New Jersey Machine-Code Toolkit
Source Code

Norman Ramsey and Mary F. Fernandez

March 7, 1996

Contents

Part I

Icon code

Chapter 1

Main program

9a ⟨*version string* 9a⟩≡ (18)

```
"0.4 of 1/25/96"
```

9b ⟨* 9b⟩≡ 9c ▷

```
link pushtrace
```

9c ⟨* 9b⟩+≡ ◁ 9b 11c ▷

```
procedure usage()
    every write(&errout, ![
        "Usage: tools [options] [specfile ...]",
        "Options:",
        ⟨usage lines 9e⟩
        "'-' may be used in place of a file name to mean stdin or stdout"])
    stop()
end
```

Defines:
  `usage`, used in chunk 18.
Uses `file` 345a and `used` 132a.

9d ⟨*cases for arguments* 9d⟩≡ (18) 9f ▷

```
"-alt"        : alt := &output
```

9e ⟨*usage lines* 9e⟩≡ (9c) 9g ▷

```
"    -ascii-dag file   write an ascii representation of matching-statement code",
```

Uses `file` 345a.

9f ⟨*cases for arguments* 9d⟩+≡ (18) ◁ 9d 10a ▷

```
"-ascii-dag" : ascii_dag   := openfile(get(args),"w")
```

Uses `openfile` 22.

9g ⟨*usage lines* 9e⟩+≡ (9c) ◁ 9e 10b ▷

```
"    -ascii-tree file  write an ascii representation of matchin-statement trees",
```

Uses `file` 345a and `tree` 262.

10a ⟨*cases for arguments* 9d⟩+≡ (18) ◁9f 10c▷
```
"-ascii-tree" : ascii_tree := openfile(get(args),"w")
```
Uses `openfile` 22 and `tree` 262.

10b ⟨*usage lines* 9e⟩+≡ (9c) ◁9g 10d▷
```
"    -asm-encoder file generate assembly-emitting encoding procedures on file.[ch]",
```
Uses `file` 345a.

10c ⟨*cases for arguments* 9d⟩+≡ (18) ◁10a 10e▷
```
"-asm-encoder" : asmencoderfilename  := get(args)
```

10d ⟨*usage lines* 9e⟩+≡ (9c) ◁10b 10f▷
```
"    -asm-grammar file generate assembly-language grammar on file",
```
Uses `file` 345a.

10e ⟨*cases for arguments* 9d⟩+≡ (18) ◁10c 10g▷
```
"-asm-grammar" : asmgrammarfilename  := get(args)
```
Uses `asmgrammarfilename` 16a.

10f ⟨*usage lines* 9e⟩+≡ (9c) ◁10d 10h▷
```
"    -byteorder O      use order O (one of [blm]) to emit words",
```

10g ⟨*cases for arguments* 9d⟩+≡ (18) ◁10e 10i▷
```
"-byteorder"  : emittername := "emit" || get(args)
```
Uses `emittername` 16a.

10h ⟨*usage lines* 9e⟩+≡ (9c) ◁10f 10j▷
```
"    -c                generate C code [default]",
```

10i ⟨*cases for arguments* 9d⟩+≡ (18) ◁10g 10k▷
```
"-c" : { Generate_C(); interface_extension := ".h"; implementation_extension := ".c" }
```
Uses `Generate_C` 323a, `implementation_extension` 16a, and `interface_extension` 16a.

10j ⟨*usage lines* 9e⟩+≡ (9c) ◁10h 10l▷
```
"    -checker file    generate a checker on file",
```
Uses `file` 345a.

10k ⟨*cases for arguments* 9d⟩+≡ (18) ◁10i 10m▷
```
"-checker"    :  { checkerfilename := get(args) }
```

10l ⟨*usage lines* 9e⟩+≡ (9c) ◁10j 11a▷
```
"    -count            count invocations of encoding procedures",
```

10m ⟨*cases for arguments* 9d⟩+≡ (18) ◁10k 11b▷
```
"-count"      : gen_counters := 1
```
Uses `gen_counters` 16a.

ge #17

11a ⟨*usage lines* 9e⟩+≡ (9c) ◁10l 11d▷
```
"    -debug-bal file   extra debugging info for balancer",
```
Uses `debug` 362a and `file` 345a.

11b ⟨*cases for arguments* 9d⟩+≡ (18) ◁10m 11e▷
```
"-debug-bal" : baldebug := openfile(get(args),"w")
```
Uses `baldebug` 11c, `debug` 362a, and `openfile` 22.

11c ⟨* 9b⟩+≡ ◁9c 14k▷
```
global baldebug
```
Defines:
`baldebug`, used in chunks 11b and 153.

11d ⟨*usage lines* 9e⟩+≡ (9c) ◁11a 11f▷
```
"    -debug-checker file  write debugging info about checker",
```
Uses `debug` 362a and `file` 345a.

11e ⟨*cases for arguments* 9d⟩+≡ (18) ◁11b 11g▷
```
"-debug-checker" : cdebug := openfile(get(args),"w")
```
Uses `cdebug` 16a, `debug` 362a, and `openfile` 22.

11f ⟨*usage lines* 9e⟩+≡ (9c) ◁11d 11h▷
```
"    -debug-heur file  write debugging info about heuristic scores",
```
Uses `debug` 362a and `file` 345a.

11g ⟨*cases for arguments* 9d⟩+≡ (18) ◁11e 11i▷
```
"-debug-heur" : hdebug := openfile(get(args),"w")
```
Uses `debug` 362a, `hdebug` 16a, and `openfile` 22.

11h ⟨*usage lines* 9e⟩+≡ (9c) ◁11f 11j▷
```
"    -debug-match file write debugging info about matching statements",
```
Uses `debug` 362a and `file` 345a.

11i ⟨*cases for arguments* 9d⟩+≡ (18) ◁11g 11k▷
```
"-debug-match" : mdebug := openfile(get(args),"w")
```
Uses `debug` 362a, `mdebug` 16a, and `openfile` 22.

11j ⟨*usage lines* 9e⟩+≡ (9c) ◁11h 11l▷
```
"    -debug-solver     turn on solver debugging",
```
Uses `debug` 362a.

11k ⟨*cases for arguments* 9d⟩+≡ (18) ◁11i 12a▷
```
"-debug-solver" : debug_on()
```
Uses `debug` 362a and `debug_on` 362a.

11l ⟨*usage lines* 9e⟩+≡ (9c) ◁11j 12b▷
```
"    -debug-split file write debugging info when splitting tree nodes",
```
Uses `debug` 362a, `file` 345a, `split` 265a, and `tree` 262.

12a ⟨*cases for arguments* 9d⟩+≡ (18) ◁11k 12c▷
```
"-debug-split" : sdebug := openfile(get(args),"w")
```
Uses `debug` 362a, `openfile` 22, `sdebug` 16a, and `split` 265a.

12b ⟨*usage lines* 9e⟩+≡ (9c) ◁11l 12f▷
```
"    -decoder file     from matcher, generate decoder on file",
```
Uses `file` 345a.

12c ⟨*cases for arguments* 9d⟩+≡ (18) ◁12a 12g▷
```
"-decoder" : decoderout := openfile(get(args),"w")
```
Uses `openfile` 22.

12d ⟨*bogus usage lines* 12d⟩≡
```
"    -dis file         generate disassembly code on file",
```
Uses `file` 345a.

12e ⟨*bogus cases for arguments* 12e⟩≡
```
"-dis"        : disassemblyfile  := openfile(get(args),"w")
```
Uses `openfile` 22.

12f ⟨*usage lines* 9e⟩+≡ (9c) ◁12b 12h▷
```
"    -dot file         write dot(1) commands showing matching statements",
```
Uses `file` 345a.

12g ⟨*cases for arguments* 9d⟩+≡ (18) ◁12c 12i▷
```
"-dot" : dotfile   := openfile(get(args),"w")
```
Uses `openfile` 22.

12h ⟨*usage lines* 9e⟩+≡ (9c) ◁12f 12j▷
```
"    -emit-bits n      unit of emission is n bits (default 8)",
```

12i ⟨*cases for arguments* 9d⟩+≡ (18) ◁12g 12k▷
```
"-emit-bits"   : emit_unit_bits := integer(get(args)) | stop("-emitbits not integer")
```
Uses `emit_unit_bits` 16a.

12j ⟨*usage lines* 9e⟩+≡ (9c) ◁12h 12l▷
```
"    -encoder file     generate encoding procedures on file.[ch]",
```
Uses `file` 345a.

12k ⟨*cases for arguments* 9d⟩+≡ (18) ◁12i 12m▷
```
"-encoder" : encoderfilename  := get(args)
```

12l ⟨*usage lines* 9e⟩+≡ (9c) ◁12j 13a▷
```
"    -expand-spec file writes an expanded version of the input specification",
```
Uses `file` 345a, `input` 44a, and `version` 16a.

12m ⟨*cases for arguments* 9d⟩+≡ (18) ◁12k 13b▷
```
"-expand-spec" : spec  := openfile(get(args),"w")
```
Uses `openfile` 22.

13a ⟨*usage lines* 9e⟩+≡ (9c) ◁12l 13c▷
```
  "    -fieldnames file  write arrays of field names on file.[ch]",
```
Uses `field` 53 and `file` 345a.

13b ⟨*cases for arguments* 9d⟩+≡ (18) ◁12m 13d▷
```
  "-fieldnames" : fieldnamesbase := get(args)
```
Uses `fieldnamesbase` 16a.

13c ⟨*usage lines* 9e⟩+≡ (9c) ◁13a 13e▷
```
  "    -foldemit         fold constants in calls to emit functions",
```

13d ⟨*cases for arguments* 9d⟩+≡ (18) ◁13b 13f▷
```
  "-foldemit"   : simplify_emits := 1
```
Uses `simplify_emits` 16a.

13e ⟨*usage lines* 9e⟩+≡ (9c) ◁13c 13g▷
```
  "    -impossible       force errors to stop with stack trace",
```

13f ⟨*cases for arguments* 9d⟩+≡ (18) ◁13d 13h▷
```
  "-impossible" : error := stop := impossible
```

13g ⟨*usage lines* 9e⟩+≡ (9c) ◁13e 13i▷
```
  "    -indirect name[:type] generated encoding procs called indirect through name",
```

13h ⟨*cases for arguments* 9d⟩+≡ (18) ◁13f 13j▷
```
  "-indirect"   : get(args) ?
                    { indirectname := tab(upto(':') | 0)
                      indirecttype := if =":" then tab(0) else indirectname
                    }
```
Uses `indirectname` 16a and `indirecttype` 16a.

13i ⟨*usage lines* 9e⟩+≡ (9c) ◁13g 13k▷
```
  "    -late-const       include constants in closure functions",
```

13j ⟨*cases for arguments* 9d⟩+≡ (18) ◁13h 13l▷
```
  "-late-const" : lateconst := 1
```
Uses `lateconst` 16a.

13k ⟨*usage lines* 9e⟩+≡ (9c) ◁13i 13m▷
```
  "    -late-zero        include zero in closure functions",
```

13l ⟨*cases for arguments* 9d⟩+≡ (18) ◁13j 14a▷
```
  "-late-zero" : latezero := 1
```
Uses `latezero` 16a.

13m ⟨*usage lines* 9e⟩+≡ (9c) ◁13k 14b▷
```
  "    -lc-cons-names    use all lower case for constructor names",
```
Uses `constructor` 155 and `lc` 382b.

14a ⟨*cases for arguments* 9d⟩+≡ (18) ◃13l 14c▹
```
  "-lc-cons-names" : lowercons := 1
```
Uses `lc` 382b and `lowercons` 16a.

14b ⟨*usage lines* 9e⟩+≡ (9c) ◃13m 14d▹
```
  "    -lc-pat-names     map pattern names to lower case in code",
```
Uses `lc` 382b and `pattern` 53.

14c ⟨*cases for arguments* 9d⟩+≡ (18) ◃14a 14e▹
```
  "-lc-pat-names" : lc_pat_names := 1
```
Uses `lc` 382b and `lc_pat_names` 16a.

14d ⟨*usage lines* 9e⟩+≡ (9c) ◃14b 14f▹
```
  "    -ledger           tell dot(1) to use ledger paper with -dot",
```

14e ⟨*cases for arguments* 9d⟩+≡ (18) ◃14c 14g▹
```
  "-ledger"     : {ledger := 1; pagesize := "11,17"}
```
Uses `pagesize` 360.

14f ⟨*usage lines* 9e⟩+≡ (9c) ◃14d 14h▹
```
  "    -matcher file     transform pattern-matching statements in file",
```
Uses `file` 345a and `pattern` 53.

14g ⟨*cases for arguments* 9d⟩+≡ (18) ◃14e 14i▹
```
  "-matcher"    : matcher := openfile(get(args),"r")
```
Uses `openfile` 22.

14h ⟨*usage lines* 9e⟩+≡ (9c) ◃14f 14l▹
```
  "    -max-decimal n    values larger than n will be printed in hex",
```
Uses `max` 396.

14i ⟨*cases for arguments* 9d⟩+≡ (18) ◃14g 15a▹
```
  "-max-decimal" : max_decimal := 0 <= get(args) |
                     error("-max-decimal value must be nonnegative integer")
```
Uses `max` 396, `max_decimal` 14k, and `value` 132a.

14j ⟨*initialize globals* 14j⟩≡ (18) 16b▹
```
  max_decimal := 2^31 - 1
```
Uses `max_decimal` 14k.

14k ⟨* 9b⟩+≡ ◃11c 15j▹
```
  global max_decimal
```
Defines:
`max_decimal`, used in chunks 14 and 312a.

14l ⟨*usage lines* 9e⟩+≡ (9c) ◃14h 15b▹
```
  "    -m3               generate Modula-3 code",
```

15a ⟨*cases for arguments* 9d⟩+≡ (18) ◁14i 15c▷
```
"-m3"         : { Generate_M3();
                  interface_extension := ".i3"; implementation_extension := ".m3" }
```
Uses `Generate_M3` 334, `implementation_extension` 16a, and `interface_extension` 16a.

15b ⟨*usage lines* 9e⟩+≡ (9c) ◁14l 15d▷
```
"    -nowarn           no warnings",
```
Uses `nowarn` 16a.

15c ⟨*cases for arguments* 9d⟩+≡ (18) ◁15a 15e▷
```
"-nowarn" : nowarn := 1
```
Uses `nowarn` 16a.

15d ⟨*usage lines* 9e⟩+≡ (9c) ◁15b 15f▷
```
"    -old-closures     use the old closure technique",
```

15e ⟨*cases for arguments* 9d⟩+≡ (18) ◁15c 15g▷
```
"-old-closures" : emit_closure_functions := emit_original_closure_functions
```
Uses `emit_closure_functions` 16a and `emit_original_closure_functions` 195b.

15f ⟨*usage lines* 9e⟩+≡ (9c) ◁15d 15h▷
```
"    -tryall           try all heuristics on every field, node (for debugging)",
```
Uses `field` 53, `node` 260a, and `tryall` 16a.

15g ⟨*cases for arguments* 9d⟩+≡ (18) ◁15e 15i▷
```
"-tryall"     : tryall := 1
```
Uses `tryall` 16a.

15h ⟨*usage lines* 9e⟩+≡ (9c) ◁15f
```
"    -verbose          extra warnings & informatory messages",
```
Uses `verbose` 16a.

15i ⟨*cases for arguments* 9d⟩+≡ (18) ◁15g
```
"-verbose"    : verbose := info_msg
```
Uses `info_msg` 15j and `verbose` 16a.

15j ⟨* 9b⟩+≡ ◁14k 16a▷
```
procedure info_msg(L[])
  write ! ([&errout, image(filename), ", line ", lineno, ": Info -- "] ||| L)
end
```
Defines:
  `info_msg`, used in chunk 15i.
Uses `line` 345a.

16a ⟨* 9b⟩+≡ ◁15j 17b▷

```
global cdebug, mdebug, sdebug, hdebug, tryall, indirectname, indirecttype
global lc_pat_names, nowarn, emptyset, emittername, pretty
global lowercons, pc_unit_bits, emit_unit_bits, simplify_emits, gen_counters
global fieldnamesbase, verbose, version
global interface_extension, implementation_extension
global emit_closure_functions, latezero, lateconst
global asmgrammarfilename
```

Defines:
- asmgrammarfilename, used in chunks 10e, 16b, and 18.
- cdebug, used in chunks 11e, 294, 296, 300, 301a, and 303.
- emit_closure_functions, used in chunks 15e, 17a, and 204a.
- emittername, used in chunks 10g, 17a, 314c, and 328b.
- emit_unit_bits, used in chunks 12i, 17a, and 187.
- emptyset, used in chunks 17a, 133b, and 355b.
- fieldnamesbase, used in chunks 13b and 18.
- gen_counters, used in chunks 10m, 16b, 204a, 218–20, and 224a.
- hdebug, used in chunks 11g, 16b, 264, and 268a.
- implementation_extension, used in chunks 10i, 15a, 17a, 217, 223a, 294, and 342c.
- indirectname, used in chunks 13h, 203b, 204a, 218–20, 222a, 224a, 233, 242, 294, 295, and 323b.
- indirecttype, used in chunks 13h, 218, and 219.
- interface_extension, used in chunks 10i, 15a, 17a, 217, 223a, 294, and 342c.
- lateconst, used in chunks 13j and 207.
- latezero, used in chunks 13l and 211a.
- lc_pat_names, used in chunks 14c, 16b, and 262.
- lowercons, used in chunks 14a, 16b, 160b, and 226a.
- mdebug, used in chunks 11i, 164, 169a, 248, 251, and 258a.
- nowarn, used in chunk 15.
- pc_unit_bits, used in chunks 17a, 32c, 62b, and 289c.
- pretty, used in chunks 17a, 19a, 20a, 197a, 199b, 202d, 204a, 205b, 216, 217, 223a, 236, 238, 295, 323a, 334, and 342c.
- sdebug, used in chunks 12a, 16b, 262, 263a, and 265–67.
- simplify_emits, used in chunks 13d and 188c.
- tryall, used in chunks 15, 16b, and 264c.
- verbose, used in chunks 15, 17a, 18, 31c, 41c, 57a, 156, 215b, 218a, 220a, 223a, 224a, 241a, 262, 294, 302a, and 303.
- version, used in chunks 12l, 18, and 370a.

16b ⟨*initialize globals* 14j⟩+≡ (18) ◁14j 17a▷

```
dotfile := ascii_dag := spec := ascii_tree := decoderout :=
  fulltree := sdebug := hdebug := encoderfilename := asmencoderfilename :=
  ledger := tryall := asmgrammarfilename :=
  lc_pat_names := alt := matcher := gen_counters := lowercons :=
  &null
```

Uses asmgrammarfilename 16a, gen_counters 16a, hdebug 16a, lc_pat_names 16a, lowercons 16a, sdebug 16a, and tryall 16a.

17a ⟨*initialize globals* 14j⟩+≡ (18) ◁16b 17c▷

```
emittername := "emitm"
emptyset := set()
verbose := nop
filename := "-"
pagesize := "8.5,11"
fetchtab := table()
pc_unit_bits := 8
emit_unit_bits := 8
fresh_variables := set()
pretty := prettyC
interface_extension := ".h"
implementation_extension := ".c"
debug := debugs := nop
emit_closure_functions := emit_optimized_closure_functions
```

Uses `debug` 362a, `debugs` 362a, `emit_closure_functions` 16a, `emitm` 378a, `emit_optimized_closure_functions` 204c, `emittername` 16a, `emit_unit_bits` 16a, `emptyset` 16a, `fetchtab` 283a, `fresh_variables` 151b, `implementation_extension` 16a, `interface_extension` 16a, `nop` 362a, `pagesize` 360, `pc_unit_bits` 16a, `pretty` 16a, `prettyC` 311, and `verbose` 16a.

It's a hack, but I use the global variable `emitterstyle` to determine whether code for `Semit` and `Stoken` should use a *direct* call to the emitter named by `emittername`, or whether it should use a style appropriate to a *closure*.

17b ⟨* 9b⟩+≡ ◁16a 17d▷

```
global emitterstyle # closure or direct, controls emission
```

Defines:
`emitterstyle`, used in chunks 17c, 199b, 216, 314b, and 315a.

17c ⟨*initialize globals* 14j⟩+≡ (18) ◁17a 17e▷

```
emitterstyle := "direct"
```

Uses `emitterstyle` 17b.

Used to make sure the program counter is unique, so that `pc - pc` is zero always.

17d ⟨* 9b⟩+≡ ◁17b 18▷

```
global the_global_pc
```

Defines:
`the_global_pc`, used in chunks 17e, 76a, 87c, 185a, 199b, 215b, 252, and 258a.

17e ⟨*initialize globals* 14j⟩+≡ (18) ◁17c

```
the_global_pc := Epc()
```

Uses `Epc` 79 and `the_global_pc` 17d.

I have to split up `main`, otherwise I get too many definitions which leads to a line more than 3000 characters wide, and TeX dies.

18 ⟨* 9b⟩+≡ ◁17d 19b▷
```
procedure main(args)
    ⟨initialize globals 14j⟩
    version := ⟨version string 9a⟩

    Generate_C()        # default
    while args[1][1:2] == "-" & *args[1] > 1 do
        case a := get(args) of {
            ⟨cases for arguments 9d⟩
            default    : usage()
       }

    verbose("NJ Machine-Code Toolkit, version ", version)

    init_parser()
    init_tests()        # see code.nw

    if /quiet then write(&errout, "Parsing...")
    if *args > 0 then
        every consume(openfile(filename := !args,"r"))
    else
        consume(&input)

    if /quiet then write(&errout, "Done parsing.")
    emit_encoding(\encoderfilename)
    emit_assembly(\asmencoderfilename)
    emit_asm_grammar(\asmgrammarfilename)
    emit_fieldnames(\fieldnamesbase)
    emit_checker(\checkerfilename)
    if \ledger then { ⟨write PostScript for ledger paper 19e⟩ }

    outspec(\spec)
    if \matcher then {⟨handle matching statements 19a⟩}
end
```
Defines:
  `main`, used in chunks 291a, 293, and 308.

Uses `asmgrammarfilename` 16a, `consume` 19c, `emit_asm_grammar` 241a, `emit_assembly` 223a, `emit_checker` 294, `emit_encoding` 217, `emit_fieldnames` 342c, `fieldnamesbase` 16a, `Generate_C` 323a, `init_parser` 30b, `init_tests` 335b, `input` 44a, `openfile` 22, `outspec` 353a, `usage` 9c, `verbose` 16a, and `version` 16a.

