeColorAttributor)
module AstStmtsAttDot :
   MakeCfgPrinter(CA)(PrintAstStmts)(EdgeColorAttributor)
```

21 Module `Cfg_ssa` : Static Single Assignment translation

Author(s): Ivan Jager

```
val v2s : Cfg.SSA.G.V.t -> string
val of_astcfg : ?tac:bool -> Cfg.AST.G.t -> Cfg.SSA.G.t
```
Translates an AST CFG into a SSA CFG.

```
val of_ast : ?tac:bool -> Ast.program -> Cfg.SSA.G.t
```
Translates an AST program into an SSA CFG.

```
val to_astcfg : ?remove_temps:bool -> ?dsa:bool -> Cfg.SSA.G.t -> Cfg.AST.G.t
```
Convert a SSA CFG to an AST CFG.

```
val to_ast : ?remove_temps:bool -> Cfg.SSA.G.t -> Ast.program
```

Convert a SSA CFG to an AST program.

```
type cfg_translation_results = {
  ssacfg : Cfg.SSA.G.t ;
  to_ssaexp : Cfg.aststmtloc -> Ast.exp -> Ssa.exp ;
```

Maps CFG location and expression to equivalent SSA expression

```
  to_ssavar : Var.t -> Var.t ;
```

Maps AST vars to SSA at end of exit node.

```
  to_astvar : Var.t -> Var.t ;
```

Maps SSA vars back to the variable they came from

```
  to_astloc : Var.t -> Cfg.aststmtloc ;
```

Maps non-phi SSA vars to the location of the AST definition

```
}
```

The translated SSA CFG and three maps. `to_ssavar` maps from the original AST variables to the corresponding SSA variable at the end of the exit node. `to_astvar` maps from SSA variables to the variables they originally came from. `to_astloc` maps non-phi SSA variables to the definition location they originally came from. Both `to_astvar` and `to_ssavar` act like the identity function for variables that don't map to anything. (This is to avoid raising exceptions for corner cases, such as when variables are never assigned.) `to_astloc` raises the `Not_found` exception.

```
val trans_cfg : ?tac:bool -> Cfg.AST.G.t -> cfg_translation_results
```

Translates an AST CFG into SSA form.

```
type ssa_translation_results = {
  cfg : Cfg.AST.G.t ;
  to_astexp : Ssa.exp -> Ast.exp ;
```

Maps SSA expressions to AST expressions.

```
}
val trans_ssacfg :
  ?remove_temps:bool ->
  ?dsa:bool -> Cfg.SSA.G.t -> ssa_translation_results
```

Translates a SSA CFG to an AST CFG.

```
val do_tac_ssacfg : Cfg.SSA.G.t -> Cfg.SSA.G.t
```

Put SSA CFG in three address code form without adding phis.

22 Module Checks : Sanity checks to provide more understandable error messages

To disable all sanity checks, disable debugging on the `Checks` module using the facilities in the `Debug` module.

Author(s): Ed Schwartz

```
exception Sanity of string
```

```
type 'a sanityf = 'a -> string -> unit
```

Sanity checks take a function-specific argument type, and a string describing the calling code. If a sanity check fails, it will raise an exception with an error message including the passed string.

22.0.9 CFG checks

```
val connected_astcfg : Cfg.AST.G.t sanityf
```

`connected_astcfg g s` raises an exception iff g is not a connected graph.

```
val connected_ssacfg : Cfg.SSA.G.t sanityf
```

`connected_ssacfg g s` raises an exception iff g is not a connected graph.

```
module MakeConnectedCheck :
   functor (C : Cfg.CFG) ->   sig
```

```
     val connected_check : C.G.t Checks.sanityf
```

`connected_check g s` raises an exception iff g is not a connected graph.

```
  end
```

Build a connected check for other graphs

```
val acyclic_astcfg : Cfg.AST.G.t sanityf
```

`acyclic_astcfg g s` raises an exception iff g is not an acyclic graph.

```
val acyclic_ssacfg : Cfg.SSA.G.t sanityf
```

`acyclic_ssacfg g s` raises an exception iff g is not an acyclic graph.

```
module MakeAcyclicCheck :
   functor (C : Cfg.CFG) ->   sig
```

```
     val acyclic_check : C.G.t Checks.sanityf
```

`acyclic_check g s` raises an exception iff g is not an acyclic graph.

```
  end
```

Build an acyclic check for other graphs

```
val exit_astcfg :
  ?allowed_exits:Cfg.bbid list ->
  ?expected_exits:Cfg.bbid list -> Cfg.AST.G.t sanityf
```

`exit_astcfg g s` raises an exception iff g contains an unspecified exit node. Allowed and expected exit node bbids can be specified using `allowed_exits`.

```
val exit_ssacfg :
  ?allowed_exits:Cfg.bbid list ->
  ?expected_exits:Cfg.bbid list -> Cfg.SSA.G.t sanityf
```

`exit_ssacfg g s` raises an exception iff g contains an unspecified exit node. Allowed and expected exit node bbids can be specified using `allowed_exits`.

```
module MakeExitCheck :
   functor (C : Cfg.CFG) ->   sig
```

```
    val exit_check :
      ?allowed_exits:Cfg.bbid list ->
      ?expected_exits:Cfg.bbid list -> C.G.t Checks.sanityf
```

`exit_check g s` raises an exception iff g contains an unspecified exit node. Allowed and expected exit node bbids can be specified using `allowed_exits`.

```
  end
```

Build an exit check for other graphs

```
val indirect_astcfg : Cfg.AST.G.t sanityf
```

`indirect_astcfg g s` raises an exception iff g contains `BB_Indirect`, the node corresponding to an unresolved indirect jump.

```
val indirect_ssacfg : Cfg.SSA.G.t sanityf
```

`indirect_ssacfg g s` raises an exception iff g contains `BB_Indirect`, the node corresponding to an unresolved indirect jump.

```
module MakeIndirectCheck :
   functor (C : Cfg.CFG) ->   sig
```

```
    val indirect_check : C.G.t Checks.sanityf
```

`indirect_check g s` raises an exception iff g contains `BB_Indirect`, the node corresponding to an unresolved indirect jump.

```
  end
```

Build an indirect check for other graphs

23 Module Coalesce : Coalesce sequential basic blocks into a single basic block.

A statement s is reorderable if s can be swapped with an adjacent reorderable statement s2 without changing the semantics of the program. For example, labels and comments are reorderable statements.

A sequence seq of basic blocks bb1, ..., bbn is sequential if

- seq is the only path from bb1 to bbn in the control flow graph.
- The first (closest to bb1) non-reorderable statement in seq dominates all other non-reorderable statements in seq.
- bbn postdominates all other basic blocks in seq.

```
val coalesce_ast :
  ?nocoalesce:Cfg.AST.G.V.t list -> Cfg.AST.G.t -> Cfg.AST.G.t
```

coalesce_ast cfg returns a new AST CFG in which sequential basic blocks in cfg are coalesced into a single basic block.

```
val coalesce_ssa :
  ?nocoalesce:Cfg.SSA.G.V.t list -> Cfg.SSA.G.t -> Cfg.SSA.G.t
```

coalesce_ssa cfg returns a new SSA CFG in which sequential basic blocks in cfg are coalesced into a single basic block.

24 Module Copy_prop : Copy propagation analysis.

```
val copyprop_ssa :
  ?stop_before:(Ssa.exp -> bool) ->
  ?stop_after:(Ssa.exp -> bool) ->
  Cfg.SSA.G.t ->
  Ssa.exp Var.VarMap.t lazy_t * Ssa.exp Var.VarMap.t * (Ssa.exp -> Ssa.exp)
```

Compute copy propagation for SSA CFGs. Returns a tuple cm,rm,lookup. cm maps all variables with known copy propagations to their copy propagated expression. rm maps all variables to their known constant expression, if any, without copy propagating any subexpressions. lookup e will perform copy propagation on all variables in e. If it propagates some expression e' and stop_before e' is true, e' will not be included in the final propagated expression. Similarly if stop_after e' holds e' will be in the final propagated expression, but propagation will not occur on subexpressions of e'.

```
val copyprop_ast :
  ?stop_before:(Ast.exp -> bool) ->
  ?stop_after:(Ast.exp -> bool) ->
```

```
  Cfg.AST.G.t ->
  Cfg.aststmtloc ->
  Ast.exp Var.VarMap.t * (Cfg.aststmtloc * Ast.exp) Var.VarMap.t *
  (Ast.exp -> Ast.exp)
```

Same as `Copy_prop.copyprop_ssa`[24] but for AST CFGs. Note that analysis results for AST CFGs are not global, and so the analysis returns information before the specified program statement. The last tuple returned is a map from variables to their constant expression, if one exists, and the location of the assignment.

```
val get_vars :
  ?stop_before:(Ssa.exp -> bool) ->
  ?stop_after:(Ssa.exp -> bool) -> Cfg.SSA.G.t -> Ssa.exp list -> Var.VarSet.t
```

`get_vars g l` uses copy propagation to return a set of variables used in the computation of any expression in `l` in the graph `g`.

25 Module `Deadcode` : Dead code elimination for SSA graphs.

```
val do_dce : ?globals:Var.t list -> Cfg.SSA.G.t -> Cfg.SSA.G.t * bool
```

Performs dead code elimination, returning the new CFG and a bool indicating whether anything changed. A statement `s` is considered live if:

- `s` has the `Liveout` attribute
- OR `s` appears in `globals`
- OR `s` defines `v`, and another statement (live or non-live) refers to `v`
- OR `s` is not a move statement.

All non-live statements are considered dead, and will be removed.

```
val do_aggressive_dce :
  ?globals:Var.t list -> Cfg.SSA.G.t -> Cfg.SSA.G.t * bool
```

Just like `Deadcode.do_dce`[25], except dead code is detected more aggressively. Specifically, a statement `s` is considered live if

- `s` has the `Liveout` attribute
- OR `s` appears in `globals`
- OR `s` defines `v`, and a **live** statement refers to `v`
- OR `s` is a conditional branch that a live statement is control-dependent upon
- OR `s` is an assertion statement
- OR `s` is a halt statement
- OR `s` is a comment statement
- OR `s` is a special statement

Although non-move statements can be dead, only dead move and cjmp statements are currently removed.

`do_aggressive_dce` is considered experimental.

26 Module `Debug` : Debugging module.

This module contains functions for printing debugging output.

Debugging output is controlled via environment variables:

- `BAP_LOGFILE` If set, debug messages will be written to the file $BAP_LOGFILE. The default is to write to standard error. Alternatively, the set_logfile function changes the file at run-time.
- `BAP_DEBUG_MODULES` Specifies which modules should have debugging enabled or disabled. The list is separated by colons (`:`). *modulename* or `!`*modulename* turn debugging on or off respectively for *modulename*. If *modulename* is the empty string, it matches all modules. The leftmost match wins. Default behavior is undefined and subject to change. Note that *modulename* refers to the name used for debug printing, which is not necesarily identical to the module name in OCaml.
- `BAP_WARN_MODULES` Specifies which modules should have warning enabled or disabled. The format is like that of `BAP_DEBUG_MODULES`. Default behavior is on (unlike debugging).
- `BAP_DEBUG_TIMESTAMPS` Specified how (and whether) timestamps will be printed with each debug message. Supported values are `unix`, `iso`, `elapsed`, and `none`.

Author(s): David Brumley

```
val set_logfile : Pervasives.out_channel -> unit
```

Sets the file to which the debug messages will be written

```
module type DEBUG =
  sig
```

```
    val debug : unit -> bool
```

Returns `true` iff debugging is enabled

```
    val warn : bool
```

Returns `true` iff warnings are enabled

```
    val pdebug : string -> unit
```

Prints given string as a debug message.

```
    val dprintf : ('a, unit, string, unit) Pervasives.format4 -> 'a
```

Prints given format string as a debug message.

```
    val dtrace :
      before:('a -> unit) -> f:('a -> 'b) -> after:('b -> unit) -> 'a -> 'b
```

dtrace before f after will return f(), but will call before first and after calling f it will pass the result to after. Also, while f is running, indentation for all debugging functions will be increased.

```
    val pwarn : string -> unit
```

Prints given string as a warning message.

```
    val wprintf : ('a, unit, string, unit) Pervasives.format4 -> 'a
```

Prints given format string as a warning message.

```
  end
```

A DEBUG module is used by client code to add debugging code that can be easily enabled or disabled through environment variables.

```
module type DEBUG_MOD_INFO =
  sig
```

```
    val name : string
```

The name to prefix all debug and warning messages with

```
    val default : [ `Debug | `NoDebug ]
```

Specifies whether debugging is enabled or disabled by default.

```
  end
```

Input information to make a DEBUG module

```
module NoDebug :
   DEBUG
```

To force debugging off, use the NoDebug module in place of a regluar DEBUG module

```
module Make :
   functor (Module :  DEBUG_MOD_INFO) -> DEBUG
```

Functor to create a DEBUG module. *

Sample code to create a DEBUG module for the "Foo" module that enables debugging by default:

```
module D = Debug.Make(struct let name = "Foo" and default = `Debug end)
open D
```

27 Module Debug_snippets : Snippets of debugging code

```
module D :
   Debug.Make(  sig
     val name : string
     val default : [> `Debug ]
  end )
```

AST debugging visitor

```
val v : Ssa_visitor.nop
val print_astcfg : Cfg.AST.G.t -> unit
val print_ast : Ast.program -> unit
```

SSA debugging visitor

```
val v : Ssa_visitor.nop
val print_ssacfg : Cfg.SSA.G.t -> unit
val print_ssa : Ssa.stmt list -> unit
val intv_to_string : Big_int_Z.big_int * Type.typ -> string
```

28 Module Depgraphs : Dependency graphs.

This module contains utilities for computing dependency graphs over graph representations. Dependency graphs can reflect data dependencies (DDG), control dependencies (CDG), or both (PDG). In addition, this module contains some support for usedef chains, and can identify undefined variables.

28.0.10 Control Dependence Graphs

```
module type CDG =
  sig
    module G :
      sig
        type t
        module V :
          sig
            type t
          end
      end
    val compute_cd : G.t -> G.V.t -> G.V.t list
```

This function computes control dependencies. This implements the algorithm in the Tiger Book p.454 (ML version) with the exception we do not add a new node before entry. Therefore all nodes are control dependent on BB_Entry.

Note that BB_Exit will not be control dependent on anything, thus a lone node in the graph (you can prune it away if you want using other utilities in BAP)

Returns a map from a node to its parents in the CDG tree.

```
val compute_cdg : G.t -> G.t
```

computes the control dependence graph (cdg), which turns the result of `compute_cd` below into a graph

```
end
```

Control dependence module type

```
module MakeRevCfg :
   functor (C : Cfg.CFG) ->   sig
     type t = C.G.t
     module V :
     Graph.Sig.VERTEX  with type label = C.G.V.label and type t = C.G.V.t
     val pred : t ->
       V.t -> V.t list
     val succ : t ->
       V.t -> V.t list
     val nb_vertex : t -> int
     val fold_vertex : (V.t -> 'a -> 'a) -> t -> 'a -> 'a
     val iter_vertex : (V.t -> unit) -> t -> unit
  end
```

Functor to reverse a CFG

```
module MakeCDG :
   functor (C : Cfg.CFG) -> CDG  with module G = C.G
```

Functor to produce control dependence analysis module for a CFG

```
module CDG_SSA :
   CDG  with module G = Cfg.SSA.G
```

Control dependence graphs for SSA CFGs

```
module CDG_AST :
   CDG  with module G = Cfg.AST.G
```

Control dependence graphs for AST CFGs

28.0.11 Data Dependence Graphs

```
module DDG_SSA :
  sig
```

```
    type location = Cfg.SSA.G.V.t * int
```

A statement location is identified by a basic block and the nth statement in the block.

```
    module SS :
    Set.S  with type elt = location
```

A set of locations

```
    val compute_dd :
      Cfg.SSA.G.t ->
      Var.VarSet.t * location Var.VarHash.t *
      location list Var.VarHash.t
```

`compute_dd cfg` returns the tuple `vars,fd,fu`. `vars` is the set of variables used by the graph. `fd` is a hashtbl from vars to their definition location. `fu` is a hashtbl from vars to their use locations. Unlike graphs such as DDG and PDG (below), we do not assume that vars are defined on entry and used on exit.

```
    val compute_dds : Cfg.SSA.G.t ->
      (location, location) Hashtbl.t -> unit
```

`compute_dds g h` adds a mapping for each data dependency in `g` to `h`.
Deprecated.

```
    val compute_ddg : Cfg.SSA.G.t -> Cfg.SSA.G.t
```

`compute_ddg g` returns a data dependency graph for the CFG `g`.

```
    val stmtlist_to_single_stmt : Cfg.SSA.G.t -> Cfg.SSA.G.t
```

`stmtlist_to_single_stmt g` returns a CFG equivalent to `g`, but that has one statement per vertex. This makes edges in a dependency graph more precise.

```
  end
```

Data dependence graphs for SSA CFGs

```
module DDG_AST :
  sig
```

```
    type location = Cfg.AST.G.V.t * int
```

A statement location is identified by a basic block and the nth statement in the block.

```
    module SS :
    Set.S  with type elt = location
```

A set of locations

```
val compute_dd :
  Cfg.AST.G.t ->
  Var.VarSet.t * location Var.VarHash.t *
  location list Var.VarHash.t
```

`compute_dd cfg` returns the tuple `vars,fd,fu`. `vars` is the set of variables used by the graph. `fd` is a hashtbl from vars to their definition location. `fu` is a hashtbl from vars to their use locations. Unlike graphs such as DDG and PDG (below), we do not assume that vars are defined on entry and used on exit.

```
val compute_dds : Cfg.AST.G.t ->
  (location, location) Hashtbl.t -> unit
```

`compute_dds g h` adds a mapping for each data dependency in `g` to `h`.
Deprecated.

```
val compute_ddg : Cfg.AST.G.t -> Cfg.AST.G.t
```

`compute_ddg g` returns a data dependency graph for the CFG `g`.

```
end
```

Data dependence graphs for AST CFGs

28.0.12 Program Dependence Graphs

Program dependence graphs are the union of the control and data dependence graphs.

```
module PDG_AST :
  sig
```

```
val compute_pdg : Cfg.AST.G.t -> Cfg.AST.G.t
```

`compute_pdg g` returns the program dependence graph for `g`.

```
  end
```

Program dependence graphs for AST CFGs

28.0.13 Use/Def and Def/Use Analyses

```
module UseDef_AST :
  sig
```

```
type location = Cfg.AST.G.V.t * int
```

A statement location is identified by a basic block and the nth statement in the block.

```
    module LocationType :
      sig
        type t =
          | Undefined
          | Loc of Depgraphs.UseDef_AST.location
        val compare : 'a -> 'a -> int
        val to_string : t -> string
      end
    module LS :
    Set.S  with type elt = LocationType.t
    val usedef :
      Cfg.AST.G.t ->
      (location, LS.t Var.VarMap.t)
      Hashtbl.t *
      (location -> Var.t -> LS.t)
```

Given an AST CFG, returns 1) a hash function mapping locations to the definitions reaching that location and 2) a function that returns the definitions for a (variable, location) pair

XXX: Aren't these almost the same thing?

```
    val defuse :
      Cfg.AST.G.t ->
      (location, LS.t) Hashtbl.t *
      (location -> LS.t)
```

Same as `usedef`, but for def use chains. That is, these functions map definitions to their possible uses.

```
  end
```

Use/Def and Def/Use chains on AST CFGs

```
module DEFS_AST :
  sig
    type location = Cfg.AST.G.V.t * int
```

A statement location is identified by a basic block and the nth statement in the block.

```
    val undefined :
      Cfg.AST.G.t ->
      location Var.VarHash.t *
      location Var.VarHash.t
```

Return variables that might be referenced before they are defined. The output of this function is a good starting point when trying to find inputs of a program. Returns a hash of values that are always undefined, and sometimes undefined (depending on program path).

```
val undefinedvars : Cfg.AST.G.t -> Var.VarSet.t * Var.VarSet.t
```

Returns a tuple with set of variables that are always undefined, and a set of variables that may be undefined.

```
val defs : Cfg.AST.G.t -> Var.VarSet.t * Var.VarSet.t
```

Returns a tuple with a set of variables that are defined, and a set of variables that are referenced but never defined.

Deprecated.

```
val vars : Cfg.AST.G.t -> Var.VarSet.t
```

Return a set of all variables defined or referenced.

```
end
```

Various functions relating to variable definitions in AST CFGs

```
module DEFS_SSA :
  sig
```

```
val defs : Cfg.SSA.G.t -> Var.VarSet.t * Var.VarSet.t
```

Returns a tuple with a set of variables that are defined, and a set of variables that are referenced but never defined.

Deprecated.

```
  end
```

Various functions relating to variable definitions in SSA CFGs

29 Module `Disasm` : General disassembly stuff

```
val arch_to_x86_mode : Arch.arch -> Disasm_i386.mode
val disasm_instr :
  Arch.arch ->
  (Big_int_Z.big_int -> char) ->
  Big_int_Z.big_int -> Ast.stmt list * Big_int_Z.big_int
val is_temp : Var.t -> bool
val is_decode_error : Ast.stmt -> bool
```

30 Module Disasm_i386 : Native lifter of x86 instructions to the BAP IL

```
module VH :
   Var.VarHash
module D :
   Debug.Make(  sig
     val name : string
     val default : [> `NoDebug ]
  end )
val compute_segment_bases : bool Pervasives.ref
exception Disasm_i386_exception of string
type binopf = Ast.exp -> Ast.exp -> Ast.exp
type mode =
  | X86
  | X8664
val type_of_mode : mode -> Type.typ
val width_of_mode : mode -> int
type order =
  | Low
  | High
type direction =
  | Forward
  | Backward
type operand =
  | Oreg of int
  | Ovec of int
  | Oseg of int
  | Oaddr of Ast.exp
  | Oimm of Big_int_Z.big_int
type jumptarget =
  | Jabs of operand
  | Jrel of Type.addr * Type.addr
module Pcmpstr :
  sig
     type ssize =
       | Bytes
       | Words
     val ssize_to_string : ssize -> string
```

```
type ssign =
  | Signed
  | Unsigned
val ssign_to_string : ssign -> string
type agg =
  | EqualAny
  | Ranges
  | EqualEach
  | EqualOrdered
val agg_to_string : agg -> string
type outselectsig =
  | LSB
  | MSB
val outselectsig_to_string : outselectsig -> string
type outselectmask =
  | Bitmask
  | Bytemask
val outselectmask_to_string : outselectmask -> string
val sig_to_mask : outselectsig -> outselectmask
type imm8cb = {
  ssize : ssize ;
  ssign : ssign ;
  agg : agg ;
  negintres1 : bool ;
  maskintres1 : bool ;
  outselectsig : outselectsig ;
  outselectmask : outselectmask ;
}
type out =
  | Index
  | Mask
val out_to_string : out -> string
type len =
  | Implicit
  | Explicit
val len_to_string : len -> string
type pcmpinfo = {
  out : out ;
  len : len ;
}
```

Information about the type of pcmp instruction.

```
  end

type offsetinfo = {
  offlen : Type.typ ;
  offtyp : Type.typ ;
  offop : operand ;
  offsrcoffset : int ;
  offdstoffset : int ;
}
type opcode =
  | Bswap of (Type.typ * operand)
  | Retn of (Type.typ * operand) option * bool
  | Nop
  | Mov of Type.typ * operand * operand * Ast.exp option
  | Movs of Type.typ
  | Movzx of Type.typ * operand * Type.typ * operand
  | Movsx of Type.typ * operand * Type.typ * operand
  | Movdq of Type.typ * operand * Type.typ * operand * bool
  | Movoffset of (Type.typ * operand) * offsetinfo list
  | Lea of Type.typ * operand * Ast.exp
  | Call of operand * Type.addr
  | Shift of Type.binop_type * Type.typ * operand * operand
  | Shiftd of Type.binop_type * Type.typ * operand * operand
   * operand
  | Rotate of Type.binop_type * Type.typ * operand * operand * bool
  | Bt of Type.typ * operand * operand
  | Bs of Type.typ * operand * operand * direction
  | Jump of jumptarget
  | Jcc of jumptarget * Ast.exp
  | Setcc of Type.typ * operand * Ast.exp
  | Hlt
  | Cmps of Type.typ
  | Scas of Type.typ
  | Stos of Type.typ
  | Push of Type.typ * operand
  | Pop of Type.typ * operand
  | Pushf of Type.typ
  | Popf of Type.typ
  | Popcnt of Type.typ * operand * operand
  | Sahf
  | Lahf
  | Add of (Type.typ * operand * operand)
  | Adc of (Type.typ * operand * operand)
  | Inc of Type.typ * operand
  | Dec of Type.typ * operand
  | Sub of (Type.typ * operand * operand)
```

```
 | Sbb of (Type.typ * operand * operand)
 | Cmp of (Type.typ * operand * operand)
 | Cmpxchg of (Type.typ * operand * operand)
 | Cmpxchg8b of operand
 | Xadd of (Type.typ * operand * operand)
 | Xchg of (Type.typ * operand * operand)
 | And of (Type.typ * operand * operand)
 | Or of (Type.typ * operand * operand)
 | Xor of (Type.typ * operand * operand)
 | Test of (Type.typ * operand * operand)
 | Ptest of (Type.typ * operand * operand)
 | Not of (Type.typ * operand)
 | Neg of (Type.typ * operand)
 | Mul of (Type.typ * operand)
 | Imul of Type.typ * (bool * operand) * operand
  * operand
 | Div of Type.typ * operand
 | Idiv of Type.typ * operand
 | Cld
 | Rdtsc
 | Cpuid
 | Xgetbv
 | Stmxcsr of operand
 | Ldmxcsr of operand
 | Fnstcw of operand
 | Fldcw of operand
 | Fld of operand
 | Fst of (operand * bool)
 | Punpck of (Type.typ * Type.typ * order * operand *
operand * operand option)
 | Ppackedbinop of (Type.typ * Type.typ * binopf * string * operand *
operand * operand option)
 | Pbinop of (Type.typ * binopf * string * operand *
operand * operand option)
 | Pmov of (Type.typ * Type.typ * Type.typ * operand * operand *
Type.cast_type * string)
 | Pmovmskb of (Type.typ * operand * operand)
 | Pcmp of (Type.typ * Type.typ * Type.binop_type * string * operand *
operand * operand option)
 | Palignr of (Type.typ * operand * operand *
operand option * operand)
 | Pcmpstr of (Type.typ * operand * operand * operand *
Pcmpstr.imm8cb * Pcmpstr.pcmpinfo)
 | Pshufb of Type.typ * operand * operand
  * operand option
```

```
  | Pshufd of Type.typ * operand * operand
   * operand option * operand
  | Leave of Type.typ
  | Interrupt of operand
  | Interrupt3
  | Sysenter
  | Syscall
val pref_lock : int
val repnz : int
val repz : int
val hint_bnt : int
val hint_bt : int
val pref_cs : int
val pref_ss : int
val pref_ds : int
val pref_es : int
val pref_fs : int
val pref_gs : int
val pref_opsize : int
val pref_addrsize : int
val standard_prefs : int list
type rex = {
  rex_w : bool ;
  rex_r : bool ;
  rex_x : bool ;
  rex_b : bool ;
}
type vex = {
  vex_nr : bool ;
  vex_nx : bool ;
  vex_nb : bool ;
  vex_map_select : int ;
  vex_we : bool ;
  vex_v : int ;
  vex_l : bool ;
  vex_pp : int ;
}
type prefix = {
  addrsize : Type.typ ;
  opsize : Type.typ ;
  bopsize : Type.typ ;
  mopsize : Type.typ ;
```

```
  repeat : bool ;
  nrepeat : bool ;
  addrsize_override : bool ;
  opsize_override : bool ;
  rex : rex option ;
  vex : vex option ;
  r_extend : int ;
  rm_extend : int ;
  sib_extend : int ;
}
val disfailwith : string -> 'a
```

disfailwith is a non-fatal disassembly exception.