19a ⟨*handle matching statements* 19a⟩≡ (18)
```
codelex(matcher)
P_CodeFile()
⟨if tokens are left over, complain 19d⟩
x := super_simplify(genheader(codeheader))
PPxwrite(PPnew(\decoderout), pretty(x))
every resolve_case_patterns(!matching_stmts)
every t := tree(cs := !matching_stmts) do { ⟨manipulate decision tree t 20a⟩ }
```
Uses `codeheader` 48c, `codelex` 351a, `genheader` 282d, `matching_stmts` 48c, `pretty` 16a, `resolve_case_patterns` 247, `super_simplify` 90a, and `tree` 262.

19b ⟨* 9b⟩+≡ ◁18 19c▷
```
global showpnf
```
Defines:
`showpnf`, used in chunk 73.

19c ⟨* 9b⟩+≡ ◁19b 20b▷
```
procedure consume(input)
    lex(input)
    P_Spec()
    ⟨if tokens are left over, complain 19d⟩
    return
end
```
Defines:
`consume`, used in chunk 18.
Uses `input` 44a.

19d ⟨*if tokens are left over, complain* 19d⟩≡ (19)
```
while (token ~== EOF) do {
    error("Leftover token ", token, " = ", image(tval))
    lex()
}
```
Uses `EOF` 345b, `token` 345a, and `tval` 345a.

19e ⟨*write PostScript for ledger paper* 19e⟩≡ (18)
```
write(\dotfile,"%!  PostScript")
write(\dotfile,"/setpapertraybyname {")
write(\dotfile,"  { /trayname exch def statusdict trayname known {")
write(\dotfile,"    statusdict trayname get exec } if} stopped")
write(\dotfile,"    { handleerror } if}bind def")
write(\dotfile,"/ledgertray setpapertraybyname")
```

20a ⟨*manipulate decision tree* `t` 20a⟩≡ (19a)

```
checktree(t, cs)
outtree(\ascii_tree, t)
if \dotfile | \ascii_dag | \decoderout | \alt then {
    d := tree2dag(t)
    outtree(\ascii_dag, d)
    dotouttree(\dotfile, d)
    PPxwrite(PPnew(\decoderout), pretty(super_simplify(gencase(cs, d))))
    printalternates(\alt, d)
    write(\alt,"\n", repl("=", 50))
}
```

Uses `checktree` 272a, `dotouttree` 360, `gencase` 283b, `outtree` 359b, `pretty` 16a, `printalternates` 21a, `super_simplify` 90a, and `tree2dag` 278b.

20b ⟨* 9b⟩+≡ ◁19c 21a▷

```
record alternator(constraints, thennode, elsenode)

procedure alternates(n)
    local alt
    if *n.children = 2 then {
        every i := 1 to 2 do
            if alt := alternates(n.children[i].node) &
               n.children[3-i].node === alt.elsenode then
                   suspend alternator(alt.constraints|||
                       [constraint(n.field, n.children[i].lo, n.children[i].hi)],
                       alt.thennode, alt.elsenode)
        every i := 1 to 2 do
            suspend alternator(
                       [constraint(n.field, n.children[i].lo, n.children[i].hi)],
                       n.children[i].node, n.children[3-i].node)
    }
end
```

Defines:
`alternates`, used in chunks 21a and 360.
Uses `constraint` 53, `constraints` 132a, `field` 53, and `node` 260a.

21a ⟨* 9b⟩+≡ ◁20b 21b▷

```
procedure printalternates(file, n)
    local alt
    if alt := alternates(n) then {
        write(file)
        writes(file, "if (")
        l := copy(alt.constraints)
        while genconstraint(file, get(l)) do writes(file, " && ")
        write(file, "1) {")
        writes(file, "    /*")
        writes(file, \alt.thennode.cs.arms[1].original.pattern.name |
                        "[" || alt.thennode.cs.arms[1].original.line || "]")
        write(file, " */")
        write(file, "} else {")
        writes(file, "    /*")
        writes(file, \alt.elsenode.cs.arms[1].original.pattern.name |
                        "[" || alt.elsenode.cs.arms[1].original.line || "]")
        write(file, " */")
        write(file, "} else assert(0);")
    } else
        every printalternates(file, (!n.children).node)
end
```

Defines:
`printalternates`, used in chunk 20a.
Uses `alternates` 20b, `constraints` 132a, `file` 345a, `genconstraint` 21b, `line` 345a, `node` 260a, and `pattern` 53.

21b ⟨* 9b⟩+≡ ◁21a 22▷

```
procedure genconstraint(file, c)
    r := sort(c.lo ++ c.hi)
    writes(file, "(", get(r), "<=", c.field.name, "<", get(r))
    while writes(file, " || ", get(r), "<=", c.field.name, "<", get(r))
    writes(file, ")")
    return
end
```

Defines:
`genconstraint`, used in chunk 21a.
Uses `field` 53 and `file` 345a.

openfile makes it possible to open standard input or output using the file name "-."

22 ⟨* 9b⟩+≡ ◁21b

```
procedure openfile(name, mode)
    /mode := "r"
    if name == "-" then
       case mode of {
          "r" : return &input
          "w" : return &output
          default : stop("bogus file mode: ", mode)
       }
    else
      return open(name, mode) | stop("Can't open file ", name, " for ", mode)
end
```

Defines:
openfile, used in chunks 9–12, 14g, 18, 223a, 241a, 294, and 295.
Uses bogus 113b, file 345a, and input 44a.

Chapter 2

Environments and environment errors

An environment is a stack of symbol tables. The default environment is the top-level environment, `globals`. `newscope` creates a copy of an environment that can be added to without having a side effect on the original. `add_to_rho` does the adding.

`lookup` searches for the first binding of string `val` in the environment. `lookuptype` searches for `val` in the environment and if it exists, checks that its type is `ty`.

23a ⟨* 23a⟩≡ 24a ▷

```
procedure is_defined(ident, rho)
  return (⟨value of ident in rho 23b⟩)   # parens handle newline from -L
end
procedure lookup(ident, rho)
  return  (⟨value of ident in rho 23b⟩) | error("‘", ident, "’ is undefined")
end
procedure lookuptype(ident, ty, rho)
  type(v := ⟨value of ident in rho 23b⟩) == ty | typeerror(v, ty, ident, rho)
  return v
end
```

Defines:
`is_defined`, used in chunks 74b, 171, and 172.
`lookup`, used in chunks 73, 74b, 159b, 171, 172, and 354b.
`lookuptype`, used in chunks 35, 40a, 43d, 50b, 73, and 297a.
Uses `typeerror` 26b.

23b ⟨*value of* `ident` *in* `rho` 23b⟩≡ (23a)

```
{ /rho := globals; \(!rho)[ident] }
```

Uses `globals` 30a.

24a ⟨* 23a⟩+≡ ◁23a 24b▷

```
procedure add_to_rho(name, val, rho)
#write("rho ", *rho, " ", name, " ", expimage(val))
    add_to_frame(name, val, rho[1]) | impossible("bogus environment")
    return rho
end

procedure add_to_frame(name, val, frame)
    (/frame[name] := val) | deferror(name)
    return frame
end
```

Defines:
`add_to_frame`, used in chunk 158b.
`add_to_rho`, used in chunks 54, 165, and 250a.
Uses `bogus` 113b, `deferror` 26a, and `expimage` 98a.

24b ⟨* 23a⟩+≡ ◁24a 24c▷

```
procedure newscope(rho)
    return push(copy(rho), table())
end
```

Defines:
`newscope`, used in chunks 165 and 248.

Here we extend an environment with a new frame. I take care not to modify the environment.

24c ⟨* 23a⟩+≡ ◁24b 25a▷

```
procedure extendscope(rho, frame)
  (type(frame) == "table" & type(rho) == "list") | impossible("rho extension")
  return push(copy(rho), frame)
end
```

Defines:
`extendscope`, used in chunk 158b.

25a ⟨* 23a⟩+≡ ◁24c 25c▷

```
procedure envimage(env, envname)
    local hidden
    /envname := "env"
    s := ""
    ⟨if env is a list, convert to a table and create string hidden 25b⟩
    if *env = 0 then s ||:= "\nEnvironment " || envname || " is empty"
    every p := !sort(env) do s ||:= pairimage(envname, p[1], p[2])
    if \hidden then {
       s ||:= "  -------- hidden --------\n"
       return s || hidden
    } else
       return s
end
```

Defines:
envimage, used in chunks 87a, 153, 248, 253a, 258a, and 363c.
Uses pairimage 25c.

25b ⟨*if* env *is a list, convert to a table and create string* hidden 25b⟩≡ (25a)

```
if type(env) == "list" then {
  t := table()
  every e := !env & ident := key(e) do
    (/t[ident] := e[ident]) |
    { /hidden := "" ; hidden ||:= pairimage(envname, ident, e[ident]) }
  env := t
}
```

Uses pairimage 25c.

25c ⟨* 23a⟩+≡ ◁25a 26a▷

```
procedure pairimage(envname, ident, v)
  return "\n  " || envname || "[" || expimage(ident) ||"]" || " = " ||
    case type(v) of {
#      "pattern" : "<pattern> " || patternimage(v)
#      "field"   : "<field> "   || fieldimage(v)
      "string"  : image(v)
      default   : expimage(v)
    }
end
```

Defines:
pairimage, used in chunk 25.
Uses expimage 98a, field 53, fieldimage 353b 353b, pattern 53, and patternimage 355b.

26a ⟨* 23a⟩+≡ ◁25c 26b▷

```
procedure deferror(t, v)
    error(t, " ", v," is already defined.")
end
```

Defines:
 `deferror`, used in chunks 24a, 33, and 45d.
Uses `defined` 132a.

26b ⟨* 23a⟩+≡ ◁26a 27▷

```
procedure typeerror(x, typename, ident, rho)
    error("Expected ", (\ident || " to be a " | ""), typename,
          "; found ", type(x), " ", expimage(x))
end
```

Defines:
 `typeerror`, used in chunks 23a, 44b, 73, 74b, and 297a.
Uses `expimage` 98a.

26c ⟨*name of* `rho` 26c⟩≡

```
if rho === globals then "globals" else "rho"
```

Uses `globals` 30a.

2.1 Injection and projection

Sometimes an identifier could have more than one meaning, depending on the precise context in which it is used. For example, the name of a constructor operand denotes an instance when it is used in an application, but a pattern when it is used as a pattern identifier. We use "injection" to define the meaning of such an identifier; the meaning will be "projected" when the identifier is actually looked up. Meanings can be projected into three spaces: patterns, integers, and constructor operands. Projection never produces a null value; null values are used to indicate inability to project, and `project` fails.

27 ⟨* 23a⟩+≡ ◁26b

```
record inject(pattern, integer, consop)

procedure project(x, ty)
  return if type(x) == ty then x
         else if type(x) == "inject" then case ty of {
           "pattern" : \x.pattern
           "integer" : \x.integer
           "consop"  : \x.consop
         } else if ty == "integer" then case type(x) of {
           "pattern" | "input" : fail
           exptypes() : x
           default    : impossible("I need to rethink projection")
         }
end
```

Defines:
`inject`, used in chunks 116b, 122c, 158b, 165, 166b, 248, and 250a.
`project`, used in chunks 73, 74b, 171, and 172.
Uses `exptypes` 81b, `fail` 396, `input` 44a, and `pattern` 53.

I don't really know if the last clause here is right, but it was better than simply allowing anything to be projected into an integer, which I think would be unwarranted optimism.

Chapter 3

Grammar for the specification language

29 ⟨* 29⟩≡ 34b ▷

⟨*terminal symbols* 51d⟩
```
%%
# EBNF grammar for toolkit specification language
```
⟨*grammar* 32a⟩
```
%%
```
⟨*toplevel* 30a⟩

Step one is to initialize the globals

30a ⟨*toplevel* 30a⟩≡ (29) 30b ▷

```
global symtab           # table of top-level symbols
global globals          # top-level environment
global equivclasses     # list of equivalence classes of fields
global constructors     # table mapping constructor name to Stype
global conslist         # list of all constructors ever defined
global instructionctype # the type of instruction constructors
global bit_zero_is_lsb  # non-null if bits are numbered with 0 == least significant
global vanishing_latent_patlabel  # should vanish
global fieldname_literals, operands_and_ids, warned_literals # find conflicts
```

Defines:
- `bit_zero_is_lsb`, used in chunks 30b, 33c, 34a, and 112b.
- `conslist`, used in chunks 30b, 31d, and 156.
- `constructors`, used in chunks 30b, 31d, 42a, 46b, 156, 161, 163, 178c, 193b, 194a, and 218b.
- `equivclasses`, used in chunks 30b and 33a.
- `fieldname_literals`, used in chunks 30 and 31.
- `globals`, used in chunks 23b, 26c, 30b, 35a, 41a, 44b, 156, and 248.
- `instructionctype`, used in chunks 30b, 42a, 162, 202e, 218–20, 222b, 224, 226, 233, 234, 241b, 242, and 295.
- `operands_and_ids`, used in chunks 30 and 31.
- `symtab`, used in chunks 30b, 33, 37b, 43–45, 48b, 112, 166b, 168b, 182b, 230, 235, 250a, 343a, 354a, and 355a.
- `vanishing_latent_patlabel`, used in chunks 30b, 91b, 97a, 114c, 118a, 174d, and 175.
- `warned_literals`, used in chunks 30b and 31c.

Uses `constructor` 155 and `defined` 132a.

30b ⟨*toplevel* 30a⟩+≡ (29) ◁ 30a 31d ▷

```
procedure init_parser()
    every symtab | constructors := table()
    globals := [symtab]
    every unchecked_fields | guaranteed_fields := set()
    every fieldname_literals | operands_and_ids | warned_literals := set()
    every conslist | equivclasses := []
    instructionctype := constype("(anonymous constructor type)", set())
    bit_zero_is_lsb := 1
    vanishing_latent_patlabel := latent_patlabel()
    ⟨initialization 36d⟩
end
```

Defines:
- `init_parser`, used in chunk 18.

Uses `bit_zero_is_lsb` 30a, `conslist` 30a, `constructor` 155, `constructors` 30a, `constype` 155, `equivclasses` 30a 353a, `fieldname_literals` 30a, `globals` 30a, `guaranteed_fields` 340f, `instructionctype` 30a, `latent_patlabel` 53, `operands_and_ids` 30a, `symtab` 30a 353a, `unchecked_fields` 340f, `vanishing_latent_patlabel` 30a, and `warned_literals` 30a.

Because of the strange way we allow users to specify field-name literals in field bindings, we want to make sure there's never a clash between the names used in fieldinfo statements and the names used in operands and equations.

31a ⟨*note new field name* `ii1` 31a⟩≡ (35c)
```
  if member(operands_and_ids, ii1) then {
    ⟨warn about clash in ii1 31c⟩
  } else
    insert(fieldname_literals, ii1)
```
Uses `fieldname_literals` 30a and `operands_and_ids` 30a.

31b ⟨*note new operand or id* `ii1` 31b⟩≡ (43c 47c)
```
  if member(fieldname_literals, ii1) then {
    ⟨warn about clash in ii1 31c⟩
  } else
    insert(operands_and_ids, ii1)
```
Uses `fieldname_literals` 30a and `operands_and_ids` 30a.

31c ⟨*warn about clash in* `ii1` 31c⟩≡ (31)
```
  every (if not member(warned_literals, ii1) then warning else verbose)(
    ii1 || " is used as a field-name literal and an operand or id..." |
    "  the literal takes priority in field bindings"
  )
  insert(warned_literals, ii1)
```
Uses `field` 53, `literal` 43e, `used` 132a, `verbose` 16a, and `warned_literals` 30a.

`conslist` contains a list of all constructors ever defined, even those that were later removed from the `constructors` table. Clients that use `conslist` are therefore obligated to check that `constructors[x.name] === x` before using a constructor `x` found in `conslist`. Similar checking is required before using the members of a constructor type. The easy way to do this is to use the `kept_constructors` generator, which provides the retained constructors of a given type, or the retained untyped constructors if no type is given.

31d ⟨*toplevel* 30a⟩+≡ (29) ◃30b 33b▹
```
  procedure kept_constructors(constype)
    local thelist
    thelist := (\constype).members | conslist
    suspend 1(c := !thelist, constructors[c.name] === c)
  end
```
Defines:
`kept_constructors`, used in chunks 106a, 129b, 168a, 179a, 183b, 194a, 201a, 218–22, 224, 238a, 241a, 294, 295, 297a, 299, 317e, and 330a.

Uses `conslist` 30a, `constructors` 30a, and `constype` 155.

Two parsers are generated from this grammar: that for a specification, and that for a code file containing pattern-matching statements. No client ever uses `Parsers` directly; they use `P_Spec` or `P_CodeFile`, but the union makes it possible for both parsers to handle end-of-file properly. The silly tokens avoid parsing conflicts. (Incidentally, the lexer for a `CodeFile` starts out in a different state, but that shouldn't matter to the parser generator.)

32a ⟨*grammar* 32a⟩≡ (29) 32b ▷

```
  Parsers   : "bogus spec marker" Spec | "bogus code marker" CodeFile;
```

Uses `bogus` 113b and `Spec` 32b.

The parts of a specification may appear in any order.

32b ⟨*grammar* 32a⟩+≡ (29) ◁ 32a 32c ▷

```
  Spec      : { Fieldspec | Patterns | Constructors | Placeholder
              | FetchSpec | PCSpec   | RelocSpec    | AsmSpec
              | BitSpec
              } ;
```

Defines:
`Spec`, used in chunk 32a.
Uses `Fieldspec` 33a and `Placeholder` 35a.

32c ⟨*grammar* 32a⟩+≡ (29) ◁ 32b 33a ▷

```
  PCSpec : "pc_unit_bits" INT /* if ii2 > 0 then pc_unit_bits := ii2
                               else error("pc_unit_bits must be positive") */;
```

Uses `INT` 51d 345b and `pc_unit_bits` 16a.

3.1 Fields and field information

Fields declared in the same `fields` declaration are members of the same equivalence class. Fields in the same equivalence class share the same *shape*, i.e., the width of the unit in which they are defined. Constraints on fields with different classes may not be combined with `&`, even if the classes have the same shape. Equivalence classes are useful for distinguishing CISC instruction fragments that must be combined to form a complete instruction.

Internally, the toolkit uses little-endian bit numbering—that is, bit 0 is always the least significant bit.

33a ⟨*grammar* 32a⟩+≡ (29) ◁32c 34a▷

```
  Fieldspec : "fields" "of" Ident "(" INT ")"
              { Ident INT ":" INT  /* newfield(ii1, ii2, ii4+1) */ }
              /* if ii5 % 8 ~= 0 then
                    error("element size ", ii5, " is not a multiple of 8 bits.")
                 ⟨if needed, flip bit numbering of fields ii7 using size ii5 33c⟩
                 (/symtab[ii3] := equivclass(ii3, ii7, ii5)) |
                          deferror(type(symtab[ii3]) || " ", image(ii3))
                 put(equivclasses, symtab[ii3])
                 every (!ii7).class := symtab[ii3] */
             ;
```

Defines:
`Fieldspec`, used in chunks 32b, 35c, and 36b.
Uses `deferror` 26a, `equivclass` 53, `equivclasses` 30a 353a, `Ident` 52, `INT` 51d 345b, `newfield` 33b, and `symtab` 30a 353a.

33b ⟨*toplevel* 30a⟩+≡ (29) ◁31d 36c▷

```
  procedure newfield(name, lo, hi)
    return (/symtab[name] := field(name, lo, hi)) | deferror("Field", image(name))
  end
```

Defines:
`newfield`, used in chunk 33a.
Uses `deferror` 26a, `field` 53, and `symtab` 30a 353a.

ßChange #26

33c ⟨*if needed, flip bit numbering of fields* `ii7` *using size* `ii5` 33c⟩≡ (33a)

```
  if /bit_zero_is_lsb then {
    if !ii7 then bit_numbering_used := 1
    every f := !ii7 do {
      f.lo := ii5 - f.hi
      f.hi := ii5 - f.lo
    }
  }
```

Uses `bit_numbering_used` 34b and `bit_zero_is_lsb` 30a.

3.2 Bit numbering

34a ⟨*grammar* 32a⟩+≡ (29) ◁33a 35a▷

```
BitSpec      : "bit" Zero "is" Significance Significant
                 /* xxx := if ii4 == "least" then 1 else &null
                    if xxx ~=== bit_zero_is_lsb then
                      ⟨possible warnings around changes in bit numberings 34c⟩
                    bit_zero_is_lsb := xxx
                    bit_numbering_set := 1
                  */
             ;
Zero         : INT /* ii1 = 0 | error("expected ‘0’") */;
Significance : Ident
                /* ii1 == ("most"|"least") | error("expected ‘most’ or ‘least’") */;
Significant  : Ident /* ii1 == "significant" | error("expected ‘significant’") */;
```

Uses `bit_numbering_set` 34b, `bit_zero_is_lsb` 30a, `Ident` 52, and `INT` 51d 345b.

34b ⟨* 29⟩+≡ ◁29 42b▷

```
global bit_numbering_set, bit_numbering_used
```

Defines:
`bit_numbering_set`, used in chunk 34.
`bit_numbering_used`, used in chunks 33c, 34c, and 112a.

34c ⟨*possible warnings around changes in bit numberings* 34c⟩≡ (34a)

```
if \bit_numbering_set then
  warning("Multiple inconsistent bit numberings --- hope you really mean it!")
if \bit_numbering_used then
  warning("You change the bit numbering, but you’ve already used the old numbering" ||
          "... seems like a crazy idea, but I’ll do it")
```

Uses `bit_numbering_set` 34b, `bit_numbering_used` 34b, and `used` 132a.

3.3 Placeholders

Placeholders are emitted for instructions that refer to unresolved relocatable addresses. They are associated with the equivalence class of the fields used in the instruction. A placeholder is defined by a pattern with the same length (but not necessarily the same shape) as the equivalence class for which it is a placeholder. By construction, a placeholder can't refer to constructors that might use it, because no such constructor can have been defined yet. In other words, the definitions of placeholders must precede the definition of an instruction that outputs a pattern with fields in the associated equivalence classes.

35a ⟨*grammar* 32a⟩+≡ (29) ◁34a 35c▷

```
  Placeholder : "placeholder" "for" Ident "is" Pattern
               /* class := lookuptype(ii3, "equivclass")
                  (/class.holder := pnf(ii5, globals)) |
                  error("Placeholder for ", ii3, " is already defined")
                  ⟨make sure placeholder of class is the right length 35b⟩
                */;
```

Defines:
`Placeholder`, used in chunk 32b.
Uses `defined` 132a, `equivclass` 53, `globals` 30a, `Ident` 52, `lookuptype` 23a, and `pnf` 72.

35b ⟨*make sure placeholder of* `class` *is the right length* 35b⟩≡ (35a)

```
  if pattern_length(class.holder) ~= class.size then
    error("Length of placeholder `", patimage(class.holder),
      "' \nfor ", class.name, " does not match class size ", class.size)
```

Uses `patimage` 358a and `pattern_length` 70c.

`fieldinfo` associates *name, value* pairs with one or more fields. The values of `checked`(`unchecked`) fields are (are not) checked at run time. A `sparse` binding of field names is used to specify that only a small set of values of the field are meaningful. The bindings associate a string name with a field value. The `names` construct binds string names to each possible value of a field. In both cases we return a table mapping names to values.

35c ⟨*grammar* 32a⟩+≡ (29) ◁35a 36a▷

```
  Fieldspec         : "fieldinfo" { Fieldinfo } ;
  Fieldinfo         : IdentBinding "is" "[" { Fielditem } "]"
                      /*  every fieldinfo(lookuptype(!ii1, "field"), !ii4) */;
  IdentBinding      : Ident /* [ii1] */ | "[" { Ident } "]";
  Fielditem         : Ident  | SparseFieldNames | FieldNameList ;
  SparseFieldNames  : "sparse" "[" FieldNameBindings "]" /* sparse_name_table(ii3) */ ;
  FieldNameBindings : FieldNameBinding {"," FieldNameBinding}  /* push(ii2, ii1) */;
  FieldNameBinding  : FieldName "=" Integer /* enumbinding(ii1, ii3) */ ;
  FieldNameList     : "names" "[" {FieldName} "]" /* full_name_table(ii3) */ ;
  FieldName         : (String | Ident) /* {⟨note new field name ii1 31a⟩}; ii1 */;
```

Defines:
`IdentBinding`, used in chunk 50a.
Uses `enumbinding` 341b, `field` 53, `fieldinfo` 340a, `Fieldspec` 33a, `full_name_table` 341c, `Ident` 52, `lookuptype` 23a, `sparse_name_table` 341a, and `String` 52.

We reserve the following similar syntax for places where we aren't referring to field names:

36a ⟨*grammar* 32a⟩+≡ (29) ◁35c 36b▷
```
SparseNames : "sparse" "[" Bindings "]" /* sparse_name_table(ii3) */ ;
Bindings    : Binding {"," Binding}  /* push(ii2, ii1) */;
Binding     : (String|Ident) "=" Integer /* enumbinding(ii1, ii3) */ ;
DenseNames  : "names" "[" {String|Ident} "]" /* full_name_table(ii3) */ ;
NameTable   : SparseNames | DenseNames;
```
Uses `enumbinding` 341b, `full_name_table` 341c, `Ident` 52, `sparse_name_table` 341a, and `String` 52.

I lump the word size of the target machine into the field specs, not because it makes all that much sense, but because I didn't know where else to put it.

36b ⟨*grammar* 32a⟩+≡ (29) ◁36a 37a▷
```
Fieldspec : "wordsize" INT /* wordsize := ii2 */;
```
Uses `Fieldspec` 33a, `INT` 51d 345b, and `wordsize` 36c.

36c ⟨*toplevel* 30a⟩+≡ (29) ◁33b 36e▷
```
global wordsize          # word size of the machine the application runs on
```
Defines:
`wordsize`, used in chunks 36, 94–96, 112c, 130c, 182b, 190a, 193a, 301a, 318e, 320c, and 332b.

36d ⟨*initialization* 36d⟩≡ (30b) 45b▷
```
wordsize := 32
```
Uses `wordsize` 36c.

3.4 Pattern definitions

Patterns parse into abstract syntax; patbinding makes them concrete and binds them to identifiers.

36e ⟨*toplevel* 30a⟩+≡ (29) ◁36c 38a▷
```
global patlhs            # hold lhs for later action
```
Defines:
`patlhs`, used in chunk 37a.

37a ⟨*grammar* 32a⟩+≡ (29) ◁36b 37b▷

```
Patterns  : "patterns" { PatBinding };
PatBinding: (Ident /* patlhs := ii1 */) "is" BindingRHS
          | "[" {Ident} "]" "is" Pattern /* patbinding(ii2, ii5) */
          ;
BindingRHS: Pattern /* patbinding(patlhs, ii1) */
          | "any" "of" "[" {Ident} "]" "," "which" "is" Pattern
              /* patbinding(copy(ii4), ii9)
                 l := []
                 every put(l, Pident("_" ~== !ii4))
                 patbinding(patlhs, Por(l)) */
          ;
```

Uses `Ident` 52, `patbinding` 41a, `patlhs` 36e, `Pident` 39a, and `Por` 39a.

Parsing labels relies on a grotesque hack to keep things LL(1); we allow an arbitrary sequent to precede colon, then patch things up semantically afterwards.

37b ⟨*grammar* 32a⟩+≡ (29) ◁37a 39c▷

```
Pattern   : Disjunct     { "|" Disjunct }      /* Por(push(ii2, ii1))   */ ;
Disjunct  : Sequent      { ";" Sequent
                         | ":" Sequent /* colon_mark(ii2) */
                         }      /* Pseq(colons_to_labels(push(ii2, ii1)))  */ ;
Sequent   : Conjunct     { "&" Conjunct }      /* Pand(push(ii2, ii1))  */ ;

Conjunct  : ["..."] DotsR    /* if \ii1 then Pseq([dots_pattern(),ii2]) else ii2 */;
DotsR     : Atomic ["..."]   /* if \ii2 then Pseq([ii1,dots_pattern()]) else ii1 */ ;

Atomic    : (Ident /* atomicid := ii1 */)
            (                                   /* Pident(atomicid) */
            | Relop (Generator | Expr)          /* Pcon(atomicid, ii1, ii2) */
            | {"^" Opname} "(" ConstructorArgs ")"
                  /* Papp(push(ii1, \symtab[atomicid] | atomicid), ii3) */
            ) /* ii2 */
          | String {"^" Opname} "(" ConstructorArgs ")"
                  /* Papp(push(ii2, ii1), ii4) */
          | "(" Pattern ")"
          | "[" {Ident /* Pident(ii1) */} "]"   /* Plist(ii2) */
          ;
ConstructorArgs : AppExpr {"," AppExpr} /* push(ii2, ii1) */
                | /* [] */
                ;
```

Uses `atomicid` 38b, `colon_mark` 38a, `colons_to_labels` 38a, `dots_pattern` 60c, `Ident` 52, `Pand` 39a, `Papp` 39a, `Pcon` 39a, `Pident` 39a, `Plist` 39a, `Por` 39a, `Pseq` 39a, `String` 52, and `symtab` 30a 353a.

We used to "explode" compound constructor names at this syntactic step, but this turned out to be not what we wanted on the RHS of constructor definitions. For example, in

```
pattern load is ld | lb
constructor load^"2" a,b,c,d is load(a,b); load(c,d)
```

we cannot evaluate `Papp(name, args)` for the `load`s on the right-hand side until we have an environment telling use whether `load` stands for `ld` or `lb` in this particular context. (The original semantics made it stand for `ld|lb`, which we decided was rarely what anyone wanted.)

Here's the colon fix:

38a ⟨*toplevel* 30a⟩+≡ (29) ◁36e 38b▷

```
  record colon_mark(x)

  procedure colons_to_labels(l)
    every i := 1 to *l & type(l[i]) == "colon_mark" do {
      l[i] := l[i].x  # strip out tag
      (l[i-1] := undo_identifier_syntax(l[i-1])) |
        error("Colon must be preceded by an identifier")
    }
    return l
  end

  procedure undo_identifier_syntax(seq)
    if type(seq) == "Pand" & *seq.patterns = 1 & con := seq.patterns[1] &
       type(con) == "Pident"
    then
      return Plabel(con.name)
  end
```

Defines:
`colon_mark`, used in chunk 37b.
`colons_to_labels`, used in chunk 37b.
Uses `Pand` 39a, `Pident` 39a, and `Plabel` 39a.

`atomicid` is an auxiliary variable.

38b ⟨*toplevel* 30a⟩+≡ (29) ◁38a 39a▷

```
  global atomicid
```

Defines:
`atomicid`, used in chunk 37b.

The following records represent the nodes in a pattern's abstract syntax tree.

39a ⟨*toplevel* 30a⟩+≡ (29) ◁38b 39b▷

```
record Por (patterns)                   # disjunction
record Pseq(patterns)                   # sequence
record Pand(patterns)                   # conjunction
record Pcon(name, relop, value)         # constraint on field
record Pident(name)                     # identifier standing for a pattern
record Plabel(name)                     # pattern label
record Papp(cons, args)                 # constructor applied to arguments
                                        #   (args are AppExprs)
record Plist(patterns)                  # list of patterns in square brackets
```

Defines:
- `Pand`, used in chunks 37b, 38a, 73, 76, 77, 160a, and 363a.
- `Papp`, used in chunks 37b, 39b, 73, 76, 77, 172, and 363a.
- `Pcon`, used in chunks 37b, 73, 76, 77, and 363a.
- `Pident`, used in chunks 37, 38a, 73, 76, 77, 160a, and 363a.
- `Plabel`, used in chunks 38a, 73, 76, and 77.
- `Plist`, used in chunks 37b and 73.
- `Por`, used in chunks 37, 39b, 73, 76, 77, and 363a.
- `Pseq`, used in chunks 37b, 73, 76, 77, and 363a.

Uses `constraint` 53, `constructor` 155, `field` 53, `label` 369a, `pattern` 53, and `value` 132a.

The name of an applied constructor depends on the environment in which it is evaluated. We use `explode_names` to generate all possible constructor names denoted by a constructor application, then generate the applications.

39b ⟨*toplevel* 30a⟩+≡ (29) ◁39a 40b▷

```
procedure explode_apps(opcode, args, rho)
  l := []
  every c := explode_names(opcode, rho) do {
    put(l, Papp(cons_named(c), args))
  }
  return if *l = 1 then l[1] else Por(l)
end
```

Defines:
- `explode_apps`, used in chunks 73 and 77b.

Uses `cons_named` 163a, `explode_names` 159b, `Papp` 39a, and `Por` 39a.

39c ⟨*grammar* 32a⟩+≡ (29) ◁37b 40a▷

```
Relop     : "=" | "!=" | "<=" | ">=" | "<" | ">" ;
Generator : "{" Integer "to" Integer [ "columns" INT /* ii2 */ ] "}"
                                /* Gfor(ii2, ii4+1, \ii5 | 1) */
          | "[" { Integer } "]"     /* Glist(ii2) */
          ;
```

Uses `Gfor` 40b, `Glist` 40b, and `INT` 51d 345b.

There are a couple of special patterns. These alone produce patterns directly, not an AST. (`pnf`, for example, isn't prepared to cope with a pattern with an empty sequence).

40a ⟨*grammar* 32a⟩+≡ (29) ◁39c 42a▷

```
Atomic : "epsilon"   /* epsilon() */
       | "some" Ident /* wildcard(lookuptype(ii2, "equivclass")) */
       ;
```

Uses `epsilon` 60b, `equivclass` 53, `Ident` 52, `lookuptype` 23a, and `wildcard` 60a.

3.4.1 Pattern syntax

A generator is represented by a `Glist`. The list of values is either given explicitly or generated on the spot by `Gfor`. Note that `hi` is one *more* than the largest value generated.

40b ⟨*toplevel* 30a⟩+≡ (29) ◁39b 41a▷

```
record Glist(values)
procedure Gfor(lo, hi, cols)
    local l, r
    l := []
    r := (hi - lo) / cols
    every put(l, lo + (0 to r - 1) + r * (0 to cols - 1))
    return Glist(l)
end
```

Defines:
`Gfor`, used in chunk 39c.
`Glist`, used in chunks 39c, 74a, and 363a.

Binding a pattern requires putting it in normal form. If there is a generator in the pattern, the normal form is a list. In that case, `id` must also be a list.

41a ⟨*toplevel* 30a⟩+≡ (29) ◁40b 43e▷

```
procedure patbinding(id, ast)
    local p
    ⟨verbosely announce binding 41c⟩
    p := pnf(ast, globals)
    ⟨insist p have no conditionals nor field bindings 41b⟩
    case type(id) || "," || type(p) of {
        "string,pattern" : patbind(id, p, globals)
        "list,list"      : if *id = *p then
                             while patbind(get(id), get(p), globals)
                           else
                             error("identifier list/generator length mismatch: ",
                                   *id, " vs ", *p)
        "string,list"    : error("generator must be bound to an identifier list")
        "list,pattern"   : error("identifier list must be bound to a generator")
        default          : impossible("pattern binding")
    }
    return
end
```

Defines:
`patbinding`, used in chunk 37a.
Uses `globals` 30a, `patbind` 54, `pattern` 53, and `pnf` 72.

41b ⟨*insist* `p` *have no conditionals nor field bindings* 41b⟩≡ (41a)

```
case type(p) of {
  "pattern" : insist_global_pattern(p)
  "list"    : every insist_global_pattern(!p)
}
```

Uses `insist_global_pattern` 71b and `pattern` 53.

41c ⟨*verbosely announce binding* 41c⟩≡ (41a)

```
case type(id) of {
  "string" : verbose("Pattern ", id)
  "list"   : verbose(*id, " patterns")
}
```

Uses `verbose` 16a.

3.5 Constructors

A constructor consists of an `Opcode`, one or more `Operands`, an optional constructor type `ConstType`, and one or more `Branches`. The default type of a constructor is `instruction`. One-branch syntax is radically different from multibranch syntax and I haven't the time to explain the details now. Each branch consists of a set of `Equations` followed by a required output `Pattern`. Branches are evaluated in order: the constructor's value is the output pattern of the first branch whose constraints are satisfied.

42a ⟨*grammar* 32a⟩+≡ (29) ◃40a 43a▹

```
Constructors : "constructors" (/*see_newline()*/) { Constructor /* see_newline() */} ;
Constructor  : Opcode Operands [ ":" ConstType ] NLBranches
                 /* note_constructor(ii1, ii2, \ii3 | instructionctype, ii4) */
               ;
Operands : SeeWhite {Operand} StopWhite /* process_operands(ii2) */;
```

Uses `constructors` 30a, `ConstType` 43d, `instructionctype` 30a, `note_constructor` 156, `process_operands` 42b, and `see_newline` 348a.

To process operands, we combine adjacent literals and remove trailing white space.

42b ⟨* 29⟩+≡ ◃34b 44b▹

```
procedure process_operands(ops)
  if type(ops[-1]) == "literal" & ops[-1].s ? (white(), pos(0)) then
    pull(ops) # discard trailing white space
  l := []
  every x := !ops do
    if type(x) == "literal" & type(l[-1]) == "literal" then
      l[-1].s ||:= x.s
    else
      put(l, x)
  return l
end
```

Defines:
`process_operands`, used in chunk 42a.
Uses `literal` 43e and `white` 350a.

43a ⟨*grammar* 32a⟩+≡ (29) ◁42a 43b▷
```
  NLBranches   : Branches
               | NEWLINE (Branches | /* [ [ [], &null ] ] */)
               ;
  Branches     : SingleBranch /* [ii1] */
               | WhenBranch {WhenBranch} [OtherwiseBranch]
                  /* push(ii2, ii1); put(ii2, \ii3); ii2 */
               ;
  SingleBranch : "{" Equations "}" [ "is" Pattern ] /* [ ii2, \ii4 | &null] */
               | "is" Pattern    /* [ [] , ii2 ] */
               ;
  WhenBranch      : "when" "{" Equations "}" "is" Pattern /* [ii3, ii6] */;
  OtherwiseBranch : "otherwise" "is" Pattern /* [[], ii3] */;
```
Uses `NEWLINE` 51d 345b.

43b ⟨*grammar* 32a⟩+≡ (29) ◁43a 43c▷
```
  Opcode    : Opname { "^" Opname }  /* push(ii2, ii1) */ ;
  Opname    : Ident /* \symtab[ii1] | ii1 */ | String ;
```
Uses `Ident` 52, `String` 52, and `symtab` 30a 353a.

43c ⟨*grammar* 32a⟩+≡ (29) ◁43b 43d▷
```
  Operand      : Ident ["!"] /* {⟨note new operand or id ii1 31b⟩};
                              name_to_input(ii1, ii2) */
               | (Literal | GlobOperator | White) /* literal(ii1) */
               ;
  Literal      : String | Integer /* string(ii1) */ | Relop | "=>"
               | "[" | "]" | "(" | ")" | "+" | "-" | "/"
               | "&" | "@" | "#" | "%" | ";" | "|"
               ;
  GlobOperator : "*" | "$" | ","
               ;
```
Uses `Ident` 52, `literal` 43e, `name_to_input` 44b, and `String` 52.

A `ConstType` is defined implicitly by the first declaration of a constructor of that type.

43d ⟨*grammar* 32a⟩+≡ (29) ◁43c 45c▷
```
  ConstType : Ident /* (/symtab[ii1] := constype(ii1, set())) |
                      lookuptype(ii1, "constype")  */;
```
Defines:
  `ConstType`, used in chunk 42a.
Uses `constype` 155, `Ident` 52, `lookuptype` 23a, and `symtab` 30a 353a.

The `literal` type serves to hold strings that are to be emitted literally.

43e ⟨*toplevel* 30a⟩+≡ (29) ◁41a 44a▷
```
  record literal(s)       # holds string or list to be emitted literally
```
Defines:
  `literal`, used in chunks 31c, 42b, 43c, 47, 76, 77, 101c, 115b, 120a, 125d, 171, 182b, 196b, 199b, 202d, 205b, 221a, 232, 234, 236, 238a, 242, 282d, 284, 285, 287a, 296, 297a, 312a, 321a, 326a, 332c, and 334.

Every operand that is not of type `literal` is of type `input`:

44a ⟨*toplevel* 30a⟩+≡ (29) ◁43e 46a▷
```
record input(name, meaning)                    # input name and meaning
```
Defines:
`input`, used in chunks 12l, 18, 19c, 22, 27, 44b, 53, 79, 158a, 162a, 165, 167a, 168b, 170a, 172, 183a, 194b, 195a, 199a, 201–3, 213a, 221a, 232, 234, 242, 250a, 297, and 362b.

There are five legitimate types for `meaning`:

`constype`	Constructor of the given type.
`field`	The given field.
`null`	Integer.
`string`	Relocatable address (the value of the string is always `"reloc"`).
`integer`	Extended field; the meaning is its width in bits. Finding the field requires looking up the input's name in the global symbol table.

An `Operand` may be the name of a symbol or the name of a constructor type trailed by zero or more digits (e.g., `Address`, `Address1`, etc). Note that variables used as relocatable addresses live in the symbol table. Finally, if not recognized, it is treated as a variable. Only a field may be sign-extended with a bang.

44b ⟨* 29⟩+≡ ◁42b 45a▷
```
procedure name_to_input(name, bang)
  i := input(name,
    if name ? type(ct := symtab[tab(reversetrailing('0123456789_'))]) == "constype" then
      mark_ct_as_used(ct)
    else case type(x := symtab[name]) of {
      "field"       : if /bang then x else fwidth(x)
      "null"        : x
      "relocatable" : "reloc"
      "constype"    : impossible("missed first-round search for constype")
      default       : typeerror(x, "free variable, field, or constructor input", name,
                                globals)
  })
  if \bang & type(i.meaning) ~== "integer" then
    typeerror(i.meaning, "field (only fields can be sign-extended)", name, globals)
  return i
end
```
Defines:
`name_to_input`, used in chunk 43c.
Uses `constructor` 155, `constype` 155, `field` 53, `fwidth` 112d, `globals` 30a, `input` 44a, `mark_ct_as_used` 45e, `relocatable` 45a, `reversetrailing` 46a, `symtab` 30a 353a, and `typeerror` 26b.

Here's all the relocation goo.

45a ⟨* 29⟩+≡ ◁44b 45d▷

```
record relocatable()    # type of a name that means relocatable address
global the_relocatable  # we only need one...
```

Defines:
`relocatable`, used in chunks 44, 45, 79, 230b, and 370a.
`the_relocatable`, used in chunk 45.
Uses `address` 369a.

45b ⟨*initialization* 36d⟩+≡ (30b) ◁36d 50e▷

```
the_relocatable := relocatable()
```

Uses `relocatable` 45a and `the_relocatable` 45a.

45c ⟨*grammar* 32a⟩+≡ (29) ◁43d 46b▷

```
RelocSpec : "relocatable" Ident {Ident} /* every make_relocatable(ii2 | !ii3) */;
```

Uses `Ident` 52, `make_relocatable` 45d, and `relocatable` 45a.

45d ⟨* 29⟩+≡ ◁45a 45e▷

```
procedure make_relocatable(name)
  return (/symtab[name] := the_relocatable) | deferror("Relocatable name", image(name))
end
```

Defines:
`make_relocatable`, used in chunk 45c.
Uses `deferror` 26a, `symtab` 30a 353a, and `the_relocatable` 45a.

A constructor type is marked when it is first used as an input.

45e ⟨* 29⟩+≡ ◁45d 48b▷

```
procedure mark_ct_as_used(type)
  if /type.used := lineno then
    {⟨sort members of type 45f⟩}
  return type
end
```

Defines:
`mark_ct_as_used`, used in chunk 44b.
Uses `used` 132a.

45f ⟨*sort members of type* 45f⟩≡ (45e)

```
t := table()
every m := !type.members do
  t[m.name] := m
t := sort(t)
type.members := []
every put(type.members, (!t)[2])
```

`reversetrailing(c)` generates, from right to left, the positions at which the rest of the subject is composed entirely of characters from `c`.

46a ⟨*toplevel* 30a⟩+≡ (29) ◁44a 47b▷

```
procedure reversetrailing(c)
  local i
  suspend *&subject + 1
  i := *&subject
  while any(c, &subject, 0 < i) do { suspend i; i -:= 1 }
end
```

Defines:
`reversetrailing`, used in chunk 44b.

3.5.1 Deleting constructors

For tests, or for other purposes, we may want to delete constructors from a specification. There are two ways: say what we want to keep, or say what we want to throw away.

46b ⟨*grammar* 32a⟩+≡ (29) ◁45c 46c▷

```
Constructors : "discard" {Opcode} /* every discard_cons_named(explode_names(!ii2)) */
             | "keep"    {Opcode}
                  /* s := set()
                     every insert(s, is_constructor(explode_names(!ii2), warning))
                     every k := key(constructors) do
                       if not member(s, constructors[k]) then
                         delete(constructors, k)
                   */
             ;
```

Uses `constructors` 30a, `discard_cons_named` 163b, `explode_names` 159b, and `is_constructor` 163a.