```
val unimplemented : string -> 'a
val (&) : int -> int -> int
val (>>) : int -> int -> int
val (<<) : int -> int -> int
val ite : Type.typ -> Ast.exp -> Ast.exp -> Ast.exp -> Ast.exp
val r1 : Type.typ
val r4 : Type.typ
val r8 : Type.typ
val r16 : Type.typ
val r32 : Type.typ
val r64 : Type.typ
val r128 : Type.typ
val r256 : Type.typ
val xmm_t : Type.typ
val ymm_t : Type.typ
val st_t : Type.typ
val nv : string -> Type.typ -> Var.t
```

Only use this for registers, not temporaries

```
val nt : string -> Type.typ -> Var.t
type multimodereg = {
  v32 : Var.t ;
  v64 : Var.t ;
}
val nmv : string -> Type.typ -> string -> Type.typ -> multimodereg
val gv : mode -> multimodereg -> Var.t
val ge : mode -> multimodereg -> Ast.exp
val rbp : multimodereg
```

```
val rsp : multimodereg
val rsi : multimodereg
val rdi : multimodereg
val rip : multimodereg
val rax : multimodereg
val rbx : multimodereg
val rcx : multimodereg
val rdx : multimodereg
val rflags : multimodereg
val cf : Var.t
val pf : Var.t
val af : Var.t
val zf : Var.t
val sf : Var.t
val oF : Var.t
val df : Var.t
val fs_base : multimodereg
val gs_base : multimodereg
val cs : Var.t
val ds : Var.t
val es : Var.t
val fs : Var.t
val gs : Var.t
val ss : Var.t
val gdt : multimodereg
val ldt : multimodereg
val fpu_ctrl : Var.t
val mxcsr : Var.t
val nums : multimodereg array
val ymms : Ast.var array
val st : Ast.var array
val mvs : multimodereg -> Var.t list
val shared_regs : Ast.var list
val shared_multi_regs : multimodereg list
val regs_x86 : Ast.var list
val regs_x86_64 : Ast.var list
val regs_full : Ast.var list
val regs_of_mode : mode -> Ast.var list
```

```
val o_rax : operand
val o_rcx : operand
val o_rdx : operand
val o_rbx : operand
val o_rsp : operand
val o_rbp : operand
val o_rsi : operand
val o_rdi : operand
val o_es : operand
val o_cs : operand
val o_ss : operand
val o_ds : operand
val o_fs : operand
val o_gs : operand
val mem : multimodereg
module R32 :
  sig
     val eax : Var.t
     val ecx : Var.t
     val edx : Var.t
     val ebx : Var.t
     val esp : Var.t
     val ebp : Var.t
     val esi : Var.t
     val edi : Var.t
     val mem : Var.t
  end
module R64 :
  sig
     val rax : Var.t
     val rcx : Var.t
     val rdx : Var.t
     val rbx : Var.t
     val rsp : Var.t
     val rbp : Var.t
     val rsi : Var.t
     val rdi : Var.t
```

```
    val mem : Var.t
    val r8 : Var.t
    val r9 : Var.t
    val r10 : Var.t
    val r11 : Var.t
    val r12 : Var.t
    val r13 : Var.t
    val r14 : Var.t
    val r15 : Var.t
  end
val cf_e : Ast.exp
val pf_e : Ast.exp
val af_e : Ast.exp
val zf_e : Ast.exp
val sf_e : Ast.exp
val of_e : Ast.exp
val df_e : Ast.exp
val seg_cs : 'a option
val seg_ss : 'a option
val seg_ds : 'a option
val seg_es : 'a option
val seg_fs : multimodereg option
val seg_gs : multimodereg option
val df_to_offset : mode -> Ast.exp -> Ast.exp
val bap_to_eflags : Ast.exp list
val bap_to_flags : Ast.exp list
val bap_to_lflags : Ast.exp list
val eflags_e : Ast.exp
val flags_e : Ast.exp
val lflags_e : Ast.exp
val eflags_to_bap : (Var.t * (Ast.exp -> Ast.exp)) option list
val flags_to_bap : (Var.t * (Ast.exp -> Ast.exp)) option list
val lflags_to_bap : (Var.t * (Ast.exp -> Ast.exp)) option list
val assns_eflags_to_bap : (Ast.exp -> Ast.stmt list) list
val assns_flags_to_bap : (Ast.exp -> Ast.stmt list) list
val assns_lflags_to_bap : (Ast.exp -> Ast.stmt list) list
val load_s : mode -> Ast.var option -> Type.typ -> Ast.exp -> Ast.exp
```

```
val lt : int64 -> Type.typ -> Ast.exp
val l64 : int64 -> Ast.exp
val l32 : int64 -> Ast.exp
val l16 : int64 -> Ast.exp
val int64_of_mode : mode -> int64 -> Ast.exp
val it : int -> Type.typ -> Ast.exp
val i64 : int -> Ast.exp
val i32 : int -> Ast.exp
val int_of_mode : mode -> int -> Ast.exp
val bt : Z.t -> Type.typ -> Ast.exp
val b64 : Z.t -> Ast.exp
val b32 : Z.t -> Ast.exp
val b16 : Z.t -> Ast.exp
val big_int_of_mode : mode -> Z.t -> Ast.exp
val lowbits2elemt : int -> Type.typ
val bits2genreg : int -> multimodereg
val reg2bits : multimodereg -> int
val bits2segreg : int -> Var.t
val bits2segrege : int -> Ast.exp
val bits2ymm : int -> Ast.var
val bits2ymme : int -> Ast.exp
val bits2ymm128e : int -> Ast.exp
val bits2ymm64e : int -> Ast.exp
val bits2ymm32e : int -> Ast.exp
val bits2xmm : int -> Ast.exp
val bits2xmm64e : int -> Ast.exp
val bits2xmm32e : int -> Ast.exp
val ymm0 : Ast.var
val bits2reg64e : mode -> int -> Ast.exp
val bits2reg32e : mode -> int -> Ast.exp
val bits2reg16e : mode -> int -> Ast.exp
val bits2reg8e : mode -> ?has_rex:bool -> int -> Ast.exp
val reg2xmm : multimodereg -> Ast.exp
val eaddr16 : mode -> int -> Ast.exp
val eaddr16e : mode -> int -> Ast.exp
val ah_e : mode -> Ast.exp
val ch_e : mode -> Ast.exp
val dh_e : mode -> Ast.exp
```

```
val bh_e : mode -> Ast.exp
module ToIR :
  sig
    val move : Ast.var -> Ast.exp -> Ast.stmt
    val store_s :
      Disasm_i386.mode ->
      Ast.var option -> Type.typ -> Ast.exp -> Ast.exp -> Ast.stmt
    val storem : Ast.var -> Type.typ -> Ast.exp -> Ast.exp -> Ast.stmt
    val op2e_s :
      Disasm_i386.mode ->
      Ast.var option -> bool -> Type.typ -> Disasm_i386.operand -> Ast.exp
    val assn_s :
      Disasm_i386.mode ->
      Ast.var option ->
      bool -> bool -> Type.typ -> Disasm_i386.operand -> Ast.exp -> Ast.stmt
    val op_dbl : Type.typ -> (Type.typ * Disasm_i386.operand) list
    val op2e_dbl_s :
      Disasm_i386.mode -> Ast.var option -> bool -> Type.typ -> Ast.exp
    val assn_dbl_s :
      Disasm_i386.mode ->
      Ast.var option ->
      bool -> bool -> Type.typ -> Ast.exp -> Ast.stmt list * Ast.exp
    val compute_jump_target :
      Disasm_i386.mode ->
      Ast.var option -> bool -> Disasm_i386.jumptarget -> Ast.exp
    val jump_target :
      Disasm_i386.mode ->
      Ast.var option -> bool -> Disasm_i386.jumptarget -> Ast.exp
    val bytes_of_width : Type.typ -> int
    val bits_of_width : Type.typ -> int
    val string_incr : Disasm_i386.mode -> Type.typ -> Ast.var -> Ast.stmt
    val rep_wrap :
      ?check_zf:int ->
      mode:Disasm_i386.mode ->
      addr:Z.t -> next:Z.t -> Ast.stmt list -> Ast.stmt list
    val reta : Type.attribute list
    val calla : Type.attribute list
    val compute_sf : Ast.exp -> Ast.exp
    val compute_zf : Type.typ -> Ast.exp -> Ast.exp
    val compute_pf : Type.typ -> Ast.exp -> Ast.exp
    val set_sf : Ast.exp -> Ast.stmt
```

```
    val set_zf : Type.typ -> Ast.exp -> Ast.stmt
    val set_pf : Type.typ -> Ast.exp -> Ast.stmt
    val set_pszf : Type.typ -> Ast.exp -> Ast.stmt list
    val set_apszf : Type.typ -> Ast.exp -> Ast.exp -> Ast.exp -> Ast.stmt list
    val set_aopszf_add :
      Type.typ -> Ast.exp -> Ast.exp -> Ast.exp -> Ast.stmt list
    val set_flags_add :
      Type.typ -> Ast.exp -> Ast.exp -> Ast.exp -> Ast.stmt list
    val set_apszf_sub :
      Type.typ -> Ast.exp -> Ast.exp -> Ast.exp -> Ast.stmt list
    val set_aopszf_sub :
      Type.typ -> Ast.exp -> Ast.exp -> Ast.exp -> Ast.stmt list
    val set_flags_sub :
      Type.typ -> Ast.exp -> Ast.exp -> Ast.exp -> Ast.stmt list
    val to_ir :
      Disasm_i386.mode ->
      Z.t ->
      Z.t ->
      Ast.var option ->
      int list -> bool -> bool -> Disasm_i386.opcode -> Ast.stmt list
    val add_labels : ?asm:string -> Type.addr -> Ast.stmt list -> Ast.stmt list
  end
val cc_to_exp : int -> Ast.exp
val parse_instr :
  mode ->
  (Big_int_Z.big_int -> char) ->
  Big_int_Z.big_int ->
  int list * prefix * opcode * Big_int_Z.big_int
val parse_prefixes : mode -> int list -> 'a -> Var.t option * int list
val disasm_instr :
  mode ->
  (Big_int_Z.big_int -> char) ->
  Big_int_Z.big_int -> Ast.stmt list * Big_int_Z.big_int
```

31 Module Djgraph

```
module type G =
  sig
    include Dominator.G
    val iter_edges : (V.t -> V.t -> unit) -> t -> unit
```

```
  end

type edge_type =
  | D
  | BJ
  | CJ
module type MakeType =
   functor (Gr :  G) ->   sig

     include Graph.Sig.I
     val dj_graph : Gr.t -> Gr.V.t -> t
     val to_underlying : V.t -> Gr.V.t
     val level : V.t -> int

  end

module Make :
   MakeType
```

32 Module `Dominator` : Dominator module for use with the ocamlgraph library.

All of the functions in this module assume that the graph is not modified between calling one of these functions and using the returned functions. Such mutation results in undefined behavior.

Author(s): Ivan Jager

```
val (|>) : 'a -> ('a -> 'b) -> 'b
exception Unreachable
module type G =
  sig

     type t
     module V :
     Graph.Sig.COMPARABLE
     val pred : t -> V.t -> V.t list
     val succ : t -> V.t -> V.t list
     val fold_vertex : (V.t -> 'a -> 'a) -> t -> 'a -> 'a
     val iter_vertex : (V.t -> unit) -> t -> unit
     val nb_vertex : t -> int

  end

module Make :
   functor (G : G) ->   sig
```

```
module H :
Hashtbl.Make(Dominator.G.V)
module S :
Set.Make(Dominator.G.V)
type idom = G.V.t -> G.V.t
```

function from `n` to `n`'s immediate dominator

```
type idoms = G.V.t -> G.V.t -> bool
```

`idoms x y` is true when `x` is `y`'s immediate dominator

```
type dom_tree = G.V.t -> G.V.t list
```

function from `x` to a list of nodes immediately dominated by `x`

```
type dominators = G.V.t -> G.V.t list
```

function from node to a list of nodes that dominate it. the list is sorted by depth in the dominator tree.

```
type dominees = G.V.t -> G.V.t list
```

function from `x` to a list of nodes that are dominated by `x`.

```
type dom = G.V.t -> G.V.t -> bool
```

`dom x y` returns true iff `x` dominates `y`

```
type sdom = G.V.t -> G.V.t -> bool
```

`sdom x y` returns true iff `x` strictly dominates `y`.

```
type dom_frontier = G.V.t -> G.V.t list
```

function from `x` to a list of nodes not dominated by `x`, but with predecessors which are dominated by `x`

```
type dom_functions = {
  idom : idom ;
  idoms : idoms ;
  dom_tree : dom_tree ;
  dominators : dominators ;
  dominees : dominees ;
  dom : dom ;
  sdom : sdom ;
  dom_frontier : dom_frontier ;
}
val set_of_list : S.elt list -> S.t
```

```
val pseudo_topological_fold :
  (H.key -> 'a -> 'a) ->
  'a ->
  H.key list ->
  (H.key -> H.key list) -> 'a
```

Fold over the nodes. Function `f` is applied in reverse-topologicalish order.

```
val pseudo_topological_sort :
  H.key list ->
  (H.key -> H.key list) ->
  H.key list
```

Given the entry nodes into a graph, and a successor function, returns the nodes in pseudo topological order.

```
val compute_idom : Dominator.G.t ->
  H.key -> H.key -> H.key
```

Computes the dominator tree, using the Lengauer-Tarjan algorithm. `compute_idom cfg s0` returns a function `idom : V.t -> V.t` s.t. `idom x` returns the immediate dominator of `x`

```
val dominators_to_dom : ('a -> S.t) -> S.elt -> 'a -> bool
```

Given a function from a node to it's dominators, returns a function `dom : V.t -> V.t -> bool` s.t. `dom x y` returns true when `x` dominates `y`

```
val dominators_to_sdom : (G.V.t -> S.t) -> S.elt -> G.V.t -> bool
```

Given a function from a node to it's dominators, returns a function `sdom : V.t -> V.t -> bool` s.t. `sdom x y` returns true when `x` strictly dominates `y`

```
val dom_to_sdom : (G.V.t -> G.V.t -> bool) -> G.V.t -> G.V.t -> bool
val dominators_to_sdominators : (S.elt -> S.t) ->
  S.elt -> S.t
```

Given a a function from a node to it's dominators, returns a function from a node to it's strict dominators.

```
val dominators_to_idoms : (S.elt -> S.t) ->
  S.elt -> S.elt -> bool
```

Given a function from a node to it's dominators, returns a function `idoms : G.V.t -> G.V.t -> bool` s.t. `idoms x y` returns true when `x` is the immediate dominator of `y`.

```
val dominators_to_dom_tree :
  Dominator.G.t ->
  ?pred:(Dominator.G.t -> S.elt -> S.elt list) ->
  (S.elt -> S.t) ->
  H.key -> S.t
```

Computes a dominator tree (function from x to a list of nodes immediately dominated by x) for the given CFG and dominator function. Note: The dominator tree is also called `IDom` by Muchnick. Note: If you are computing a post-dominator tree, then the optional argument pred should be G.succ.

```
val dom_tree_to_dominees : (S.elt -> S.elt list) ->
  H.key -> S.elt list
```

Computes the transitive closure of a dominator tree.

```
val idom_to_dom_tree :
  Dominator.G.t ->
  (G.V.t -> H.key) -> H.key -> G.V.t list
```

Computes a dominator tree (function from x to a list of nodes immediately dominated by x) for the given CFG and idom function.

```
val idom_to_idoms : idom -> G.V.t -> G.V.t -> bool
val compute_dom_frontier :
  Dominator.G.t ->
  dom_tree ->
  idom -> H.key -> G.V.t list
```

Computes the dominance frontier. As specified in section 19.1 of Modern Compiler Implementation in ML by Andrew Appel.

```
val idom_to_dominators : ('a -> 'a) -> 'a -> 'a list
val idom_to_dom : (G.V.t -> G.V.t) -> G.V.t -> G.V.t -> bool
val compute_all : Dominator.G.t -> H.key -> dom_functions
```

Computes all dominance functions.

This function computes some things eagerly and some lazily, so don't worry about it doing extra work to compute functions you don't need, but also don't call it if you aren't going to use anything it returns.

Returns a record containing all dominance functions for the given graph and entry node.

```
end
```

33 Module `Flatten_mem` : Break complicated memory write statements a series of flat ones of form `Store(Var v, ...)`.

This makes it easier to execute the memory operations sequentially.

Author(s): Ed Schwartz

```
val flatten_memexp :
  Ast.var -> Ast.attrs -> Ast.exp -> Ast.exp * Ast.stmt list
```

`flatten_memexp memvl atts e` returns a tuple `(flate, stmts)` where `flate` contains no nested Stores, and is equivalent to `e`, provided `stmts` are executed immediately before evaluating `flate`.

`val flatten_stores : Ast.stmt -> Ast.stmt list`

Converts a nested memory assignment to multiple flat (non-nested) assignments. Non-memory write statements are not changed. `Let` expressions that evaluate to a memory are removed using substitution, although this may change.

For example, `Move(memv, Store(Store(memv, idx1, value1), idx2, value2))` would be converted to `Move(memv, Store(Var memv, idx1, value1)) :: Move(memv, Store(Var memv, idx2, value2)) :: []`.

`val flatten_loads : Ast.exp -> Ast.exp`

Converts a memory load expression `e` into a form that can be sequentially evaluated concretely. In particular, this means that there are no `Let` bindings that bind a memory state to a variable, because that cannot be implemented concretely.

`val flatten_mem_program : Ast.program -> Ast.program`

Flattens all memory loads and stores in a program.

34 Module `Formulap` : Printing formulas

```
module VH :
   Var.VarHash
module D :
   Debug.Make(  sig
     val name : string
     val default : [> `NoDebug ]
  end )
```

`val freevars : Ast.exp -> VH.key list`

Returns a list of free variables in the given expression

```
class virtual fpp :
  object
     method virtual forall : Formulap.VH.key list -> unit
     method virtual ast_exp : Ast.exp -> unit
     method virtual assert_ast_exp :
       ?exists:Ast.var list -> ?foralls:Ast.var list -> Ast.exp -> unit
     method virtual valid_ast_exp :
       ?exists:Ast.var list -> ?foralls:Ast.var list -> Ast.exp -> unit
     method virtual counterexample : unit
```

```
  end
class virtual fpp_oc :
  object
     inherit Formulap.fpp [34]
     method virtual close : unit
     method virtual flush : unit
  end
type fppf = Pervasives.out_channel -> fpp_oc
class virtual stream_fpp :
  object
     method virtual and_start : unit
```

Begin a list of constraints at the start of the formula

```
     method virtual and_constraint : Ast.exp -> unit
```

Add a constraint to a list of constraints

```
     method virtual and_close_constraint : unit
```

Close constraint at end of the formula
E.g., print ')'

```
     method virtual and_end : unit
```

End a list of constraints at the end of the formula
E.g., print 'true'

```
     method virtual let_begin : Ast.var -> Ast.exp -> unit
```

Begin a let binding

```
     method virtual let_end : Ast.var -> unit
```

End a let binding

```
     method virtual open_stream_benchmark : unit
```

Open a new benchmark for a streaming formula, which is assumed to use the theory of bitvectors and arrays

```
     method virtual close_benchmark : unit
```

Close the benchmark

```
     method virtual counterexample : unit
```

Request a counter-example

```
method virtual predeclare_free_var : Ast.var -> unit
```

Declaring a variable consists of calling predeclare_free_var to register the name and type, and then calling print_free_var.

```
method virtual print_free_var : Ast.var -> unit
method virtual assert_ast_exp_begin :
  ?exists:Ast.var list -> ?foralls:Ast.var list -> unit -> unit
method virtual assert_ast_exp_end : unit
method virtual valid_ast_exp_begin :
  ?exists:Ast.var list -> ?foralls:Ast.var list -> unit -> unit
method virtual valid_ast_exp_end : unit
end

class virtual stream_fpp_oc :
  object
```

`inherit Formulap.stream_fpp` [34]

```
method virtual close : unit
method virtual flush : unit
end

type stream_fppf = Pervasives.out_channel -> stream_fpp_oc
```

35 Module `Func_boundary` : Function boundary identification for x86

Author(s): Tiffany (Youzhi) Bao

```
val start_addresses : Asmir.asmprogram -> Type.addr list
```

`start_addresses p` identifies a list of function start addresses in `p` using heuristics. Raises `Invalid_argument` if called on a non-x86 program.

```
val end_address_at : Cfg.AST.G.t -> Type.addr
```

`end_address_at p addr scheme` returns the identified end address of the function in `cfg`.

```
val get_function_ranges :
  Asmir.asmprogram -> (string * Type.addr * Type.addr) list
```

`get_function_ranges p` finds functions using the symbol table, and if that fails, uses `start_addresses` to identify funtions.

36 Module Fwp : Forward weakest preconditions.

Author(s): ejs

```
val fwp :
  ?normalusage:bool ->
  ?simp:(Ast.exp -> Ast.exp) ->
  ?k:int -> Type.formula_mode -> Gcl.t -> Ast.exp -> Ast.exp
```

`fwp mode p q` is the same as `dwp`, but uses an alternate formulation of dwp. It is arguably easier to understand, and generates smaller formulas for programs that do not have `Assume` statements.

```
val fwp_uwp :
  ?simp:(Ast.exp -> Ast.exp) ->
  ?k:int -> Type.formula_mode -> Gcl.Ugcl.t -> Ast.exp -> Ast.exp
```

`fwp_uwp` is a version of `fwp` that operates directly on a CFG.

```
val fwp_lazyconc :
  ?simp:(Ast.exp -> Ast.exp) ->
  ?k:int -> ?cf:bool -> Type.formula_mode -> Gcl.t -> Ast.exp -> Ast.exp
```

`fwp_lazyconc` is like `Fwp.fwp`[36] but with concrete evaluation and lazy merging turned on.

```
val fwp_lazyconc_uwp :
  ?simp:(Ast.exp -> Ast.exp) ->
  ?k:int -> ?cf:bool -> Type.formula_mode -> Gcl.Ugcl.t -> Ast.exp -> Ast.exp
```

`fwp_lazyconc_uwp` is a version of `fwp_lazyconc` that operates directly on a CFG.

37 Module Gcl : Dijkstra's Guarded Command Language

Type declarations for the Guarded Command Language, and functions to traslate BAP programs to GCL.

BAP programs are converted to GCL form to compute the weakest precondition, since weakest preconditions are typically defined for GCL programs.

Author(s): : Ivan Jager

Type of GCL statements

```
type t =
  | Assume of Ast.exp
  | Assign of Var.t * Ast.exp
  | Assert of Ast.exp
  | Choice of t * t
  | Seq of t * t
  | Skip
```

A GCL statement. `Skip` does nothing. `Assign(v,e)` assigns an expression `e` to an lvalue `v`. `Seq(a,b)` evaluates `a` and then moves on to `b`. `Choice(a,b)` non-deterministically chooses to execute `a` or `b`. `Assert e` terminates in failure if `e` is untrue, and otherwise does nothing. `Assume e` does not start the program if `e` is untrue, and otherwise does nothing.

```
val to_string : t -> string
```

`to_string p` converts the GCL program `p` to a string for debugging purposes.

```
val size : t -> int
```

`size p` computes the number of statements in `p`.

Functions to convert BAP programs to GCL

```
val of_astcfg :
  ?entry:Cfg.AST.G.V.t -> ?exit:Cfg.AST.G.V.t -> Cfg.AST.G.t -> t
```

`of_astcfg cfg` converts the AST CFG `cfg` to GCL form.

The conversion is non-trivial, since GCL is a structured language, but AST CFGs are not. The current implementation proceeds by choosing a node `n` in reverse topological order from `cfg`. The interesting case is when `n` has two successors `x` and `y`. The current implmentation gets the GCL for `x` and `y` and finds the longest suffix shared between them and the unique parts of `x` and `y` not shared in the suffix. The two GCLs are then merged as `Seq(Choice(Seq(Assume(cond), Unique_x), Seq(Assume(not cond), Unique_y)), Common_suffix)`. This is not guaranteed to return the smallest GCL, but is simple and works well in practice.

Raises `Not_found` if `cfg` contains cycles.

```
val of_ast : Ast.program -> t
```

`of_ast` is the same as `Gcl.of_astcfg`[37], except that it converts an AST program instead of an AST CFG.

```
val passified_of_ssa :
  ?entry:Cfg.SSA.G.V.t ->
  ?exit:Cfg.SSA.G.V.t ->
  ?mode:Type.formula_mode -> Cfg.SSA.G.t -> t * Ast.var list
```

`passified_of_ssa cfg` converts a SSA CFG `cfg` to a passified GCL program. Passified GCL programs do not contain `Assign(v,e)` statements. Instead, all assignments `Assign(v,e)` are replaced with `Assert(v == e)` or `Assume(v == e)` statements, depending on whether the formula will be evaluated for satisfiability or validity.

Passification is used by the efficient weakest precondition algorithms in BAP, inlcuding DWP (`Wp.dwp`[95]) and Flanagan and Saxe's algorithm (`Wp.flanagansaxe`[95]).

Raises `Not_found` if `cfg` contains cycles.

```
val passified_of_astcfg :
  ?entry:Cfg.AST.G.V.t ->
```

```
  ?exit:Cfg.AST.G.V.t ->
  ?mode:Type.formula_mode ->
  Cfg.AST.G.t -> t * Ast.var list * (Var.t -> Var.t)
```

passified_of_astcfg is the same as Gcl.passified_of_ssa[37], except that it takes an ASG CFG as input.

The gclhelp program representation

```
type gclhelp =
  | CAssign of Cfg.AST.G.V.t
  | CChoice of Ast.exp * gclhelp * gclhelp
  | Cunchoice of gclhelp * gclhelp
  | CSeq of gclhelp list
```

Intermediate program representation that is somewhere between CFG and GCL, which can be useful for converting to non-GCL structured languages.

```
val gclhelp_of_astcfg :
  ?entry:Cfg.AST.G.V.t -> ?exit:Cfg.AST.G.V.t -> Cfg.AST.G.t -> gclhelp
```

gclhelp_of_astcfg cfg converts cfg to the gclhelp representation.

```
val gclhelp_to_string : gclhelp -> string
```

gclhelp_to_string gclh converts the gclhelp representation gclh to a string

37.0.14 Unstructured GCL

```
module Ugcl :
  sig
```

```
    type stmt = Gcl.t
```

Ugcl statements are just Gcl statements. However, the Choice constructor is not used in Ugcl statements.

```
    type t = Cfg.AST.G.t * (Cfg.AST.G.V.label -> stmt)
```

Ugcl programs are a CFG, annotated with a map from each basic block to a Ugcl statement.

```
    val of_ssacfg :
      ?entry:Cfg.AST.G.V.t ->
      ?exit:Cfg.AST.G.V.t -> ?mode:Type.formula_mode -> Cfg.SSA.G.t -> t
```

Convert a SSACFG to Ugcl

```
  end
```

GCL over unstructured programs (CFGs).

38 Module Grammar_private_scope : Define a Scope solely for the Parser and its helper functions.

```
val default_scope : unit -> (string, Ast.var) Hashtbl.t * 'a Stack.t
val cur_scope : Grammar_scope.Scope.t Pervasives.ref
val get_scope : unit -> Grammar_scope.Scope.t
val set_scope : Grammar_scope.Scope.t -> unit
val reset_scope : unit -> unit
```

39 Module Grammar_scope : Scope module for parsing.

ejs: I moved this out of the grammar.mly file so that external users could access it.

```
val strip_nums : bool Pervasives.ref
```

Whether or not to strip the trailing _number from variable names

```
val stripnum : string -> string
val err : string -> 'a
module Scope :
  sig
    type t = (string, Var.t) Hashtbl.t * string Stack.t
    val create : Var.t list -> (string, Var.t) Hashtbl.t * 'a Stack.t
    val empty_scope : unit -> t
    val add_var : ('a, 'b) Hashtbl.t * 'c -> 'a -> 'b -> 'b
    val add : (string, Var.t) Hashtbl.t * 'a -> string -> Type.typ -> Var.t
    val add_push :
      (string, Var.t) Hashtbl.t * string Stack.t -> string -> Type.typ -> Var.t
    val pop : ('a, 'b) Hashtbl.t * 'a Stack.t -> unit
    val get_lval_if_defined :
      (string, Var.t) Hashtbl.t * 'a -> string -> Type.typ option -> Var.t
```

Gets lval if defined, otherwise raises Not_found

```
    val get_lval :
      (string, Var.t) Hashtbl.t * 'a -> string -> Type.typ option -> Var.t
  end
val empty_scope : unit -> Scope.t
val create_scope_from_decls :
  Var.t list -> (string, Var.t) Hashtbl.t * 'a Stack.t
```

40 Module GraphDataflow : Dataflow module for use with the ocamlgraph library.

You should not need to use this module directly in BAP. Instead, use CfgDataflow[18].

Author(s): Ivan Jager

```
module type G =
  sig
```

```
    type t
```

```
    module V :
```

```
    Graph.Sig.COMPARABLE
```

```
    module E :
```

```
    Graph.Sig.EDGE  with type vertex = V.t
```

```
    val pred_e : t -> V.t -> E.t list
```

```
    val succ_e : t -> V.t -> E.t list
```

```
    val fold_vertex : (V.t -> 'a -> 'a) -> t -> 'a -> 'a
```

```
  end
```

Types of graph that data flow is defined on

```
type direction =
  | Forward
  | Backward
```

Data flow direction. `Forward` dataflow propagates values in the same direction as edges in the control flow graph, while `Backward` dataflow propagates values in the reverse direction.

```
module type BOUNDED_MEET_SEMILATTICE =
  sig
```

```
    type t
```

The type of a latice element

```
    val top : t
```

Top of the latice (the bound)

```
    val meet : t ->
      t ->
      t
```

The meet operator. `meet v1 v2` should form a lattice. In particular, `meet v1 Top = meet Top v1 = v1`, and `meet Bottom _ = meet _ Bottom = Bottom`.

```
    val equal : t ->
      t -> bool
```

Equality checking for latice values. Returns true when the two latice elements are the same.

```
  end
```

The lattice the dataflow is defined on. See here[http://en.wikipedia.org/wiki/Meet-semilattice] for more information.

```
module type BOUNDED_MEET_SEMILATTICE_WITH_WIDENING =
  sig
```

```
    include GraphDataflow.BOUNDED_MEET_SEMILATTICE
```

```
    val widen : t -> t -> t
```

The widening operator W.

I'm not going to repeat the formal definition here, since it is intended for infinite chains, and we often have finite (but long chains), which are just as bad.

Suffice to say if you have some long finite chain over the lattice values, applying the widening operator to the chain should cause it to stablize quickly.

If `A < B`, then `widen B A` should probably equal the bottom element of the lattice. If `A <= B`, then `widen A B = A`.

```
  end
```

The lattice the dataflow is defined on, with a widening operator.

```
module type OPTIONS =
  sig
```

```
    type t
```

Type of extra information that can be specified each time the dataflow analysis is applied.

```
    val default : t
```

Default options when none are specified

```
  end
```

Run-time options for the data-flow analysis

```
module NOOPTIONS :
   OPTIONS  with type t = unit
```

A No-op options module

```
module type DATAFLOW =
  sig
```

```
    module L :
    GraphDataflow.BOUNDED_MEET_SEMILATTICE
    module G :
    GraphDataflow.G
    module O :
    GraphDataflow.OPTIONS
    val node_transfer_function : O.t -> GraphDataflow.G.t -> G.V.t -> L.t -> L.t
```

The transfer function over node elements, e.g., basic blocks.

```
    val edge_transfer_function :
      O.t -> GraphDataflow.G.t -> GraphDataflow.G.E.t -> L.t -> L.t
```

The transfer function over edge elements, e.g., conditions.

```
    val s0 : O.t -> GraphDataflow.G.t -> G.V.t
```

The starting node for the analysis.

```
    val init : O.t -> GraphDataflow.G.t -> L.t
```

The initial lattice value given to node `s0`. All other nodes start out with `Top`.

```
    val dir : O.t -> GraphDataflow.direction
```

The dataflow direction.

```
  end
```

A dataflow problem is defined by a lattice over a graph.

```
module type DATAFLOW_WITH_WIDENING =
  sig
    module L :
    GraphDataflow.BOUNDED_MEET_SEMILATTICE_WITH_WIDENING
    module G :
    GraphDataflow.G
    module O :
    GraphDataflow.OPTIONS
    val node_transfer_function : O.t -> GraphDataflow.G.t -> G.V.t -> L.t -> L.t
```

The transfer function over node elements, e.g., basic blocks.

```
    val edge_transfer_function :
      O.t -> GraphDataflow.G.t -> GraphDataflow.G.E.t -> L.t -> L.t
```

The transfer function over edge elements, e.g., conditions.

```
val s0 : O.t -> GraphDataflow.G.t -> G.V.t
```

The starting node for the analysis.

```
val init : O.t -> GraphDataflow.G.t -> L.t
```

The initial lattice value given to node `s0`. All other nodes start out with `Top`.

```
val dir : O.t -> GraphDataflow.direction
```

The dataflow direction.

```
end
```

A dataflow problem with widening.

```
module Make :
   functor (D : DATAFLOW) ->   sig
```

```
val worklist_iterate :
  ?init:(D.O.t -> D.G.t -> D.L.t) ->
  ?opts:D.O.t -> D.G.t -> (D.G.V.t -> D.L.t) * (D.G.V.t -> D.L.t)
```

`worklist_iterate g` returns a worklist algorithm for graph `g` as a pair of functions `in,out`. `in`, when given a node `v`, computes the lattice value going in to that node, `v`. `out`, when given a node `v`, computes the lattice value exiting `v`.

```
end
```

Build a custom dataflow algorithm for the given dataflow problem `D`.

```
module MakeWide :
   functor (D : DATAFLOW_WITH_WIDENING) ->   sig
```

```
val worklist_iterate_widen :
  ?init:(D.O.t -> D.G.t -> D.L.t) ->
  ?nmeets:int ->
  ?opts:D.O.t -> D.G.t -> (D.G.V.t -> D.L.t) * (D.G.V.t -> D.L.t)
```

Same as `worklist_iterate`, but additionally employs the widening operator as lattice values propagate over backedges in the CFG. Backedges are identified by observing when lattices values flow in cycles.

```
end
```

Build a custom dataflow algorithm for the given dataflow problem with widening operator `D`.

41 Module Hacks : Hacks

```
module C :
   Cfg.AST
module D :
   Debug.Make(  sig
     val name : string
     val default : [> `NoDebug ]
  end )
val ra_final : Var.t
val ra0 : Var.t
val function_end : string
val attrs : Type.attribute list
val save_ra0 : Ast.stmt
val ret_to_jmp : ?ra:Ast.var -> Ast.stmt list -> Ast.stmt list
val attrs : Type.attribute list
val assert_noof : Ast.stmt list -> Ast.stmt list
type color =
  | White
  | Gray
  | Black
val remove_cycles : Cfg.AST.G.t -> Cfg.AST.G.t
val repair_node : Cfg.AST.G.t -> C.G.vertex -> C.G.t
```

Fix outgoing edges of `n` in `g`

```
val repair_cfg : Cfg.AST.G.t -> Reachable.AST.gt
```

Repair cfg whose graph is inconsistent with its statements

```
val uniqueify_labels : Ast.program -> Ast.program
```

Rename labels so they are always unique. This is useful for traces.

```
module Rm :
   functor (C : Cfg.CFG) ->   sig
     val remove_error_if_disconnected : Hacks.C.G.t -> Hacks.C.G.t
     val remove_indirect : Hacks.C.G.t -> Hacks.C.G.t
     val exit_indirect : Hacks.C.G.t -> Hacks.C.G.t
  end
val ast_remove_indirect : Cfg.AST.G.t -> Cfg.AST.G.t
```

```
val ast_exit_indirect : Cfg.AST.G.t -> Cfg.AST.G.t
val ssa_remove_indirect : Cfg.SSA.G.t -> Cfg.SSA.G.t
val ssa_exit_indirect : Cfg.SSA.G.t -> Cfg.SSA.G.t
val replace_unknowns : Ast.program -> Ast.program
```
Replace unknown expressions with constant zero

```
val append_file : string -> string -> unit
```
Append src to dst

```
val bberror_assume_false : Cfg.AST.G.t -> C.G.t
```
Add an "assume false" statement to BB_Error and add an edge to BB_Exit.

Useful for testing validity with a bounded number of loops.

```
val add_sink_exit : Cfg.AST.G.t -> C.G.t
```
Add edges from all sinks (other than BB_Exit) to BB_Exit

```
val ast_replacer :
  ?eq:(Ast.exp -> Ast.exp -> bool) ->
  needle:Ast.exp -> haystack:Ast.exp -> replacement:Ast.exp -> Ast.exp
```
Replace occurences of needle with replacement in haystack

```
val ssa_replacer :
  ?eq:(Ssa.exp -> Ssa.exp -> bool) ->
  needle:Ssa.exp -> haystack:Ssa.exp -> replacement:Ssa.exp -> Ssa.exp
```
Replace occurences of needle with replacement in haystack

```
val filter_specials : Ast.stmt list -> Ast.stmt list
val filter_calls_cfg : Cfg.AST.G.t -> Cfg.AST.G.t
```

42 Module Input : Use this to read in a program.

TODO: Add convenience functions to get SSA directly, and maybe more input options.

```
val speclist : (Arg.key * Arg.spec * Arg.doc) list
```
A speclist suitable to pass to Arg.parse. Add this to the speclist for your program.

```
val stream_speclist : (Arg.key * Arg.spec * Arg.doc) list
```
A speclist with only streaming inputs

```
val trace_speclist : (Arg.key * Arg.spec * Arg.doc) list
```
A speclist with only trace inputs

```
val get_program :
  unit -> Ast.program * Grammar_scope.Scope.t * Arch.arch option
```

Get the program as specified by the commandline.

```
val get_stream_program : unit -> Ast.program Stream.t * Arch.arch option
val init_ro : bool Pervasives.ref
val streamrate : int64 Pervasives.ref
val get_arch : Arch.arch option -> Arch.arch
```

`get_arch (Some x)` returns `x`, and `get_arch None` raises an informational exception.

43 Module `Llvm_codegen` : LLVM code generation backend for BAP IL programs.

Convert BAP programs and expressions to LLVM native code objects. The LLVM native code objects are LLVM functions, even though the original BAP object may not represent a function. BAP register variables are converted to LLVM globals, and BAP temporaries are converted to local LLVM variables in the generated LLVM function. Memory operations can be converted using several different modes, as specified by the `Llvm_codegen.memimpl`[43] type.

```
type memimpl =
  | Real
```

Memory reads and writes directly access the true main memory. **This is unsafe.**

```
  | Func
```

Replace each memory byte read or written with calls to `set_memory` and `get_memory`.

```
  | FuncMulti
```

Replace each memory operation with calls to `set_memory_multi` or `get_memory_multi`. These functions can access multiple bytes in the same function call.

Conversion modes for emulating memory

```
val string_to_memimpl : string -> memimpl option
```

`string_to_memimpl "Real"` returns `Some Real`, and so on. `string_to_memimpl "Garbage"` returns `None`

```
class codegen : ?opts:bool -> memimpl ->
  object
```

```
    method convert_cfg : Cfg.AST.G.t -> Llvm.llvalue
```

`convert_cfg cfg` converts the BAP cfg `cfg` to LLVM.

```
    method convert_straightline_f : Ast.program -> Llvm.llvalue
```

`convert_straightline_f p` converts the BAP AST program `p` to LLVM.

```
    method convert_exp : Ast.exp -> Llvm.llvalue
```

`convert_exp e` converts the BAP expression `e` to LLVM.

`method convert_type : Type.typ -> Llvm.lltype`

`convert_type t` converts the BAP type `t` to the corresponding LLVM type.

`method convert_var : Var.t -> Llvm.llvalue`

`convert_var v` converts the BAP variable `v` to a LLVM variable. If `v` is a temporary, the variable will be a local LLVM variable. If `v` is not a temporary, it will be converted to a LLVM global.

`method dump : unit`

Print the current LLVM bytecode to the screen.

`method output_bitcode : Pervasives.out_channel -> bool`

`output_bitcode oc` outputs the LLVM bitcode for all functions generated by the current converter to the output channel `oc`.

`method eval_fun : ?ctx:(Var.t * Ast.exp) list -> Llvm.llvalue -> Ast.exp`

`eval_fun ctx f` evaluates the LLVM object `f` with the input context `ctx`, which is a list of variable assignments. If `f` was produced with `convert_exp` or `convert_var`, the returned value will be the evaluated expression or variable value. Otherwise, the return value of the BAP program is returned. If the program terminated because it reached a `Halt` instruction, the evaluated halt expression is returned. Otherwise, `true` is returned.

`end`

`codegen opts memimpl` constructs a class instance for converting BAP constructs to LLVM.

44 Module `Lnf` : Loop nesting forest definitions

```
module type G =
  sig
    include Graph.Builder.S
    val remove_edge_e : G.t -> G.E.t -> G.t
    val remove_edge : G.t -> G.V.t -> G.V.t -> G.t
    val v2s : G.V.t -> string
  end
type 'a lnt = {
  headers : 'a list ;
  body : 'a list ;
```

```
  children : 'a lnf ;
}
type 'a lnf = 'a lnt list
val validate_lnf : 'a lnf -> bool
val validate_lnt : 'a lnt -> bool
val string_of_lnf : ('a -> string) -> 'a lnf -> string
val string_of_lnt : ('a -> string) -> 'a lnt -> string
module type MakeType =
   functor (Gr :  G) ->   sig
     val lnf : Gr.G.t -> Gr.G.V.t -> Gr.G.V.t Lnf.lnf
  end
module Dot :
   functor (Gr :  G) ->   sig
     val to_dot :
       ?e2s:(Gr.G.E.t -> string) -> Gr.G.t -> Gr.G.V.t Lnf.lnf -> string
  end
```

45 Module Lnf_havlak

```
module Make :
   Lnf.MakeType
```

46 Module Lnf_interface : Interface to loop nesting forest (LNF) algorithms.

```
module Steensgard :
   Lnf.MakeType
```
Steensgard

```
module Havlak :
   Lnf.MakeType
```
Havlak

```
module Reduced_Havlak :
   Lnf.MakeType
```
Reduced Havlak

```
module Sreedhar :
   Lnf.MakeType
```

Sreedhar

```
val lnflist : (string * (module Lnf.MakeType)) list
```

A list of all supported LNF algorithms.

47 Module Lnf_ramalingam

```
module type RamalingamHelper =
   functor (Gr :  Lnf.G) ->   sig
     type t
     val init : Gr.G.t -> Gr.G.V.t -> t
     val find_headers : t -> Gr.G.V.t list -> Gr.G.V.t list
  end
module Make :
   functor (RH : RamalingamHelper) -> Lnf.MakeType
```

48 Module Lnf_reduced_havlak

```
module Make :
   Lnf.MakeType
```

49 Module Lnf_sreedhar

```
module Make :
   Lnf.MakeType
```

50 Module Lnf_steensgard

```
module Make :
   Lnf.MakeType
```

51 Module Memory2array : Convert Type.TMem memories to Type.Array memories.

`Type.Array` memories ensure that all memory reads and writes are of the element type, which is almost always `Reg 8` (one byte). This module converts `Type.TMem` memories, which do not have these restrictions, to `Type.Array` memories.

Author(s): Edward J. Schwartz

```
val coerce_prog : Ast.program -> Ast.program
```

`coerce_prog p` returns a new AST program in which all variables of type `Type.TMem` are replaced with variables of type `Type.Array`. All memory reads and writes are of type `Reg 8`.

```
val coerce_exp : Ast.exp -> Ast.exp
```

Like `Memory2array.coerce_prog`[51] but for expressions

Stateful Variants

The stateful variants of the coerce functions are like their non-stateful versions, but also read and update an external state that maps `Type.TMem` variables to `Type.Array` variables. These functions are useful when coercing multiple parts of a program so that a `Type.TMem` variable is mapped to the same `Type.Array` variable in each part of the program.

An optional scope argument can also be passed to each stateful variant so that the parser scope can be kept consistent with converted variables. This allows the parser to understand references to a converted `Type.Array` variable.

```
type state
```

Map of `Type.TMem` variables to their `Type.Array` type equivalent.

```
val create_state : unit -> state
```

Create an empty state usable by `Memory2array.coerce_prog_state`[51], `Memory2array.coerce_exp_state`[51], or `Memory2array.coerce_rvar_state`[51].

```
val coerce_prog_state :
  ?scope:(string, Ast.var) Hashtbl.t * 'a ->
  state -> Ast.program -> Ast.program
```

`coerce_prog_state state p` is like `Memory2array.coerce_prog`[51], but uses and updates the external state `state`.

```
val coerce_exp_state :
  ?scope:(string, Ast.var) Hashtbl.t * 'a ->
  state -> Ast.exp -> Ast.exp
```

Like `Memory2array.coerce_prog_state`[51], but for expressions.

```
val coerce_rvar_state :
  ?scope:(string, Ast.var) Hashtbl.t * 'a ->
  state -> Ast.var -> Ast.var
```

Like `Memory2array.coerce_prog_state`[51], but for referenced variables.

Deprecated functions

```
val split_loads : Ast.exp -> Ast.exp -> Type.typ -> Ast.exp -> Ast.exp
val split_writes :
  Ast.exp -> Ast.exp -> Type.typ -> Ast.exp -> Ast.exp -> Ast.exp
```

52 Module Parser : Bap interface to the parser.

```
val handle_exception : Lexing.lexbuf -> exn -> 'a
val program_from_lexbuf :
  ?scope:Grammar_scope.Scope.t ->
  Lexing.lexbuf -> Ast.program * Grammar_scope.Scope.t
val program_from_file :
  ?scope:Grammar_scope.Scope.t -> string -> Ast.program * Grammar_scope.Scope.t
val exp_from_lexbuf :
  ?scope:Grammar_scope.Scope.t ->
  Lexing.lexbuf -> Ast.exp * Grammar_scope.Scope.t
val exp_from_string :
  ?scope:Grammar_scope.Scope.t -> string -> Ast.exp * Grammar_scope.Scope.t
val exp_from_file :
  ?scope:Grammar_scope.Scope.t -> string -> Ast.exp * Grammar_scope.Scope.t
```

53 Module Pp : Pretty printing

```
module VH :
   Var.VarHash
module F :
   Format
module D :
   Debug.Make(  sig
     val name : string
     val default : [> `Debug ]
  end )
val output_varnums : bool Pervasives.ref
val many_parens : bool Pervasives.ref
val typ_to_string : Type.typ -> string
val ct_to_string : Type.cast_type -> string
val binop_to_string : Type.binop_type -> string
val unop_to_string : Type.unop_type -> string
val reasonable_size_varctx : int
val printed_varctx_warning : bool Pervasives.ref
type varctx = string VH.t * (string, unit) Hashtbl.t
val var_to_string :
  ?ctx:string VH.t * (string, unit) Hashtbl.t -> VH.key -> string
```

```
class pp : F.formatter ->
  object
     method var : Pp.VH.key -> unit
     method vars : Pp.VH.key list -> unit
     method typ : Type.typ -> unit
     method attrs : Ast.attrs -> unit
     method attr : Type.attribute -> unit
     method du : Var.defuse -> unit
     method label : Type.label -> unit
     method int : Big_int_Z.big_int -> Type.typ -> unit
     method ast_exp : ?prec:int -> Ast.exp -> unit
     method ast_endian : Ast.exp -> unit
     method ast_stmt : Ast.stmt -> unit
     method ast_program : Ast.stmt list -> unit
     method ssa_endian : Ssa.exp -> unit
     method ssa_exp : Ssa.exp -> unit
     method ssa_stmt : Ssa.stmt -> unit
     method ssa_stmts : Ssa.stmt list -> unit
     method close : unit
  end

class pp_oc : Pervasives.out_channel ->
  object
     inherit Pp.pp [53]
     method close : unit
  end

val buf : Buffer.t
val ft : Format.formatter
val pp2string_with_pp : 'a -> ('a -> 'b -> 'c) -> 'b -> string
val pp2string : (pp -> 'a -> 'b) -> 'a -> string
val make_varctx : unit -> pp
```

Create a context for use with *_to_string_in_varctx functions.

```
val attr_to_string : Type.attribute -> string
val label_to_string : Type.label -> string
val ssa_exp_to_string : Ssa.exp -> string
val ssa_stmt_to_string : Ssa.stmt -> string
val ast_exp_to_string : Ast.exp -> string
```

```
val ast_stmt_to_string : Ast.stmt -> string
val ast_prog_to_string : Ast.stmt list -> string
val label_to_string_in_varctx : < label : 'a -> 'b; .. > -> 'a -> string
val ssa_exp_to_string_in_varctx : < ssa_exp : 'a -> 'b; .. > -> 'a -> string
val ssa_stmt_to_string_in_varctx :
  < ssa_stmt : 'a -> 'b; .. > -> 'a -> string
val ast_exp_to_string_in_varctx :
  < ast_exp : prec:int -> 'a -> 'b; .. > -> 'a -> string
val ast_stmt_to_string_in_varctx :
  < ast_stmt : 'a -> 'b; .. > -> 'a -> string
```

54 Module Prune_unreachable : Code for removing unreachable nodes in a CFG.

```
val prune_unreachable_ast : Cfg.AST.G.t -> Cfg.AST.G.t
```

prune_unreachable_ast g returns an AST CFG in which nodes unreachable from BB_Entry are removed.

```
val prune_unreachable_ssa : Cfg.SSA.G.t -> Cfg.SSA.G.t
```

Same as prune_unreachable_ast but for SSA CFGs.

55 Module Reachable : Reachability analysis

```
module type G =
  sig
    include Graph.Builder.S
    val remove_vertex : G.t -> G.V.t -> G.t
    val copy_map : G.t -> G.t
    val v2s : G.V.t -> string
  end
```

Graph signature needed for reachability analysis

```
module type Reach =
  sig
    type gt
```

Graph type.

```
type vt
```

Vertex type.

```
val iter_reachable : (vt -> unit) ->
  gt -> vt -> unit
```

`iter_reachable f g v` calls `f v'` for every vertex `v'` that is reachable from `v`.

```
val iter_unreachable : (vt -> unit) ->
  gt -> vt -> unit
```

`iter_unreachable f g v` calls `f v'` for every vertex `v'` that is unreachable from `v`.

```
val fold_reachable : (vt -> 'a -> 'a) ->
  gt -> vt -> 'a -> 'a
```

Fold over reachable vertices.

```
val fold_unreachable : (vt -> 'a -> 'a) ->
  gt -> vt -> 'a -> 'a
```

Fold over unreachable vertices.

```
val reachable : gt -> vt -> vt list
```

Return a list of reachable vertices.

Return a list of unreachable vertices.

```
val unreachable : gt -> vt -> vt list
val remove_unreachable : gt -> vt -> gt
```

Return a new graph with unreachable nodes removed.

```
val remove_unreachable_copy : gt -> vt -> gt
```

Same as `remove_unreachable`, but implemented differently. Creates a new graph and copies all reachable nodes. This can be more efficient for large graphs when many vertices are unreachable.

```
end
```

Output signature of reachability analysis

```
module Make :
   functor (BI : G) -> Reach  with type gt = BI.G.t and type vt = BI.G.V.t
```

Functor that builds reachability analysis

55.0.15 Reachability analyses for Control Flow Graphs

```
module AST :
   Reach  with type gt = Cfg.AST.G.t and type vt = Cfg.AST.G.V.t
```

Reachability analysis for AST CFGs

```
module SSA :
   Reach  with type gt = Cfg.SSA.G.t and type vt = Cfg.SSA.G.V.t
```

Reachability analysis for SSA CFGs

56 Module Sccvn : Strongly connected component based value numbering.

Currently we only implement the RPO algorithm, described in "SCC-Based Value Numbering" by Keith Cooper and Taylor Simpson. http://citeseer.ist.psu.edu/41805.html

TODO: This has been hacked on a bit and could use some cleanup. (Removing silly things and making it easier to understand.)

TODO: canonicalize constants in HInt

Author(s): Ivan Jager

```
val replacer : ?opt:bool -> Cfg.SSA.G.t -> Cfg.SSA.G.t * bool
```

Use SCCVN to elliminate redundant expressions, replacing them with a previously computed value. Some variables will no longer be used after this, so it may be beneficial to run dead code elimination after.

Returns the new CFG and a bool indicating whether anything changed.

```
val aliased : Cfg.SSA.G.t -> Ssa.exp -> Ssa.exp -> bool option
```

`aliased cfg` returns a function `f` to tell whether two values are aliased. `f x y` returns: Some true when `x=y`, Some false when `x<>y`, or None when it could not statically determine whether `x=y`.

57 Module Smtexec : Interface for executing command line driven SMT solvers.

XXX: This module is designed for Unix systems and is not portable.

Author(s): ejs

```
type model = (string * Big_int_Z.big_int) list option
type result =
  | Valid
```

The formula was valid or unsatisfiable.

| `Invalid of model`

The formula was invalid or satisfiable.

| `SmtError of string`

The solver failed. Possible reasons for this include the formula having invalid syntax and the solver running out of memory.

| `Timeout`

The solver took too long to solve the formula.

The result of solving a formula.

```
val result_to_string : result -> string
```

Convert a result to a string

```
val print_model : model -> unit
```

Print a model to `stdout`.

```
class type smtexec =
  object
    method in_path : unit -> bool
    method printer : Formulap.fppf
    method solve_formula_file :
      ?timeout:int -> ?remove:bool -> ?getmodel:bool -> string -> Smtexec.result
    method solvername : string
    method streaming_printer : Formulap.stream_fppf
  end
```

A hack so that we can subtype solver instances. If ocaml <3.11 had first order modules, we wouldn't need this.

```
module type SOLVER =
  sig
    val solvername : string
```

Solver name

```
    val in_path : unit -> bool
```

`in_path ()` returns `true` if and only if the solver appears to be in the in the `PATH`.

```
    val solve_formula_file :
      ?timeout:int -> ?remove:bool -> ?getmodel:bool -> string -> Smtexec.result
```

`solve_formula_file f` solves the formula in `f`.

```
val check_exp_validity :
  ?timeout:int ->
  ?remove:bool ->
  ?exists:Ast.var list -> ?foralls:Ast.var list -> Ast.exp -> Smtexec.result
```

check_exp_validity e tests the validity of e. The timeout and remove options are the same as in Smtexec.SOLVER.solve_formula_file[57].

```
val create_cfg_formula :
  ?remove:bool ->
  ?exists:Ast.var list -> ?foralls:Ast.var list -> Cfg.AST.G.t -> string
```

create_cfg_formula cfg computes the weakest precondition for the CFG program cfg, using postcondition true. The weakest precondition is then written to a file, and the name of this file is returned.

remove, exists, and foralls behave the same as above.

XXX: Select weakest precondition method using vc.mli

XXX: Give this a better name

```
val si : Smtexec.smtexec
```

An object to enable subtyping

```
end
```

Interface for a solver.