3.6 Equations

Extensions and ranges are guaranteed unique within a set of equations, and they are reset before each set of equations. The grammar is bit awkward because we're trying to keep it LL(1).

46c ⟨*grammar* 32a⟩+≡ (29) ◁46b 47a▷

```
Equations : [Equation { "," Equation } /* push(ii2, ii1) */] /* \ii1 | [] */ ;
Equation  : Expr Relop Expr /* eqn(ii1, ii2, ii3) */ ;
```

Uses `eqn` 81c.

Applications are forbidden in expressions except in special contexts.

47a ⟨*grammar* 32a⟩+≡ (29) ◁46c 47c▷
```
Expr      : AppExpr /* if has_app_or_literal(ii1) then
                         error("Application or literal string not legal")
                       else ii1 */
          ;
```
Uses `has_app_or_literal` 47b and `literal` 43e.

47b ⟨*toplevel* 30a⟩+≡ (29) ◁46a 48a▷
```
procedure has_app_or_literal_f(e)
  return type(e) == ("Eapp"|"literal")
end

procedure has_app_or_literal(e)
  suspend expwalk(e, has_app_or_literal_f)
end
```
Defines:
`has_app_or_literal`, used in chunk 47a.
Uses `Eapp` 79, `expwalk` 114a, and `literal` 43e.

47c ⟨*grammar* 32a⟩+≡ (29) ◁47a 49a▷
```
AppExpr   : Term { AOp Term }
            /* every t := !ii2 do ii1 := binop(ii1, t[1], t[2]); ii1 */ ;
Term      : Factor { Mop Factor }
            /* every f := !ii2 do ii1 := binop(ii1, f[1], f[2]); ii1 */ ;
AOp       : "+" | "-" ;
Mop       : "*" | "/" ;
Factor    : Integer
          | String /* literal(ii1) */   # legal only in constructor apps
          | "_" /* fresh_variable("_") */
          | IDENT ( [ Bitrange ] [ "!" ]          /* SyntaxRange(ii1, ii2) */
                  | "(" AppExpr {"," AppExpr} ")" /* push(ii3, ii2) */
                  )
            /* ⟨note new operand or id ii1 31b⟩
               if type(ii2) == "SyntaxRange" then mkfactor(ii1, ii2.bits, ii2.bang)
               else Eapp(ii1, ii2) */
          | "(" AppExpr ")"
          | "-" Factor /* binop(0, ii1, ii2) */
          ;
Bitrange  : "@" "[" INT [":" INT /*ii2*/] "]" /* [ii3, (\ii4|ii3)+1] */ ;
```
Uses `binop` 110a, `constructor` 155, `Eapp` 79, `fresh_variable` 151b, `IDENT` 51d 345b 346b, `INT` 51d 345b, `literal` 43e, `mkfactor` 48b, `String` 52, and `SyntaxRange` 48a.

Can't use a square-bracket syntax for bit ranges because it creates parsing conflict with list of names on left-hand side of pattern binding, list of patterns in square brackets, and optional name in arm of case statement.

48a ⟨*toplevel* 30a⟩+≡ (29) ◁47b 48c▷

```
record SyntaxRange(bits, bang)
```

Defines:
`SyntaxRange`, used in chunk 47c.

48b ⟨* 29⟩+≡ ◁45e 50d▷

```
procedure mkfactor(ident, range, ext)
  e := ident
  if \range then {
    e := mkslice(e, range[1], range[2])
    w := e.n
  }
  if \ext then {
    /w := if type(f := symtab[ident]) == "field" then fwidth(f)
          else error("Can't sign-extend ", ident, " (not a field)")
    e := Ewiden(e, w)
  }
  return e
end
```

Defines:
`mkfactor`, used in chunk 47c.
Uses `Ewiden` 79, `field` 53, `fwidth` 112d, `mkslice` 112a, and `symtab` 30a 353a.

3.7 Code containing matching statements

Using the global value `succptr` is OK for now, but it will break horribly if we ever permit nested matching statements.

48c ⟨*toplevel* 30a⟩+≡ (29) ◁48a

```
global matching_stmts, codeheader
```

Defines:
`codeheader`, used in chunks 19a and 49a.
`matching_stmts`, used in chunks 19a and 49a.

49a ⟨*grammar* 32a⟩+≡ (29) ◁47c 49b▷
```
  CodeFile      : (/*codeheader := arm(filename, lineno)*/)
                  ({CODELINE} /* codeheader.code := ii1 */)
                  { Casestmt {CODELINE} /* ii1.trailer.code := ii2 ; ii1 */ }
                  /* codeheader.original := codeheader; matching_stmts := ii3 */;
  Casestmt      : CASELINE { Casearm } [ElseArm] "endmatch"
                 /* x := matching_stmt(ii2, ii1, succptr) ; put(x.arms, \ii3)
                    x.trailer := arm(filename, lineno); x */
                ;
  Casearm       : ("|" /*arm(filename, lineno) */)
                  Pattern OptEquations OptName "=>" {CODELINE}
                  /* ii1.pattern := ii2; ii1.eqns := ii3; ii1.name := ii4; ii1.code := ii6
                     ii1.original := ii1 */;   # value is ii1
  ElseArm       : ("else"          /* arm(filename, lineno, epsilon()) */) {CODELINE}
                  /* ii1.code := ii2; ii1.original := ii1 */ ;
  OptName       : [ "[" Ident "]" ];
  OptEquations  : [ "{" Equations "}" ];
```
Uses `arm` 259b, `CASELINE` 51d 345b, `codeheader` 48c, `CODELINE` 51d 345b, `epsilon` 60b, `Ident` 52, `matching_stmt` 259b, `matching_stmts` 48c, `pattern` 53, `succptr` 346b, and `value` 132a.

3.8 Specifications for fetching words

I introduce the nonterminal `Address` in order to avoid making `address` a reserved word.

49b ⟨*grammar* 32a⟩+≡ (29) ◁49a 50a▷
```
  FetchSpec : "fetch" ((INT | "any") "using" String  /* fetchtab[ii1] := ii3 */)
            | "address" (Add "using" String          /* fetchtab[ii1] := ii3 */
                        | "type" "is" String         /* fetchtab[ii1] := ii3 */
                        | "to" IntIdent "using" String     /* fetchtab[ii2] := ii4 */
                        )
            ;

  Add      : Ident /* (ii1 == "add") | error("expected 'add', 'type', or 'to'") */;
  IntIdent : Ident /* (ii1 == "integer") | error("expected", image("integer")) */;
```
Uses `address` 369a, `fetchtab` 283a, `Ident` 52, `INT` 51d 345b, and `String` 52.

3.9 Assembly syntax

50a ⟨*grammar* 32a⟩+≡ (29) ◁49b 50b▷
```
AsmSpec : "assembly"
   ( "operand" {IdentBinding "is" OperandSyntaxSpec
                               /* every asmoperand(!ii1, ii3[1], ii3[2]) */}
   | "component"  {Globbing "is" GlobTarget  /* asmopcode (ii1, ii3) */}
   | "opcode"  {Globbing "is" GlobTarget  /* asmopcode (ii1, ii3, 1) */}
   | "syntax"  (/*see_newline()*/) { AsmSyntax /* see_newline() */}
   );
```
Uses `asmopcode` 227a, `asmoperand` 229b, `IdentBinding` 35c, and `see_newline` 348a.

50b ⟨*grammar* 32a⟩+≡ (29) ◁50a 50c▷
```
OperandSyntaxSpec : String ["using" OperandNameSpec ] /* [ii1, ii2] */
                  | OperandNameSpec /* ["%s", ii1] */
                  ;
OperandNameSpec   : NameTable
                  | "field" Ident /* lookuptype(ii2, "field") */
                  ;
```
Uses `field` 53, `Ident` 52, `lookuptype` 23a, and `String` 52.

50c ⟨*grammar* 32a⟩+≡ (29) ◁50b 51a▷
```
AsmSyntax: Opcode Operands NEWLINE
              /* every set_asmsyntax(is_constructor(explode_names(ii1), warning), ii2) */;
```
Uses `explode_names` 159b, `is_constructor` 163a, `NEWLINE` 51d 345b, and `set_asmsyntax` 231a.

3.9.1 Globbing expressions

A globbing expression is a string, a list (concatenation), a wildcard, or a list of alternatives in braces.

50d ⟨* 29⟩+≡ ◁48b 51c▷
```
record glob_any(alternatives, number)   # braced list of alternate globbing patterns
record glob_wildcard()
global the_glob_wildcard                # only need one value
record glob_dollar(number)              # $n on right-hand side
```
Defines:
- `glob_any`, used in chunks 51a, 227c, and 228b.
- `glob_dollar`, used in chunks 51b, 228c, and 229a.
- `glob_wildcard`, used in chunks 50e, 227c, and 228b.
- `the_glob_wildcard`, used in chunks 50e and 51a.

Uses `value` 132a.

50e ⟨*initialization* 36d⟩+≡ (30b) ◁45b
```
the_glob_wildcard := glob_wildcard()
```
Uses `glob_wildcard` 50d and `the_glob_wildcard` 50d.

51a ⟨*grammar* 32a⟩+≡ (29) ◁50c 51b▷
```
  Globbing         : SeeWhite GlobPattern StopWhite White /* ii2 */;
  GlobPattern      : {GlobAlternatives | Literal | Ident}
                              /* number_braces(cat_adjacent_strings(ii1)) */;
  GlobAlternatives : "{" GlobPattern {"," GlobPattern } "}" /* glob_any(push(ii3, ii2))*/
                   | "*"                                  /* the_glob_wildcard */
                   ;
```
Uses cat_adjacent_strings 51c, glob_any 50d, Ident 52, number_braces 228b, and the_glob_wildcard 50d.

51b ⟨*grammar* 32a⟩+≡ (29) ◁51a 52▷
```
  GlobTarget       : SeeWhite GlobTargets StopWhite White /* ii2 */;
  GlobTargets      : {GlobTargetSpecial | GlobTargetLiteral}
                                          /* cat_adjacent_strings(ii1) */;
  GlobTargetSpecial: "$" (Integer /* glob_dollar(ii1) */ | "$")
                   ;
  GlobTargetLiteral: Literal | Ident | "*" | "{" | "}" | "," ;
```
Uses cat_adjacent_strings 51c, glob_dollar 50d, and Ident 52.

51c ⟨* 29⟩+≡ ◁50d
```
  procedure cat_adjacent_strings(l)
    m := []
    every x := !l do
      if type(x) == "string" & type(m[-1]) == "string" then
        m[-1] ||:= x
      else
        put(m, x)
    return m
  end
```
Defines:
cat_adjacent_strings, used in chunk 51.

3.10 Lexical elements

51d ⟨*terminal symbols* 51d⟩≡ (29)
```
  %term IDENT
  %term INT
  %term CASELINE
  %term CODELINE
  %term NEWLINE
  %term WHITESPACE
```
Defines:
CASELINE, used in chunks 49a, 346a, and 347a.
CODELINE, used in chunks 49a, 346, and 347.
IDENT, used in chunks 47c, 52, 346a, 348a, and 349a.
INT, used in chunks 32–34, 36b, 39c, 47c, 49b, 52, 346a, and 349a.
NEWLINE, used in chunks 43a, 50c, 346a, and 348a.
WHITESPACE, used in chunks 52, 346a, and 348b.

52 ⟨*grammar* 32a⟩+≡ (29) ◁51b
```
  Ident     : IDENT | "_" ;
  String    : '"' ;
  Integer   : INT | "'" /* ord(ii1) */ ;
  White     : WHITESPACE;
  SeeWhite  : /* see_whitespace() */;
  StopWhite : /* ignore_whitespace() */;
```
Defines:
`Ident`, used in chunks 33–37, 40a, 43, 45c, and 49–51.
`String`, used in chunks 35–37, 43, 47c, 49b, and 50b.

Uses `IDENT` 51d 345b 346b, `ignore_whitespace` 348b, `INT` 51d 345b, `see_whitespace` 348b, and `WHITESPACE` 51d 345b.

Chapter 4

Patterns

4.1 Semantics, constructors, and mutators

Patterns are built up from *constraints* by applying `;`, `&` and `|` (sequence, conjunction and disjunction), and are represented in disjunctive normal form. A constraint restricts a field's value to be within a certain interval: $lo \leq f < hi$. Patterns consisting solely of constraints are created with `conspat`, which relates a field to a value. The only relational operators now supported are equality and inequality, but others could easily be added by adding cases to `conspat`.

A field `f` is that part of a word from bits `f.lo` through bits `f.hi-1` inclusive. Notice that this is *not* the syntax used in the input, which is like what's used in architecture manuals.

If a pattern contains field bindings with free variables, it may have conditions attached to its disjuncts. A condition is an expression over those same free variables.

53 ⟨* 53⟩≡ 54 ▷

```
record field(name, lo, hi, class)       # field is bits lo..hi-1 (note NOT like input)
record equivclass(name, fields, size, holder)   # equivalence class of fields

record pattern(disjuncts, name)             # list of disjuncts, name of pattern
record disjunct(sequents, name, conditions) # list of sequents, name of disjunct,
                                            #          set of expressions
record sequent(constraints, class)          # list of contraints, class
record patlabel(name, original_name, offset) # marks a location in a sequence
record latent_patlabel(instance)            # could become a patlabel or vanish
record constraint(field, lo, hi)            # constrain lo <= field.value < hi
record fieldbinding(field, code)            # code to compute field

# invariant: constraints of the same sequent have distinct fields
```

Defines:
`constraint`, used in chunks 20b, 39a, 59, 60a, 66–69, 73, 75a, 83a, 90, 91, 101a, 114c, 119b, 134, 139c, 187, 252, 258b, 261, 267a, 270, 273b, 306a, 357a, and 358a.

disjunct, used in chunks 54, 55, 58–60, 62a, 63b, 83a, 90, 91, 99c, 114b, 115b, 118a, 160b, 185, 196b, 197b, 199b, 205, 248, 253a, 267a, 303, 355d, and 358a.
equivclass, used in chunks 33a, 35a, and 40a.
field, used in chunks 13a, 15f, 20b, 21b, 25c, 31c, 33b, 35c, 39a, 44b, 48b, 50b, 59, 66–68, 73, 76b, 83a, 90c, 94b, 101a, 112–14, 119b, 158–60, 165, 166b, 168b, 170, 171, 182b, 183a, 187, 191c, 193–95, 198b, 202d, 230, 235, 237, 242, 244, 250a, 252, 253c, 256, 258b, 260a, 264, 265b, 267a, 270, 273, 277–80, 284–86, 288b, 297a, 301b, 303, 306a, 319–21, 331, 332, 335c, 337c, 339–43, and 357–60.
fieldbinding, used in chunks 63c, 68a, 71b, 75a, 83a, 90c, 101a, 114c, 119b, 166b, 168b, 187, 191c, 250a, 252, 258b, 288b, 357a, and 358a.
latent_patlabel, used in chunks 30b, 57c, 58b, 65b, 83a, 97a, 99c, 101a, 114c, 119b, 174–76, 257, and 356b.
patlabel, used in chunks 55, 57, 58a, 65b, 67b, 71b, 79, 83a, 90c, 97a, 99c, 101a, 109a, 114b, 119b, 174d, 175, 248, 257, and 356b.
pattern, used in chunks 14, 21a, 25c, 27, 39a, 41, 49a, 54, 55, 58–63, 69c, 70c, 73, 76–79, 83a, 90c, 91a, 99, 109a, 114b, 116b, 118a, 122c, 158–60, 169a, 172, 175, 179a, 183b, 185c, 187, 227c, 228b, 248–51, 257, 259, 260, 263, 265–67, 269c, 270b, 273a, 288, 299, 354, 358a, 360, 363a, and 364a.
sequent, used in chunks 57c, 59, 60c, 63c, 65a, 66d, 68–71, 83a, 90c, 91b, 101a, 109a, 114b, 119a, 186b, 257, and 358a.

Uses constraints 132a, input 44a, location, and value 132a.

N.B. patlabel offsets are in PC units, not in bits.

During the specification process, names may be bound to a particular pattern. In that way, a pattern and its disjuncts are associated with a name. The name information is saved so it can be referred to by code that appears in case statements. The procedure patbind also saves the pattern and name in the current environment.

54 ⟨* 53⟩+≡ ◁53 55▷

```
procedure patbind(name, p, env)
    if name == "_" then return
    if *p.disjuncts = 1 & \p.disjuncts[1].name then
        p := pattern([disjunct(p.disjuncts[1].sequents)])       # copy w/o names
    add_to_rho(name, p, env)
    /p.name := name
    every d := !p.disjuncts & /d.name & d.name := name
   return
end
```

Defines:
patbind, used in chunk 41a.

Uses add_to_rho 24a, disjunct 53, and pattern 53.

4.1.1 Explanation of pattern labels

A label `L:` in any sequence makes `L` available for use in field bindings throughout that sequence. However, we wish to limit the scope of `L` to the literal sequence in which it appears; when we reuse a pattern by using a name bound with `patterns` or by applying a constructor, labels in the pattern so denoted should not be visible. In other words, the scope of a label `L` is the enclosing pattern or constructor definition.

Conjoining two sequences preserves labels. This is a pain to implement (see `merge_sequences`, especially the chunk ⟨*suck initial labels into* `l` 65b⟩).

In a single `when` clause of a constructor definition, a label `L` can be used as inputs to the set of equations as long as it appears in all disjuncts of the associated pattern. (This is, of course, one way of propagating the label into field bindings.)

We enforce the scoping of labels by substituting `Epatlabel` expressions for variables that refer to labels. (We do this after running the solver, when binding patterns into what we might call "top-level normal form." Of course this has nothing to do with patterns themselves; it's a property of the expressions in field bindings, but we do null out the `name` field of a `patlabel` so it can mark the location without interfering with other labels that are differently scoped but identically named.) At the emission level (untyped constructor), we'll rewrite `patlabel` expressions into the form `Epc() + offset`. At the recognition level, we'll insert a binding of the name of the label to an expression of the same form.

The invariant on `patlabel`s is that the `name` field is non-null during parsing and conversion to normal form, but the `name` field must be null in any pattern that results from a `lookup` or a `crhs` call. We might actually be able to do something with this in ML by parameterising the type by `unit` or `string`.

Another invariant we want to maintain is that no `patlabel` should appear in more than one sequence. Conjunction can't do it, because it destroys all labels. Disjunction can't do it, because collecting a larger group of sequences together can't violate the invariant. But concatentation can do it, because we replicate sequences to get an outer product. So we have to be careful in `seqpx`. We exploit the invariant by setting `name` fields null when binding label names.

A `latent_patlabel` is used to implement label binding of free constructor-typed arguments in matching statements. The label has to be associated with the constructor-input pattern, but making it a label at constructor-input time will just put it in the wrong scope. Instead, we keep the latent pattern label around until we know whether it will refer to a real instance `Einstance(name)` or a binding instance `Ebinding_instance(name)`. In the former case it vanishes; in the latter, it turns into a real label. `eliminate_instances` does the work.

55 ⟨* 53⟩+≡ ◁54 56▷

```
procedure bind_and_remove_patlabel_names(p, keepname, free_vars_ok)
  if type(!(!p.disjuncts).sequents) == "patlabel" then &null else return p  # prem opt
  q := pattern([], p.name)
  every put(q.disjuncts, barpm_d(!p.disjuncts, keepname, free_vars_ok))
  return q
```

```
  end

  procedure barpm_d(d, keepname, free_vars_ok)
    t := table()
    every s := !d.sequents & type(s) == "patlabel" & \s.name do {
      (/t[s.name] := Epatlabel(s)) |
        error("Duplicate labels ", s.name, " in disjunct ", expimage(d))
      if s.name == free_variables(d) then &null
      else \free_vars_ok |
           warning("Label ", s.name, " not used in disjunct ", expimage(d))
      if /keepname then s.name := &null
    }
    return subst_tab(d, t)
  end

  procedure bind_patlabel_names(p, free_vars_ok)
    return bind_and_remove_patlabel_names(p, 1, free_vars_ok)
  end
```

Defines:
`bind_and_remove_patlabel_names`, used in chunk 166a.
`bind_patlabel_names`, used in chunk 248.
Uses `disjunct` 53, `Epatlabel` 79, `expimage` 98a, `free_variables` 82b, `patlabel` 53, `pattern` 53, `subst_tab` 87a, and `used` 132a.

56 ⟨* 53⟩+≡ ◁55 57b▷

```
  procedure all_disjuncts_ids(p, idgen, warningmsg, warning2)
    local alld
    alld := set()
    every insert(alld, idgen(p.disjuncts[1]))
    every d := p.disjuncts[2 to *p.disjuncts] do {
      s := set()
      every i := idgen(d) do
        if member(alld, i) then
          insert(s, i)
        else
          ⟨warn of discarding i 57a⟩
      if *s < *alld then
        every i := !alld & not member(s, i) do
          ⟨warn of discarding i 57a⟩
      alld := s
    }
    return alld
  end
```

Defines:
`all_disjuncts_ids`, used in chunks 57b and 248.

57a ⟨*warn of discarding* i 57a⟩≡ (56)
```
if \warningmsg then
  verbose(warningmsg, image(i), warning2)
```
Uses verbose 16a.

57b ⟨* 53⟩+≡ ◁56 57c▷
```
procedure patlabels_defined_in_all_disjuncts_of(p)
  s := all_disjuncts_ids(p, patlabels_defined_in)
  suspend !s
end

procedure patlabels_defined_in(d)
  suspend (seq := !d.sequents, type(seq) == "patlabel", seq.name)
end
```
Uses all_disjuncts_ids 56 and patlabel 53.

57c ⟨* 53⟩+≡ ◁57b 57d▷
```
procedure set_patlabel_offsets(d)
  local offset
  offset := 0
  every i := 1 to *d.sequents do
    case type (s := d.sequents[i]) of {
      "patlabel" : s.offset := bits_to_pcunits(offset)
      "sequent"  : offset +:= s.class.size
      "dots_sequent" | "latent_patlabel" : &null
      default                            : impossible("sequent type")
    }
  return
end
```
Defines:
set_patlabel_offsets, used in chunks 184, 248, and 300a.
Uses bits_to_pcunits 62b, dots_sequent 60c, latent_patlabel 53, patlabel 53, and sequent 53.

We sometimes have to make fresh pattern labels (as with constructor application).