```
module STP :
   SOLVER
module STPSMTLIB :
   SOLVER
module CVC3 :
   SOLVER
module CVC3SMTLIB :
   SOLVER
module YICES :
   SOLVER
module Z3 :
   SOLVER
module BOOLECTOR :
   SOLVER
val solvers : (string, smtexec) Hashtbl.t
```

A Hashtbl that maps solver names to the corresponding Smtexec.SOLVER[57] module.

58 Module Smtlib1 : Output to SMTLIB1 format

```
module D :
   Debug.Make(  sig
     val name : string
     val default : [> `NoDebug ]
  end )
exception No_rule
module VH :
   Var.VarHash
type sort =
  | BitVec
  | Bool
val use_booleans : bool Pervasives.ref
```

This printer has to deal with a number of differences between the BAP IL and SMTLIB. One of the most striking is that SMTLIB has separate types for booleans and bitvectors of one bit; BAP only has bitvectors of one bit. There is no one size fits all solution. Booleans have many nice features, such as n-ary functions. Some functions only work on bitvectors (like casts).

The primary printing functions are ast_exp_base and ast_exp_bool_base. ast_exp_base prints a BAP function e, and if e is a 1-bit bap bitvector, it treats it as a bitvector in SMTLIB. ast_exp_bool_base only works on 1-bit bap bitvectors, and treats them as a boolean in SMTLIB. Either function can raise the No_rule exception. When the check argument is set, the function should raise No_rule before printing anything.

ast_exp, ast_exp_bool, and ast_exp_bv are wrappers for the *_base functions. They catch the No_rule exceptions, and send the request to the other base function if necessary. They print, rather than returning lazy evaluations, since they can never return No_rule.

```
class pp : Format.formatter ->
  object
    inherit Formulap.fpp [34]
    inherit Formulap.stream_fpp [34]
    val used_vars : (string, Smtlib1.VH.key) Hashtbl.t
    val ctx : (string * Smtlib1.sort) Smtlib1.VH.t
    val mutable unknown_counter : int
    val mutable let_counter : int
    method bool_to_bv : check:bool -> Ast.exp -> unit
    method bv_to_bool : check:bool -> Ast.exp -> unit
    method flush : unit
    method extend : Smtlib1.VH.key -> string -> Smtlib1.sort -> unit
    method unextend : Smtlib1.VH.key -> unit
```

```
method var : Smtlib1.VH.key -> unit
method and_start : unit
method and_constraint : Ast.exp -> unit
method and_close_constraint : unit
method and_end : unit
method let_begin : Ast.var -> Ast.exp -> unit
```

Seperate let_begin and let_end to allow for streaming generation of formulas in utils/streamtrans.ml

```
method let_end : Ast.var -> unit
method let_middle : Smtlib1.sort -> Ast.exp -> unit
method letme : Ast.var -> Ast.exp -> Ast.exp -> Smtlib1.sort -> unit Lazy.t
```

Returns a lazy expression that prints let v = e1 in e2. Never raises No_rule.

```
method varname : Smtlib1.VH.key -> string
method varsort : Smtlib1.VH.key -> Smtlib1.sort
method declare_given_freevars : Smtlib1.VH.key list -> unit
method declare_new_freevars : Ast.exp -> unit
method typ : Type.typ -> unit
method decl_no_print : Ast.var -> unit
method predeclare_free_var : Ast.var -> unit
method print_free_var : Ast.var -> unit
method decl : Smtlib1.VH.key -> unit
method tryit : (check:bool -> Ast.exp -> unit) -> Ast.exp -> unit
method ast_exp_base : check:bool -> Ast.exp -> unit
```

Prints the BAP expression e in SMTLIB format. If e is a 1-bit bitvector in BAP, then e is printed as a SMTLIB 1-bit bitvector. Raises the No_rule exception if it is not possible to print e as a bitvector (e.g., it should be printed as a boolean).

```
method ast_exp : Ast.exp -> unit
```

Evaluate an expression to a bitvector, preferring bools instead of 1-bit bvs.

```
method ast_exp_bv : Ast.exp -> unit
```

Evaluates an expression to a bitvector, preferring bitvectors over booleans.

```
method ast_exp_bool_base : check:bool -> Ast.exp -> unit
```

Try to evaluate an expression to a boolean. If no good rule exists, then raises the No_rule exception.

```
method ast_exp_bool : Ast.exp -> unit
```

Try to evaluate an expression to a boolean. If no good rule exists, uses bitvector conversion instead.

```
    method forall : Formulap.VH.key list -> unit
    method exists : Smtlib1.VH.key list -> unit
    method open_benchmark_with_logic : string -> unit
    method open_stream_benchmark : unit
    method open_benchmark : Ast.exp -> unit
    method close_benchmark : unit
    method assert_ast_exp_begin :
      ?exists:Smtlib1.VH.key list -> ?foralls:Formulap.VH.key list -> unit -> unit
    method assert_ast_exp_end : unit
    method assert_ast_exp :
      ?exists:Smtlib1.VH.key list ->
      ?foralls:Formulap.VH.key list -> Ast.exp -> unit
    method valid_ast_exp_begin :
      ?exists:Smtlib1.VH.key list -> ?foralls:Formulap.VH.key list -> unit -> unit
    method valid_ast_exp_end : unit
    method valid_ast_exp :
      ?exists:Smtlib1.VH.key list ->
      ?foralls:Formulap.VH.key list -> Ast.exp -> unit
    method formula : unit -> unit
    method counterexample : unit
    method close : unit
  end

class pp_oc : Pervasives.out_channel ->
  object

    inherit Smtlib1.pp [58]
    inherit Formulap.fpp_oc [34]
    inherit Formulap.stream_fpp_oc [34]
    method close : unit
  end
```

59 Module Smtlib2 : Output to SMTLIB2 format

Author(s): ejs

```
module D :
   Debug.Make(  sig
```

```
    val name : string
    val default : [> `NoDebug ]
  end )
exception No_rule
module VH :
   Var.VarHash
type sort =
  | BitVec
  | Bool
type option =
  | SetOptionProduceAssignments
```

Z3 only

```
val use_booleans : bool Pervasives.ref
```

This printer has to deal with a number of differences between the BAP IL and SMTLIB. One of the most striking is that SMTLIB has separate types for booleans and bitvectors of one bit; BAP only has bitvectors of one bit. There is no one size fits all solution. Booleans have many nice features, such as n-ary functions. Some functions only work on bitvectors (like casts).

The primary printing functions are ast_exp_base and ast_exp_bool_base. ast_exp_base prints a BAP function e, and if e is a 1-bit bap bitvector, it treats it as a bitvector in SMTLIB. ast_exp_bool_base only works on 1-bit bap bitvectors, and treats them as a boolean in SMTLIB. Either function can raise the No_rule exception. When the check argument is set, the function should raise No_rule before printing anything.

ast_exp, ast_exp_bool, and ast_exp_bv are wrappers for the *_base functions. They catch the No_rule exceptions, and send the request to the other base function if necessary. They print, rather than returning lazy evaluations, since they can never return No_rule.

```
class pp : ?opts:option list -> Format.formatter ->
  object
    inherit Formulap.fpp [34]
    val used_vars : (string, Smtlib2.VH.key) Hashtbl.t
    val ctx : (string * Smtlib2.sort) Smtlib2.VH.t
    val mutable unknown_counter : int
    val mutable let_counter : int
    method bool_to_bv : check:bool -> Ast.exp -> unit
    method bv_to_bool : check:bool -> Ast.exp -> unit
    method flush : unit
    method extend : Smtlib2.VH.key -> string -> Smtlib2.sort -> unit
    method unextend : Smtlib2.VH.key -> unit
    method var : Smtlib2.VH.key -> unit
    method and_start : unit
    method and_constraint : Ast.exp -> unit
```

```
method and_close_constraint : unit
method and_end : unit
method let_begin : Smtlib2.VH.key -> Ast.exp -> unit
```

Seperate let_begin and let_end to allow for streaming generation of formulas in utils/streamtrans.ml

```
method let_end : Smtlib2.VH.key -> unit
method let_middle : Smtlib2.sort -> Ast.exp -> unit
method letme :
  Smtlib2.VH.key -> Ast.exp -> Ast.exp -> Smtlib2.sort -> unit Lazy.t
```

Returns a lazy expression that prints let v = e1 in e2. Never raises No_rule.

```
method varname : Smtlib2.VH.key -> string
method varsort : Smtlib2.VH.key -> Smtlib2.sort
method declare_given_freevars : Smtlib2.VH.key list -> unit
method declare_new_freevars : Ast.exp -> unit
method typ : Type.typ -> unit
method decl_no_print : Smtlib2.VH.key -> unit
method predeclare_free_var : Smtlib2.VH.key -> unit
method print_free_var : Smtlib2.VH.key -> unit
method decl : Smtlib2.VH.key -> unit
method tryit : (check:bool -> Ast.exp -> unit) -> Ast.exp -> unit
method ast_exp_base : check:bool -> Ast.exp -> unit
```

Prints the BAP expression e in SMTLIB format. If e is a 1-bit bitvector in BAP, then e is printed as a SMTLIB 1-bit bitvector. Raises the No_rule exception if it is not possible to print e as a bitvector (e.g., it should be printed as a boolean).

```
method ast_exp : Ast.exp -> unit
```

Evaluate an expression to a bitvector, preferring bools instead of 1-bit bvs.

```
method ast_exp_bv : Ast.exp -> unit
```

Evaluates an expression to a bitvector, preferring bitvectors over booleans.

```
method ast_exp_bool_base : check:bool -> Ast.exp -> unit
```

Try to evaluate an expression to a boolean. If no good rule exists, then raises the No_rule exception.

```
method ast_exp_bool : Ast.exp -> unit
```

Try to evaluate an expression to a boolean. If no good rule exists, uses bitvector conversion instead.

```
    method forall : Formulap.VH.key list -> unit
    method exists : Smtlib2.VH.key list -> unit
    method open_benchmark_with_logic : string -> unit
    method open_stream_benchmark : unit
    method open_benchmark : Ast.exp -> unit
    method close_benchmark : unit
    method assert_ast_exp_begin :
      ?exists:Smtlib2.VH.key list -> ?foralls:Formulap.VH.key list -> unit -> unit
    method assert_ast_exp_end : unit
    method assert_ast_exp :
      ?exists:Smtlib2.VH.key list ->
      ?foralls:Formulap.VH.key list -> Ast.exp -> unit
    method valid_ast_exp_begin :
      ?exists:Smtlib2.VH.key list -> ?foralls:Formulap.VH.key list -> unit -> unit
    method valid_ast_exp_end : unit
    method valid_ast_exp :
      ?exists:Smtlib2.VH.key list ->
      ?foralls:Formulap.VH.key list -> Ast.exp -> unit
    method formula : unit
    method counterexample : unit
    method close : unit
  end

class pp_oc : ?opts:option list -> Pervasives.out_channel ->
  object

    inherit Smtlib2.pp [59]
    inherit Formulap.fpp_oc [34]
    inherit Formulap.stream_fpp_oc [34]
    method close : unit
  end
```

60 Module `Solver` : Primary interface to SMT solvers.

Author(s): Thanassis Avgerinos, Sang Kil Cha, ejs

```
exception Solver_error of string
module type Solver_Out =
  sig

    type set
```

Solver context

```
val is_satisfiable : Ast.exp -> bool
```

`is_sat e` returns true iff e is satisfiable.

```
val is_valid : Ast.exp -> bool
```

`is_valid e` returns true iff e is satisfiable

```
class solver :
  object
    val s : Solver.Solver_Out.set
    method add_binding : Ast.var -> Ast.exp -> unit
```

Add a `Ast.Let` binding to context

```
    method add_constraint : Ast.exp -> unit
```

Add boolean constraint to context

```
    method del_binding : Ast.var -> unit
```

Delete boolean constraint from context

```
    method is_sat : bool
```

true iff context is satisfiable

```
    method is_valid : Ast.exp -> bool
```

true iff context is valid

```
    method model_exp : Ast.exp -> Ast.exp
```

`model_exp e` evaluates `e` in the current model

```
    method pop : unit
```

Pop solver context from context stack

```
    method push : unit
```

Push current solver context onto context stack

```
    method pp_sol : string
  end
```

Objects wrapping around a solver context

```
val newsolver : unit -> solver
```

Create a new solver object

```
end
```

Output signature for a solver

```
module Z3 :
   Solver_Out
```

Z3 direct binding interface

```
module STPExec :
   Solver_Out
```

STP command line interface

```
module STPSMTLIBExec :
   Solver_Out
```

STP command line interface in SMTLIB mode

```
module CVC3Exec :
   Solver_Out
```

CVC3 command line interface

```
module CVC3SMTLIBExec :
   Solver_Out
```

CVC3 command line interface in SMTLIB mode

```
module YICESExec :
   Solver_Out
```

Yices command line interface

```
val solvers : (string, unit -> STPExec.solver) Hashtbl.t
```

A hashtbl that maps a solver name to a solver creation function. Useful for subtyping.

61 Module Ssa : Static Single Assignment form.

This is the intermediate language where most analysis should be happening.
Author(s): Ivan Jager

```
type var = Var.t
type exp =
  | Load of exp * exp * exp * Type.typ
```

Load(arr,idx,endian, t)

```
  | Store of exp * exp * exp * exp * Type.typ
```

Store(arr,idx,val, endian, t)

```
  | BinOp of Type.binop_type * exp * exp
  | UnOp of Type.unop_type * exp
  | Var of var
  | Lab of string
  | Int of Big_int_Z.big_int * Type.typ
  | Cast of Type.cast_type * Type.typ * exp
```

Cast to a new type.

```
  | Unknown of string * Type.typ
  | Ite of exp * exp * exp
  | Extract of Big_int_Z.big_int * Big_int_Z.big_int * exp
  | Concat of exp * exp
  | Phi of var list
```

Joins variables that were assigned over different paths

```
type attrs = Type.attributes
type stmt =
  | Move of var * exp * attrs
```

Assign the exp on the right to the var on the left

```
  | Jmp of exp * attrs
```

Jump to a label/address

```
  | CJmp of exp * exp * exp * attrs
```

Conditional jump. If e1 is true, jumps to e2, otherwise jumps to e3

```
  | Label of Type.label * attrs
```

A label we can jump to

```
  | Halt of exp * attrs
  | Assert of exp * attrs
  | Assume of exp * attrs
  | Special of string * Var.defuse * attrs
  | Comment of string * attrs
```

A comment to be ignored

```
val val_false : exp
val val_true : exp
val lab_of_exp : exp -> Type.label option
```

If possible, make a label that would be refered to by the given expression.

```
val full_value_eq : 'a -> 'a -> bool
val quick_value_eq : 'a -> 'a -> bool
val num_exp : exp -> int
val getargs_exp :
  exp ->
  exp list * Type.typ list * Type.binop_type list * Type.unop_type list *
```

```
  string list * Type.cast_type list * var list * Big_int_Z.big_int list
val quick_exp_eq : exp -> exp -> bool
```

quick_exp_eq e1 e2 returns true if and only if the subexpressions in e1 and e2 are *physically* equal.

```
val full_exp_eq : 'a -> 'a -> bool
```

full_exp_eq e1 e2 returns true if and only if e1 and e2 are structurally equivalent.

```
val (===) : 'a -> 'a -> bool
val num_stmt : stmt -> int
val getargs_stmt :
  stmt ->
  exp list * var list * Type.label list * attrs list *
  string list * exp list
val quick_stmt_eq : stmt -> stmt -> bool
```

quick_stmt_eq returns true if and only if the subexpressions in e1 and e2 are *physically* equal.

```
val full_stmt_eq : 'a -> 'a -> bool
```

full_stmt_eq returns true if and only if e1 and e2 are structurally equivalent.

```
val get_attrs : stmt -> attrs
val exp_true : exp
val exp_false : exp
```

62 Module Ssa_cond_simplify : Simplify predicates so that VSA and other abstract interpretations can use them.

```
val simplifycond_ssa : Cfg.SSA.G.t -> Cfg.SSA.G.t
```

Simplify conditions used in edge labels

```
val simplifycond_targets_ssa : Ssa.exp list -> Cfg.SSA.G.t -> Cfg.SSA.G.t
```

Simplify conditions used in edge labels. This version is for resolving the target expressions passed as arguments. It does so by never copy propagating beyond one of the variables used in any of the target expressions. This should ensure that the simplified conditions are "in terms of" variables in the target expression. This is important for VSA cfg recovery.

63 Module Ssa_convenience : Utility functions for SSAs.

It's useful to have these in a separate file so it can use functions from Typecheck and elsewhere.

```
val unknown : Type.typ -> string -> Ssa.exp
val binop : Type.binop_type -> Ssa.exp -> Ssa.exp -> Ssa.exp
val unop : Type.unop_type -> Ssa.exp -> Ssa.exp
val concat : Ssa.exp -> Ssa.exp -> Ssa.exp
val extract : int -> int -> Ssa.exp -> Ssa.exp
val exp_and : Ssa.exp -> Ssa.exp -> Ssa.exp
val exp_or : Ssa.exp -> Ssa.exp -> Ssa.exp
val exp_eq : Ssa.exp -> Ssa.exp -> Ssa.exp
val exp_not : Ssa.exp -> Ssa.exp
val exp_implies : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( +* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( -* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( ** ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( <<* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( >>* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( >>>* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( &* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( |* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( ^* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( ==* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( <>* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( <* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( >* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( <=* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( >=* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( =* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
```

bitwise equality

```
val ( ++* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( %* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( $%* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( /* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val ( $/* ) : Ssa.exp -> Ssa.exp -> Ssa.exp
val cast : Type.cast_type -> Type.typ -> Ssa.exp -> Ssa.exp
val cast_low : Type.typ -> Ssa.exp -> Ssa.exp
```

```
val cast_high : Type.typ -> Ssa.exp -> Ssa.exp
val cast_signed : Type.typ -> Ssa.exp -> Ssa.exp
val cast_unsigned : Type.typ -> Ssa.exp -> Ssa.exp
val exp_int : Big_int_Z.big_int -> int -> Ssa.exp
val it : int -> Type.typ -> Ssa.exp
val exp_ite : ?t:Type.typ -> Ssa.exp -> Ssa.exp -> Ssa.exp -> Ssa.exp
val parse_ite : Ssa.exp -> (Ssa.exp * Ssa.exp * Ssa.exp) option
val parse_implies : Ssa.exp -> (Ssa.exp * Ssa.exp) option
val rm_duplicates : Ssa.exp -> Ssa.exp
```

Duplicate any shared nodes. Useful for using physical location as a unique identity.

XXX: I think this would be much faster if we only duplicated things that actually occur more than once.

```
val parse_extract : Ssa.exp -> (Z.t * Big_int_Z.big_int) option
val parse_concat : Ssa.exp -> (Ssa.exp * Ssa.exp) option
val rm_ite : Ssa.exp -> Ssa.exp
val rm_extract : Ssa.exp -> Ssa.exp
val rm_concat : Ssa.exp -> Ssa.exp
val last_meaningful_stmt : Ssa.stmt list -> Ssa.stmt
val min_symbolic : signed:bool -> Ssa.exp -> Ssa.exp -> Ssa.exp
val extract_element : Type.typ -> Ssa.exp -> int -> Ssa.exp
val extract_byte : Ssa.exp -> int -> Ssa.exp
val extract_element_symbolic : Type.typ -> Ssa.exp -> Ssa.exp -> Ssa.exp
val extract_byte_symbolic : Ssa.exp -> Ssa.exp -> Ssa.exp
val reverse_bytes : Ssa.exp -> Ssa.exp
val concat_explist : Ssa.exp BatEnum.t -> Ssa.exp
```

64 Module Ssa_simp : SSA simplifications

This uses all supported simplifications to try to optimize as much as possible.

```
module D :
   Debug.Make(  sig
     val name : string
     val default : [> `NoDebug ]
  end )
val simp_cfg :
  ?liveout:Var.t list ->
  ?usedc:bool -> ?usesccvn:bool -> ?usemisc:bool -> Cfg.SSA.G.t -> Cfg.SSA.G.t
```

```
val simp_astcfg :
  ?liveout:Var.t list ->
  ?usedc:bool -> ?usesccvn:bool -> ?usemisc:bool -> Cfg.AST.G.t -> Cfg.AST.G.t
```

65 Module Ssa_simp_misc : Misc optimizations

```
module C :
   Cfg.SSA
module D :
   Debug.Make(  sig
     val name : string
     val default : [> `Debug ]
  end )
val cfg_jumpelim : C.G.t -> C.G.t * bool
```

Look for conditional jumps that have a constant as the expression and replace with a jump

66 Module Ssa_slice : A module to perform chopping on SSAs

```
module CHOP_SSA :
  sig
     module Traverse :
     Graph.Traverse.Dfs(Cfg.SSA.G)
     module SG :
       sig
         type t = Cfg.SSA.G.t
         module V :
         Cfg.SSA.G.V
         val iter_vertex : (Cfg.SSA.G.vertex -> unit) -> Cfg.SSA.G.t -> unit
         val iter_succ :
           (Cfg.SSA.G.vertex -> unit) -> Cfg.SSA.G.t -> Cfg.SSA.G.vertex -> unit
       end
     module Comp :
     Graph.Components.Make(SG)
     val get_scc : SG.t ->
       Cfg.SSA.G.vertex -> Cfg.SSA.G.vertex -> SG.t
```

```
    val compute_cds :
      Depgraphs.CDG_SSA.G.t ->
      (Cfg.SSA.G.vertex * int, Cfg.SSA.G.vertex * int) Hashtbl.t -> unit
    val add_jmp_stmts :
      Cfg.SSA.G.t ->
      (Cfg.SSA.G.vertex * int, Cfg.SSA.G.vertex * int) Hashtbl.t -> unit
    val get_dds :
      Depgraphs.CDG_SSA.G.t ->
      (Depgraphs.DDG_SSA.location, Depgraphs.DDG_SSA.location) Hashtbl.t
    val slice :
      Depgraphs.CDG_SSA.G.t -> Cfg.SSA.G.V.t -> int -> Depgraphs.CDG_SSA.G.t
    val chop : SG.t -> int -> 'a -> int -> int -> Depgraphs.CDG_SSA.G.t
  end
```

67 Module `Ssa_visitor` : Visitor for SSA programs and expressions.

Visitors are a systematic method of exploring and modifying programs and expressions.

Users create visitors that describe what actions to take when various constructs are encountered. For instance, the `visit_exp` method is called whenever an expression is encountered. This visitor is passed to `accept` functions, which takes an object to visit, and calls the visitor's methods are the proper time.

```
class type t =
  object
```

`method visit_exp : Ssa.exp -> Ssa.exp Type.visit_action`

Called when visiting an expression

`method visit_stmt : Ssa.stmt -> Ssa.stmt Type.visit_action`

Called when visiting a statement.

Note that in a Move() or Let(), referenced variables will be visited first, so that this can be used to add the assigned variable to your context.

FIXME: would be nice to be able to add stmts... We may change this.

`method visit_rvar : Ssa.var -> Ssa.var Type.visit_action`

Called when visiting a refenenced variable. (IE: inside an expression)

`method visit_avar : Ssa.var -> Ssa.var Type.visit_action`

Called when visiting an assigned variable. (IE: On the LHS of a Move

```
  end
```

The type for a visitor

`class nop : t`

A nop visitor that visits all children, but does not change anything. This visitor can be inherited from to build a new one.

67.0.16 Accept functions

`val rvar_accept : #t -> Ssa.var -> Ssa.var`

Visit a referenced variable

`val avar_accept : #t -> Ssa.var -> Ssa.var`

Visit an assigned variable

`val exp_accept : #t -> Ssa.exp -> Ssa.exp`

Visit an expression

`val stmt_accept : #t -> Ssa.stmt -> Ssa.stmt`

Visit a statement

`val stmts_accept : #t -> Ssa.stmt list -> Ssa.stmt list`

Visit a list of statements

`val prog_accept : #t -> Cfg.SSA.G.t -> Cfg.SSA.G.t`

Visit a SSA program/CFG

`val cfg_accept : #t -> Cfg.SSA.G.t -> Cfg.SSA.G.t`

Alias of `Ssa_visitor.prog_accept`[67.0.16].

68 Module `Steensgard` : Steensgard's loop nesting algorithm

See Steensgaard, B. (1993). Sequentializing Program Dependence Graphs for Irreducible Programs (No. MSR-TR-93-14).

```
type 'a loopinfo =
  | BB of 'a
  | Other of 'a loopinfo list
  | Loop of 'a loopinfo list
```

Loop information for CFGs with vertices of type `'a`.

`val steensgard_ast : Cfg.AST.G.t -> Cfg.AST.G.V.t loopinfo`

`val steensgard_ssa : Cfg.SSA.G.t -> Cfg.SSA.G.V.t loopinfo`

69 Module Stp : Output to STP format (same as CVCL or CVC3)

```
module D :
   Debug.Make(  sig
     val name : string
     val default : [> `NoDebug ]
  end )
module VH :
   Var.VarHash
class pp : Format.formatter ->
  object
     inherit Formulap.fpp [34]
     val used_vars : (string, Stp.VH.key) Hashtbl.t
     val ctx : string Stp.VH.t
     val mutable unknown_counter : int
     val mutable let_counter : int
     method and_start : unit
     method and_constraint : Ast.exp -> unit
     method and_close_constraint : unit
     method and_end : unit
     method let_begin : Stp.VH.key -> Ast.exp -> unit
     method let_end : Stp.VH.key -> unit
     method open_stream_benchmark : unit
     method close_benchmark : unit
     method declare_given_freevars : Stp.VH.key list -> unit
     method declare_new_freevars : Ast.exp -> unit
     method predeclare_free_var : Stp.VH.key -> unit
     method flush : unit
     method extend : Stp.VH.key -> string -> unit
     method unextend : Stp.VH.key -> unit
     method var : Stp.VH.key -> unit
     method varname : Stp.VH.key -> string
     method typ : Type.typ -> unit
     method decl_no_print : Stp.VH.key -> unit
     method print_free_var : Stp.VH.key -> unit
     method decl : Stp.VH.key -> unit
     method ast_exp : Ast.exp -> unit
```

```
    method forall : Formulap.VH.key list -> unit
    method exists : Stp.VH.key list -> unit
    method assert_ast_exp_begin :
      ?exists:Stp.VH.key list -> ?foralls:Formulap.VH.key list -> unit -> unit
    method assert_ast_exp_end : unit
    method assert_ast_exp :
      ?exists:Stp.VH.key list -> ?foralls:Formulap.VH.key list -> Ast.exp -> unit
    method valid_ast_exp_begin :
      ?exists:Stp.VH.key list -> ?foralls:Formulap.VH.key list -> unit -> unit
    method valid_ast_exp_end : unit
    method valid_ast_exp :
      ?exists:Stp.VH.key list -> ?foralls:Formulap.VH.key list -> Ast.exp -> unit
```

Is e a valid expression (always true)?

```
    method counterexample : unit
    method close : unit
  end

class pp_oc : Pervasives.out_channel ->
  object

    inherit Stp.pp [69]
    inherit Formulap.fpp_oc [34]
    inherit Formulap.stream_fpp_oc [34]
    method close : unit
  end
```

70 Module `Structural_analysis` : Structural Analysis

This is an implementation of structural analysis based on Advanced Compiler Design & Implementation by Steven S Muchnick.

TODO: Add support for proper and improper intervals.

```
module D :
   Debug.Make(  sig

     val name : string
     val default : [> `NoDebug ]

  end )

type region_type =
  | Block
```

```
  | IfThen
  | IfThenElse
  | Case
  | Proper
  | SelfLoop
  | WhileLoop
  | NaturalLoop
  | Improper
type node =
  | BBlock of Cfg.bbid
  | Region of region_type * node list
module C :
   Cfg.AST
module Node :
  sig
     type t = Structural_analysis.node
     val compare : 'a -> 'a -> int
     val hash : 'a -> int
     val equal : 'a -> 'a -> bool
  end
val rtype2s : region_type -> string
val node2s : node -> string
val nodes2s : node list -> string
module G :
   Graph.Imperative.Digraph.ConcreteBidirectional(Node)
module PC :
   Graph.Path.Check(G)
module DFS :
   Graph.Traverse.Dfs(G)
val printg : G.t -> unit
val graph_of_cfg : C.G.t -> G.t
val find_backedges : G.t ->
  G.vertex ->
  (G.vertex * G.vertex) list
val structural_analysis : C.G.t -> G.vertex
```

71 Module `Switch_condition` : Rewrite the outgoing edge conditions from (indirect jump) switches so they are in terms of the input variable, and not the target destination.

Author(s): ejs

```
val add_switch_conditions_disasm :
  Asmir_disasm.vsaresult -> Cfg.SSA.G.t * bool
```

Adds switch conditions to a SSA graph. Takes the information returned by the VSA CFG recovery analysis as input. Returns a modified CFG where as many indirect jumps are rewritten as posible and a success flag. The success flag is true when all indirect jumps were rewritten.

```
val add_switch_conditions_ssacfg :
  Asmir.asmprogram -> Cfg.SSA.G.t -> Cfg.SSA.G.t * bool
```

Like `add_switch_conditions_disasm`, but starts from an arbitrary SSA CFG. Re-runs VSA analysis.

72 Module Symbeval : A module to perform AST Symbolic Execution

TODO list: * * - Cleanup & make readable

```
module VH :
   Var.VarHash
module VM :
   Var.VarMap
module D :
   Debug.Make(  sig
     val name : string
     val default : [> 'NoDebug ]
  end )
module AddrMap :
   Map.Make(  sig
     type t = Big_int_Z.big_int
     val compare : Z.t -> Z.t -> int
  end )
type mem = Ast.exp AddrMap.t * Var.t
type instr = Ast.stmt
type varid = Ast.exp
type varval =
  | Symbolic of Ast.exp
  | ConcreteMem of mem
type label_kind = Type.label
type form_type =
```

```
  | Equal
  | Rename
type ('a, 'b) ctx = {
  pred : 'b ;
  mutable delta : 'a ;
  sigma : (Type.addr, instr) Hashtbl.t ;
  lambda : (label_kind, Type.addr) Hashtbl.t ;
  pc : Type.addr ;
}
val eq_concrete_mem :
  'a AddrMap.t * 'b -> 'a AddrMap.t * 'b -> Z.t * Type.typ
val mem_binop :
  Type.binop_type ->
  'a AddrMap.t * 'b -> 'a AddrMap.t * 'b -> Z.t * Type.typ
val empty_mem : Var.t -> varval
val empty_smem : Ast.var -> varval
val val_true : varval
val val_false : varval
val is_true_val : varval -> bool
val is_false_val : varval -> bool
val is_symbolic : varval -> bool
val is_concrete_scalar : Ast.exp -> bool
val is_concrete_mem_or_scalar : varval -> bool
val is_concrete_mem : varval -> bool
val concrete_val_tuple : varval -> Big_int_Z.big_int * Type.typ
val context_update : 'a VH.t -> VH.key -> 'a -> unit
val context_copy : 'a VH.t -> 'a VH.t
val symb_to_exp : varval -> Ast.exp
val concmem_to_mem : varval -> mem
val symb_to_string : varval -> string
val conc2symb : Ast.exp AddrMap.t -> Ast.var -> varval
val varval_to_exp : varval -> Ast.exp
val normalize : Z.t -> Type.typ -> Z.t
module type MemLookup =
  sig
    type t
```

Lookup type

```
    val create : unit -> t
```

Initial lookup table
Clear the lookup table

```
val clear : t -> t
```

Deep copy the lookup table

```
val copy : t -> t
```

Print vars

```
val print_values : t -> unit
```

Print memories

```
val print_mem : t -> unit
val lookup_var : t -> Var.t -> Symbeval.varval
```

Look up the value of a variable
Update the value of a variable.

```
val update_var : t -> Var.t -> Symbeval.varval -> t
```

Remove the value for a variable.

```
val remove_var : t -> Var.t -> t
```

Update memory

```
val update_mem :
  Symbeval.varval -> Ast.exp -> Ast.exp -> Ast.exp -> Symbeval.varval
```

Lookup memory

```
val lookup_mem : Symbeval.varval -> Ast.exp -> Ast.exp -> Ast.exp
  end
module type EvalTune =
  sig
    val eval_symb_let : bool
  end
module type FlexibleFormula =
  sig
    type t
    type init
    type output
    val init : init -> t
    val add_to_formula : t ->
      Ast.exp -> Symbeval.form_type -> t
    val output_formula : t -> output
```

```
  end
module type Assign =
   functor (MemL : MemLookup) -> functor (Form :  FlexibleFormula) ->   sig
     val assign :
       Var.t ->
       Symbeval.varval ->
       (MemL.t, Form.t) Symbeval.ctx -> (MemL.t, Form.t) Symbeval.ctx
```

Assign a variable. Does not modify the entire context in place.

```
  end
```

Module that handles how Assignments are handled.

```
module Make :
   functor (MemL : MemLookup) -> functor (Tune :  EvalTune) -> functor (Assign :  Assign)
-> functor (Form :  FlexibleFormula) ->   sig
     module MemL :
     MemL
     module Assign :
     Assign(MemL)(Form)
     module Form :
     Form
     type myctx = (MemL.t, Form.t) Symbeval.ctx
     exception Halted of Symbeval.varval option * myctx
     exception UnknownLabel of Symbeval.label_kind
     exception Error of string * myctx
     exception AssertFailed of myctx
     exception AssumptionFailed of myctx
     val inst_fetch : ('a, 'b) Hashtbl.t -> 'a -> 'b
     val label_decode :
       (Symbeval.label_kind, 'a) Hashtbl.t -> Symbeval.label_kind -> 'a
     val lookup_var : MemL.t -> Var.t -> Symbeval.varval
     val update_var : MemL.t -> Var.t -> Symbeval.varval -> MemL.t
     val update_mem :
       Symbeval.varval -> Ast.exp -> Ast.exp -> Ast.exp -> Symbeval.varval
     val remove_var : MemL.t -> Var.t -> MemL.t
     val lookup_mem : Symbeval.varval -> Ast.exp -> Ast.exp -> Ast.exp
     val assign :
       Var.t ->
       Symbeval.varval ->
```

```
      (MemL.t, Form.t) Symbeval.ctx ->
      (MemL.t, Form.t) Symbeval.ctx
    val copy : MemL.t -> MemL.t
    val print_values : MemL.t -> unit
    val print_mem : MemL.t -> unit
    val eval_symb_let : bool
    val add_constraint : Form.t -> Ast.exp -> Symbeval.form_type -> Form.t
    val output_formula : Form.t -> Form.output
    val create_state : Form.init ->
      (MemL.t, Form.t) Symbeval.ctx
    val initialize_prog : ('a, 'b) Symbeval.ctx -> Symbeval.instr list -> unit
    val cleanup_delta : (MemL.t, 'a) Symbeval.ctx -> unit
    val build_default_context :
      Symbeval.instr list ->
      Form.init ->
      (MemL.t, Form.t) Symbeval.ctx
    val eval_expr : MemL.t -> Ast.exp -> Symbeval.varval
    val eval_stmt :
      (MemL.t, Form.t) Symbeval.ctx ->
      Ast.stmt -> (MemL.t, Form.t) Symbeval.ctx list
    val eval :
      (MemL.t, Form.t) Symbeval.ctx ->
      (MemL.t, Form.t) Symbeval.ctx list
    val eval_straightline :
      ?step:((MemL.t, Form.t) Symbeval.ctx ->
             (MemL.t, Form.t) Symbeval.ctx) ->
      (MemL.t, Form.t) Symbeval.ctx ->
      (MemL.t, Form.t) Symbeval.ctx list
```

Evaluate as long as there is exactly one choice of state.

```
  end

module MemVHBackEnd :
  sig
    type t = Symbeval.varval Symbeval.VH.t
    val copy : 'a Symbeval.VH.t -> 'a Symbeval.VH.t
    val clear : 'a Symbeval.VH.t -> 'a Symbeval.VH.t
    val create : unit -> 'a Symbeval.VH.t
    val fold :
      'a Symbeval.VH.t -> (Symbeval.VH.key -> 'a -> 'b -> 'b) -> 'b -> 'b
    val num_values : 'a Symbeval.VH.t -> int
```

Number of variable locations stored in state

```
    val num_mem_locs : Symbeval.varval Symbeval.VH.t -> int
```

Number of concrete memory locations stored in state

```
    val find_var : 'a Symbeval.VH.t -> Symbeval.VH.key -> 'a
    val update_var :
      'a Symbeval.VH.t -> Symbeval.VH.key -> 'a -> 'a Symbeval.VH.t
    val remove_var : 'a Symbeval.VH.t -> Symbeval.VH.key -> 'a Symbeval.VH.t
  end
module MemVMBackEnd :
  sig
    type t = Symbeval.varval Symbeval.VM.t
    val copy : 'a -> 'a
    val create : unit -> 'a Symbeval.VM.t
    val clear : 'a -> 'b Symbeval.VM.t
    val fold :
      'a Symbeval.VM.t -> (Symbeval.VM.key -> 'a -> 'b -> 'b) -> 'b -> 'b
    val num_values : 'a Symbeval.VM.t -> int
```

Number of variable locations stored in state

```
    val num_mem_locs : Symbeval.varval Symbeval.VM.t -> int
```

Number of concrete memory locations stored in state

```
    val find_var : 'a Symbeval.VM.t -> Symbeval.VM.key -> 'a
    val update_var :
      'a Symbeval.VM.t -> Symbeval.VM.key -> 'a -> 'a Symbeval.VM.t
    val remove_var : 'a Symbeval.VM.t -> Symbeval.VM.key -> 'a Symbeval.VM.t
  end
module type MemBackEnd =
  sig
    type t
    val copy : t -> t
    val clear : t -> t
    val create : unit -> t
    val fold : t -> (Ast.var -> Symbeval.varval -> 'a -> 'a) -> 'a -> 'a
    val num_values : t -> int
    val find_var : t -> Ast.var -> Symbeval.varval
    val update_var : t -> Ast.var -> Symbeval.varval -> t
    val remove_var : t -> Ast.var -> t
```

```
  end

module type Foldable =
  sig

     type t
     val fold : t -> (Ast.var -> Symbeval.varval -> 'a -> 'a) -> 'a -> 'a

  end

module BuildMemLPrinters :
   functor (F : Foldable) ->   sig

     val print_values : F.t -> unit
     val print_mem : F.t -> unit
     val print_var : F.t -> string -> unit

  end

module BuildSymbolicMemL :
   functor (BE : MemBackEnd) ->   sig

     include BE
     val lookup_var : t -> Ast.var -> Symbeval.varval
     val update_mem :
       Symbeval.varval -> Ast.exp -> Ast.exp -> Ast.exp -> Symbeval.varval
     val lookup_mem : Symbeval.varval -> Ast.exp -> Ast.exp -> Ast.exp

     include struct ...  end

  end

module SymbolicMemL :
   BuildSymbolicMemL(MemVHBackEnd)
module BuildConcreteMemL :
   functor (BE : MemBackEnd) ->   sig

     include BE
     val lookup_var : t -> Ast.var -> Symbeval.varval
     val update_mem :
       Symbeval.varval -> Ast.exp -> Ast.exp -> 'a -> Symbeval.varval
     val lookup_mem : Symbeval.varval -> Ast.exp -> 'a -> Ast.exp

     include struct ...  end

  end

module ConcreteMemL :
   BuildConcreteMemL(MemVHBackEnd)
module BuildConcreteUnknownZeroMemL :
   functor (BE : MemBackEnd) ->   sig
```

```
    include struct ...  end
    val lookup_mem : Symbeval.varval -> Ast.exp -> 'a -> Ast.exp
  end
module ConcreteUnknownZeroMemL :
   BuildConcreteUnknownZeroMemL(MemVHBackEnd)
module PredAssign :
   functor (MemL : MemLookup) -> functor (Form :  FlexibleFormula) ->   sig
    val assign :
      Ast.var ->
      Symbeval.varval ->
      (MemL.t, Form.t) Symbeval.ctx -> (MemL.t, Form.t) Symbeval.ctx
  end
```

Symbolic assigns are represented as Lets in the formula, except for temporaries. If you use this, you should clear out temporaries after executing each instruction.

```
module StdAssign :
   functor (MemL : MemLookup) -> functor (Form :  FlexibleFormula) ->   sig
    val assign :
      Var.t ->
      Symbeval.varval -> (MemL.t, 'a) Symbeval.ctx -> (MemL.t, 'a) Symbeval.ctx
  end
```

Symbolic assigns are represented in delta

```
module DontEvalSymbLet :
  sig
    val eval_symb_let : bool
  end
module EvalSymbLet :
  sig
    val eval_symb_let : bool
  end
module StdForm :
  sig
    type t = Ast.exp
    type init = unit
    type output = Ast.exp
    val init : unit -> Ast.exp
    val add_to_formula : Ast.exp -> Ast.exp -> 'a -> Ast.exp
    val output_formula : 'a -> 'a
```

```
  end
```

Just build a straightforward expression; does not use Lets

```
module LetBind :
  sig
     type t = Ast.exp -> Ast.exp
     type init = unit
     type output = Ast.exp
     val init : unit -> 'a -> 'a
     val add_to_formula :
       (Ast.exp -> 'a) -> Ast.exp -> Symbeval.form_type -> Ast.exp -> 'a
     val output_formula : (Ast.exp -> 'a) -> 'a
  end
```

Uses Lets for assignments, continuation style.

```
module LetBindOld :
  sig
     type f =
       | And of Ast.exp
       | Let of (Var.t * Ast.exp)
     type t = f list
     type init = unit
     type output = Ast.exp
     val init : unit -> 'a list
     val add_to_formula : f list ->
       Ast.exp -> Symbeval.form_type -> f list
     val output_formula : f list -> Ast.exp
  end
```

Uses Lets for assignments

```
module FlexibleFormulaConverterToStream :
   functor (Form :  FlexibleFormula  with type init=unit  with type output = Ast.exp)
->   sig
     type fp = Formulap.fpp_oc
     type t = {
       printer : Formulap.fpp_oc ;
       form_t : Form.t ;
     }
     type init = Formulap.fpp_oc
```

```
    type output = unit
    val init : Formulap.fpp_oc -> t
    val add_to_formula : t ->
      Ast.exp -> Symbeval.form_type -> t
    val output_formula : t -> unit
  end
module StdFormFakeStream :
   FlexibleFormulaConverterToStream(StdForm)
module LetBindFakeStream :
   FlexibleFormulaConverterToStream(LetBind)
module LetBindOldFakeStream :
   FlexibleFormulaConverterToStream(LetBindOld)
module LetBindStreamSat :
  sig
    type t = {
      formula_printer : Formulap.stream_fpp_oc ;
      formula_filename : string ;
      free_var_printer : Formulap.stream_fpp_oc ;
      free_var_filename : string ;
      close_funs : (unit -> unit) Stack.t ;
      mutable free_vars : Var.VarSet.t ;
      mutable defined_vars : Var.VarSet.t ;
    }
    type init = string * Formulap.stream_fppf
    type output = unit
    val init : string * Formulap.stream_fppf -> t
    val add_to_formula : t ->
      Ast.exp -> Symbeval.form_type -> t
    val output_formula : t -> unit
  end
```

Print let formulas in a streaming fashion. For example, given a trace: 1. x = 5 2. assert (x = 5) 3. y = 10

The formula (for smtlib) would look like: (and (let x = 5 in (x = 5) (let y = 10 in (true))))

```
module Symbolic :
   Make(SymbolicMemL)(DontEvalSymbLet)(StdAssign)(StdForm)
module SymbolicSlowMap :
   Make(BuildSymbolicMemL(MemVMBackEnd))(EvalSymbLet)(StdAssign)(StdForm)
module SymbolicSlow :
   Make(SymbolicMemL)(EvalSymbLet)(StdAssign)(StdForm)
```

```
module Concrete :
   Make(ConcreteUnknownZeroMemL)(EvalSymbLet)(StdAssign)(StdForm)
val concretely_execute :
  ?s:Type.addr ->
  ?i:Ast.stmt list -> instr list -> Concrete.myctx
```

Execute a program concretely

```
val eval_expr : Symbolic.MemL.t -> Ast.exp -> varval
```

73 Module Symbeval_search : A module to try out search strategies on symbolic execution

```
module D :
   Debug.Make(  sig
     val name : string
     val default : [> `Debug ]
  end )
module type Symb =
  sig
     module MemL :
       sig
         type t
       end
     module Form :
       sig
         type t
         type init = unit
         type output = Ast.exp
         val init : init -> t
         val add_to_formula : t ->
           Ast.exp -> Symbeval.form_type -> t
         val output_formula : t -> output
       end
     type myctx = (MemL.t, Form.t) Symbeval.ctx
     exception Halted of Symbeval.varval option * myctx
     exception Error of string * myctx
```

```
    exception UnknownLabel of Symbeval.label_kind
    exception AssertFailed of myctx
    exception AssumptionFailed of myctx
    val init : Ast.stmt list -> Form.init -> myctx
    val eval : myctx -> myctx list
    val eval_expr : MemL.t -> Ast.exp -> Symbeval.varval
  end

module type Strategy =
   functor (Symbolic :  Symb) ->   sig

    type myctx = Symbolic.myctx
    type t
    type data
    type initdata
    val start_at : myctx ->
      initdata -> t
    val pop_next : t ->
      ((myctx * data) *
       t)
      option
    val add_next_states : t ->
      myctx ->
      data ->
      myctx list -> t
  end

module NaiveSymb :
  sig

    include Symbeval.SymbolicSlow
    val init :
      Symbeval.instr list ->
      Symbeval.SymbolicSlow.Form.init ->
      (Symbeval.SymbolicSlow.MemL.t, Symbeval.SymbolicSlow.Form.t) Symbeval.ctx
    val eval :
      (Symbeval.SymbolicSlow.MemL.t, Symbeval.SymbolicSlow.Form.t) Symbeval.ctx ->
      (Symbeval.SymbolicSlow.MemL.t, Symbeval.SymbolicSlow.Form.t) Symbeval.ctx
      list
  end

module FastSymb :
  sig
```

```
    include Symbeval.Symbolic
    val init :
      Symbeval.instr list ->
      Symbeval.Symbolic.Form.init ->
      (Symbeval.Symbolic.MemL.t, Symbeval.Symbolic.Form.t) Symbeval.ctx
    val eval :
      (Symbeval.Symbolic.MemL.t, Symbeval.Symbolic.Form.t) Symbeval.ctx ->
      (Symbeval.Symbolic.MemL.t, Symbeval.Symbolic.Form.t) Symbeval.ctx list
  end

module MakeSearch :
   functor (S' : Strategy) -> functor (Symbolic :  Symb) ->   sig
    module S :
    S'(Symbolic)
    val search :
      Ast.exp ->
      Symbolic.Form.t list ->
      S.t -> Symbolic.Form.t list
    val eval_ast_program :
      S.initdata ->
      Ast.stmt list -> Ast.exp -> Symbolic.Form.output
  end

module Q :
  sig
    type 'a t = 'a list * 'a list
    val empty : 'a list * 'b list
    val enqueue : 'a * 'b list -> 'b -> 'a * 'b list
    val enqueue_all : 'a * 'b list -> 'b list -> 'a * 'b list
    val dequeue : 'a list * 'a list -> 'a * ('a list * 'a list)
    val dequeue_o : 'a list * 'a list -> ('a * ('a list * 'a list)) option
  end
```

A purely functional queue with amortised constant time enqueue and dequeue.

```
module UnboundedBFS :
   MakeSearch(functor (Symbolic :  Symb) ->   sig
    type myctx = Symbolic.myctx
    type t = Symbolic.myctx Symbeval_search.Q.t
    type data = unit
    type initdata = unit
```

```
    val start_at : 'a -> unit -> 'b list * 'a list
    val pop_next :
      'a list * 'a list -> (('a * unit) * ('a list * 'a list)) option
    val add_next_states : 'a * 'b list -> 'c -> unit -> 'b list -> 'a * 'b list
  end )
module UnboundedBFSNaive :
  UnboundedBFS(NaiveSymb)
module UnboundedBFSFast :
  UnboundedBFS(FastSymb)
val bfs_ast_program : Ast.stmt list -> Ast.exp -> NaiveSymb.Form.output
val bfs_ast_program_fast : Ast.stmt list -> Ast.exp -> FastSymb.Form.output
module MaxdepthBFS :
  MakeSearch(functor (Symbolic :  Symb) ->   sig
    type myctx = Symbolic.myctx
    type data = int
    type initdata = int
    type t = (myctx * data)
      Symbeval_search.Q.t
    val start_at : 'a -> 'b -> 'c list * ('a * 'b) list
    val pop_next : 'a list * 'a list -> ('a * ('a list * 'a list)) option
    val add_next_states :
      'a * ('b * int) list -> 'c -> int -> 'b list -> 'a * ('b * int) list
  end )
module MaxdepthBFSNaive :
  MaxdepthBFS(NaiveSymb)
module MaxdepthBFSFast :
  MaxdepthBFS(FastSymb)
val bfs_maxdepth_ast_program :
  MaxdepthBFSNaive.S.initdata ->
  Ast.stmt list -> Ast.exp -> NaiveSymb.Form.output
val bfs_maxdepth_ast_program_fast :
  MaxdepthBFSFast.S.initdata ->
  Ast.stmt list -> Ast.exp -> FastSymb.Form.output
module UnboundedDFS :
  MakeSearch(functor (Symbolic :  Symb) ->   sig
    type myctx = Symbolic.myctx
    type t = myctx list
    type data = unit
    type initdata = unit
```

```
    val start_at : 'a -> unit -> 'a list
    val pop_next : 'a list -> (('a * unit) * 'a list) option
    val add_next_states : 'a list -> 'b -> unit -> 'a list -> 'a list
  end )
module UnboundedDFSNaive :
  UnboundedDFS(NaiveSymb)
module UnboundedDFSFast :
  UnboundedDFS(FastSymb)
val dfs_ast_program : Ast.stmt list -> Ast.exp -> NaiveSymb.Form.output
val dfs_ast_program_fast : Ast.stmt list -> Ast.exp -> FastSymb.Form.output
module MaxdepthDFS :
  MakeSearch(functor (Symbolic :  Symb) ->   sig
    type myctx = Symbolic.myctx
    type data = int
    type initdata = int
    type t = (myctx * data) list
    val start_at : 'a -> 'b -> ('a * 'b) list
    val pop_next : 'a list -> ('a * 'a list) option
    val add_next_states :
      ('a * int) list -> 'b -> int -> 'a list -> ('a * int) list
  end )
module MaxdepthDFSNaive :
  MaxdepthDFS(NaiveSymb)
module MaxdepthDFSFast :
  MaxdepthDFS(FastSymb)
val dfs_maxdepth_ast_program :
  MaxdepthDFSNaive.S.initdata ->
  Ast.stmt list -> Ast.exp -> NaiveSymb.Form.output
val dfs_maxdepth_ast_program_fast :
  MaxdepthDFSFast.S.initdata ->
  Ast.stmt list -> Ast.exp -> FastSymb.Form.output
module EdgeMap :
  Map.Make(  sig
    type t = Big_int_Z.big_int * Big_int_Z.big_int
    val compare : 'a -> 'a -> int
  end )
module MaxrepeatDFS :
  MakeSearch(functor (Symbolic :  Symb) ->   sig
```

```
    type myctx = Symbolic.myctx
    type data = int Symbeval_search.EdgeMap.t
    type initdata = int
    type t = (myctx * data) list *
      int
    val start_at : 'a -> 'b -> ('a * 'c Symbeval_search.EdgeMap.t) list * 'b
    val pop_next : 'a list * 'b -> ('a * ('a list * 'b)) option
    val add_next_states :
      (('a, 'b) Symbeval.ctx * int Symbeval_search.EdgeMap.t) list * int ->
      ('c, 'd) Symbeval.ctx ->
      int Symbeval_search.EdgeMap.t ->
      ('a, 'b) Symbeval.ctx list ->
      (('a, 'b) Symbeval.ctx * int Symbeval_search.EdgeMap.t) list * int
  end )

module MaxrepeatDFSNaive :
   MaxrepeatDFS(NaiveSymb)
module MaxrepeatDFSFast :
   MaxrepeatDFS(FastSymb)
val maxrepeat_ast_program :
  MaxrepeatDFSNaive.S.initdata ->
  Ast.stmt list -> Ast.exp -> NaiveSymb.Form.output
val maxrepeat_ast_program_fast :
  MaxrepeatDFSFast.S.initdata ->
  Ast.stmt list -> Ast.exp -> FastSymb.Form.output
```

74 Module `Syscall_id` : Statically identify obvious use of system call numbers on AST CFGs.

Finds uses like `mov $20, %eax. sysenter` that a compiler would emit.

Assumes that `%eax` contains the system call number before the system call is executed.

Author(s): ejs

```
val find_syscalls :
  Cfg.AST.G.t -> (Cfg.AST.G.V.t * Big_int_Z.big_int list) list
```

find_syscalls `cfg` returns a list of tuples `(bb, n)`. A tuple `(bb, n)` in the output list denotes that *all* executions reaching `bb` perform system call `n`.

The non-existance of a tuple `(bb, n)` does not imply that an execution reaching `bb` will *not* perform system call `n`; it only means that the static analysis was unable to prove *all* executions reaching `bb` will perform system call `n`.

75 Module Syscall_models : A module with IL models of system calls.

```
module R32 :
   Asmir_vars.X86.R32
module R64 :
   Asmir_vars.X86.R64
```

We are going to index each system call model by the value of eax. Because this is going to be quite time-consuming we will use (for now) specials for system call models that are unimplemented.