57d ⟨* 53⟩+≡ ◁57c 58a▷
```
procedure freshen_patlabels(p)
  t := table()
  every l := subterms_matching(p, "patlabel") do
    /t[l] := copy(l)
  return if *t = 0 then p else subst_table_elements(p, t)
end
```
Defines:
freshen_patlabels, used in chunks 62a and 169a.
Uses patlabel 53, subst_table_elements 87b, and subterms_matching 82a.

4.1.2 Pattern constructor functions

The constructors build the basic patterns.

58a ⟨* 53⟩+≡ ◁57d 58b▷
```
procedure label2pattern(name)
    return pattern([disjunct([patlabel(name, name)])])
end
```
Defines:
`label2pattern`, used in chunks 73 and 250a.
Uses `disjunct` 53, `patlabel` 53, and `pattern` 53.

58b ⟨* 53⟩+≡ ◁58a 59▷
```
procedure latent_label2pattern(name)
    return pattern([disjunct([latent_patlabel(name)])])
end
```
Defines:
`latent_label2pattern`, used in chunk 168a.
Uses `disjunct` 53, `latent_patlabel` 53, and `pattern` 53.

59 ⟨* 53⟩+≡ ◃58b 60a▹

```
procedure constraints2pattern(L[])
    return pattern([disjunct([sequent(L, L[1].field.class)])])
end
procedure conspat(f, op, val)
    local max, p, q
    type(val) == "integer" | impossible("non-integer constraint")
    max := 2 ^ fwidth(f)
    return case op of {
        "="  : constraints2pattern(constraint(f, val, val+1))
        "<"  : { if val <= 0 then warning("Impossible constraint ", f.name, " < ", val)
                 if val > max then
                   error("Constraint value too large in ", f.name, " < ",val)
                 constraints2pattern(constraint(f, 0, val))
               }
        "<=" : { if val >= max then
                   error("Constraint value too large in ", f.name, " <= ", val)
                 constraints2pattern(constraint(f, 0, val+1))
               }
        ">"  : { if val >= max-1 then
                   warning("Impossible constraint ", f.name, " > ", val)
                 if val < 0 then
                   error("Constraint value too small in ", f.name, " > ",val)
                 constraints2pattern(constraint(f, val+1, max))
               }
        ">="  : { if val >= max then
                   warning("Impossible constraint ", f.name, " >= ", val)
                 if val < 0 then
                   error("Constraint value too small in ", f.name, " >= ",val)
                 constraints2pattern(constraint(f, val, max))
               }
        "!=" : {
            if val > 0       then p := constraints2pattern(constraint(f, 0, val))
            if val + 1 < max then q := constraints2pattern(constraint(f, val+1, max))
            orpx(\p, \q) | \p | \q | impossible("lost inequality")
        }
    }
end
```

Defines:
  conspat, used in chunks 74a, 75a, and 158b.
  constraints2pattern, used in chunks 60a, 75a, 166b, 168b, and 250a.

Uses constraint 53, disjunct 53, field 53, fwidth 112d, max 396, orpx 61c, pattern 53, sequent 53, and value 132a.

60a ⟨* 53⟩+≡ ◁59 60b▷
```
procedure wildcard(class)    # return a pattern that matches anything in class
    return constraints2pattern(constraint(f := class.fields[1], 0, 2^fwidth(f)))
end
```
Defines:
`wildcard`, used in chunk 40a.
Uses `constraint` 53, `constraints2pattern` 59, `fwidth` 112d, and `pattern` 53.

60b ⟨* 53⟩+≡ ◁60a 60c▷
```
procedure epsilon()     # pattern with shape of length 0
    static e
    initial e := pattern([disjunct([], "epsilon")], "epsilon")
    return e
end
```
Defines:
`epsilon`, used in chunks 40a, 49a, 70a, 99c, and 358a.
Uses `disjunct` 53 and `pattern` 53.

A special kind of sequent, "`dots`," is used to implement the ellipsis. Only one instance is ever used. An invariant property is that `dots` may appear only at the ends of a sequence. The invariant is maintained by the concatenation mutator, which forces `dots` to disappear "in the middle." The purpose of dots is to relax the shape-matching rules used in conjunction (q.v.).

60c ⟨* 53⟩+≡ ◁60b 61a▷
```
record dots_sequent()           # a very special sequent
global dots
procedure dots_pattern()
    static d
    initial d := pattern([disjunct([dots := dots_sequent()], "...")], "...")
    return d
end
```
Defines:
`dots`, used in chunks 63a, 65a, 66c, and 356.
`dots_pattern`, used in chunks 37b and 363b.
`dots_sequent`, used in chunks 57c, 83a, 90c, 101a, 114b, and 119b.
Uses `disjunct` 53, `pattern` 53, and `sequent` 53.

4.1.3 Pattern mutator functions

The mutators `&`, `|`, and `;`, combine patterns. When generators are used in the specification, they are represented as lists of patterns. It is therefore necessary to be able to combine (with `andp`, `orp`, and `seqp`) not just two patterns, but a pattern and a list. `mix` does the job.

61a ⟨* 53⟩+≡ ◁60c 61b▷

```
procedure andp(p1, p2)
    return mix(andpx, p1, p2)
end

procedure orp(p1, p2)
    return mix(orpx, p1, p2)
end

procedure seqp(p1, p2)
    return mix(seqpx, p1, p2)
end
```

Defines:
`andp`, used in chunk 73.
`orp`, used in chunks 73, 165, and 168a.
`seqp`, used in chunks 70a and 73.
Uses `andpx` 63b, `mix` 61b, `orpx` 61c, and `seqpx` 62a.

61b ⟨* 53⟩+≡ ◁61a 61c▷

```
procedure mix(op, p1, p2) # can destroy its list argument because
                          # lists are never saved or bound to names
    return case type(p1) || "," || type(p2) of {
        "pattern,pattern" : op(p1, p2)
        "list,pattern"    : { every i := 1 to *p1 do p1[i] := op(p1[i], p2) ; p1 }
        "pattern,list"    : { every i := 1 to *p2 do p2[i] := op(p1, p2[i]) ; p2 }
        "list,list"       : { l := []; every put(l, op(!p1, !p2)); l }
        default           : error("Invalid combination of patterns.")
    }
end
```

Defines:
`mix`, used in chunk 61a.
Uses `pattern` 53.

61c ⟨* 53⟩+≡ ◁61b 62a▷

```
procedure orpx(p1, p2)
    return pattern(p1.disjuncts ||| p2.disjuncts)
end
```

Defines:
`orpx`, used in chunks 59 and 61a.
Uses `pattern` 53.

The responsibility for maintaining the no-multiple-appearance invariant over `patlabels` is here. The key features is that, by the invariant, we can count on the same label not appearing in both `d1` and `d2`. Therefore, when we combine them, we can get away with simply freshening the pattern labels. And if there's only one disjunct on each side, we don't even have to do that.

62a ⟨* 53⟩+≡ ◁61c 62b▷

```
procedure seqpx(p1, p2)
    local p, count1, count2
    p := pattern([])
    count1 := *p1.disjuncts
    count2 := *p2.disjuncts
    every d1 := !p1.disjuncts & d2 := !p2.disjuncts do {
      d := disjunct(concat_sequences(d1.sequents, d2.sequents), &null,
                    conjoin_conditions(d1.conditions, d2.conditions))
      if count1 > 1 | count2 > 1 then # d1 or d2 will be reused
        d := freshen_patlabels(d)
      put(p.disjuncts, d)
    }
    return p
end
```

Defines:
`seqpx`, used in chunks 61a, 168a, and 250a.
Uses `concat_sequences` 63a, `conjoin_conditions` 62c, `disjunct` 53, `freshen_patlabels` 57d, and `pattern` 53.

62b ⟨* 53⟩+≡ ◁62a 62c▷

```
procedure bits_to_pcunits(n)
  if n % pc_unit_bits ~= 0 then
    error("pc is incremented in units of ", pc_unit_bits, ", but some pattern is ", n,
          " bits wide")
  else
     return n / pc_unit_bits
end
```

Defines:
`bits_to_pcunits`, used in chunks 57c and 257.
Uses `pattern` 53 and `pc_unit_bits` 16a.

62c ⟨* 53⟩+≡ ◁62b 63a▷

```
procedure conjoin_conditions(c1, c2)
  return if /c1 then c2
         else if /c2 then c1
         else c1 ++ c2
end
```

Defines:
`conjoin_conditions`, used in chunks 62a and 63b.

concat_sequences maintains the invariant on dots.

63a ⟨* 53⟩+≡ ◁62c 63b▷

```
procedure concat_sequences(s1, s2)
  local seq
  seq := if s1[-1] === dots & *s1 > 1 then
           if s2[1] === dots & *s2 > 1 then
             s1[1:-1] ||| s2[2:0]
           else
             s1[1:-1] ||| s2
         else
           if s2[1] === dots & *s2 > 1 then
             s1       ||| s2[2:0]
           else
             s1       ||| s2
  return seq
end
```

Defines:
concat_sequences, used in chunk 62a.
Uses dots 60c.

It is desirable to preserve names when patterns are combined with field bindings in pattern-matching statements. Because it's late at night and I'm lazy, I'm making this work only when field bindings are anded in on the right.

63b ⟨* 53⟩+≡ ◁63a 65a▷

```
procedure andpx(p1, p2)
    local keepname
    ⟨make keepname name of p1 if p2 is field binding, null otherwise 63c⟩
    p := pattern([], keepname)
    every d1 := !p1.disjuncts & d2 := !p2.disjuncts do {
        put(p.disjuncts,
            d := not_contradictory(
                     disjunct(merge_sequences(d1.sequents, d2.sequents), &null,
                              conjoin_conditions(d1.conditions, d2.conditions))))
        if \keepname then d.name := d1.name
    }
    return p
end
```

Defines:
andpx, used in chunk 61a.
Uses conjoin_conditions 62c, disjunct 53, merge_sequences 65a, not_contradictory 69a, and pattern 53.

63c ⟨*make* keepname *name of* p1 *if* p2 *is field binding, null otherwise* 63c⟩≡ (63b)

```
if *p2.disjuncts  = 1 & d := p2.disjuncts[1] &
   *d.sequents    = 1 & s := d.sequents[1]   & type(s) == "sequent" &
   *s.constraints = 1 & type(s.constraints[1]) == "fieldbinding" then
  keepname := p1.name
```

Uses constraints 132a, fieldbinding 53, and sequent 53.

`merge_sequences` merges from the left if neither sequence has a `dots` there; otherwise it merges from the right. Direction is determined by `inc`, which is added to a list index, and by `add`, which is used to add a sequent to the result.

64a ⟨*set* `grab`, `add` *and* `first` *for right-to-left* 64a⟩≡ (65a)
```
grab := pull
add := push
first := -1
```

64b ⟨*set* `grab`, `add` *and* `first` *for left-to-right* 64b⟩≡ (65a)
```
grab := get
add := put
first := 1
```

65a ⟨* 53⟩+≡ ◁63b 66e▷

```
procedure merge_sequences(l1, l2)
    local grab, first, add
    l1 := copy(l1)
    l2 := copy(l2)
    if (l1|l2)[1] === dots then
      if (l1|l2)[-1] === dots then
        ⟨complain about dots 66c⟩
      else {⟨set grab, add and first for right-to-left 64a⟩}
    else {⟨set grab, add and first for left-to-right 64b⟩}

    l := []
    ⟨suck initial labels into l 65b⟩
    # invariant: initial elements, if present, are not labels
    while l1[first] ~=== dots & l2[first] ~=== dots do {
      s1 := grab(l1)
      s2 := grab(l2)
      if s1.class ~=== s2.class then
        ⟨complain about class mismatch 66d⟩
      else add(l, x := sequent(merge_constraints(s1.constraints, s2.constraints),
                               s1.class))
      ⟨suck initial labels into l 65b⟩
    }
    # invariant: initial elements, if present, are not labels
    if l1[first] === dots then
      while add(l, dots ~=== grab(l2))
    else if l2[first] === dots then
      while add(l, dots ~=== grab(l1))
    else if l1[first] then
      ⟨complain of right sequence too short 66a⟩
    else if l2[first] then
      ⟨complain of left sequence too short 66b⟩
    # else they're an exact match; do nothing
    return l
end
```

Defines:
  `merge_sequences`, used in chunk 63b.
Uses `constraints` 132a, `dots` 60c, `merge_constraints` 66e, and `sequent` 53.

65b ⟨*suck initial labels into* `l` 65b⟩≡ (65a)

```
every ll := l1 | l2 do
  while type(ll[first]) == ("patlabel" | "latent_patlabel") do
    add(l, grab(ll))
```

Uses `latent_patlabel` 53 and `patlabel` 53.

66a ⟨*complain of right sequence too short* 66a⟩≡ (65a)
```
error("Shapes differ for &; right-hand sequence (",
       sequenceimage(l2), ") too short to conjoin with (", sequenceimage(l1), ")")
```
Uses `sequenceimage` 356a.

66b ⟨*complain of left sequence too short* 66b⟩≡ (65a)
```
error("Shapes differ for &; left-hand sequence (",
       sequenceimage(l1), ") too short to conjoin with (", sequenceimage(l2), ")")
```
Uses `sequenceimage` 356a.

66c ⟨*complain about dots* 66c⟩≡ (65a)
```
error("Illegal conjunction; dots on both left and right of (",
       sequenceimage(l1), ") & (", sequenceimage(l2), ")")
```
Uses `dots` 60c and `sequenceimage` 356a.

66d ⟨*complain about class mismatch* 66d⟩≡ (65a)
```
error("Shapes differ for &; left sequent from class `", s1.class.name,
      "'; right sequent from class `", s2.class.name, "', in\n\t(",
      expimage(s1), ") & (", expimage(s2), ")")
```
Uses `expimage` 98a and `sequent` 53.

`merge_constraints` merges two sets of constraints, preserving the invariant that a field may appear at most once on any list of constraints.

66e ⟨* 53⟩+≡ ◁65a 67b▷
```
procedure merge_constraints(l1, l2)
    local c1, c2, S1, S2, maxlo, minhi
    S1 := set(l1)
    S2 := set()
    every c2 := !l2 do
      if c1 := !S1 & c1.field === c2.field then { # doubly-constrained field
          if type(c1) == type(c2) == "constraint" then {
              delete(S1, c1)
              ⟨merge c1 and c2, inserting result in S2 66f⟩
          } else
             ⟨complain of constraint conflict 67a⟩
      } else
          insert(S2, c2)
    return sort(S1 ++ S2)
end
```
Defines:
`merge_constraints`, used in chunk 65a.
Uses `constraint` 53 and `field` 53.

66f ⟨*merge* `c1` *and* `c2`*, inserting result in* `S2` 66f⟩≡ (66e)
```
maxlo := c1.lo < c2.lo | c1.lo
minhi := c1.hi > c2.hi | c1.hi
if maxlo >= minhi then maxlo := minhi := 0      # canonical impossible constraint
insert(S2, constraint(c1.field, maxlo, minhi))
```
Uses `constraint` 53 and `field` 53.

67a ⟨*complain of constraint conflict* 67a⟩≡ (66e)
```
error("Code and constraint (or inconsistent code) for field `", c1.field.name,
      "\n\t", constraintimage(c1), "\n\t", constraintimage(c2))
```
Uses `constraint` 53, `constraintimage` 357a 357a, and `field` 53.

67b ⟨* 53⟩+≡ ◁66e 68a▷
```
procedure strip_patlabels(s)
  if type(!s) == "patlabel" then &null else return s
  l := []
  every put(l, 1(x := !s, type(x) ~== "patlabel"))
  return l
end
```
Uses `patlabel` 53.

4.2 Preparing patterns for emission

Fields within a sequent can be constrained either by field bindings or by range constraints. Given a sequent S and a field F, S may contain at most one field binding of F and at most one constraint on F. (The old invariant didn't permit both.) Here are the semantics:

- on output (encoding), a field binding is a string that is an rvalue in the target language (e.g. C). The rvalue will be assigned to the field f in the output pattern
- on input (decoding), a field binding must be an identifier. that identifier will be declared as a readonly variable in an inner scope, and it will be initialized to the value of the field
- on output, a constraint may determine the value of a field uniquely, in which case that value is assigned to the field. if the constraint doesn't determine the value uniquely, there must be a field binding that does. on input, a constraint limits the number of word sequences that can match.

Priority:

- on input, there can never be a conflict between a constraint and a field binding; they can be handled independently.
- on output, the conflict should be handled as follows: before handing the pattern to an emit procedure, we run a transformation over it that removes the constraint and turns it into a target-code constraint which guarantees that the value in the binding satisfies the constraint. That transformation is `sanitize_for_output`.

sanitize_for_output accepts a sequent and returns a smaller sequent in which at most one constraint is present for each field. Superfluous range constraints are transformed into conditions and bound to the disjunct d.

68a ⟨* 53⟩+≡ ◁67b 68c▷
```
procedure sanitize_for_output(seq, conditions)
    local fields, outconstraints
    ⟨if no field is overconstrained, return seq 68b⟩
    warning("Sanitizing ", expimage(seq))
    fields := set()
    outconstraints := []        # will be constraints of new sequent
    every insert(fields, (!seq.constraints).field)
    every f := !fields do {
       ((fb := !seq.constraints).field == f, type(fb) == "fieldbinding") | (fb := &null)
       ((rc := !seq.constraints).field == f, type(rc) == "constraint")   | (rc := &null)
       if \fb & \rc then {
         put(outconstraints, fb)
         insert_constraint_conditions(conditions, rc, fb.code)
       } else {
         put(outconstraints, \fb | \rc)
       }
    }
    return sequent(outconstraints, seq.class)
end
```
Defines:
sanitize_for_output, used in chunk 186a.
Uses constraint 53, constraints 132a, expimage 98a, field 53, fieldbinding 53, insert_constraint_conditions 68c, and sequent 53.

68b ⟨if no field is overconstrained, return seq 68b⟩≡ (68a)
```
if type(seq) == "sequent" &
   c1 := !seq.constraints & c2 := !seq.constraints &
   c1 ~=== c2 & c1.field === c2.field & type(c1) ~== type(c2) then &null
else
   return seq
```
Uses constraints 132a, field 53, and sequent 53.

68c ⟨* 53⟩+≡ ◁68a 69a▷
```
procedure insert_constraint_conditions(conditions, rc, code)
  every insert_condition(conditions, eqn(                0 < rc.lo, "<=", code) |
                                     eqn(2^fwidth(rc.field) > rc.hi, ">",  code))
  return
end
```
Defines:
insert_constraint_conditions, used in chunk 68a.
Uses eqn 81c, field 53, fwidth 112d, and insert_condition 123c.

4.3 Eliminating contradictions

On the Intel 486, some modifiers may be used only with certain opcodes, e.g., `grp3.Eb | grp3.Ev`. When such a disjunction is conjoined with a particular opcode, the result contains contradictions, e.g.

`(grp3.Eb | grp3.Ev) & grp3.Eb` $\equiv$ `grp3.Eb |` *impossible.*

`not_contradictory` can be applied to a disjunct; it succeeds if and only if the disjunct contains no impossible constraints. We might one day decide to use it to eliminate disjuncts with conditions that are evidently always false.

69a ⟨* 53⟩+≡ ◁68c 69b▷

```
procedure not_contradictory(d)
 if s := !d.sequents    & type(s) == "sequent" &
    c := !s.constraints & type(c) == "constraint" & c.lo >= c.hi then fail
 else return d
end
```

Defines:
`not_contradictory`, used in chunk 63b.
Uses `constraint` 53, `constraints` 132a, `fail` 396, and `sequent` 53.

`eliminate_contradictions` returns a pattern without any impossible disjuncts

69b ⟨* 53⟩+≡ ◁69a 70a▷

```
procedure eliminate_contradictions(p)
  return if d := !p.disjuncts   &
            s := !d.sequents    & type(s) == "sequent" &
            c := !s.constraints & type(c) == "constraint" & c.lo >= c.hi then
    {impossible("contradiction now eliminated at conjunction")
     ⟨new pattern with contradictions eliminated 69c⟩}
  else p
end
```

Defines:
`eliminate_contradictions`, used in chunk 165.
Uses `constraint` 53, `constraints` 132a, and `sequent` 53.

69c ⟨*new pattern with contradictions eliminated* 69c⟩≡ (69b)

```
l := []
every d := !p.disjuncts do
  if s := !d.sequents    & type(s) == "sequent" &
     c := !s.constraints & type(c) == "constraint" & c.lo >= c.hi then
    &null
  else
    put(l, d)
pattern(l, p.name)
```

Uses `constraint` 53, `constraints` 132a, `pattern` 53, and `sequent` 53.

4.4 Functions on patterns

place_holder takes a disjunct and returns a disjunct that may be used as a place holder for it.

70a ⟨* 53⟩+≡ ◁69b 70c▷
```
procedure place_holder(d)
  local shape, class
  shape := shapeof(d)
  ⟨complain if some class in shape has no placeholder 70b⟩
  p := epsilon()
  every class := !shape do p := seqp(p, class.holder)
  *p.disjuncts = 1 | impossible("Some placeholder has multiple disjuncts")
  return p.disjuncts[1]
end
```
Defines:
place_holder, used in chunks 196b and 205b.
Uses epsilon 60b, seqp 61a, and shapeof 71a.

70b ⟨*complain if some class in* shape *has no placeholder* 70b⟩≡ (70a)
```
if /(class := !shape).holder then
    error("No placeholder is defined for class ", class.name)
```
Uses defined 132a.

pattern_length computes the length in bits of a pattern in normal form.

70c ⟨* 53⟩+≡ ◁70a 70d▷
```
procedure pattern_length(p)
  local shapes, class
  shapes := maplist(disjunct_length, p.disjuncts)
  *shapes > 0 | error("length of impossible pattern")
  if !shapes ~= shapes[1] then error("Length of pattern ", patimage(p), " is not fixed")
  return shapes[1]
end
```
Defines:
pattern_length, used in chunk 35b.
Uses disjunct_length 70d, patimage 358a, and pattern 53.

70d ⟨* 53⟩+≡ ◁70c 71a▷
```
procedure disjunct_length(d)
  n := 0
  every s := !d.sequents & type(s) == "sequent" do
    n +:= s.class.size
  return n
end
```
Defines:
disjunct_length, used in chunk 70c.
Uses sequent 53.

71a ⟨* 53⟩+≡ ◁70d 71b▷

```
procedure shapeof(d)
  l := []
  every s := !d.sequents & type(s) == "sequent" do
    put(l, s.class)
  return l
end
```

Defines:
shapeof, used in chunk 70a.
Uses sequent 53.