```
val x86_is_system_call : Ast.stmt -> bool
val syscall_reg : Arch.arch -> Var.t
val linux_get_name : Arch.arch -> int -> string
val linux_get_model : Arch.arch -> int -> Ast.stmt list option
val linux_syscall_to_il : Arch.arch -> int -> Ast.stmt list
```

76 Module Template

```
module D :
   Debug.Make(  sig
     val name : string
     val default : [> `Debug ]
  end )
module type Types =
  sig
     type environ
     type state
     type result_exp
     type result_stmt
  end
module Abstract :
   functor (Typ :  Types) ->   sig
     type process_exp = Typ.environ -> Typ.state -> Typ.result_exp
     type process_stmt = Typ.environ -> Typ.state -> Typ.result_stmt
     module type Exprs =
       sig
```

```
    val var : Var.t -> Template.Abstract.process_exp
    val int : Big_int_Z.big_int * Type.typ -> Template.Abstract.process_exp
    val lab : string -> Template.Abstract.process_exp
    val ite : Ast.exp * Ast.exp * Ast.exp -> Template.Abstract.process_exp
    val extract :
      Big_int_Z.big_int * Big_int_Z.big_int * Ast.exp ->
      Template.Abstract.process_exp
    val concat : Ast.exp * Ast.exp -> Template.Abstract.process_exp
    val binop :
      Type.binop_type * Ast.exp * Ast.exp -> Template.Abstract.process_exp
    val unop : Type.unop_type * Ast.exp -> Template.Abstract.process_exp
    val cast :
      Type.cast_type * Type.typ * Ast.exp -> Template.Abstract.process_exp
    val lett : Var.t * Ast.exp * Ast.exp -> Template.Abstract.process_exp
    val load :
      Ast.exp * Ast.exp * Ast.exp * Type.typ -> Template.Abstract.process_exp
    val store :
      Ast.exp * Ast.exp * Ast.exp * Ast.exp * Type.typ ->
      Template.Abstract.process_exp
    val unknown : 'a -> Template.Abstract.process_exp
  end

module type Stmts =
  sig

    val move : Var.t * Ast.exp * Ast.attrs -> Template.Abstract.process_stmt
    val halt : Ast.exp * Ast.attrs -> Template.Abstract.process_stmt
    val jmp : Ast.exp * Ast.attrs -> Template.Abstract.process_stmt
    val cjmp :
      Ast.exp * Ast.exp * Ast.exp * Ast.attrs -> Template.Abstract.process_stmt
    val assertt : Ast.exp * Ast.attrs -> Template.Abstract.process_stmt
    val assume : Ast.exp * Ast.attrs -> Template.Abstract.process_stmt
    val comment : 'a -> Template.Abstract.process_stmt
    val label : 'a -> Template.Abstract.process_stmt
    val special :
      string * Var.defuse option * Ast.attrs -> Template.Abstract.process_stmt
  end

module Make :
functor (Expr :  Exprs) -> functor (Stmt :  Stmts) ->   sig
```

```
        val expr : Ast.exp -> Template.Abstract.process_exp
        val stmt : Ast.stmt -> Template.Abstract.process_stmt
      end
  end
```

77 Module Test_common : Functions used by BAP library and tests

```
exception RangeNotFound of Type.addr * Type.addr
val leave_files : bool Pervasives.ref
val speclist : (string * Arg.spec * string) list
val check_file : string -> unit
```

General system functions *

```
val mkdir_and_ignore : string -> unit
val rm_and_ignore : string -> unit
val rm_and_ignore_list : string list -> unit
val stp : string
```

STP helpers *

```
val does_stp_work : unit -> bool
module SolverCheck :
   functor (S : Smtexec.SOLVER) ->   sig
     val check_solver_path : unit -> unit
  end
val check_stp_path : unit -> unit
val pin_path : string Pervasives.ref
```

pin helpers *

```
val pin : string
val gentrace_path_64 : string
val gentrace_path_32 : string
val gentrace_path_of_arch : Arch.arch -> string
val gentrace : string
val pin_out_suffix : string
val find_pin_out : string list -> string -> string
val check_pin_setup : Arch.arch -> unit
val get_arch : string -> Arch.arch
```

Get architecture of binary

```
val run_trace :
  ?arch:Arch.arch ->
  ?tag:string ->
  ?pin_args:string list ->
  ?pintool_args:string list -> string -> string list -> string
val find_funs : ?msg:string -> ('a * 'b * 'c) list -> 'a list -> 'b * 'c
```

Common functions across multiple tests *

```
val find_fun : ?msg:string -> ('a * 'b * 'c) list -> 'a -> 'b * 'c
val find_call : Ast.stmt list -> Type.addr
val inject_stmt :
  Ast.stmt list -> Type.addr -> string -> Ast.stmt -> Ast.stmt list
val halt_stmt : Ast.stmt
val check_bigint_answer : Ast.exp -> Z.t -> unit
val check_eax :
  (Symbeval.varval Var.VarHash.t, 'a) Symbeval.ctx -> Z.t -> unit
val check_functions : string -> ('a * 'b * 'c) list -> 'a list -> unit
val typecheck : Ast.stmt list -> unit
val find_prog_chunk :
  Ast.stmt list -> Type.addr -> Type.addr -> Ast.stmt list
val summarize : OUnit.test_result -> unit
val summarize_results : OUnit.test_result list -> 'a option
val backwards_taint : Arch.arch -> Ast.program -> Traces_backtaint.LocSet.t
```

78 Module `To_c` : C output.

TODO: Make the interface consistent with that of Pp.

```
module VH :
   Var.VarHash
class pp : ?debug_labels:bool -> Format.formatter ->
  object
     val special : string
     val mem_load : string
     val mem_store : string
     method ast_program : ?name:string -> Ast.stmt list -> unit
```

Pretty print a program using the given formater

```
    method format_exp : ?prec:int -> Ast.exp -> unit
    method ast_stmt : Ast.stmt -> unit
    method ast_stmts : Ast.stmt list -> unit
    method format_typ : Type.typ -> unit
    method format_initializations : Ast.var list -> unit
    method ast_exp : ?prec:int -> Ast.exp -> unit
    method format_var : ?print_type:bool -> Ast.var -> unit
    method binop_to_string : Type.binop_type -> string
    method typ : Type.typ -> unit
    method format_name : Ast.var -> unit
    method format_value : Ast.exp -> unit
    method format_cast :
      (int -> Ast.exp -> unit) -> Type.cast_type -> Type.typ -> Ast.exp -> unit
  end
```

79 Module Traces : A module to perform trace analysis

```
module D :
   Debug.Make(  sig

     val name : string
     val default : [> `NoDebug ]

  end )

module DV :
   Debug.Make(  sig

     val name : string
     val default : [> `NoDebug ]

  end )
```

So here's how we will do partial symbolic execution on traces: 1. A trace is a list of AST stmts as executed by the program 2. Execute the trace and at each instruction:

a) check if it is a taint introduction stmt b) if it is, update the memory context with the symbolic variables c) If it a regular stmt, read the new concrete values and taint flags and store them in a map d) whenever the symbolic evaluator requests a value that is known and untainted, provide it with the value from the map

- if it is tainted let the evaluator worry about it

```
val consistency_check : bool Pervasives.ref
```

Optional consistency check between trace and bap evaluation. Tainted values should always be equal in the BAP evaluation and the trace. Non-tainted values do not have to match, since their values are assumed to be constant.

```
val checkall : bool Pervasives.ref
```

Option used to force checking of an entire trace.

```
type stmt_info = {
  assignstmt : Ast.stmt ;
  assigned_time : int ;
}
```

For each register, map it to an assignment and record the value of !current_time when the assignment happened.

```
val current_time : int Pervasives.ref
```

Current time in executor (number of statements executed)

```
val last_special : Symbeval.instr Pervasives.ref
```

Last special concretely executed

```
val last_time : int Pervasives.ref
```

The time that the last special was assigned at

```
val reg_to_stmt : (Symbeval.VH.key * int option, stmt_info) Hashtbl.t
```

Map each register to the assembly instruction that set it. Useful for interpreting consistency failures.

```
val dce : bool Pervasives.ref
val allow_symbolic_indices : bool Pervasives.ref
val padding : bool Pervasives.ref
val memtype : Arch.arch -> Type.typ
val endtrace : string
val tassignattr : Type.attribute
val dontkeepattr : Type.attribute
```

Statements we want to execute during concrete execution, but do not want to keep in the final output

```
type value = {
  exp : Ast.exp ;
  usg : Type.usage ;
  tnt : bool ;
}
type environment = {
  vars : (string, value) Hashtbl.t ;
```

```
  memory : (Big_int_Z.big_int, value) Hashtbl.t ;
  symbolic : (Big_int_Z.big_int, Ast.exp) Hashtbl.t ;
  symbolicvar : (int, Ast.exp) Hashtbl.t ;
}
val global : environment
val gamma : Arch.arch -> Asmir.varctx
```

Create a lookup for our variables

```
val name_to_var : Arch.arch -> string -> Var.t option
```

Convert name of a register to a var for that register

```
val var_lookup : string -> value option
val mem_lookup : Big_int_Z.big_int -> value option
val sym_lookup : int -> Ast.exp
val dsa_rev_map : Symbeval.VH.key Symbeval.VH.t option Pervasives.ref
```

This is a map from DSA variable names to standard variable names.

```
val dsa_var : Symbeval.VH.key -> Symbeval.VH.key option
```

Convert a DSA var to a normal var

```
val dsa_orig_name : Symbeval.VH.key -> string option
```

Get original variable name from DSA var

```
val dsa_var_lookup : Symbeval.VH.key -> value option
```

Looks for concrete value information by a DSA name

```
val concrete_val : string -> Ast.exp option
val dsa_concrete_val : Symbeval.VH.key -> Ast.exp option
val concrete_mem : Big_int_Z.big_int -> Ast.exp option
val symbolic_mem : Big_int_Z.big_int -> Ast.exp
val taint_val : string -> bool option
val dsa_taint_val : Symbeval.VH.key -> bool option
val taint_mem : Big_int_Z.big_int -> bool option
val bound : string -> bool
val in_memory : Big_int_Z.big_int -> bool
val add_var : string -> Ast.exp -> Type.usage -> bool -> unit
val add_mem : Big_int_Z.big_int -> Ast.exp -> Type.usage -> bool -> unit
val add_symbolic : Big_int_Z.big_int -> Ast.exp -> unit
val add_new_var : string -> Ast.exp -> Type.usage -> bool -> unit
val del_var : string -> unit
val dsa_del_var : Symbeval.VH.key -> unit
```

```
val del_mem : Big_int_Z.big_int -> unit
val del_symbolic : Big_int_Z.big_int -> unit
val cleanup : unit -> unit
val conc_mem_fold : (Big_int_Z.big_int -> value -> 'a -> 'a) -> 'a -> 'a
val conc_mem_iter : (Big_int_Z.big_int -> value -> unit) -> unit
val symb_re : Str.regexp
val get_symb_num : Var.t -> int
val bytes_from_file : string -> int list
val badregs : (string, unit) Hashtbl.t
```

A list of registers that we will never check for correctness.

```
val checkallregs : (string, unit) Hashtbl.t
```

A list of registers that we should only check when checking all instructions. This includes EFLAGS registers because we will not consider an instruction tainted if the only operand that is tainted is EFLAGS.

```
val isnotallbad : Var.t -> bool
val isbad : Var.t -> bool
val print_vars : unit -> unit
val assert_vars : Arch.arch -> 'a Symbeval.VH.t -> Ast.stmt
```

Build a statement asserting that each operand is equal to its value in the trace

```
val assign_vars :
  Arch.arch -> Ast.var -> (Var.t -> Ast.var) -> bool -> Ast.stmt list
```

Build statements assigning concrete operands

```
val symbtoexp : Symbeval.varval -> Ast.exp
```

Get the expression for a symbolic value

```
val typ_to_bytes : Type.typ -> int
val get_byte : int -> Z.t -> Z.t
val num_to_bit : Z.t -> Z.t
val keep_taint : Type.attribute -> bool
val unwrap_taint : Type.attribute -> Type.context
val filter_taint : Type.attribute list -> Type.context list
val get_int : Ast.exp -> Z.t
val taint_to_bool : int -> bool
val hd_tl : 'a list -> 'a * 'a list
val is_mem : Var.t -> bool
val is_temp : Var.t -> bool
val is_tconcassign : Type.attribute -> bool
```

```
val is_symbolic : Var.t -> bool
val clean_delta : 'a Symbeval.VH.t -> unit
```
remove temporaries from delta

```
val add_eflags : Z.t -> Type.usage -> bool -> unit
val allvars : Ast.program -> Symbeval.VH.key list
```
Get the vars used in a program

```
val find_memv : Ast.program -> Symbeval.VH.key
val regs : (string, string * int * Type.typ) Hashtbl.t
val is_seed_label : string -> bool
val get_tid : Ast.stmt list -> int option
module ThreadVar :
  sig
     type t = int * Var.t
     val hash : 'a * Var.t -> int
     val equal : 'a -> 'a -> bool
     val compare : 'a * Var.t -> 'a * Var.t -> int
  end
```
Support muti-threaded traces by seperating variables per threadId

```
module TVMake :
   Hashtbl.Make(ThreadVar)
module TVHash :
  sig
     include TVMake
     include struct ...  end
  end
val create_thread_map_state : unit -> Var.t TVHash.t
val lookup_thread_map : Var.t TVHash.t -> int option -> Var.t -> Var.t
val explicit_thread_stmts_int :
  int option -> Ast.program -> Ast.var TVHash.t -> Ast.program
val explicit_thread_stmts : Ast.program -> Ast.var TVHash.t -> Ast.program
```
Rename variables so they are unique to the thread they are found in

```
val add_to_conc : Type.context -> unit
val update_concrete : Ast.stmt -> bool
val get_first_atts : Ast.stmt list -> Type.context list * Ast.stmt list
```
Fetching the first stmt with attributes containing taint info

Initializing the trace contexts

```
val remove_jumps : Ast.stmt list -> Ast.stmt list
```

Removing all jumps from the trace

```
val is_loaded : Ast.stmt -> bool
```

Detect 'loaded module' specials

```
val remove_loaded : Ast.stmt list -> Ast.stmt list
```

Remove 'Loaded module' specials

```
val remove_specials : Ast.program -> Symbeval.instr list
```

Removing all specials from the traces

```
val append_halt : Ast.stmt list -> Ast.stmt list
val trace_to_blocks : Ast.stmt list -> Ast.stmt list list
```

A trace is a sequence of instructions. This function takes a list of ast statements and returns a list of lists of ast stmts. Each one of those sublists will represent the IL of the executed assembly instruction

Strips the last jump of the block

```
val print : ('a, Pervasives.out_channel, unit) Pervasives.format -> 'a
val print_block : Ast.stmt list -> unit
val trace_length : 'a list -> unit
module Status :
   Util.StatusPrinter
val trace_dce : Ast.stmt list -> Ast.stmt list
```

In a nutshell, remove any computation that is not referenced inside of an Assert.

```
module TraceConcreteDef :
  sig
     include Symbeval.MemVHBackEnd
     val lookup_var :
       Symbeval.varval Symbeval.VH.t -> Symbeval.VH.key -> Symbeval.varval
     val normalize : Z.t -> Type.typ -> Z.t
     val update_mem :
       Symbeval.varval -> Ast.exp -> Ast.exp -> Ast.exp -> Symbeval.varval
     val lookup_mem : Symbeval.varval -> Ast.exp -> 'a -> Ast.exp
     include struct ...  end
  end
module TraceConcreteAssign :
   functor (MemL : Symbeval.MemLookup) -> functor (Form :  Symbeval.FlexibleFormula)
->   sig
```

```
    val assign :
      Symbeval.VH.key ->
      Symbeval.varval -> (MemL.t, 'a) Symbeval.ctx -> (MemL.t, 'a) Symbeval.ctx
  end

module TraceConcrete :
   Symbeval.Make(TraceConcreteDef)(Symbeval.EvalSymbLet)(TraceConcreteAssign)(Symbeval.StdForm)
val check_delta :
  (Symbeval.varval Symbeval.VH.t, 'a) Symbeval.ctx ->
  int option -> TraceConcrete.MemL.t
```

Check all variables in delta to make sure they agree with operands loaded from a trace. We should be able to find bugs in BAP and the trace code using this.

```
val counter : int Pervasives.ref
val get_symbolic_seeds :
  Arch.arch -> Ast.var -> Ast.stmt -> Ast.stmt list * Ast.stmt list
val trace_transform_stmt : Ast.stmt -> (Ast.exp -> Ast.exp) -> Ast.stmt list
```

Transformations needed for traces.

```
val get_next_label : Ast.stmt list list -> Type.addr option
val run_block :
  Arch.arch ->
  ?next_label:Z.t option ->
  ?transformf:(Symbeval.instr -> (Ast.exp -> Ast.exp) -> Symbeval.instr list) ->
  (TraceConcrete.MemL.t, TraceConcrete.Form.t) Symbeval.ctx ->
  Ast.var -> Ast.var TVHash.t -> Ast.program -> Symbeval.instr list
```

Running each block separately

```
val run_blocks :
  Arch.arch -> Ast.program list -> Ast.var -> int -> Ast.stmt list
val to_dsa_stmt :
  Ast.stmt ->
  Symbeval.VH.key Symbeval.VH.t -> Symbeval.VH.key Symbeval.VH.t -> Ast.stmt
```

Convert a stmt to DSA form

Returns The DSA'ified statement

```
val to_dsa : Ast.stmt list -> Ast.stmt list * Symbeval.VH.key Symbeval.VH.t
```

Convert a trace to DSA form

Returns A tuple of the DSA version of the program and a hash table mapping DSA vars to the original vars.

```
val concrete : Arch.arch -> Ast.program -> Ast.stmt list
```

Perform concolic execution on the trace and output a set of constraints

```
module TaintConcreteDef :
  sig
    include Symbeval.MemVHBackEnd
    include Symbeval.Symbolic
    val bytes : int array Pervasives.ref
    val conc2symb : 'a -> 'b -> 'c
    val lookup_var :
      Symbeval.varval Symbeval.VH.t -> Symbeval.VH.key -> Symbeval.varval
    val lookup_mem : Symbeval.varval -> Ast.exp -> 'a -> Ast.exp
  end
module TaintConcrete :
   Symbeval.Make(TaintConcreteDef)(Symbeval.DontEvalSymbLet)(Symbeval.StdAssign)(Symbeval.StdFor
val concrete_rerun : string -> Symbeval.instr list -> Symbeval.instr list
```

Concretely execute a trace without using any operand information

```
val formula_size : Ast.exp -> Z.t
module PredAssignTraces :
   functor (MemL : Symbeval.MemLookup) -> functor (Form :  Symbeval.FlexibleFormula)
->   sig
    val assign :
      Symbeval.VH.key ->
      Symbeval.varval ->
      (MemL.t, Form.t) Symbeval.ctx -> (MemL.t, Form.t) Symbeval.ctx
  end
```

Symbolic assigns are represented as Lets in the formula, except for temporaries. If you use this, you should clear out temporaries after executing each instruction.

This should be exactly the same as Symbeval.PredAssign, except that when we call is_temp we need to use the non-dsa variable name.

```
module type FormulaAdapter =
  sig
    type user_init
    type form_init
    val adapt : user_init -> form_init
  end
```

Modules that convert user_init data to a FlexibleFormula's init type

```
module type TraceSymbolic =
  sig
```

```
    type user_init
    type init
    type output
    type state
    val create_state : user_init -> state
    val symbolic_run : user_init ->
      Arch.arch -> Ast.stmt list -> state
    val symbolic_run_blocks : state ->
      Arch.arch -> Ast.stmt list -> state
    val generate_formula : user_init ->
      Arch.arch -> Ast.stmt list -> output
    val output_formula : state -> output
    val formula_valid_to_invalid : Arch.arch -> ?min:int -> Ast.stmt list -> unit
    val trace_valid_to_invalid : Arch.arch -> Ast.stmt list -> unit
    val output_exploit : user_init -> Arch.arch -> Ast.stmt list -> unit
  end

module MakeTraceSymbolic :
   functor (Tune :  Symbeval.EvalTune) -> functor (Assign :  Symbeval.Assign) -> functor
(FormAdapt :  FormulaAdapter) -> functor (Form :  FlexibleFormula  with type init = FormAdapt.fo
->   sig
    module SymbolicEval :
    Symbeval.Make(Symbeval.SymbolicMemL)(Tune)(Assign)(Form)
    type user_init = FormAdapt.user_init
    type init = Form.init
    type output = Form.output
    type state = {
      symstate : SymbolicEval.myctx ;
      h : Var.t Var.VarHash.t ;
      rh : Var.t Var.VarHash.t ;
    }
    val status : int Pervasives.ref
    val count : int Pervasives.ref
    val create_state : FormAdapt.user_init -> state
    val symbolic_run_block : Arch.arch ->
      state -> Ast.stmt -> state
    val symbolic_run_blocks : state ->
      Arch.arch -> Ast.program -> state
    val symbolic_run :
      FormAdapt.user_init ->
```

```
      Arch.arch -> Ast.stmt list -> state
    val generate_formula :
      FormAdapt.user_init -> Arch.arch -> Ast.program -> Form.output
    val output_formula : state -> Form.output
  end

type standard_user_init = string * Smtexec.smtexec
module MakeTraceSymbolicStandard :
   functor (Tune :  Symbeval.EvalTune) -> functor (Assign :  Symbeval.Assign) -> functor
(FormAdapt :  FormulaAdapter  with type user_init = standard_user_init) -> functor (Form
:  FlexibleFormula  with type init = FormAdapt.form_init  with type output = unit) ->
sig
    module TraceSymbolic :
    Traces.MakeTraceSymbolic(Tune)(Assign)(FormAdapt)(Form)
    include TraceSymbolic
    val formula_valid_to_invalid : Arch.arch -> ?min:int -> Ast.program -> unit
    val trace_valid_to_invalid : Arch.arch -> Ast.stmt list -> unit
    val formula_storage : string
    val answer_storage : string
    val parse_answer_to : (string * Z.t) list option -> string -> unit
    val output_exploit :
      string * Smtexec.smtexec -> Arch.arch -> Ast.program -> unit
  end

module StreamPrinterAdapter :
  sig
    type user_init = Traces.standard_user_init
    type form_init = Symbeval.LetBindStreamSat.init
    val adapt : user_init -> string * Formulap.stream_fppf
  end

module OldPrinterAdapter :
  sig
    type user_init = Traces.standard_user_init
    type form_init = Formulap.fpp_oc
    val adapt : user_init -> Formulap.fpp_oc
  end

module TraceSymbolic :
   MakeTraceSymbolicStandard(Symbeval.DontEvalSymbLet)(PredAssignTraces)(OldPrinterAdapter)(Symb
module TraceSymbolicStream :
```

```
   MakeTraceSymbolicStandard(Symbeval.DontEvalSymbLet)(PredAssignTraces)(StreamPrinterAdapter)(S
val shellcode : string
val winshellcode : string
val nop : char
val nopsled : int -> string
val pin_offset : int64
val get_last_jmp_exp : Ast.stmt list -> (Ast.exp * Ast.attrs) * Ast.stmt list
val hijack_control : Ast.exp -> Ast.stmt list -> Ast.stmt list * Ast.stmt
val control_flow :
  Big_int_Z.big_int -> Arch.arch -> Ast.stmt list -> Ast.stmt list
val limited_control : Arch.arch -> Ast.stmt list -> Ast.stmt list
val get_last_load_exp : Ast.stmt list -> Ast.exp * Ast.exp * Ast.stmt list
```

Return the last load expression

```
val inject_payload_gen :
  Arch.arch -> Ast.exp -> int list -> Ast.program -> Ast.stmt list
```

Injecting a payload at an exp

XXX: Consolidate other payload functions to use this one.

```
val inject_payload :
  Arch.arch ->
  Big_int_Z.big_int ->
  int list -> Ast.stmt list -> Ast.stmt list * Ast.stmt list
val string_to_bytes : string -> int list
val add_payload :
  Big_int_Z.big_int -> string -> Arch.arch -> Ast.stmt list -> Ast.stmt list
val add_payload_after :
  Big_int_Z.big_int -> string -> Arch.arch -> Ast.stmt list -> Ast.stmt list
val add_payload_from_file :
  Big_int_Z.big_int -> string -> Arch.arch -> Ast.stmt list -> Ast.stmt list
val add_payload_from_file_after :
  Big_int_Z.big_int -> string -> Arch.arch -> Ast.stmt list -> Ast.stmt list
exception Found_load of Ast.exp
val inject_shellcode : int -> Arch.arch -> Ast.stmt list -> Ast.stmt list
val add_pivot :
  Big_int_Z.big_int ->
  Big_int_Z.big_int -> string -> Arch.arch -> Ast.stmt list -> Ast.stmt list
```

Use pivot to create exploit

```
val add_pivot_file :
  Big_int_Z.big_int ->
  Big_int_Z.big_int -> string -> Arch.arch -> Ast.stmt list -> Ast.stmt list
```

Use pivot to create exploit

```
val add_seh_pivot :
  Big_int_Z.big_int ->
  Big_int_Z.big_int ->
  Big_int_Z.big_int -> string -> Arch.arch -> Ast.program -> Ast.stmt list
```

Transfer control by overwriting sehaddr with gaddr.

```
val add_seh_pivot_file :
  Big_int_Z.big_int ->
  Big_int_Z.big_int ->
  Big_int_Z.big_int -> string -> Arch.arch -> Ast.program -> Ast.stmt list
```

Transfer control by overwriting sehaddr with gaddr.

```
val add_assignments : Ast.stmt list -> Ast.stmt list
```

80 Module `Traces_backtaint` : Backwards taint analysis on traces.

Given a list of data sink locations, computes the set of data source locations whose data flowed to user specific sinks.

Author(s): Brian Pak

```
module Loc :
  sig
```

```
    type t =
      | V of Var.t
      | M of Big_int_Z.big_int
```

```
    val compare : t -> t -> int
```

```
  end
```

A location is a register (BAP variable) or a memory location (address)

```
module LocSet :
   Set.S  with type elt = Loc.t
```

```
val backwards_taint : Ast.stmt list -> LocSet.t -> LocSet.t
```

`backwards_taint t sinkset` returns the set of all source locations (input bytes) from which data flowed to at least one location in `sinkset` in the trace `t`.

For example, if `sinkset` is a faulting operand, the function will return the list of input bytes whose data contributed to that operand.

Note: This function performs taint analysis, which is a data analysis, and does not capture control dependencies, which can also be used to affect program values.

```
val print_locset : LocSet.t -> unit
```

Prints the contents of a location set to `stdout`.

```
val exp_to_locset : Ast.exp -> LocSet.t
```

`exp_to_locset` returns a location set that contains the locations of all variables and loaded memory addresses referenced in the expression `e`.

```
val identify_fault_location : Ast.program -> LocSet.t
```

Given a trace of a crash, try to identify the faulting location. The algorithm for this is simple: Look for the last indirect jump or memory operation. If this operation is an indirect jump, the target is the faulting location. For a memory reference, the addresses are the faulting location.

81 Module `Traces_stream` : Functions for streaming processing of traces.

Author(s): Spencer Whitman, Edward J. Schwartz

```
val concrete : bool -> Arch.arch -> Ast.program -> Ast.program
```

`concrete ret` returns a streaming function for concretely executing a trace. If `ret` is true, the transformed trace is return. If false, the empty program is returned.

```
val generate_formula :
  string ->
  Smtexec.smtexec -> (Arch.arch -> Ast.program -> unit) * (unit -> unit)
```

`generate_formula f solver` returns two functions for symbolically executing a program and outputting the formula in `f` using `solver`. The first function is a streaming function that does the symbolic execution. The second function finishes outputting the formula to file.

82 Module `Traces_surgical` : Transformations needed for traces.

```
module D :
   Traces.D
module NameSet :
   Set.Make(String)
val mem_name : int64 -> string
val used_vars : (int64, Ast.var) Hashtbl.t
val newvar : int64 -> Type.typ -> Ast.var
val trace_transform_stmt2 : Ast.stmt -> (Ast.exp -> Ast.exp) -> Ast.stmt list
```

Transformations needed for traces.

```
val get_symbolic_seeds2 : 'a -> Ast.stmt -> Ast.stmt list
val run_and_subst_block :
  Arch.arch ->
  (Traces.TraceConcrete.MemL.t, Traces.TraceConcrete.Form.t) Symbeval.ctx ->
  Ast.var ->
  Ast.var Traces.TVHash.t -> Symbeval.instr list -> Symbeval.instr list
```

Running each block separately

```
val run_and_subst_blocks :
  Arch.arch ->
  Symbeval.instr list list ->
  Ast.var -> Ast.var Traces.TVHash.t -> int -> Symbeval.instr list
val dicer : NameSet.t -> Ast.stmt list -> Ast.stmt list -> Ast.stmt list
val concrete_substitution : Arch.arch -> Ast.program -> Symbeval.instr list
val check_slice : Arch.arch -> Ast.program -> Symbeval.instr list
val slice : NameSet.elt -> Ast.stmt list -> Ast.stmt list
```

83 Module Tunegc : Automatically tune Garbage collection parameters.

Decrease the aggressiveness of OCaml's garbage collector, since BAP often uses very large objects.

```
val set_gc : unit -> unit
```

`set_gc ()` tunes the garbage collection parameters so they are more appropriate for BAP.

84 Module Type : Type declarations for BAP.

Author(s): Ivan Jager, Ivan Jager

```
type addr = Big_int_Z.big_int
```

Addresses are big_ints

```
type label =
  | Name of string
```

For named labels

```
  | Addr of addr
```

For addresses. Cast REG_type as unsigned when comparing.

Labels are program locations that can be jumped to.

```
type typ =
  | Reg of int
```

an N-bit bitvector (use 1 for booleans).

| `TMem of typ * typ`

Memory of given index type, element type.

| `Array of typ * typ`

Array of index type, element type.

The IR type of a BAP expression

```
val reg_1 : typ
val reg_8 : typ
val reg_16 : typ
val reg_32 : typ
val reg_64 : typ
val reg_128 : typ
val reg_256 : typ
```

`Array` memories can only be updated or accessed in terms of their element type, which is usually `Reg 8`. `TMem` memories can be updated or accessed by any type, for instance both `Reg 8` and `Reg 32`. Native code is generally lifted with `TMem` memories for simplicity, and converted to `Array` type when needed. SMT solvers require `Array` type memories.

```
type cast_type =
```

| `CAST_UNSIGNED`

0-padding widening cast.

| `CAST_SIGNED`

Sign-extending widening cast.

| `CAST_HIGH`

Narrowning cast. Keeps the high bits.

| `CAST_LOW`

Narrowing cast. Keeps the low bits.

Different forms of casting

```
type binop_type =
```

| `PLUS`

Integer addition. (commutative, associative)

| `MINUS`

Subtract second integer from first.

| `TIMES`

Integer multiplication. (commutative, associative)

| `DIVIDE`

Unsigned integer division.

| SDIVIDE

Signed integer division.

| MOD

Unsigned modulus.

| SMOD

Signed modulus.

| LSHIFT

Left shift.

| RSHIFT

Right shift, fill with 0.

| ARSHIFT

Right shift, sign extend.

| AND

Bitwise and. (commutative, associative)

| OR

Bitwise or. (commutative, associative)

| XOR

Bitwise xor. (commutative, associative)

| EQ

Equals (commutative) (associative on booleans)

| NEQ

Not equals (commutative) (associative on booleans)

| LT

Unsigned less than

| LE

Unsigned less than or equal to

| SLT

Signed less than

| SLE

Signed less than or equal to

Binary operations implemented in the IR

```
type unop_type =
```

| NEG

Negate (2's complement)

| NOT

Bitwise not

Unary operations implemented in the IR

```
type pos = string * int
```

The position of a statement in a source file

Extra attributes

```
type taint_type =
  | Taint of int
type usage =
  | RD
  | WR
  | RW
type context = {
  name : string ;
  mem : bool ;
  t : typ ;
  index : addr ;
  value : Big_int_Z.big_int ;
  usage : usage ;
  taint : taint_type ;
}
```

Information about a concrete operand from a trace

```
type attribute =
  | Pos of pos
```

The position of a statement in the source file

```
  | Asm of string
```

Assembly representation of the following IL code

```
  | Address of addr
```

The address corresponding to lifted IL.

```
  | Target of label
```

An address this insn may jump to (esp. function specials)

```
  | Liveout
```

Statement should be considered live by deadcode elimination

```
  | StrAttr of string
```

Generic printable and parseable attribute

```
  | NamedStrAttr of string * string
```

Generic printable and parseable attribute

```
  | Context of context
```

Information about the instruction operands from a trace.

| `ThreadId of int`

Executed by a specific thread

| `ExnAttr of exn`

Generic extensible attribute, but no parsing

| `InitRO`

The memory in this assignment is stored in the binary

| `Synthetic`

Operation was added by an analysis

| `SpecialBlock`

Start of a special block

| `TaintIntro of int * string * int`

Attributes are extra information contained in a `stmt`.

```
type attributes = attribute list
```

```
type 'a visit_action =
```

| `SkipChildren`

Do not visit the children. Return the node as is.

| `DoChildren`

Continue exploring children of the current node. Changes to children will propagate up.

| `ChangeTo of 'a`

Replace the current object with the specified one.

| `ChangeToAndDoChildren of 'a`

Replace the current object with the given one, and visit children of the **replacement** object.

Visitors are a systematic method for exploring and changing objects of type `'a`. In BAP, `'a` can be `stmt`, `exp`, etc. The children of statements are expressions; the children of expressions are subexpressions and variables.

```
type formula_mode =
```

| `Sat`
| `Validity`
| `Foralls`

Specifies whether generated VCs will be evaluated for satisfiability or validity. Alternatively, quantifiers can be used.

85 Module Typecheck : Type checking and inference for AST programs.

Author(s): Ed Schwartz

`exception TypeError of string`

Exception raised when a type check error occurs.

85.0.17 Type inference of expressions

`val infer_ast : Ast.exp -> Type.typ`

`infer_ast e` returns the type of the expression `e`.

`val infer_ssa : Ssa.exp -> Type.typ`

`infer_ssa e` returns the type of the `Ssa.exp`[61] expression `e`.

85.0.18 Type checking

`val typecheck_expression : Ast.exp -> unit`

`typecheck_expression e` type checks the expression `e`.

Raises `TypeError` if the expression does not type check.

`val typecheck_stmt : Ast.stmt -> unit`

`typecheck_expression s` type checks the statement `s`.

Raises `TypeError` if the statement does not type check.

`val typecheck_prog : Ast.stmt list -> unit`

`typecheck_expression p` type checks the program `p`.

Raises `TypeError` if the program does not type check.

85.0.19 Helper functions

`val is_integer_type : Type.typ -> bool`

`is_integer_type t` returns true iff `t` is a register type.

`val is_mem_type : Type.typ -> bool`

`is_integer_type t` returns true iff `t` is of memory or array type.

`val index_type_of : Type.typ -> Type.typ`

`index_type_of t` returns the index type for memory loads and stores in a memory of type `t`.

Raises `Invalid_arg` if `t` is a non-memory type.

`val value_type_of : Type.typ -> Type.typ`

`index_type_of t` returns the value type for memory loads and stores in a memory of type `t`. For instance, a value type of `Reg 8` means the memory is byte addressable.

Raises `Invalid_arg` if `t` is a non-memory type.

`val bits_of_width : Type.typ -> int`

`bits_of_width t` returns the number of bits in the register type `t`.

Raises `Invalid_argument` if `t` is not of register type.

`val bytes_of_width : Type.typ -> int`

`bytes_of_width t` returns the number of bytes in the register type `t`.

Raises `Invalid_argument` if `t` is not of register type, or if t is a register type but its size is not expressible in bytes.

86 Module `Unroll` : Loop unrolling

`type unrollf = ?count:int -> Cfg.AST.G.t -> Cfg.AST.G.t`

The type signature of a loop unrolling function. Count specifies the number of times to unroll loops.

`type node_mapping = (Cfg.AST.G.V.t * Cfg.AST.G.V.t) list`

The type signature of the mapping from the original nodes to the node in the unrolled cfg

`val unroll_loops_steensgard : unrollf`

Unroll loops identified by Steensgard's loop forest algorithm:

Steensgaard, B. (1993). Sequentializing Program Dependence Graphs for Irreducible Programs (No. MSR-TR-93-14).

`val unroll_loops_sa : unrollf`

Unroll loops identified by structural analysis. The current structural analysis implementation is incomplete and may fail (raise an exception).

`val unroll_loops : unrollf`

Unroll loops using the default loop identification algorithm, which is unspecifed and may change.

```
val unroll_loops_with_mapping :
  ?count:int -> Cfg.AST.G.t -> Cfg.AST.G.t * node_mapping
```

Unroll loops using the default loop identification algorithm, which is unspecifed and may change.

87 Module `Util` : Utility functions that are not BAP specific.

Author(s): Ivan Jager

`exception Timeout`

The timeout exception, which is raised by `timeout`.

`val id : 'a -> 'a`

The identity function

`val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c`

Curry a tupled function

`val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c`

The opposite of `curry`

`val (<@) : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`

`(f <@ g) x = f (g x)`

`val hd_tl : 'a list -> 'a * 'a list`

Return the head and tail of a list

`val foldn : ?t:int -> ('a -> int -> 'a) -> 'a -> int -> 'a`

`foldn f i n` is f (... (f (f i n) (n-1)) ...) 0

`val foldn64 : ?t:int64 -> ('a -> int64 -> 'a) -> 'a -> int64 -> 'a`

Same as `foldn` but for `int64` integers.

`val mapn : (int -> 'a) -> int -> 'a list`

`mapn f n` is the same as `f 0; f 1; ...; f n`

`val gc_keepalive : 'a -> unit`

`gc_keepalive e` keeps the objects reachable in `e` from being garbage collected. This function is needed because `ignore e` does not serve this purpose.

87.0.20 List utility functions

`val list_mem : ?eq:('a -> 'a -> bool) -> 'a -> 'a list -> bool`

`list_mem` behaves like `List.mem`, but element equality can be specified with `eq`, which defaults to `(=)`.

```
val list_argmax :
  ?compare:('a -> 'a -> int) -> ('b -> 'a) -> 'b list -> 'b * 'a
```

Returns the arg max of `f` where the arguments come from `l`

`val list_union : 'a list -> 'a list -> 'a list`

Returns a union b, assuming a and b are sets

`val list_intersection : 'a list -> 'a list -> 'a list`

Returns a intersect b, assuming a and b are sets

`val list_does_intersect : 'a list -> 'a list -> bool`

Returns true if the intersection of two lists is not empty

`val list_difference : 'a list -> 'a list -> 'a list`

Returns a - b, assuming a and b are sets

`val list_subset : 'a list -> 'a list -> bool`

returns true when all elements in a are also in b

`val list_set_eq : 'a list -> 'a list -> bool`

Returns true when both sets contain the same elements

`val list_unique : 'a list -> 'a list`

`list_unique l` returns a list of elements that occur in `l`, without duplicates. (uses `=` and `Hashtbl.hash`)

`val list_pop : 'a list Pervasives.ref -> 'a`

Pop the first element off a list ref.

`val list_push : 'a list Pervasives.ref -> 'a -> unit`

Push an element onto the front of a list ref.

`val list_last : 'a list -> 'a`

Returns the last element of a list

`val list_last_option : 'a list -> 'a option`

If `l` is non-empty, `list_last_option l` returns `Some (list_last l)`. Otherwise, it returns `None`.

`val list_insert : 'a list -> 'a list -> int -> 'a list`

`list_insert l li i` Insert all elements in `li` into `l` just before index `i`

`val list_remove : 'a list -> int -> int -> 'a list`

`list_remove l i n` removes `n` elements in `l` starting at position `i`.

`val list_delete : 'a list -> 'a -> 'a list`

Deletes the first occurrence of e (if it exists) in the list and returns the updated list

`val list_compare : ('a -> 'a -> int) -> 'a list -> 'a list -> int`

Lexicographic compare of lists

`val list_cart_prod2 : ('a -> 'b -> unit) -> 'a list -> 'b list -> unit`

Calls f with each element in the cartesian product of the input lists

```
val list_cart_prod3 :
  ('a -> 'b -> 'c -> unit) -> 'a list -> 'b list -> 'c list -> unit
```

Calls f with each element in the cartesian product of the input lists

```
val list_cart_prod4 :
  ('a -> 'b -> 'c -> 'd -> unit) ->
  'a list -> 'b list -> 'c list -> 'd list -> unit
```

Calls f with each element in the cartesian product of the input lists

`val list_permutation : 'a list list -> ('a list -> unit) -> unit`

calls f with every value in the cartesian product of the sets in setlist. For instance, if setlist is `[1;2]; [2;3]`, then this function will call f `1;2`, f `1;3`, f `2;2`, and f `2;3` in some order.

`list_find_option p l` returns `Some x` for the first element `x` such that `p x` is true. If no such elements `x` exists, returns `None`.

`val list_find_option : ('a -> 'b option) -> 'a list -> 'b option`

Calls f on each element of l, and returns the first Some(x) returned. If no Some(x) are returned, None is returned.

`val list_partition_last : 'a list -> 'a list * 'a`

`list_partition_last l` returns `(lst,last)` where `lst` is `List.tl l` and `last` is the last element in `l`.

Raises `Invalid_argument` if `lst` is empty

`val list_shortest_first : (int -> int -> 'a) -> 'b list -> 'c list -> 'a`

`list_shortest_first f l1 l2` Calls `f` with its shorter list argument first, and the longer one second.

87.0.21 File/IO functions

`val change_ext : string -> string -> string`

Change the extension of a filename

`val list_directory : ?sort_files:bool -> string -> string list`

Get list of file name in directory (ordered by name)

`val trim_newline : string -> string`

Remove trailing newline

87.0.22 Simple algorithms

```
val union_find : ('a -> 'b list) -> 'a list -> 'a list list
```

`union_find map items`, where `map` is a mapping from items to their dependencies, finds independent elements in `items`

```
val split_common_prefix :
  ?eq:('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list * 'a list * 'a list
```

`split_common_prefix ?eq l1 l2` returns `(lcom, l1', l2')` where `l1'` and `l2'` are the longest lists such that `lcom::l1' eq l1` and `lcom::l2' eq l2`.

```
val split_common_suffix :
  ?eq:('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list * 'a list * 'a list
```

`split_common_suffix ?eq l1 l2` returns `(lcom, l1', l2')` where `l1'` and `l2'` are the longest lists such that `l1'::lcom eq l1` and `l2'::lcom eq l2`.

```
val apply_option : ('a -> 'a) option -> 'a -> 'a
```

`apply_option fopt` is equal to `f` when `fopt = Some f`, and is equal to the identity function when `fopt = None`.

```
val memoize : ?size:int -> ('a -> 'b) -> 'a -> 'b
```

Memoize the results of a function

87.0.23 Hash table functions

```
val get_hash_keys : ?sort_keys:bool -> ('a, 'b) Hashtbl.t -> 'a list
```

Get the keys from a hash table. If a key has multiple bindings, it is included once per binding

```
val get_hash_values : ?sort_values:bool -> ('a, 'b) Hashtbl.t -> 'b list
```

Get the values from a hash table. If a key has multiple bindings, the value is included once per binding

```
module HashUtil :
   functor (H : Hashtbl.S) ->   sig
```

```
    val hashtbl_eq : ?eq:('a -> 'a -> bool) -> 'a H.t -> 'a H.t -> bool
```

Test if two `Hashtbl`s are equal. Keys are tested for equality with `eq`, which defaults to `(=)`.

```
    val hashtbl_replace : 'a H.t -> H.key -> 'a -> unit
```

Implementation of `Hashtbl.replace` to work around OCaml bug.

```
    val get_hash_keys : ?sort_keys:bool -> 'a H.t -> H.key list
```

Get the keys from a hash table. If a key has multiple bindings, it is included once per binding

```
val get_hash_values : ?sort_values:bool -> 'a H.t -> 'a list
```

Get the values from a hash table. If a key has multiple bindings, the value is included once per binding

```
end
```

Extension of `Hashtbl`s

```
val hashtbl_eq :
  ?eq:('a -> 'a -> bool) -> ('b, 'a) Hashtbl.t -> ('b, 'a) Hashtbl.t -> bool
```

Version of hashtbl_eq for polymorphic `Hashtbl`s

87.0.24 Arithmetic `int64` operations

```
val int64_ucompare : int64 -> int64 -> int
```

Unsigned compare

```
val int64_udiv : int64 -> int64 -> int64
```

Unsigned division

```
val int64_urem : int64 -> int64 -> int64
```

Unsigned remainder

```
val int64_umax : int64 -> int64 -> int64
```

Unsigned max

```
val int64_umin : int64 -> int64 -> int64
```

Unsigned min

```
val int64_to_binary : ?pad:int -> int64 -> string
```

Convert `int64` to binary string for printing

87.0.25 Arithmetic `big_int` operations

```
val big_int_ucompare : Big_int_Z.big_int -> Big_int_Z.big_int -> int
```

Unsigned compare

```
val big_int_umax :
  Big_int_Z.big_int -> Big_int_Z.big_int -> Big_int_Z.big_int
```

Unsigned max

```
val big_int_umin :
  Big_int_Z.big_int -> Big_int_Z.big_int -> Big_int_Z.big_int
```

Unsigned min

```
val big_int_to_binary : ?pad:int -> Big_int_Z.big_int -> string
```

Convert `big_int` to binary string for printing

```
val big_int_to_hex : ?pad:int -> Big_int_Z.big_int -> string
```

Convert `big_int` to hex string for printing

```
val big_int_of_string : string -> Big_int_Z.big_int
```

Convert string to `big_int`

```
val big_int_of_binstring :
  ?e:[ `Big | `Little ] -> string -> Big_int_Z.big_int
```

Convert binary string to `big_int`

```
val run_with_remapped_fd :
  Unix.file_descr -> Unix.file_descr -> (unit -> 'a) -> 'a
```

execute f with fd_from remapped to fd_to. useful for redirecting output of external code; e.g., redirecting stdout when calling STP code.

87.0.26 Printing/status functions

```
module StatusPrinter :
  sig
```

```
    val init : string -> int -> unit
```

`init s n` starts a new process called `s` with `n` steps.

```
    val inc : unit -> unit
```

`inc ()` indicates a step has completed.

```
    val stop : unit -> unit
```

`stop ()` indicates the process has completed.

```
  end
```

Status printer module

```
val timeout : secs:int -> f:('a -> 'b) -> x:'a -> 'b
```

`timeout n f x` runs `f x` for `n` seconds. If `f x` returns `v`, `timeout n f x` returns `v`. Otherwise `timeout n f x` raises `Util.Timeout`[87].

```
val timeout_option : secs:int -> f:('a -> 'b) -> x:'a -> 'b option
```

`timeout_option n f x` runs `f x` for `n` seconds. If `f x` returns `v`, `timeout_option n f x` returns `Some v`. Otherwise `timeout_option n f x` returns `None`.

```
val print_separated_list : ('a -> string) -> string -> 'a list -> string
```

`print_separated_list printer sep l` converts `l` to a string by computing `printer e` for each element `e`, and concatenating the results with `sep`.

```
val print_mem_usage : unit -> unit
```

If the `UtilMemUse` module has debugging enabled, prints information about the memory use of the process.

88 Module `Var` : The type of variables.

Author(s): Ivan Jager

```
type t = private
  | V of int * string * Type.typ
```

The type for a variable identifier. The int should uniquely identify the variable. The string is to make it easier for humans to read. A variable's type is embedded in it.

```
val newvar : string -> Type.typ -> t
```

`newvar s t` creates a fresh variable of type `t` and human readable string `s`.

```
val renewvar : t -> t
```

`renewvar v` creates a fresh variable with the same name and type as `v`. `renewvar v` is equivalent to `newvar (Var.name v) (Var.typ v)`.

```
val typ : t -> Type.typ
```

`typ v` returns the type of `v`.

```
val name : t -> string
```

`name v` returns the name of `v`.

88.0.27 Comparison functions

```
val hash : t -> int
val equal : t -> t -> bool
val compare : t -> t -> int
```

88.0.28 Data structures for storing variables

```
module VarHash :
   Hashtbl.S  with type key = t
module VarMap :
   BatMap.S  with type key = t
module VarSet :
   BatSet.S  with type elt = t
type defuse = {
  defs : t list ;
  uses : t list ;
}
```

Variable definition and use type.

89 Module `Var_temp` : Recognizing and creating temporary variables

A temporary is a variable introduced by BAP's lifting process that is only referenced inside one assembly block. The evaluator (and other BAP analyses) use this information to throw away any state stored for these temporaries once the temporary becomes out of scope (i.e., out of that assembly block).

```
val is_temp_name : string -> bool
```

`is_temp_name s` returns true iff s denotes a temporary variable name.

```
val is_temp : Var.t -> bool
```

`is_temp v` is equivalent to `is_temp_name (Var.name v)`.

```
val nt : string -> Type.typ -> Var.t
```

`nt n t` creates a new temporary variable with name `n` and type `t`.

90 Module `Vc` : Interface to verification generation procedures.

Verification condition procedures take as input a program `p` and post-condition `q`, and return a logical formula that is `true` when executions in `p` end in a state satisfying the post-condition `q`.

Author(s): Ed Schwartz

90.1 VC-specific Types

```
type options = {
  k : int ;
```

Expressions larger than `k` will be given their own temporary variable assignment in the formula. Only used by DWP algorithms.

`mode : Type.formula_mode ;`

Indicates whether the formula should be valid for validity or satisfiability. This is currently only needed for VC algorithms that use passification, since passification introduces new free variables into the formula.

`full_subst : bool ;`

`true` indicates that full substitution should be performed, even for `Let` expressions, in which case exponential blowup may occur. `false` enables partial substitution only.

`}`

Various options that can modify the behavior of VC generation algorithms.

```
val default_options : options
```

Sensible default VC options

```
type ast_vc = options -> Ast.program -> Ast.exp -> Ast.exp * Ast.var list
```

Type of VC algorithm that naturally operates on a `Ast.program`[8] program.

```
type cfg_vc = options -> Cfg.AST.G.t -> Ast.exp -> Ast.exp * Ast.var list
```

Type of VC algorithm that naturally operates on a `Cfg.AST.G.t` program.

```
type ssa_vc = options -> Cfg.SSA.G.t -> Ast.exp -> Ast.exp * Ast.var list
```

Type of VC algorithm that naturally operates on a `Cfg.SSA.G.t` program.

```
type t =
  | AstVc of ast_vc
  | CfgVc of cfg_vc
  | SsaVc of ssa_vc
```

Any type of VC algorithm

```
val vc_astprog :
  t -> options -> Ast.program -> Ast.exp -> Ast.exp * Ast.var list
```

`vc_astprog vc opts prog post` runs `vc` to compute the VC of `prog` with post-condition `post`, starting from a `Ast.program`[8] program.

```
val vc_astcfg :
  t -> options -> Cfg.AST.G.t -> Ast.exp -> Ast.exp * Ast.var list
```

`vc_astcfg` is like `vc_astprog`, but for AST CFG programs.

```
val vc_ssacfg :
  t -> options -> Cfg.SSA.G.t -> Ast.exp -> Ast.exp * Ast.var list
```

`vc_ssacfg` is like `vc_astprog`, but for SSA CFG programs.

90.2 DWP algorithms

`val compute_dwp : ssa_vc`

DWP implementation that does not emit quantifiers.

`val compute_dwp_gen : t`

`val compute_dwp_let : ssa_vc`

DWP implementation that utilizes `Let` expressions in place of some equalities. This has the advantage of not introducing meaningless free variables.

`val compute_dwp_let_gen : t`

`val compute_fwp : ssa_vc`

Alternate formulation of DWP that is easier to understand and produces smaller formulas on programs that have no `Assume` statements. However, it produces slightly larger formulas for programs with `Assume` statements.

`val compute_fwp_gen : t`

`val compute_fwp_uwp : ssa_vc`

An unstructured implementation of `compute_fwp` that does not convert the entire program to GCL

`val compute_fwp_uwp_gen : t`

`val compute_fwp_lazyconc : ssa_vc`

Same as `compute_fwp` but with concrete evaluation and lazy merging.

`val compute_fwp_lazyconc_gen : t`

`val compute_fwp_lazyconc_uwp : ssa_vc`

An unstructured implementation of `compute_fwp_lazyconc` that does not convert the entire program to GCL

`val compute_fwp_lazyconc_uwp_gen : t`

`val compute_dwp1 : ssa_vc`

DWP implementation that uses forall quantifiers.

General form of `compute_dwp1`.

`val compute_dwp1_gen : t`

`val compute_flanagansaxe : ssa_vc`

Flanagan and Saxe's algorithm, which is like DWP but can only run backwards.

`val compute_flanagansaxe_gen : t`

90.3 "Standard" Weakest Precondition Algorithms

`val compute_wp : cfg_vc`

Weakest precondition based on Dijkstra's GCL. Produces exponentially sized formulas.

`val compute_wp_gen : t`

`val compute_passified_wp : ssa_vc`

Efficient weakest precondition on passified GCL programs.

`val compute_passified_wp_gen : t`

`val compute_uwp : ssa_vc`

Unstructured Weakest Precondition that operates on a CFG-like representation. Produces exponentially sized formulas.

`val compute_uwp_gen : t`

`val compute_uwp_efficient : ssa_vc`

Unstructured Weakest Precondition algorithm that uses passification. Output sub-exponentially sized formulas.

`val compute_uwp_efficient_gen : t`

90.4 Symbolic Execution VC algorithms

`val compute_fse_bfs : ast_vc`

Use BFS to visit all states. Produces exponentially sized formulas.

`val compute_fse_bfs_gen : t`

`val compute_fse_bfs_maxdepth : int -> ast_vc`

Use BFS to visit all states up to a certain depth. Produces exponentially sized formulas.

`val compute_fse_bfs_maxdepth_gen : int -> t`

`val compute_fse_maxrepeat : int -> ast_vc`

Use BFS to visit all states on paths that do not visit a node more than `k` times. Produces exponentially sized formulas.

`val compute_fse_maxrepeat_gen : int -> t`

90.5 All Supported VCs

`val vclist : (string * t) list`

A list of supported VCs. VCs with parameters (like maxdepth) are not included.

`val pred_vclist : (string * t) list`

A list of supported VCs that only require predicate logic (no quantifiers).

91 Module Vsa : number of bits * unsigned stride * signed lower bound * signed upper bound

```
module VM :
   Var.VarMap
module D :
   Debug.Make(  sig
     val name : string
     val default : [> `NoDebug ]
  end )
val gcd : Z.t -> Z.t -> Z.t
val lcm : Z.t -> Z.t -> Z.t
val rem : Z.t -> Z.t -> Z.t
val bits_of_width : Type.typ -> int
val mem_max : int option Pervasives.ref
exception Unimplemented of string
module SI :
  sig
     type t = int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
```

number of bits * unsigned stride * signed lower bound * signed upper bound

```
     val is_empty : 'a * Z.t * Z.t * Z.t -> bool
     val to_string : int * Z.t * Z.t * Z.t -> string
     val size : 'a * Z.t * Z.t * Z.t -> Z.t
     val highbit : int -> Z.t
     val extend : int -> Z.t -> Z.t
     val trunc : int -> Z.t -> Z.t
     val maxi : int -> Z.t
     val mini : int -> Z.t
     val top : int -> int * Z.t * Z.t * Z.t
     val empty : 'a -> 'a * Z.t * Z.t * Z.t
     val upper : int -> Z.t -> Z.t -> Z.t
     val lower : int -> Z.t -> Z.t -> Z.t
     val remove_lower_bound : int * Z.t * 'a * Z.t -> int * Z.t * Z.t * Z.t
     val remove_upper_bound : int * Z.t * Z.t * 'a -> int * Z.t * Z.t * Z.t
     val single : 'a -> 'b -> 'a * Z.t * 'b * 'b
     val of_bap_int : Z.t -> Type.typ -> int * Z.t * Z.t * Z.t
```