A pattern is a legitimate global pattern if it has no conditions and no field bindings and if no labels are named.

71b ⟨* 53⟩+≡ ◁71a 72▷

```
procedure insist_global_pattern(p)
  if d := !p.disjuncts & *\d.conditions > 0 then
    error(patimage(p), " cannot be used on the right-hand side; there are conditions")
  else if d := !p.disjuncts & type(s := !d.sequents) == "sequent" &
      type(!s.constraints) == "fieldbinding" then
    error(patimage(p), " cannot be used on the right-hand side; there are inputs")
  else if d := !p.disjuncts & type(s := !d.sequents) == "patlabel" & \s.name then
    error(patimage(p), " cannot be used on the right-hand side; there's a loose label")
  else
    return
end
```

Defines:
insist_global_pattern, used in chunk 41b.
Uses constraints 132a, fieldbinding 53, label 369a, patimage 358a, patlabel 53, sequent 53, and used 132a.

4.5 Interpreting abstract syntax into normal form

4.5.1 Tree walking with free variables

This code is used on the decoding end, where free variables in patterns are OK—they become binding instances. `pnf_recurse` produces a pattern in normal form. If the `free_env` argument is non-null, then certain free variables are permitted, and the bindings of the free variables are planted in `free_env`. In fact, only bindings of constructor-typed constructor arguments really go into this environment—bindings of free variables in field bindings are handled by the equation-solving mechanism, and pattern identifiers are never permitted to be free variables (how could they be binding instances). What about labels????

72 ⟨* 53⟩+≡ ◁71b 73▷

```
procedure pnf(p, rho, free_env)
  return eliminate_instances(pnf_recurse(p, rho, free_env))
end
```

Defines:
`pnf`, used in chunks 35a, 41a, 166a, and 248.
Uses `eliminate_instances` 174a and `pnf_recurse` 73.

pnf_recurse converts a pattern in abstract syntax into its normal form. The meaning of an applied constructor depends on the environment in which it is evaluated; we use explode_apps to produce the application(s) denoted by a constructor application.

73 ⟨* 53⟩+≡ ◁72 74b▷

```
procedure pnf_recurse(p, rho, free_env)
  x := case type(p) of {
    "pattern" : p
    "Pident"  : project(x := lookup(p.name, rho), "pattern") |
                  typeerror(x, "pattern or field operand", p.name, rho)
                # can never be a free identifier
    "Plabel"  : label2pattern(p.name)
    "Pcon"    : { f := lookuptype(p.name, "field", rho)
                  # free identifiers in constraint are always OK
                  ⟨constraint or field binding, depending on the type of p.value 74a⟩
                }
    "Pand"    : pnfreduce(andp,  p.patterns, rho, free_env)
    "Por"     : pnfreduce(orp,   p.patterns, rho, free_env)
    "Pseq"    : pnfreduce(seqp,  p.patterns, rho, free_env)
    "Papp"    : if (type(p.cons) == "list") then
                  pnf_recurse(explode_apps(p.cons, p.args, rho), rho, free_env)
                else apply_constructor(p.cons, p.args, rho, free_env)
    "Plist"   : maplist3(pnf_recurse, p.patterns, rho, free_env)
     default  : impossible("pnf_recurse")
  }
  if \showpnf then PPxwrite(PPnew(&output),
     "pnf_recurse(", image(if type(p) == "Pident" then p.name else p),
     ") returns: $t${ $o", ppexpimage(x), "$b$}")
  return x
end
```

Defines:
pnf_recurse, used in chunks 72 and 75c.

Uses andp 61a, apply_constructor 169a, constraint 53, explode_apps 39b, field 53, label2pattern 58a, lookup 23a, lookuptype 23a, orp 61a, Pand 39a, Papp 39a, pattern 53, Pcon 39a, Pident 39a, Plabel 39a, Plist 39a, pnfreduce 75c, Por 39a, ppexpimage 98d, project 27, Pseq 39a, seqp 61a, showpnf 19b, and typeerror 26b.

Before trying to differentiate a field binding from a constraint, we perform the appropriate mapping on lists of constraints (which we get when generators are used). We can take a short cut for the case of a list, because we know the suckers are always integers—generators can't be used with expressions.

74a ⟨*constraint or field binding, depending on the type of* `p.value` 74a⟩≡ (73)
```
  case type(p.value) of {
    "Glist"   : { l := []
                  every put(l, conspat(f, p.relop, !p.value.values))
                  l
                }
    default   : constraint_or_binding(f, p.relop, p.value, rho, free_env)
  }
```
Uses `conspat` 59, `constraint_or_binding` 75a, `Glist` 40b, and `value` 132a.

Now we have to deal with an expression on the right-hand side of a binding. The distinction between a constraint and a binding hinges on whether the expression is constant. Since identifiers in bindings could be free variables, field names, or 'other', we begin with a substitution.

74b ⟨* 53⟩+≡ ◁73 75a▷
```
  procedure binding_subst_f(e, rho, free_env)
    return if type(e) == "string" then
      (if is_defined(e, rho) then
         project(lookup(e, rho), "integer")
       else
         new_binding_instance(e, e, "integer", \free_env)
      ) | typeerror(lookup(e, rho), "integer", e, rho)
  end
```
Defines:
  `binding_subst_f`, used in chunk 75a.
Uses `is_defined` 23a, `lookup` 23a, `new_binding_instance` 75b, `project` 27, and `typeerror` 26b.

Now we can make the distinction:

75a ⟨* 53⟩+≡ ◁74b 75b▷
```
procedure constraint_or_binding(f, relop, val, rho, free_env)
  val := super_simplify(gsubst(val, binding_subst_f, fieldname_env_for(f) ||| rho,
                               free_env))
  val := constant(val) # might fail if expression not constant
  if type(val) == "integer" then
    return conspat(f, relop, val)
  else if relop == "=" then
    return constraints2pattern(fieldbinding(f, val))
  else
    error("`", expimage(val), "' has free variables, ",
          "so the only permissible constraint is equality")
end
```
Defines:
constraint_or_binding, used in chunk 74a.
Uses binding_subst_f 74b, conspat 59, constant 111b, constraint 53, constraints2pattern 59, expimage 98a, fail 396, fieldbinding 53, fieldname_env_for 342b, gsubst 117a, and super_simplify 90a.

Note that the error message lies; it might be possible to write a constraint in with a pattern, because constraints are now part of patterns. Must look into it.

75b ⟨* 53⟩+≡ ◁75a 75c▷
```
record binding_instance(val, type)
procedure new_binding_instance(name, val, type, env)
  if (/env[name] := binding_instance(val, type)) | env[name].type === type then
    return val
  else
    error(name, " was bound earlier as type ", expimage(env[name].type),
          " but it's used here as type ", expimage(type))
end
```
Defines:
binding_instance, used in chunks 101b and 173.
new_binding_instance, used in chunks 74b and 171.
Uses expimage 98a and used 132a.

`pnfreduce(op, l, rho)` applies `pnf_recurse` to each member of `l` then reduces the list (in the APL sense) by applying `op`. A special version is used when `op === seqp`, because that requires updating the program counter.

75c ⟨* 53⟩+≡ ◁75b 76a▷
```
procedure pnfreduce(op, l, rho, free_env)
  r := &null
  every p := pnf_recurse(!l, rho, free_env) do
     r := op(\r, p) | p
  return r
end
```
Defines:
pnfreduce, used in chunk 73.
Uses pnf_recurse 73.

76a ⟨* 53⟩+≡ ◁75c 76b▷

```
procedure equals_pc(e)
  return e === the_global_pc
end
```

Defines:
equals_pc, used in chunk 315a.
Uses the_global_pc 17d.

4.5.2 Finding free variables in abstract syntax

As far as I know, this code is used only for def-use checking.

76b ⟨* 53⟩+≡ ◁76a 77a▷

```
procedure pattern_free_variables(p)
    case type(p) of {
        "pattern" : fail        # note free var of Pcon is field, not pattern
        "Pident"  : return p.name
        "Plabel"  : fail # name is a binding instance
        "Pcon"    : suspend free_variables(p.value)
        "Pand" | "Por" | "Pseq"  :
                    suspend pattern_free_variables(!p.patterns)
        "Papp"    : suspend free_variables(!p.args)
        "string"  : return p
        "integer" : fail
        "literal" : fail
        default : impossible("pattern free variables")
    }
end
```

Defines:
pattern_free_variables, used in chunks 161d and 248.
Uses fail 396, field 53, free_variables 82b, literal 43e, Pand 39a, Papp 39a, pattern 53, Pcon 39a, Pident 39a, Plabel 39a, Por 39a, Pseq 39a, and value 132a.

Here we need label names for entry into the environment.

77a ⟨* 53⟩+≡ ◁76b 77b▷

```
procedure pattern_label_names(p)
    case type(p) of {
        "pattern" : fail
        "Pident"  : fail
        "Plabel"  : return p.name
        "Pcon"    : fail
        "Pand" | "Por" | "Pseq" : suspend pattern_label_names(!p.patterns)
        "Papp"    : fail
        "string"  : fail
        "integer" : fail
        "literal" : fail
        default : impossible("pattern label names")
    }
end
```

Defines:
pattern_label_names, used in chunks 158a, 166a, 200c, and 248.
Uses fail 396, label 369a, literal 43e, Pand 39a, Papp 39a, pattern 53, Pcon 39a, Pident 39a, Plabel 39a, Por 39a, and Pseq 39a.

The tricky part of computing the constructors applied in a pattern is that the exact set is not known until **pnf** is applied to the pattern. We make a conservative estimate by generating all possible applications using **explode_apps**.

77b ⟨* 53⟩+≡ ◁77a 78a▷

```
procedure constructors_applied_in(p)
    case type(p) of {
        "pattern" | "Pident" | "Plabel" | "Pcon" | "string" | "integer" | "literal" :
                 fail
        "Pand" | "Por" | "Pseq"  : suspend constructors_applied_in(!p.patterns)
        "Papp" : suspend (if type(p.cons) == "list" then
                            constructors_applied_in(explode_apps(p.cons,p.args))
                          else p.cons) |
                          constructors_applied_in(!p.args)
            default : impossible("pattern AST")
    }
end
```

Defines:
constructors_applied_in, used in chunks 200c and 201a.
Uses explode_apps 39b, fail 396, literal 43e, Pand 39a, Papp 39a, pattern 53, Pcon 39a, Pident 39a, Plabel 39a, Por 39a, and Pseq 39a.

4.6 Manipulations of conditions

I use a destructive `bind_condition` to add conditions. When I need something non-destructive, I use `freshen_disjuncts` to get a pattern with fresh disjuncts and conditions.

78a ⟨* 53⟩+≡ ◁77b 78b▷

```
procedure bind_condition(p, condition)
  every bind_condition_d(!p.disjuncts, condition)
  return p
end

procedure bind_condition_d(d, condition)
  /d.conditions := set()
  insert_condition(d.conditions, condition)
  return d
end
```

Defines:
`bind_condition`, used in chunks 166a and 168a.
Uses `insert_condition` 123c.

78b ⟨* 53⟩+≡ ◁78a

```
procedure freshen_disjuncts(p)
  return pattern(maplist(freshen_disjunct, p.disjuncts), p.name)
end
link maplist

procedure freshen_disjunct(d)
  d := copy(d)
  d.conditions := copy(d.conditions)
  return d
end
```

Defines:
`freshen_disjuncts`, used in chunks 166a, 168a, and 178b.
Uses `pattern` 53.

Chapter 5

Expressions and Equations

Expressions describe all the value manipulations performed by the toolkit. They include not only manipulations that can be expressed using the input expression language, but also manipulations that are generated internally, including slicing, narrowing, sign extension, division, and modulus. A special normal form is used to represent sums of terms with constant coefficients. The integer 1 may also be used as a term; it is multiplied by a coefficient to represent a constant term, if any. This normal form is represented as an Icon table in which the keys are the terms and the values are the coefficients. For example, the expression `10*x + 4` is represented by the Icon table mapping `"x"` to 10 and 1 to 4.

These expressions are used both in equation solving and to compute the values of fields. I give definitions in terms of Modula-3. `x` and `y` are expressions; `n` and `lo` are integer constants.

79 ⟨* 79⟩≡ 81b▷

```
record Eorb(x, y)                  # Word.Or(x, y)
record Eand(x, y)                  # x AND y              -- short-circuit Boolean
record Eslice(x, lo, n)            # Word.Extract(x, lo, width)
record Eshift(x, n)                # Word.Shift(x, n)
record Enarrowu(x, n)              # narrow to n bits, unsigned, with check
record Enarrows(x, n)              # narrow to n bits,   signed, with check
record Ewiden(x, n)                # sign-extend low n bits
record Ediv(x, n)                  # x DIV y
record Emod(x, n)                  # x MOD y
record Efitsu(x, n)                # unsigned value x fits in n bits
record Efitss(x, n)                # signed value x fits in n bits
record Epatlabel(l)                # reference to patlabel l
record Eforce(x)                   # force relocatable x to an integer
record Eforceable(x)               # true iff relocatable x has known address
record Epc()                       # start of pattern match or token emission
record Epc_known()                 # predicate needed in encoding
record Sstmts(x)                   # sequence of statements
record Enot(x)                     # NOT x
```

```
record Enosimp(x)                        # x, but suppress simplification
record Semit(x)                          # emit list of Stoken
record Stoken(x, n, offset)              # emit x with width n (offset from beginning)
                                         #       (offset >= 0 always)
record Einstance(cons, argt)             # cons applied to arguments in table argt
record Einstance_input(x, cons, name)    # input name from instance x of constructor cons
record Einstance_tagged(x, cons, uid)    # true iff instance x tagged as constructor cons
record Ebinding_instance(name, type, vart) # binding instance of constructor-typed input
                                           # vart is table used to find fresh vars
record Sif(arms)                         # list of Sguarded
record Sguarded(guard, x)                # if guard then x (in Sif)
record Sepsilon()                        # empty statement
record Efail(msg)                        # result of selecting from the wrong constructor
record Eapp(f, args)                     # function application
record Eclosure_loc()                    # location of a relocation closure
record Eclosure_addr(n)                  # relocatable address in a relocation closure
record Eclosure_val(n)                   # value in a relocation closure
```

Defines:

- `Eand`, used in chunks 84a, 94a, 103a, 109a, 115a, 120b, 127a, 181b, 313c, and 327c.
- `Eapp`, used in chunks 47, 81b, 84e, 107c, 115a, 121b, 130d, 171, and 172.
- `Ebinding_instance`, used in chunks 81b, 84a, 105b, 115a, 121a, 128e, 173–75, and 177a.
- `Eclosure_addr`, used in chunks 81b, 85a, 108a, 115a, 121b, 130e, 205b, 209a, 215b, and 315e.
- `Eclosure_loc`, used in chunks 81b, 85a, 108a, 115a, 121b, 130e, 199b, 215b, 315e, and 316a.
- `Eclosure_val`, used in chunks 81b, 85a, 108a, 115a, 121b, 130e, 205b, 209a, and 315e.
- `Ediv`, used in chunks 81b, 84a, 95, 103d, 109a, 115a, 120b, 127d, 135b, 145d, 146a, 314a, and 328a.
- `Efail`, used in chunks 94c, 95a, 97a, 106c, 115b, 121c, 129d, 174b, 175, 177a, 317b, and 329d.
- `Efitss`, used in chunks 84a, 96b, 104b, 115a, 120b, 127f, 182b, 190–92, 202d, 301b, 306a, 315g, and 328c.
- `Efitsu`, used in chunks 84a, 96b, 104b, 115a, 120b, 127f, 182b, 190–92, 202d, 301, 306a, 315g, and 328c.
- `Eforce`, used in chunks 81a, 84a, 96b, 104d, 115a, 120b, 127h, 168a, 183a, 191a, 199b, 212b, 215b, 300c, 316a, and 328d.
- `Eforceable`, used in chunks 84a, 96b, 104d, 115a, 120b, 127h, 191a, 316a, and 328d.
- `Einstance`, used in chunks 81, 84a, 90c, 97a, 105b, 115a, 121a, 128e, 169b, 174, 175, and 177a.
- `Einstance_input`, used in chunks 81b, 84a, 94b, 96b, 97a, 105b, 115a, 121a, 128e, 168a, 174, 175, 177a, 192c, 197b, 198a, 208b, 238b, 301c, 316f, and 329b.
- `Einstance_tagged`, used in chunks 84a, 97a, 105b, 115a, 121a, 128e, 168a, 174b, 175, 179, 180a, 316f, and 329b.
- `Emod`, used in chunks 81b, 84a, 95, 103d, 109a, 115a, 120b, 127d, 135b, 145d, 146a, 301a, 314a, and 328a.
- `Enarrows`, used in chunks 81b, 84a, 94–96, 103c, 109a, 115a, 120b, 127c, 147, 148, 171b, 183a, 191c, 192a, 313e, and 327e.
- `Enarrowu`, used in chunks 81b, 84a, 94, 96b, 103c, 109a, 115a, 120b, 127c, 191c, 192a, 313e, and 327e.
- `Enosimp`, used in chunks 81a, 84a, 90c, 104g, 115a, 120b, 128c, 188c, 316c, and 328f.
- `Enot`, used in chunks 84a, 96b, 104f, 115a, 120b, 128b, 182b, 184, 316b, and 328e.
- `Eorb`, used in chunks 81, 84a, 94a, 103a, 109a, 115a, 120b, 127a, 187, 210a, 313c, and 327c.
- `Epatlabel`, used in chunks 55, 84a, 104c, 115a, 120b, 127g, 185a, and 258a.

Epc, used in chunks 17e, 81a, 84a, 104e, 115a, 120b, 128a, 191a, 196b, 205b, 207, 299, and 315d.
Epc_known, used in chunks 81a, 84a, 104e, 115a, 120b, 128a, 191a, 196b, 205b, 207, and 315d.
Eshift, used in chunks 81b, 84a, 94–96, 103b, 109a, 113a, 115a, 120b, 127b, 150, 188c, 313d, and 327d.
Eslice, used in chunks 81b, 84a, 94–96, 103b, 109a, 112a, 115a, 120b, 127b, 140b, 141c, 149d, 150, 187, 285c, 304a, 313, 321a, 327, and 332c.
Ewiden, used in chunks 48b, 81b, 84a, 95, 96b, 103c, 109a, 115a, 120b, 127c, 140b, 141c, 147, 148, 313e, 318e, and 327e.
Semit, used in chunks 84a, 104a, 115a, 120b, 127e, 186b, 314b, and 328b.
Sepsilon, used in chunks 81a, 93a, 97b, 106b, 115b, 121c, 129c, 211b, 236, 318a, and 330b.
Sguarded, used in chunks 81a, 84b, 93b, 105c, 109a, 115b, 121c, 128f, 181, 182, 184, 199b, 205a, 211b, 236, 284b, 285a, and 287.
Sif, used in chunks 81a, 84b, 90c, 93a, 105c, 115b, 121c, 128f, 181, 182, 184, 199b, 205a, 236, 284b, 285a, 287b, 317a, and 329c.
Sstmts, used in chunks 84a, 97b, 105a, 115a, 120b, 128d, 196b, 205b, 316d, 317d, and 329a.
Stoken, used in chunks 81a, 84a, 104a, 115a, 120b, 127e, 187, 212a, and 314b.

Uses address 369a, constructor 155, input 44a, location, patlabel 53, pattern 53, relocatable 45a, token 345a, used 132a, and value 132a.

81a ⟨*argument descriptions* 81a⟩≡
```
expargs x y arms guard left right args.
special simplify Einstance Stagcase Enosimp Sif.
special hoist Epc Epc_known Eorb Sguarded Sepsilon Sfail Stoken Eforce.
```
Uses Eforce 79, Einstance 79, Enosimp 79, Eorb 79, Epc 79, Epc_known 79, hoist 207, Sepsilon 79, Sfail 182c, Sguarded 79, Sif 79, simplify 89b, Stagcase 178a, and Stoken 79.

The following types officially represent values:

81b ⟨* 79⟩+≡ ◁79 81c▷
```
procedure exptypes()
  suspend "string" | "integer" | "table" | "Eorb" | "Eslice" | "Eshift" | "Enarrowu" |
          "Enarrows" | "Ewiden" | "Ediv" | "Emod" | "Einstance_input" | "Einstance" |
          "Ebinding_instance" | "Eapp" | "Eclosure_loc" | "Eclosure_addr" |
          "Eclosure_val"
end
```
Defines:
exptypes, used in chunk 27.

Uses Eapp 79, Ebinding_instance 79, Eclosure_addr 79, Eclosure_loc 79, Eclosure_val 79, Ediv 79, Einstance 79, Einstance_input 79, Emod 79, Enarrows 79, Enarrowu 79, Eorb 79, Eshift 79, Eslice 79, and Ewiden 79.

An *equation* asserts that two expressions are equal. In this code, I extend the term to cover what is more normally called an "inequality"—a pair of expressions related by one of the usual relational operators.

81c ⟨* 79⟩+≡ ◁81b 82a▷
```
record eqn(left, op, right)            # equality or inequality
```
Defines:
eqn, used in chunks 46c, 68c, 83b, 90c, 92, 101c, 114d, 120a, 125b, 139c, 141a, 146a, 148, 150, 211a, 252, 301a, 302b, 304a, 305b, 312a, 326a, and 364b.

5.1 Searching for subterms and free variables

`subterms_matching(e, ty)` suspends all subterms of `e` that have type `ty`.

82a ⟨* 79⟩+≡ ◁81c 82b▷

```
procedure subterms_matching_f(e, types[])
  return if type(e) == !types then e
end

procedure subterms_matching(e, types[])
  suspend do_expwalk(e, subterms_matching_f, types)
end
```

Defines:
`subterms_matching`, used in chunks 57d, 191, 192a, 195–97, 204c, 205b, 288a, and 301c.
Uses `do_expwalk` 114a.

Finding free variables is a common enough case of `subterms_matching` that I define a special procedure for it:

82b ⟨* 79⟩+≡ ◁82a 82c▷

```
procedure free_variables_f(e)
  return if type(e) == "string" then e
end

procedure free_variables(e)
  suspend expwalk(e, free_variables_f)
end
```

Defines:
`free_variables`, used in chunks 55, 76b, 133a, 135, 136, 152a, 197b, 205a, 208c, and 250c.
Uses `expwalk` 114a.

5.2 Testing for equality

This is a conservative test.

82c ⟨* 79⟩+≡ ◁82b 85c▷

```
procedure exps_eq(e1, e2)
  if e1 === e2 then return e2
  e1 := untableexp(e1)
  e2 := untableexp(e2)
  return case type(e1) == type(e2) of {
    ⟨cases for exps_eq 83a⟩
  }
end
```

Defines:
`exps_eq`, used in chunks 83–85, 92e, 93b, 123c, 140b, 152a, 179a, 180a, 191b, 197c, 248, and 254.
Uses `untableexp` 86a.

83a ⟨*cases for* `exps_eq` 83a⟩≡ (82c) 83b ▷
```
  "pattern"      : fail
  "disjunct"     : fail
  "adisjunct"    : fail
  "sequent"      : fail
  "patlabel"     : e1 === e2
  "latent_patlabel" : e1 === e2  ## ???
  "dots_sequent" : fail
  "constraint"   : fail
  "fieldbinding" : fail
  "field"        : e1 === e2
  "absolute_field" : exps_eq(e1.field, e2.field) & e1.offset = e2.offset
```
Uses `absolute_field` 256, `adisjunct` 256, `constraint` 53, `disjunct` 53, `dots_sequent` 60c, `exps_eq` 82c, `fail` 396, `field` 53, `fieldbinding` 53, `latent_patlabel` 53, `patlabel` 53, `pattern` 53, and `sequent` 53.

83b ⟨*cases for* `exps_eq` 83a⟩+≡ (82c) ◁83a 83c ▷
```
  "list"         : if *e1 ~= *e2 then fail
                   else {
                     every i := 1 to *e1 do if not exps_eq(e1[i], e2[i]) then fail
                     e1
                   }
  "eqn"          : ⟨equations e1 and e2 are the same 85b⟩
```
Uses `eqn` 81c, `exps_eq` 82c, and `fail` 396.

83c ⟨*cases for* `exps_eq` 83a⟩+≡ (82c) ◁83b 83d ▷
```
  "table"        : { if *e1 ~= *e2 then fail
                     every k1 := key(e1) do
                       if k2 := key(e2) & e2[k2] = e1[k1] & exps_eq(k1, k2) then &null
                       else fail
                     e2
                   }
```
Uses `exps_eq` 82c and `fail` 396.

83d ⟨*cases for* `exps_eq` 83a⟩+≡ (82c) ◁83c 84a ▷
```
  "set"          : { if *e1 ~= *e2 then fail
                     s := set()
                     every x := !e1 do
                       if exps_eq(x, y := !e2)
                         then insert(s, y)
                       else
                         fail
                     *s = *e2
                    }
```
Uses `exps_eq` 82c and `fail` 396.

84a ⟨*cases for* `exps_eq` 83a⟩+≡ (82c) ◁83d 84b▷
```
"Eorb" | "Eand" : exps_eq(e1.x, e2.x) & exps_eq(e1.y, e2.y)
"Eslice" : e1.lo = e2.lo & e1.n = e2.n & exps_eq(e1.x, e2.x)
"Eshift" | "Enarrowu" | "Enarrows" | "Ewiden" | "Ediv" | "Emod" |
"Efitsu" | "Efitss" : e1.n = e2.n & exps_eq(e1.x, e2.x)
"Epatlabel" : e1.l === e2.l
"Eforce" | "Eforceable" | "Enot" | "Enosimp" | "Sstmts" | "Semit" : exps_eq(e1.x, e2.x)
"Epc" | "Epc_known" : true
"Stoken" : e1.offset = e1.offset & e1.n = e2.n & exps_eq(e1.x, e2.x)
"Einstance_input"  : e1.cons === e2.cons & e1.name == e2.name & exps_eq(e1.x, e2.x)
"Einstance_tagged" : e1.cons === e2.cons & e1.uid = e2.uid & exps_eq(e1.x, e2.x)
"Einstance"        : impossible("escaping Einstance")
"Ebinding_instance": impossible("escaping Ebinding_instance")
```
Uses `Eand` 79, `Ebinding_instance` 79, `Ediv` 79, `Efitss` 79, `Efitsu` 79, `Eforce` 79, `Eforceable` 79, `Einstance` 79, `Einstance_input` 79, `Einstance_tagged` 79, `Emod` 79, `Enarrows` 79, `Enarrowu` 79, `Enosimp` 79, `Enot` 79, `Eorb` 79, `Epatlabel` 79, `Epc` 79, `Epc_known` 79, `Eshift` 79, `Eslice` 79, `Ewiden` 79, `exps_eq` 82c, `Semit` 79, `Sstmts` 79, and `Stoken` 79.

84b ⟨*cases for* `exps_eq` 83a⟩+≡ (82c) ◁84a 84c▷
```
"Sif" | "Sguarded" | "Stagcase" : fail
```
Uses `fail` 396, `Sguarded` 79, `Sif` 79, and `Stagcase` 178a.

84c ⟨*cases for* `exps_eq` 83a⟩+≡ (82c) ◁84b 84d▷
```
"Glines"    : fail
"Gresynch"  : fail
"Gblock"    : fail
"Gdecl"     : fail
"Gcase"     : fail
"Gcasearm"  : fail
"Ginrange"  : fail
"Gsetname"  : fail
"Gnomatch"  : fail
"Gasgn"     : fail
"Tunsigned" : fail
```
Uses `fail` 396, `Gasgn` 282a, `Gblock` 281a, `Gcase` 282a, `Gcasearm` 282a, `Gdecl` 281a, `Ginrange` 282a, `Glines` 281a, `Gnomatch` 282a, `Gresynch` 281a, `Gsetname` 282a, and `Tunsigned` 282a.

84d ⟨*cases for* `exps_eq` 83a⟩+≡ (82c) ◁84c 84e▷
```
"string"  : e1 == e2
"integer" : e1 = e2
```

84e ⟨*cases for* `exps_eq` 83a⟩+≡ (82c) ◁84d 85a▷
```
"Eapp" : fail
```
Uses `Eapp` 79 and `fail` 396.

85a ⟨*cases for* `exps_eq` 83a⟩+≡ (82c) ◁84e

```
"Eclosure_loc" : e1
"Eclosure_addr" : e1.n = e2.n
"Eclosure_val" : e1.n = e2.n
```

Uses `Eclosure_addr` 79, `Eclosure_loc` 79, and `Eclosure_val` 79.

For matching equations, we have a plethora of possibilities. The first line is an optimization. For the second line, we test both for the same relation and "opposite" relations. Note that equality is its own opposite, so $a = b$ is equivalent to $a' = b'$ if either $a - b = a' - b'$ or $a - b = b' - a'$. Similarly for simple inequality.

85b ⟨*equations* `e1` *and* `e2` *are the same* 85b⟩≡ (83b)

```
(e1.op == e2.op, exps_eq(e1.left, e2.left), exps_eq(e1.right, e2.right)) |
(d1 := subtract(e1.left, e1.right),
  (e1.op == e2.op,
   exps_eq(d1, subtract(e2.left, e2.right))) |
  (e1.op == opposite_op(e2.op),
   exps_eq(d1, subtract(e2.right, e2.left))))
```

Uses `exps_eq` 82c, `opposite_op` 85c, and `subtract` 140c.

85c ⟨* 79⟩+≡ ◁82c 86a▷

```
procedure opposite_op(op)
  return case op of {
    "<"  : ">"
    "<=" : ">="
    ">"  : "<"
    ">=" : "<="
    "="  : "="
    "!=" : "!="
  }
end
```

Defines:
`opposite_op`, used in chunk 85b.

86a ⟨* 79⟩+≡ ◁85c 86b▷

```
procedure untableexp(e)
  local result
  if type(e) == "table" then {
    every k := key(e) & e[k] = 0 do
      delete(e, k)
    case *e of {
      0: return 0
      1: { k := key(e)
           if k === 1 then return e[k]
           else if e[k] = 1 then return k
         }
     }
  }
  return e
end
```

Defines:
`untableexp`, used in chunks 82c, 152a, 302b, 304a, and 307.

5.3 Substitution

I have two forms: single value for variable and values for variables in table.

86b ⟨* 79⟩+≡ ◁86a 87a▷

```
procedure subst_f(e, var, value)
  if type(e) === "string" then return if var == e then value else e
end

procedure subst(e, var, value)
  return if var === value then e        # optimization
         else do_gsubst(e, subst_f, [var, value])
end
```

Defines:
`subst`, used in chunks 88b, 165, 167a, 205a, 214a, and 248.
`subst_f`, used in chunk 88a.
Uses `do_gsubst` 117a and `value` 132a.

If `all` is not null, all the free variables in the expression must be bound by the table.

87a ⟨* 79⟩+≡ ◁86b 87b▷
```
procedure subst_tab_f(e, tbl, all)
  return if type(e) == "string" then
           \tbl[e] | if \all then
                        error("variable ", image(e)," is unbound in table ",
                                                  envimage(tbl, "substitions"))
                      else
                        e
end

procedure subst_tab(e, tbl, all)
  return do_gsubst(e, subst_tab_f, [tbl, all])
end
```
Defines:
`subst_tab`, used in chunks 55, 88b, 139b, 150, 168a, 169a, 182b, 252, 287b, 294, and 306b.
`subst_tab_f`, used in chunk 88a.
Uses `do_gsubst` 117a and `envimage` 25a.

Sometimes we just want to substitute for all elements of any table, even if they aren't just strings.

87b ⟨* 79⟩+≡ ◁87a 87c▷
```
procedure subst_table_elements_f(e, tbl)
  return \tbl[e]
end
procedure subst_table_elements(e, tbl)
  return do_gsubst(e, subst_table_elements_f, [tbl])
end
```
Defines:
`subst_table_elements`, used in chunks 57d, 199b, and 215b.
Uses `do_gsubst` 117a.

87c ⟨* 79⟩+≡ ◁87b 88a▷
```
procedure subst_for_pc_f(e, val)
  return if e === the_global_pc then val
end

procedure subst_for_pc(e, val)
  return do_gsubst(e, subst_for_pc_f, [val])
end
```
Defines:
`subst_for_pc`, used in chunks 284b, 285a, and 288a.
Uses `do_gsubst` 117a and `the_global_pc` 17d.

Sometimes the values in a table are expressions.

88a ⟨* 79⟩+≡ ◁87c 88b▷

```
procedure subst_values_in_table(t, var, value)
  return do_gsubst_values_in_table(t, subst_f, [var, value])
end

procedure subst_tab_values_in_table(t, var, value)
  return do_gsubst_values_in_table(t, subst_tab_f, [var, value])
end

procedure do_gsubst_values_in_table(t, f, closure)
  n := 0
  u := table()
  every k := key(t) & x := do_gsubst(t[k], f, closure) do {
    u[k] := x
    if x ~=== t[k] then n +:= 1
  }
  return if n > 0 then u else t
end
```

Defines:
`do_gsubst_values_in_table`, used in chunk 119a.
Uses `do_gsubst` 117a, `subst_f` 86b, `subst_tab_f` 87a, and `value` 132a.

The solver needs destructive substitution when `e` is a table. Fortuitously, we can use the same code for `dsubst` and `dsubst_tab`.

88b ⟨* 79⟩+≡ ◁88a 89a▷

```
procedure dsubst(e, var, value)
  return destructive_subst(e, var, value, subst)
end
procedure dsubst_tab(e, tbl, all)
  return destructive_subst(e, tbl, all, subst_tab)
end
```

Defines:
`dsubst`, used in chunk 136a.
`dsubst_tab`, used in chunk 139b.
Uses `destructive_subst` 89a, `subst` 86b, `subst_tab` 87a, and `value` 132a.

89a ⟨* 79⟩+≡ ◁88b 89b▷

```
procedure destructive_subst(e, a2, a3, substitute)
  m := copy(e)
  every k := key(m) & mm := m[k] & kk := substitute(k, a2, a3) do {
    e[k] -:= mm
    if type(kk) ~== "table" then
      e[kk] +:= mm
    else
      every v := key(kk) do e[v] +:= kk[v] * mm
  }
  every k := key(e) & e[k] = 0 do delete(e, k)
  return e
end
```

Defines:
`destructive_subst`, used in chunk 88b.

5.4 Simplification

I discovered late in the day that there need to be two kinds of simplification. Basically, some of the simplifying rules reduce the strength of operations, and after strength reduction the solver can't always invert the results. On the other hand, there are situations in which one simply can't do without simplification, as when it eliminates an unsolvable disjunct (see bug #2). A tentative fix is to designate some simplification rules as ⟨*super rewrite rules* 94b⟩, and to use them only on expressions that won't be further transformed. They are used in `super_simplify` only. ßChange #11

`simplify` is not functional; it may mutate its argument.

89b ⟨* 79⟩+≡ ◁89a 90a▷

```
procedure simplify(e)
  local left, right
  return case type(e) of {
    $define simfun simplify
      ⟨basic cases for simplification 90c⟩
      ⟨cases generated from rewrite rules 89c⟩
    $undef simfun
    default : e
  }
end
```

Defines:
`simplify`, used in chunks 81a, 133a, 165, 168a, 169a, 175, 179a, 248, and 306b.

I need an empty chunk for bootstrapping.

89c ⟨*cases generated from rewrite rules* 89c⟩≡ (89b)

90a ⟨* 79⟩+≡ ◁89b 93b▷

```
procedure super_simplify(e)
  local left, right
  return case type(e) of {
    $define simfun super_simplify
      ⟨basic cases for simplification 90c⟩
      ⟨cases generated from super rewrite rules 90b⟩
    $undef simfun
    default : e
  }
end
```

Defines:
super_simplify, used in chunks 19a, 20a, 75a, 170b, 171b, 182b, 188c, 191a, 192b, 199b, 205a, 252, 258b, 313, 321a, and 327e.

I need an empty chunk for bootstrapping.

90b ⟨*cases generated from super rewrite rules* 90b⟩≡ (90a)

90c ⟨*basic cases for simplification* 90c⟩≡ (89b 90a)

```
"eqn"      : {⟨simplify equation 92e⟩}
"pattern"  : {⟨simplify pattern 91a⟩}
"disjunct" : {⟨simplify disjunct 91b⟩}
"adisjunct": {⟨simplify adisjunct 91c⟩}
"sequent"  : if (l := simfun(e.constraints)) === e.constraints then e
             else sequent(l, e.class)
"patlabel"     : e
"dots_sequent" : e
"constraint"   : e
"fieldbinding" : if (x := simfun(e.code)) === e.code then e
                 else fieldbinding(e.field, x)
"absolute_field" : e
"set"   : {⟨simplify set of conditions 92a⟩}
"table" : {⟨simplify table 92d⟩}
"list"  : {l := maplist(simfun, e); if lists_match(e, l) then e else l}
"integer" | "string" : e
"Einstance" : {⟨simplify instance's argument table 92b⟩}
"Stagcase"  : {⟨simplify tag case's arm table 92c⟩}
"Sif"   : {⟨simplify if statement 93a⟩}
"Enosimp" : e       # suppresses simplification
```

Uses absolute_field 256, adisjunct 256, constraint 53, constraints 132a, disjunct 53, dots_sequent 60c, Einstance 79, Enosimp 79, eqn 81c, field 53, fieldbinding 53, lists_match 124a, patlabel 53, pattern 53, sequent 53, Sif 79, and Stagcase 178a.

Simplification of a disjunct could fail, implying that the disjunct's condition is always false.

91a ⟨*simplify pattern* 91a⟩≡ (90c)
```
l := []; every put(l, simfun(!e.disjuncts))
if lists_match(l, e.disjuncts) then e else pattern(l, e.name)
```
Uses `lists_match` 124a and `pattern` 53.

91b ⟨*simplify disjunct* 91b⟩≡ (90c)
```
c := simfun(e.conditions)
if member(\c, 0) then fail
else {
  l := []; every put(l, vanishing_latent_patlabel ~=== simfun(!e.sequents))
  if lists_match(l, e.sequents) then l := e.sequents
  if ss := !l & type(ss) == "sequent" &
     cc := !ss.constraints & type(cc) == "constraint" & cc.lo >= cc.hi
  then
    fail # this disjunct can't match -- eliminate it
  if c === e.conditions & l === e.sequents then e else disjunct(l, e.name, c)
}
```
Uses `constraint` 53, `constraints` 132a, `disjunct` 53, `fail` 396, `lists_match` 124a, `sequent` 53, and `vanishing_latent_patlabel` 30a.

91c ⟨*simplify adisjunct* 91c⟩≡ (90c)
```
c := simfun(e.conditions)
if member(\c, 0) then fail
else {
  l := simfun(e.aconstraints)
  if cc := !l & type(cc) == "constraint" & cc.lo >= cc.hi then
    fail # this disjunct can't match -- eliminate it
  if c === e.conditions & l === e.aconstraints then e ### & a === e.answers then e
  else adisjunct(l, e.name, c, e.length, e.patlabelbindings) ### , a)
}
```
Uses `adisjunct` 256, `constraint` 53, `disjunct` 53, and `fail` 396.

92a ⟨*simplify set of conditions* 92a⟩≡ (90c)
```
s := set()
every x := simfun(!e) do {
  if type(x) == "eqn" then {
    if      x.op == "="  & x.left === x.right then x := 1
    else if x.op == "!=" & x.left === x.right then x := 0
  }
  insert(s, x)
}
delete(s, 1)
return if member(s, 0) then set([0])
       else if sets_match(e, s) then e
       else if *s = 0 then &null
       else s
```
Uses eqn 81c and sets_match 124b.

92b ⟨*simplify instance's argument table* 92b⟩≡ (90c)
```
every k := key(e.argt) do e.argt[k] := simfun(e.argt[k])
e
```

92c ⟨*simplify tag case's arm table* 92c⟩≡ (90c)
```
every k := key(e.arms) do e.arms[k] := simfun(e.arms[k])
e
```

92d ⟨*simplify table* 92d⟩≡ (90c)
```
m := copy(e)
every k := key(e) do {
  m[k] -:= e[k]
  add_to_table(m, simfun(k), e[k])
}
every k := key(m) & m[k] = 0 do delete(m, k)
if *m > 1 then
  return if tables_match(m, e) then e else m
else
  return if !m = 1 then key(m) else if key(m) === 1 then !m else m
```
Uses add_to_table 123d and tables_match 124c.

92e ⟨*simplify equation* 92e⟩≡ (90c)
```
l := simfun(e.left)
r := simfun(e.right)
if exps_eq(l, r) then
  if e.left === 0 & e.right === 0 then e
  else eqn(0, e.op, 0)
else
  if e.left === l & e.right === r then e
  else eqn(l, e.op, r)
```
Uses eqn 81c and exps_eq 82c.

93a ⟨*simplify if statement* 93a⟩≡ (90c)
```
a := simp_arms(e.arms, simfun)
if *a = 0 then Sepsilon()
else if *a = 1 & guard_always_satisfied(a[1].guard) then a[1].x
else if a === (e.arms) then e
else Sif(a)
```
Uses guard_always_satisfied 93c, Sepsilon 79, Sif 79, and simp_arms 93b.

93b ⟨* 79⟩+≡ ◁90a 93c▷
```
procedure simp_arms(earms, simfun)
  local arms
  l := []
  arms := copy(earms)
  while a := get(arms) & g := simfun(a.guard) do
    if guard_always_satisfied(g) then { # always take this arm
      c := simfun(a.x)
      put(l, if exps_eq(g, a.guard) & exps_eq(c, a.x) then a else Sguarded(g, c))
      return if lists_match(l, earms) then earms else l
    } else if not case type(g) of {
      "integer" : g = 0
      "set"     : member(g, 0)
    } then { # sometimes take this arm
      c := simfun(a.x)
      put(l, if exps_eq(g, a.guard) & exps_eq(c, a.x) then a else Sguarded(g, c))
    }
  return if lists_match(l, earms) then earms else l
end
```
Defines:
simp_arms, used in chunk 93a.
Uses arm 259b, exps_eq 82c, guard_always_satisfied 93c, lists_match 124a, and Sguarded 79.

93c ⟨* 79⟩+≡ ◁93b 96a▷
```
procedure guard_always_satisfied(g)
  return case type(g) of {
      "integer" : g = 1
      "set"     : *g = 0
      "null"    : &null
  }
end
```
Defines:
guard_always_satisfied, used in chunks 93, 211b, 260b, 261, 266, 271b, 317c, and 329e.

94a ⟨*rewrite rules* 94a⟩≡ 94c ▷

```
Eorb(0, x) -> x
Eorb(x, 0) -> x
Eorb(N, M) -> ior(N, M)
Eorb(Eorb(x, N), M) -> Eorb(x, ior(N, M))
Eorb(N, x) -> Eorb(x, N)
Eand(1, x) -> x
Eand(0, _) -> 0
Eand(x, 1) -> x
Eand(_, 0) -> 0
Eand(x, y) ->
  if type(x) == "set" & *x = 0 then y
  else if type(x) == "set" & *y = 0 then x
  else e
Eslice(N, lo, n) -> iand(ishift(N, -lo), 2^n-1)
```

Uses `Eand` 79, `Eorb` 79, and `Eslice` 79.

94b ⟨*super rewrite rules* 94b⟩≡ 94d ▷

```
Eslice(Eshift(x, n), m, k) ->
  if m - n >= 0 then Eslice(x, m-n, k)
  else Eslice(Eshift(x, n-m), 0, k)
Eslice(y as Enarrowu(x, n), 0, m) -> if n <= m then y else e
Eslice(y as Enarrows(x, n), 0, m) -> if n <= m then y else e
Eslice(Eslice(x, l1, n1), l2, n2) ->
  if l2 >= n1 then 0 else Eslice(x, l1+l2, if n2 < n1-l2 then n2 else n1-l2)
Eslice(y as Einstance_input(x, c, name), 0, n) ->
  if ipt := inputs_of(c) & ipt.name == name & case type(ipt.meaning) of {
    "field" : fwidth(ipt.meaning) <= n
  } then y else e
Eslice(x, 0, n) -> if n = wordsize then x else e
```

Uses `Einstance_input` 79, `Enarrows` 79, `Enarrowu` 79, `Eshift` 79, `Eslice` 79, `field` 53, `fwidth` 112d, `inputs_of` 162a, and `wordsize` 36c.

94c ⟨*rewrite rules* 94a⟩+≡ ◁ 94a 95a ▷

```
Eshift(x, 0) -> x
Eshift(N, n) -> ishift(N, n)
Enarrowu(N, n) -> if 0 <= N < 2^n then N else Efail(expimage(e))
```

Uses `Efail` 79, `Enarrowu` 79, `Eshift` 79, and `expimage` 98a.

94d ⟨*super rewrite rules* 94b⟩+≡ ◁ 94b 95b ▷

```
Enarrowu(y as Eslice(x, lo, n), m) -> if n <= m then y else e
Enarrowu(x, M) -> if M >= wordsize then x else e
```

Uses `Enarrowu` 79, `Eslice` 79, and `wordsize` 36c.