```
val above : int -> Z.t -> int * Z.t * Z.t * Z.t
val below : int -> Z.t -> int * Z.t * Z.t * Z.t
val above_unsigned : int -> Z.t -> int * Z.t * Z.t * Z.t
val below_unsigned : 'a -> Z.t -> 'a * Z.t * Z.t * Z.t
val aboveeq : int -> 'a -> int * Z.t * 'a * Z.t
val beloweq : int -> 'a -> int * Z.t * Z.t * 'a
val aboveeq_unsigned : int -> 'a -> int * Z.t * 'a * Z.t
val beloweq_unsigned : 'a -> 'b -> 'a * Z.t * Z.t * 'b
val zero : 'a -> 'a * Z.t * Z.t * Z.t
val one : 'a -> 'a * Z.t * Z.t * Z.t
val minus_one : 'a -> 'a * Z.t * Z.t * Z.t
val is_reduced : int -> int * Z.t * Z.t * Z.t -> bool
val check_reduced : int -> int * Z.t * Z.t * Z.t -> unit
val check_reduced2 :
  int -> int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t -> unit
val renorm : int -> int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t
val renormtri :
  (int -> 'a -> 'b -> 'c -> int * Z.t * Z.t * Z.t) ->
  int -> 'a -> 'b -> 'c -> int * Z.t * Z.t * Z.t
val renormbin :
  (int -> 'a -> 'b -> int * Z.t * Z.t * Z.t) ->
  int -> 'a -> 'b -> int * Z.t * Z.t * Z.t
val renormun :
  (int -> 'a -> int * Z.t * Z.t * Z.t) -> int -> 'a -> int * Z.t * Z.t * Z.t
val renormbin' :
  (int * 'a * 'b * 'c -> 'd -> int * Z.t * Z.t * Z.t) ->
  int * 'a * 'b * 'c -> 'd -> int * Z.t * Z.t * Z.t
val add :
  ?allow_overflow:bool ->
  int ->
  int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t
```

Addition of strided intervals

```
val add :
  ?allow_overflow:bool ->
  int ->
  int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t
val neg :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Z.t * Z.t
```

Negation of a strided interval

```
val neg :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Z.t * Z.t
```

```
val sub :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Z.t * Z.t
```

Subtraction of strided intervals

```
val sub :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Z.t * Z.t
```

```
val minor :
  int -> Z.t -> Big_int_Z.big_int -> Z.t -> Big_int_Z.big_int -> Z.t
```

```
val maxor :
  int -> Big_int_Z.big_int -> Z.t -> Big_int_Z.big_int -> Z.t -> Z.t
```

```
val ntz : Z.t -> int
```

```
val logor :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Z.t * Z.t * Z.t
```

Bitwise OR

```
val logor :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Z.t * Z.t * Z.t
```

```
val lognot :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Z.t * Z.t
```

Bitwise NOT

```
val lognot :
  int ->
```

```
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Z.t * Z.t
val logand :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Z.t * Z.t
```

Bitwise AND

```
val logand :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Z.t * Z.t
val logxor :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Z.t * Z.t * Z.t
```

Bitwise XOR

```
val logxor :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Z.t * Z.t * Z.t
val modulus :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Z.t * Big_int_Z.big_int * Big_int_Z.big_int
```

unsigned modulus

```
val modulus :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Z.t * Big_int_Z.big_int * Big_int_Z.big_int
val toshifts : int -> int * Z.t * Z.t * Z.t -> int * int
val mk_shift :
  [< `Leftshift | `Rightshift ] ->
  (Z.t -> int -> Z.t) ->
  int ->
```

```
  int * Z.t * Z.t * Z.t ->
  int * Z.t * Z.t * Z.t -> int * Big_int_Z.big_int * Z.t * Z.t
val rshift :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Z.t * Z.t
```

Logical right-shift

```
val arshift :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Z.t * Z.t
```

Arithmetic right-shift

```
val lshift :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Z.t * Z.t
```

Left shift

```
val cast_low :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
val cast_low :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
val cast_high :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
val cast_high :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
val cast_signed :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
```

```
val cast_signed :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
val cast_unsigned :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
val cast_unsigned :
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
val extract :
  int ->
  int ->
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
val extract :
  int ->
  int ->
  int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
  int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
val concat : int -> t -> t -> int * Z.t * Z.t * Z.t
val concat : int -> t -> t -> int * Z.t * Z.t * Z.t
val yes : int * Z.t * Z.t * Z.t
val no : int * Z.t * Z.t * Z.t
val maybe : int * Z.t * Z.t * Z.t
val eq :
  int ->
  int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t
val union :
  int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t
val union :
  int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t
val intersection :
  int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t
val intersection :
  int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t
val widen :
  int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t
```

```
    val widen :
      int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t
    val fold : (Z.t -> 'a -> 'a) -> 'b * Z.t * Z.t * Z.t -> 'a -> 'a
    val binop_to_si_function :
      Type.binop_type ->
      int ->
      int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
      int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
      int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
    val unop_to_si_function :
      Type.unop_type ->
      int ->
      int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
      int * Big_int_Z.big_int * Z.t * Z.t
    val cast_to_si_function :
      Type.cast_type ->
      int ->
      int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int ->
      int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
  end

module VS :
  sig
    type region = Var.t
    type address = region * Vsa.SI.t
    type t = address list
    val global : Var.t
    val top : int -> (Var.t * (int * Z.t * Z.t * Z.t)) list
    val empty : 'a -> (Var.t * ('a * Z.t * Z.t * Z.t)) list
    val width : ('a * ('b * 'c * 'd * 'e)) list -> 'b
    val size : 'a -> ('b * ('c * Z.t * Z.t * Z.t)) list -> Z.t
    val pp_address : (string -> 'a) -> Pp.VH.key * (int * Z.t * Z.t * Z.t) -> 'a
    val pp :
      (string -> unit) -> (Pp.VH.key * (int * Z.t * Z.t * Z.t)) list -> unit
    val to_string : (Pp.VH.key * (int * Z.t * Z.t * Z.t)) list -> string
    val single : 'a -> 'b -> (Var.t * ('a * Z.t * 'b * 'b)) list
    val of_bap_int : Z.t -> Type.typ -> (Var.t * (int * Z.t * Z.t * Z.t)) list
    val remove_lower_bound :
      (region * Vsa.SI.t) list ->
      (region * (int * Big_int_Z.big_int * Z.t * Big_int_Z.big_int)) list
    val remove_upper_bound :
```

```
  (region * Vsa.SI.t) list ->
  (region * (int * Big_int_Z.big_int * Big_int_Z.big_int * Z.t)) list
val zero : 'a -> (Var.t * ('a * Z.t * Z.t * Z.t)) list
val one : 'a -> (Var.t * ('a * Z.t * Z.t * Z.t)) list
val minus_one : 'a -> (Var.t * ('a * Z.t * Z.t * Z.t)) list
val above : int -> Z.t -> (Var.t * (int * Z.t * Z.t * Z.t)) list
val below : int -> Z.t -> (Var.t * (int * Z.t * Z.t * Z.t)) list
val above_unsigned : int -> Z.t -> (Var.t * (int * Z.t * Z.t * Z.t)) list
val below_unsigned : 'a -> Z.t -> (Var.t * ('a * Z.t * Z.t * Z.t)) list
val aboveeq : int -> 'a -> (Var.t * (int * Z.t * 'a * Z.t)) list
val beloweq : int -> 'a -> (Var.t * (int * Z.t * Z.t * 'a)) list
val aboveeq_unsigned : int -> 'a -> (Var.t * (int * Z.t * 'a * Z.t)) list
val beloweq_unsigned : 'a -> 'b -> (Var.t * ('a * Z.t * Z.t * 'b)) list
val add :
  int ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list
val sub :
  int ->
  (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
  list ->
  (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
  list -> (Var.t * (int * Big_int_Z.big_int * Z.t * Z.t)) list
val makeother :
  (int ->
   int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t -> int * Z.t * Z.t * Z.t) ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list option ->
  int ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list
val logand :
  int ->
  (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
  list ->
  (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
  list ->
  (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
  list
val logor :
```

```
  int ->
  (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
  list ->
  (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
  list ->
  (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
  list
val logxor :
  int ->
  (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
  list ->
  (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
  list ->
  (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
  list
val si_to_vs_binop_function :
  ('a -> 'b -> 'c -> 'd) ->
  'a -> (Var.t * 'b) list -> (Var.t * 'c) list -> (Var.t * 'd) list
val si_to_vs_unop_function :
  ('a -> 'b -> 'c) -> 'a -> (Var.t * 'b) list -> (Var.t * 'c) list
val si_to_vs_cast_function :
  ('a -> 'b -> 'c) -> 'a -> (Var.t * 'b) list -> (Var.t * 'c) list
val concat :
  int ->
  (Var.t * Vsa.SI.t) list ->
  (Var.t * Vsa.SI.t) list -> (Var.t * (int * Z.t * Z.t * Z.t)) list
val yes : (Var.t * (int * Z.t * Z.t * Z.t)) list
val no : (Var.t * (int * Z.t * Z.t * Z.t)) list
val maybe : (Var.t * (int * Z.t * Z.t * Z.t)) list
val eq :
  int ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list
```

Slightly unconservative equality checking.

```
val equal : 'a -> 'a -> bool
val union :
  (Var.t * (int * Z.t * Z.t * Z.t)) list ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list
val intersection :
  (Var.t * (int * Z.t * Z.t * Z.t)) list ->
```

```
      (Var.t * (int * Z.t * Z.t * Z.t)) list ->
      (Var.t * (int * Z.t * Z.t * Z.t)) list
    val widen :
      (Var.t * (int * Z.t * Z.t * Z.t)) list ->
      (Var.t * (int * Z.t * Z.t * Z.t)) list ->
      (Var.t * (int * Z.t * Z.t * Z.t)) list
    val fold :
      ('a * Z.t -> 'b -> 'b) -> ('a * ('c * Z.t * Z.t * Z.t)) list -> 'b -> 'b
    val concrete :
      ?max:int -> (Var.t * ('a * Z.t * Z.t * Z.t)) list -> Z.t list option
    val numconcrete : ('a * ('b * Z.t * Z.t * Z.t)) list -> Z.t
    val binop_to_vs_function :
      Type.binop_type ->
      int ->
      (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
      list ->
      (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
      list ->
      (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
      list
    val unop_to_vs_function :
      Type.unop_type ->
      int ->
      (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
      list -> (Var.t * (int * Big_int_Z.big_int * Z.t * Z.t)) list
    val cast_to_vs_function :
      Type.cast_type ->
      int ->
      (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
      list ->
      (Var.t * (int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int))
      list
  end

module MemStore :
  sig
    type aloc = Vsa.VS.region * Big_int_Z.big_int
    module M1 :
    BatMap.Make(  sig
        type t = Vsa.VS.region
        val compare : Var.t -> Var.t -> int
```

```
  end )
module M2 :
BatMap.Make(  sig
    type t = Big_int_Z.big_int
    val compare : Z.t -> Z.t -> int
  end )
type t = Vsa.VS.t M2.t M1.t
```

This implementation may change...

```
val top : 'a M1.t
val fold : (M1.key * M2.key -> 'a -> 'b -> 'b) ->
  'a M2.t M1.t -> 'b -> 'b
```

Fold over all addresses in the MemStore

```
val pp :
  (string -> unit) ->
  (Pp.VH.key * (int * Z.t * Z.t * Z.t)) list M2.t
  M1.t -> unit
val read_concrete :
  int ->
  ?o:'a ->
  (Var.t * Vsa.SI.t) list M2.t M1.t ->
  M1.key * M2.key -> (Var.t * Vsa.SI.t) list
val read :
  int ->
  ?o:'a ->
  (Var.t * Vsa.SI.t) list M2.t M1.t ->
  (M1.key *
   (int * Z.t * M2.key * M2.key))
  list -> (Var.t * Vsa.SI.t) list
val widen_region : 'a M2.t -> 'a M2.t
val widen_mem : 'a M2.t M1.t ->
  'a M2.t M1.t
val write_concrete_strong :
  int ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list M2.t M1.t ->
  M1.key * M2.key ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list ->
  (Var.t * (int * Z.t * Z.t * Z.t)) list M2.t M1.t
val write_concrete_weak :
  int ->
```

```
  (Var.t * Vsa.SI.t) list M2.t M1.t ->
  M1.key * M2.key ->
  (Var.t * Vsa.SI.t) list ->
  (Var.t * Vsa.SI.t) list M2.t M1.t
val write_concrete_intersection :
  int ->
  (Var.t * Vsa.SI.t) list M2.t M1.t ->
  M1.key * M2.key ->
  (Var.t * Vsa.SI.t) list ->
  (Var.t * Vsa.SI.t) list M2.t M1.t
val write_concrete_weak_widen :
  int ->
  (Var.t * Vsa.SI.t) list M2.t M1.t ->
  M1.key * M2.key ->
  (Var.t * Vsa.SI.t) list ->
  (Var.t * Vsa.SI.t) list M2.t M1.t
val write :
  int ->
  (Var.t * Vsa.SI.t) list M2.t M1.t ->
  (M1.key *
   (int * Z.t * M2.key * M2.key))
  list ->
  (Var.t * Vsa.SI.t) list ->
  (Var.t * Vsa.SI.t) list M2.t M1.t
val write_intersection :
  int ->
  (Var.t * Vsa.SI.t) list M2.t M1.t ->
  (M1.key * ('a * Z.t * M2.key * M2.key))
  list ->
  (Var.t * Vsa.SI.t) list ->
  (Var.t * Vsa.SI.t) list M2.t M1.t
val equal : 'a M2.t M1.t ->
  'a M2.t M1.t -> bool
val merge_region :
  inclusive:bool ->
  f:('a -> 'a -> 'a) ->
  'a M2.t -> 'a M2.t -> 'a M2.t
val merge_mem :
  inclusive:bool ->
  f:('a -> 'a -> 'a) ->
  'a M2.t M1.t ->
  'a M2.t M1.t ->
  'a M2.t M1.t
val intersection : t -> t -> t
```

```
    val union : t -> t -> t
    val widen : t -> t -> t
  end
```

Abstract Store

```
module AbsEnv :
  sig
    type value = [ `Array of Vsa.MemStore.t | `Scalar of Vsa.VS.t ]
    type t = value Vsa.VM.t
```

This implementation may change

```
    val empty : 'a Vsa.VM.t
    val pp_value :
      (string -> unit) ->
      [< `Array of
           (Pp.VH.key * (int * Z.t * Z.t * Z.t)) list Vsa.MemStore.M2.t
           Vsa.MemStore.M1.t
       | `Scalar of (Pp.VH.key * (int * Z.t * Z.t * Z.t)) list ] ->
      unit
    val value_to_string :
      [< `Array of
           (Pp.VH.key * (int * Z.t * Z.t * Z.t)) list Vsa.MemStore.M2.t
           Vsa.MemStore.M1.t
       | `Scalar of (Pp.VH.key * (int * Z.t * Z.t * Z.t)) list ] ->
      string
    val pp :
      (string -> unit) ->
      [< `Array of
           (Pp.VH.key * (int * Z.t * Z.t * Z.t)) list Vsa.MemStore.M2.t
           Vsa.MemStore.M1.t
       | `Scalar of (Pp.VH.key * (int * Z.t * Z.t * Z.t)) list ]
      Vsa.VM.t -> unit
    val to_string :
      [< `Array of
           (Pp.VH.key * (int * Z.t * Z.t * Z.t)) list Vsa.MemStore.M2.t
           Vsa.MemStore.M1.t
       | `Scalar of (Pp.VH.key * (int * Z.t * Z.t * Z.t)) list ]
      Vsa.VM.t -> string
    val value_equal :
      [> `Array of 'a Vsa.MemStore.M2.t Vsa.MemStore.M1.t | `Scalar of 'b ] ->
      [> `Array of 'a Vsa.MemStore.M2.t Vsa.MemStore.M1.t | `Scalar of 'b ] -> bool
    val equal :
```

```
      ([> `Array of 'b Vsa.MemStore.M2.t Vsa.MemStore.M1.t | `Scalar of 'c ] as 'a)
      Vsa.VM.t -> 'a Vsa.VM.t -> bool
    val do_find_vs_int : [> `Scalar of 'a ] Vsa.VM.t -> Vsa.VM.key -> 'a
    val do_find_vs :
      [> `Scalar of (Var.t * (int * Z.t * Z.t * Z.t)) list ] Vsa.VM.t ->
      Vsa.VM.key -> (Var.t * (int * Z.t * Z.t * Z.t)) list
    val do_find_vs_opt : [> `Scalar of 'a ] Vsa.VM.t -> Vsa.VM.key -> 'a option
    val do_find_ae_int : [> `Array of 'a ] Vsa.VM.t -> Vsa.VM.key -> 'a
    val do_find_ae :
      [> `Array of 'a Vsa.MemStore.M1.t ] Vsa.VM.t ->
      Vsa.VM.key -> 'a Vsa.MemStore.M1.t
    val do_find_ae_opt : [> `Array of 'a ] Vsa.VM.t -> Vsa.VM.key -> 'a option
  end
```

Abstract Environment

```
type options = {
  initial_mem : (Type.addr * char) list ;
  sp : Var.t ;
  mem : Var.t ;
}
```

92 Module Vsa_ast : Value-Set Analysis / Value-Set Arithmetic

```
module SI :
  sig
    type t = int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
    val is_empty : t -> bool
    val to_string : t -> string
  end
```

Strided intervals

```
module VS :
  sig
    type region = Ast.var
    type address = region * Vsa_ast.SI.t
    type t = address list
    val global : region
    val to_string : t -> string
    val concrete : ?max:int -> t -> Big_int_Z.big_int list option
```

```
  end
```

Value sets

```
module MemStore :
  sig
     module M1 :
     Map.S  with type key = Var.t
     module M2 :
     Map.S  with type key = Big_int_Z.big_int
     type t = Vsa_ast.VS.t M2.t M1.t
  end
```

Memories

```
module AbsEnv :
  sig
     type value = [ `Array of Vsa_ast.MemStore.t | `Scalar of Vsa_ast.VS.t ]
     type t = value Var.VarMap.t
     val pp : (string -> unit) -> t -> unit
     val value_to_string : value -> string
  end
```

Abstract environments

```
val exp2vs : AbsEnv.t -> Ast.exp -> VS.t
```

Approximate an expression using value sets in an abstract environment

```
val vsa :
  ?nmeets:int ->
  Vsa.options ->
  Cfg.AST.G.t ->
  (Cfg.AST.G.V.t * int -> AbsEnv.t option) *
  (Cfg.AST.G.V.t * int -> AbsEnv.t option)
```

Main VSA interface. Returns functions for computing abstract environments before and after the given location.

```
val last_loc : Cfg.AST.G.t -> Cfg.AST.G.V.t -> Cfg.AST.G.V.t * int
```

Returns the last location in a basic block.

```
val build_default_arch_options : Arch.arch -> Vsa.options
```

Build default options for arch

```
val build_default_prog_options : Asmir.asmprogram -> Vsa.options
```

Build default options for program

93 Module Vsa_ssa : Value-Set Analysis / Value-Set Arithmetic

```
module SI :
  sig
     type t = int * Big_int_Z.big_int * Big_int_Z.big_int * Big_int_Z.big_int
     val is_empty : t -> bool
     val to_string : t -> string
  end
```

Strided intervals

```
module VS :
  sig
     type region = Ssa.var
     type address = region * Vsa_ssa.SI.t
     type t = address list
     val global : region
     val to_string : t -> string
     val concrete : ?max:int -> t -> Big_int_Z.big_int list option
  end
```

Value sets

```
module MemStore :
  sig
     module M1 :
     Map.S  with type key = Var.t
     module M2 :
     Map.S  with type key = Big_int_Z.big_int
     type t = Vsa_ssa.VS.t M2.t M1.t
  end
```

Memories

```
module AbsEnv :
  sig
     type value = [ `Array of Vsa_ssa.MemStore.t | `Scalar of Vsa_ssa.VS.t ]
     type t = value Var.VarMap.t
     val pp : (string -> unit) -> t -> unit
     val value_to_string : value -> string
```

```
  end
```

Abstract environments

```
val exp2vs : AbsEnv.t -> Ssa.exp -> VS.t
```

Approximate an expression using value sets in an abstract environment

```
val prepare_ssa_indirect :
  ?vs:Cfg.SSA.G.V.t list -> Cfg.SSA.G.t -> Cfg.SSA.G.t
```

Prepare SSA CFG for resolving indirect jumps

```
val vsa :
  ?nmeets:int ->
  Vsa.options ->
  Cfg.SSA.G.t ->
  (Cfg.SSA.G.V.t * int -> AbsEnv.t option) *
  (Cfg.SSA.G.V.t * int -> AbsEnv.t option)
```

Main VSA interface. Returns functions for computing abstract environments before and after the given location.

```
val last_loc : Cfg.SSA.G.t -> Cfg.SSA.G.V.t -> Cfg.SSA.G.V.t * int
```

Returns the last location in a basic block.

```
val build_default_arch_options : Arch.arch -> Vsa.options
```

Build default options for arch

```
val build_default_prog_options : Asmir.asmprogram -> Vsa.options
```

Build default options for program

94 Module `Worklist` : Worklists with in place modification

```
type 'a t
```

The type of a worklist

```
exception Empty
```

Raised when `take` or `peek` is applied to an empty worklist.

```
val create : unit -> 'a t
```

Return a new worklist, initially empty.

```
val add : 'a -> 'a t -> unit
```

`add x w` adds the element `x` at the end of the worklist `w` if `x` is not already in `w`. If `x` is already in `w`, the worklist does not change.

The current implementation of `add` is `O(n)`, where `n = length w`.

`val push : 'a -> 'a t -> unit`

`push` is a synonym for `add`.

`val add_list : 'a list -> 'a t -> unit`

`add_list l w` adds the elements in `l` to the worklist `w`.

`val filter : ('a -> bool) -> 'a t -> unit`

`filter f w` removes any element e from `w` when `f e` is false.

`val take : 'a t -> 'a`

`take w` removes and returns the first element of worklist `w`, or raises `Empty` if the worklist is empty.

`val pop : 'a t -> 'a`

`pop` is a synonym for `take`.

`val peek : 'a t -> 'a`

`peek w` returns the first element in worklist `w`, without removing it from the worklist, or raises `Empty` if the worklist is empty.

`val top : 'a t -> 'a`

`top` is a synonym for `peek`.

`val all : 'a t -> 'a list`

`all w` returns all elements in worklist `w`.

`val clear : 'a t -> unit`

Discard all elements from a worklist.

`val is_empty : 'a t -> bool`

Return `true` if the given worklist is empty, `false` otherwise.

`val length : 'a t -> int`

Return the number of elements in a worklist.

`val iter : ('a -> unit) -> 'a t -> unit`

`iter f q` applies `f` in turn to all elements of `q`, from the least recently entered to the most recently entered. The worklist itself is unchanged.

`val fold : ('a -> 'b -> 'a) -> 'a -> 'b t -> 'a`

`fold f accu q` is equivalent to `List.fold_left f accu l`, where `l` is the list of `q`'s elements. The worklist remains unchanged.

95 Module Wp : Functions for computing the weakest preconditions (WPs) of programs.

Given a program p and a postcondition q, the weakest precondition wp(p,q) describes all input states that will cause p to terminate in a state satisfying q. Generally, the precondition is solved with an SMT solver.

```
val wp : ?simp:(Ast.exp -> Ast.exp) -> Gcl.t -> Ast.exp -> Ast.exp
```

wp p q uses Dijkstra's classic WP algorithm to compute wp(p,q), applying simp to simplify each intermediate expression during the calculation. See "A Discipline of Programming" by Dijkstra, or CMU-CS-08-159 (Brumley's thesis), chapter 3.2. This algorithm may produce formulas exponential in the size of the program.

```
val passified_wp : ?simp:(Ast.exp -> Ast.exp) -> Gcl.t -> Ast.exp -> Ast.exp
```

passified_wp is similar to Wp.wp[95], but is intended for passified programs. Unlike Wp.wp[95] it does not duplicate the post-condition.

```
val build_uwp :
  (Gcl.t -> Ast.exp -> Ast.exp) -> Gcl.Ugcl.t -> Ast.exp -> Ast.exp
```

build_uwp wp uses the GCL-based weakest precondition algorithm for non-passified programs wp and builds an unstructured weakest precondition algorithm from it. See "Weakest-Precondition of Unstructured Programs" by Barnett for the general technique.

```
val dijkstra_uwp : Gcl.Ugcl.t -> Ast.exp -> Ast.exp
```

dijkstra_wp is build_uwp Wp.wp

```
val build_passified_uwp :
  ((Cfg.AST.G.V.t -> unit) -> Cfg.AST.G.t -> unit) ->
  (Gcl.t -> Ast.exp -> Ast.exp) -> Gcl.Ugcl.t -> Ast.exp -> Ast.exp
```

build_passified_uwp iter wp uses the GCL-based weakest precondition algorithm wp for passified programs and builds an unstructured weakest precondition algorithm from it. iter specifies the iteration strategy, e.g., reverse topological order. See "Weakest-Precondition of Unstructured Programs" by Barnett for the general technique.

```
val efficient_uwp : Gcl.Ugcl.t -> Ast.exp -> Ast.exp
```

efficient_uwp is build_passified_uwp RToposort.iter Wp.wp. Note that the Choice rule is not used, which is the only inefficient aspect of Wp.wp.

```
val efficient_wp : ?simp:(Ast.exp -> Ast.exp) -> Gcl.t -> Ast.exp -> Ast.exp
```

efficient_wp p q computes wp(p,q) using an algorithm that guarantees the resulting precondition will be linear in the size of p. efficient_wp expects p to be assignment-free, e.g., to be an SSA acyclic program. See CMU-CS-08-159.pdf (Brumley's thesis), chapter 3.3.

```
val flanagansaxe :
  ?simp:(Ast.exp -> Ast.exp) ->
```

```
  ?less_duplication:bool ->
  ?k:int -> Type.formula_mode -> Gcl.t -> Ast.exp -> Ast.exp
```

`flanagansaxe mode p q` computes `wp(p,q)` using Flanagan and Saxe's algorithm. The `mode` argument specifies whether the formula will be solved for satisfiability or validity.

Directionless Weakest Precondition Algorithms

```
val dwp_1st :
  ?simp:(Ast.exp -> Ast.exp) ->
  ?less_duplication:bool ->
  ?k:int -> Gcl.t -> Ast.exp -> Ast.var list * Ast.exp
```

`dwp_1st p q` returns a tuple `(vars, pc)` where `pc` is `wp(p,q)` and `vars` is a list of variables that should be quantified using foralls.

```
val dwp :
  ?simp:(Ast.exp -> Ast.exp) ->
  ?less_duplication:bool ->
  ?k:int -> Type.formula_mode -> Gcl.t -> Ast.exp -> Ast.exp
```

`dwp mode p q` is the same as `Wp.dwp_1st`[95], except it generates a precondition that does not need quantifiers. However, the `mode` argument must be used to specify whether the formula will be used for satisfiability or validity.

```
val dwp_let :
  ?simp:(Ast.exp -> Ast.exp) ->
  ?less_duplication:bool ->
  ?k:int -> Type.formula_mode -> Gcl.t -> Ast.exp -> Ast.exp
```

`dwp_let` is just like `Wp.dwp`[95], except that `dwp_let` wraps helper variables in `Let` expressions so they do not appear as free variables.

Utility Functions for Building WP Algorithms

```
val variableify :
  ?name:string ->
  int ->
  (Ast.var * Ast.exp) list -> Ast.exp -> (Ast.var * Ast.exp) list * Ast.exp
```

When given a list of variable assignments `v`, `variableify s v e` returns a tuple `(v',e')` where `v'` is an updated list of variable assignments, and `e'` is guaranteed to be at most `s` in size. Use this function to avoid duplicating large expressions in formulas.

```
val assignments_to_exp : (Ast.var * Ast.exp) list -> Ast.exp
```

Convert a list of variable assignments, such as those returned by `Wp.variableify`[95], to a single expression.

```
val assignments_to_lets : (Ast.var * Ast.exp) list -> Ast.exp -> Ast.exp
```

`assignments_to_lets vars e` converts a list of variable assignments, such as those returned by `Wp.variableify`[95], to let bindings surrounding expression `e`.