95a ⟨*rewrite rules* 94a⟩+≡ ◁94c 95c▷
```
  Ewiden(N, n) ->
    if (iand(N,2^(n-1)) = 0) then iand(N,2^n-1)
    else ior(iand(2^wordsize-1,icom(2^n-1)),N)
  Enarrows(N, n) ->
    if -(2^(n-1)) <= N < 2^(n-1) then iand(N,2^n-1) else Efail(expimage(e))
```
Uses `Efail` 79, `Enarrows` 79, `Ewiden` 79, `expimage` 98a, and `wordsize` 36c.

95b ⟨*super rewrite rules* 94b⟩+≡ ◁94d 95d▷
```
  Enarrows(y as Eslice(x, lo, n), m) -> if n < m then y else e
```
Uses `Enarrows` 79 and `Eslice` 79.

95c ⟨*rewrite rules* 94a⟩+≡ ◁95a 95f▷
```
  Enarrows(Ewiden(x, n), m) -> if n = m then x else e
  Enarrows(y as Enarrows(x, n), m) -> if n = m then y else e
```
Uses `Enarrows` 79 and `Ewiden` 79.

Note we can't rewrite `Ewiden(Enarrows(x, N), N)` as `x` because there's no way to pick up `Efitss(x, N)`. Ugh.

95d ⟨*super rewrite rules* 94b⟩+≡ ◁95b 95e▷
```
  Enarrows(x, M) -> if M >= wordsize then x else e
  Ediv(N, M) -> if N > 0 then N / M else (N - M + 1) / M  # force toward - infinity
```
Uses `Ediv` 79, `Enarrows` 79, and `wordsize` 36c.

95e ⟨*super rewrite rules* 94b⟩+≡ ◁95d 95g▷
```
  Ediv(x, N) -> simfun(Eshift(x, - exactlog2(N))) | e
```
Uses `Ediv` 79, `Eshift` 79, and `exactlog2` 97c.

95f ⟨*rewrite rules* 94a⟩+≡ ◁95c 96b▷
```
  Emod(N, M) -> {x := integer(N % M); while x < 0 do x +:= M; x}
```
Uses `Emod` 79.

95g ⟨*super rewrite rules* 94b⟩+≡ ◁95e
```
  Emod(x, N) ->
    if x := strip_const_multiple_N(x, N) then
      simfun(Eslice(x, 0, exactlog2(N))) | Emod(x, N)
    else
      simfun(Eslice(x, 0, exactlog2(N))) | e
```
Uses `Emod` 79, `Eslice` 79, `exactlog2` 97c, and `strip_const_multiple_N` 96a.

This procedure helps optimize the constant term out of modulus expressions.

96a ⟨* 79⟩+≡ ◃93c 97c▹

```
procedure strip_const_multiple_N(x, N)
  local y
  if type(x) == "table" & x[1] ~= 0 & x[1] % N = 0 then {
    y := copy(x)
    y[1] := 0
    return y
  }
end
```

Defines:
strip_const_multiple_N, used in chunk 95g.

96b ⟨*rewrite rules* 94a⟩+≡ ◃95f 97b▹

```
Efitsu(N, m) -> if 0 <= N < 2^m then 1 else 0
Efitsu(Enarrowu(x, n), m)   -> if n <= m then 1 else e
Efitsu(Enarrows(x, n), m)   -> if n <= m then 1 else e
Efitsu(Eslice(x, lo, n), m) -> if n <= m then 1 else e
Efitsu(y as Einstance_input(_, _, _), m) ->
  if known_to_fit(input_fitsu, y, &null, m) then 1 else e
Efitsu(Eshift(x, n), m)     -> simfun(Efitsu(x, m-n))
Efitsu(x, M) -> if M >= wordsize then 1 else e
Efitss(N, m) -> if -(2^(m-1)) <= N < 2^(m-1) then 1 else 0
Efitss(Enarrowu(x, n), m)   -> if n < m then 1 else e
Efitss(Enarrows(x, n), m)   -> if n < m then 1 else e
Efitss(Eslice(x, lo, n), m) -> if n < m then 1 else e
Efitss(Ewiden(Eslice(x, lo, n), m), mm) -> if n = m <= mm then 1 else e
Efitss(y as Einstance_input(_, _, _), m) ->
  if known_to_fit(input_fitss, y, &null, m) then 1 else e
Efitss(Eshift(x, n), m)     -> simfun(Efitss(x, m-n))
Efitss(x, M) -> if M >= wordsize then 1 else e
Eforce(x as Eforce(_)) -> x
Eforce(N) -> N
Eforceable(N) -> 1
Enot(N) -> if N = 0 then 1 else 0
Enot(Enot(x)) -> x
```

Uses Efitss 79, Efitsu 79, Eforce 79, Eforceable 79, Einstance_input 79, Enarrows 79, Enarrowu 79, Enot 79, Eshift 79, Eslice 79, Ewiden 79, input_fitss 193a, input_fitsu 193a, known_to_fit 192c, and wordsize 36c.

See eliminate_instances for an explanation of why the transformations below are no longer implemented as rewrite rules.

97a ⟨*former rewrite rules now implemented by* eliminate_instances 97a⟩≡

```
Einstance_tagged(Einstance(c, a), c2, uid) -> if c === c2 then 1 else 0
Einstance_input(Einstance(c, a), c2, name) -> if c === c2 then a[name]
                                              else Efail(expimage(e))
Einstance_tagged(Ewildcard(_), _, _) -> 1
Einstance_input(Ewildcard(iname), c, fname) ->
   Ewildcard(iname || "." || c.name || "." || fname)
latent_patlabel(Einstance(_, _)) -> vanishing_latent_patlabel
latent_patlabel(Ewildcard(nam)) -> patlabel(nam, nam)
```

Uses Efail 79, Einstance 79, Einstance_input 79, Einstance_tagged 79, expimage 98a, iname 160b, latent_patlabel 53, patlabel 53, and vanishing_latent_patlabel 30a.

97b ⟨*rewrite rules* 94a⟩+≡ ◁96b

```
Sstmts(l) -> case *l of { 0 : Sepsilon(); 1: l[1]; default : e }
```

Uses Sepsilon 79 and Sstmts 79.

97c ⟨* 79⟩+≡ ◁96a 98a▷

```
procedure exactlog2(n)
  local log
  log := 0
  while 2^log < n do log +:= 1
  if n = 2^log then return log else fail
end
```

Defines:
exactlog2, used in chunk 95.
Uses fail 396.

5.5 Imaging

`expimage` prints a readable form of an expression. It is used only for diagnostic purposes; a separate procedure is used to produce C (or Modula-3) code that implements an expression.

98a ⟨* 79⟩+≡ ◁97c 98b▷

```
procedure expimage(e, pp, precedence, associativity)
  local leadingsign, prefix
  static nopp
  initial {⟨initialize prec and assoc 109a⟩
           ⟨initialize nopp 98c⟩}
  /pp := nopp
  /precedence := 0
  /associativity := "L"
  return case type(e) of {
    ⟨cases for expimage 99a⟩
    default    : (proc(type(e) || "image") | image)(e)
  }
end
```

Defines:
`expimage`, used in chunks 24–26, 55, 66d, 68a, 75, 94c, 95a, 97–108, 112a, 121, 124b, 132–34, 136a, 138–41, 146a, 150, 152b, 153, 160a, 173, 174b, 176a, 177a, 179a, 189b, 190a, 258a, 262, 263, 266, 288b, 294, 296, 300, 302a, 303, 305, 306, 337d, 356b, 359b, 363, and 364.

`expimage` can optionally use a prettyprinter, but by default it uses none.

98b ⟨* 79⟩+≡ ◁98a 98d▷

```
record ppspec(be, en, in, ou, nl, on, cn) # { } t b m o c
```

Defines:
`ppspec`, used in chunk 98.

98c ⟨initialize nopp 98c⟩≡ (98a)

```
nopp := ppspec("", "", "", "", "", "", "")
```

Uses `ppspec` 98b.

98d ⟨* 79⟩+≡ ◁98b 100b▷

```
procedure ppexpimage(e)
  static pp
  initial pp := ppspec("${", "$}", "$t", "$b", "$n", "$o", "$c")
  return expimage(e, pp)
end
```

Defines:
`ppexpimage`, used in chunks 73, 156, 164, 169a, 179a, 205a, 207, 214b, 221a, 248, 249, and 253a.

Uses `expimage` 98a and `ppspec` 98b.

99a ⟨*cases for* `expimage` 99a⟩≡ (98a) 99b ▷

```
  "arm" : pp.in || "| @" || e.file || ":" || e.line || ": " || pp.be ||
          expimage(e.pattern, pp) || pp.en || " => " || pp.cn || expimage(e.code, pp) ||
          pp.ou
```

Uses `arm` 259b, `expimage` 98a, `file` 345a, `line` 345a, and `pattern` 53.

99b ⟨*cases for* `expimage` 99a⟩+≡ (98a) ◁ 99a 99c ▷

```
  "pattern"  : if *e.disjuncts > 0 then {
                 s := commaseparate(maplist3(expimage, e.disjuncts, pp, prec["|"]),
                                   " " || pp.cn || "| ")
                 bracket(s, pp, "|", precedence) ### || ("[[" || \e.name || "]]" | "")
               } else "<NOMATCH>"
```

Uses `bracket` 108b, `expimage` 98a, `pattern` 53, and `prec` 108b.

99c ⟨*cases for* `expimage` 99a⟩+≡ (98a) ◁ 99b 100a ▷

```
  "disjunct" : { s := "(" || (if \e.name then e.name else "?noname?") || ") "
                 s ||:= if *\e.conditions > 0 then
                        "{" || pp.be || expimage(e.conditions, pp) || pp.en || "} => " ||
                        pp.in || pp.cn
                      else ""
                 s ||:= if *e.sequents = 0 then "epsilon"
                        else {
                          prefix := ""
                          every i := 1 to *e.sequents do {
                            s ||:= prefix || expimage(e.sequents[i], pp, prec[";"])
                            prefix :=
                              (if type(e.sequents[i]) == ("patlabel"|"latent_patlabel")
                               then ": "
                               else "; ") || pp.on
                          }
                        }
                 s ||:= if *\e.conditions > 0 then pp.ou else ""
                 bracket(s, pp, ";", precedence) ### || ("[" || \e.name || "]" | "")
               }
```

Uses `bracket` 108b, `disjunct` 53, `epsilon` 60b, `expimage` 98a, `latent_patlabel` 53, `patlabel` 53, and `prec` 108b.

100a ⟨*cases for* expimage 99a⟩+≡ (98a) ◁99c 101a▷
```
"adisjunct" : {  s := "(" || (if \e.name then e.name else "?noname?") || ") "
                 s ||:= if \e.patlabelbindings then {
                  "[" || pp.in || pp.be || bindingimage(e.patlabelbindings, pp) ||
                         pp.en || pp.ou || "] " || pp.cn
                } else ""
                s ||:= if *\e.conditions > 0 then
                      "{" || pp.be || expimage(e.conditions, pp) || pp.en || "} => " ||
                      pp.in || pp.cn
                    else ""
               s ||:= "LENGTH = " || expimage(e.length) || ", "
               s ||:= if *e.aconstraints = 0 then "<MATCH>"
                      else commaseparate(maplist3(expimage, e.aconstraints, pp, prec["&"]),
                                      " & " || pp.on)
               s ||:= if *\e.conditions > 0 then pp.ou else ""
###               if *\e.answers > 0 then {
###                 l := []
###                 every k := key(e.answers) do
###                   put(l, k || " = " || expimage(e.answers[k], pp, prec["="]))
###                 s ||:= pp.in || pp.on || "{" || pp.in || pp.be ||
###                     commaseparate(l, ", " || pp.cn) || pp.en || pp.ou || "}" || pp.ou
###               }
               bracket(s, pp, "&", precedence)
             }
```
Uses adisjunct 256, bindingimage 100b, bracket 108b, expimage 98a, and prec 108b.

100b ⟨* 79⟩+≡ ◁98d 108b▷
```
procedure bindingimage(t, pp)
  local sep, id
  sep := ""
  s := ""
  every id := key(t) do {
    s ||:= sep || pp.be || id || " -> " || expimage(t[id], pp) || pp.en
    sep := ", " || pp.on
  }
  return s
end
```
Defines:
bindingimage, used in chunk 100a.
Uses expimage 98a.

101a ⟨*cases for* expimage 99a⟩+≡ (98a) ◁100a 101b▷
```
  "sequent"  : { s := if *e.constraints = 0 then "some " || e.class.name
                      else commaseparate(maplist3(expimage, e.constraints, pp, prec["&"]),
                                          " & " || pp.on)
                  pp.be || bracket(s, pp, "&", precedence) || pp.en
                }
  "patlabel"        : \e.name | e.original_name || "#" || image(e)[17:-3] ||
                      ("@" || \e.offset | "") || ":"
  "latent_patlabel" : "(?" || expimage(e.instance, pp, prec[","]) || ":)"
  "dots_sequent"    : "..."
  "constraint"      : bracket(stringininterval(expimage(e.field, pp, prec["="]),
                                               e.lo, e.hi), pp, "=", precedence)
  "fieldbinding"    : { s := expimage(e.field) || " = " || expimage(e.code, pp,prec["="])
                        bracket(s, pp, "=", precedence)
                      }
  "field"  : e.name
  "absolute_field" : "{" || expimage(e.field) || " at " || e.offset || "}"
```
Uses absolute_field 256, bracket 108b, constraint 53, constraints 132a, dots_sequent 60c, expimage 98a, field 53, fieldbinding 53, latent_patlabel 53, patlabel 53, prec 108b, sequent 53, and stringininterval 358b.

101b ⟨*cases for* expimage 99a⟩+≡ (98a) ◁101a 101c▷
```
  "binding_instance" : expimage(e.val) || " : " || expimage(e.type)
  "constype" : "constructor-type " || e.name
```
Uses binding_instance 75b, constructor 155, constype 155, and expimage 98a.

101c ⟨*cases for* expimage 99a⟩+≡ (98a) ◁101b 103a▷
```
  "list"     : commaseparate(maplist3(expimage, e, pp, prec[","]), ", " || pp.on)
  "set"      : commaseparate(maplist3(expimage, sort(e), pp, prec[","]), ", " || pp.on)
  "eqn"      : bracket(expimage(e.left, pp, prec["="]) || " " || e.op || " " || pp.on ||
                       expimage(e.right, pp, prec["="]), pp, "=", precedence)
  "table"    : {                     # standard normal form
      ⟨make s represent table e, but elide additive and multiplicative units 102a⟩
      pp.in || pp.be || s || pp.en || pp.ou
    }
  "string"   : e
  "literal"  : image(e.s)
  "integer"  : string(e)
  "null"     : image(e)
```
Uses bracket 108b, eqn 81c, expimage 98a, literal 43e, and prec 108b.

The old version printed in any old order, but the new version prints positive terms first, then negative terms, with the constant term last in its group. ßChange #23

102a ⟨*make* s *represent table* e*, but elide additive and multiplicative units* 102a⟩≡ (101c)

```
{
  s := ""; leadingsign := ""
  ⟨add positive terms to s 102b⟩
  leadingsign := " - " || pp.on
  ⟨add negative terms to s 102c⟩
  s := if s == "" then "0" else bracket(s, pp, "+", precedence)
}
```

Uses bracket 108b.

102b ⟨*add positive terms to* s 102b⟩≡ (102a)

```
every e[k := 1 ~=== key(e)] > 0 do {
    s ||:= leadingsign
    s ||:= (1 ~= abs(e[k])) || "*"    # print coefficient if not 1
    s ||:= expimage(k, pp, prec["+"])
    leadingsign := " + " || pp.on
}
if e[1] > 0 then s ||:= leadingsign || string(e[1])
```

Uses expimage 98a and prec 108b.

102c ⟨*add negative terms to* s 102c⟩≡ (102a)

```
every e[k := 1 ~=== key(e)] < 0 do {
    s ||:= leadingsign
    s ||:= (1 ~= abs(e[k])) || "*"    # print coefficient if not 1
    s ||:= expimage(k, pp, prec["+"])
}
if e[1] < 0 then s ||:= leadingsign || string(-e[1])
```

Uses expimage 98a and prec 108b.

102d ⟨*old make* s *represent table* e*, but elide additive and multiplicative units* 102d⟩≡

```
{
  # print constant term if nonzero
  if e[1] ~= 0 then { s := string(e[1]) ; leadingsign := " + " || pp.on }
  else s := leadingsign := ""
  # print every nonconstant term k
  every k := 1 ~=== key(e) do {
      s ||:= if e[k] < 0 then " - " else leadingsign
      leadingsign := " + " || pp.on
      s ||:= (1 ~= abs(e[k])) || "*"    # print coefficient if not 1
      s ||:= expimage(k, pp, prec["+"])
  }
  s := if s == "" then "0" else bracket(s, pp, "+", precedence)
}
```

Uses bracket 108b, constant 111b, expimage 98a, and prec 108b.

103a ⟨*cases for* `expimage` 99a⟩+≡ (98a) ◁101c 103b▷

```
  "Eorb"     : { s := expimage(e.x, pp, prec["Eorb"], "L")
                 t := expimage(e.y, pp, prec["Eorb"], "R")
                 bracket(s || " ORB " || pp.on || t, pp, "Eorb", precedence)
               }
  "Eand"     : { s := expimage(e.x, pp, prec["Eand"], "L")
                 t := expimage(e.y, pp, prec["Eand"], "R")
                 bracket(s || " AND " || t, pp, "Eand", precedence)
               }
```

Uses `bracket` 108b, `Eand` 79, `Eorb` 79, `expimage` 98a, and `prec` 108b.

103b ⟨*cases for* `expimage` 99a⟩+≡ (98a) ◁103a 103c▷

```
  "Eslice"   : { s := expimage(e.x, pp, prec["Eslice"])
                 s ||:= "[" || e.lo || ":" || (e.lo + e.n - 1) || "]"
                 bracket(s, pp, "Eslice", precedence)
               }
  "Eshift"   : { s := expimage(e.x, pp, prec["Eshift"])
                 s ||:= if e.n < 0 then (" >> " || -e.n) else (" << " || e.n)
                 bracket(s, pp, "Eshift", precedence)
               }
```

Uses `bracket` 108b, `Eshift` 79, `Eslice` 79, `expimage` 98a, and `prec` 108b.

103c ⟨*cases for* `expimage` 99a⟩+≡ (98a) ◁103b 103d▷

```
  "Enarrowu" : { s := expimage(e.x, pp, prec["Enarrowu"])
                 s ||:= "[" || e.n || "]"
                 bracket(s, pp, "Enarrowu", precedence)
               }
  "Enarrows" : { s := expimage(e.x, pp, prec["Enarrows"])
                 s ||:= "[" || e.n || "!]"
                 bracket(s, pp, "Enarrows", precedence)
               }
  "Ewiden"   : { s := expimage(e.x, pp, prec["Ewiden"])
                 s ||:= "!" || e.n
                 bracket(s, pp, "Ewiden", precedence)
               }
```

Uses `bracket` 108b, `Enarrows` 79, `Enarrowu` 79, `Ewiden` 79, `expimage` 98a, and `prec` 108b.

103d ⟨*cases for* `expimage` 99a⟩+≡ (98a) ◁103c 104a▷

```
  "Ediv"     : { s := expimage(e.x, pp, prec["Ediv"])
                 s ||:= " DIV " || e.n
                 bracket(s, pp, "Ediv", precedence)
               }
  "Emod"     : { s := expimage(e.x, pp, prec["Emod"])
                 s ||:= " MOD " || e.n
                 bracket(s, pp, "Emod", precedence)
               }
```

Uses `bracket` 108b, `Ediv` 79, `Emod` 79, `expimage` 98a, and `prec` 108b.

104a ⟨*cases for* expimage 99a⟩+≡ (98a) ◃103d 104b▹
```
  "Semit"    : commaseparate(maplist3(expimage, e.x, pp, prec[","]), ";" || pp.cn)
  "Stoken"   : { s := "EMIT@" || e.offset || "(" || pp.in ||
                        expimage(e.x, pp, prec[","]) || ", " ||
                        pp.on || e.n || ")" || pp.ou
                 bracket(pp.cn || s, pp, "app", precedence)
               }
```
Uses bracket 108b, expimage 98a, prec 108b, Semit 79, and Stoken 79.

104b ⟨*cases for* expimage 99a⟩+≡ (98a) ◃104a 104c▹
```
  "Efitsu"   : { s := "FITSU(" || expimage(e.x, pp, prec[","]) || ", " || pp.on || e.n||")"
                 bracket(s, pp, "app", precedence)
               }
  "Efitss"   : { s := "FITSS(" || expimage(e.x, pp, prec[","]) || ", " || pp.on ||e.n ||")"
                 bracket(s, pp, "app", precedence)
               }
```
Uses bracket 108b, Efitss 79, Efitsu 79, expimage 98a, and prec 108b.

104c ⟨*cases for* expimage 99a⟩+≡ (98a) ◃104b 104d▹
```
  "Epatlabel" : "LOCATION_OF(" || expimage(e.l, pp, prec[","]) || ")"
```
Uses Epatlabel 79, expimage 98a, and prec 108b.

104d ⟨*cases for* expimage 99a⟩+≡ (98a) ◃104c 104e▹
```
  "Eforce"   : { s := "FORCE(" || expimage(e.x, pp, prec[","]) || ")"
                 bracket(s, pp, "app", precedence)
               }
  "Eforceable":{ s := "FORCEABLE(" || expimage(e.x, pp, prec[","]) || ")"
                 bracket(s, pp, "app", precedence)
               }
```
Uses bracket 108b, Eforce 79, Eforceable 79, expimage 98a, and prec 108b.

104e ⟨*cases for* expimage 99a⟩+≡ (98a) ◃104d 104f▹
```
  "Epc"       : "FORCE(<program counter>)"
  "Epc_known" :" FORCEABLE(<program counter>)"
```
Uses Epc 79 and Epc_known 79.

104f ⟨*cases for* expimage 99a⟩+≡ (98a) ◃104e 104g▹
```
  "Enot"      : { s := "NOT " || expimage(e.x, pp, prec["NOT"])
                  bracket(s, pp, "NOT", precedence)
                }
```
Uses bracket 108b, Enot 79, expimage 98a, and prec 108b.

104g ⟨*cases for* expimage 99a⟩+≡ (98a) ◃104f 105a▹
```
  "Enosimp"   : expimage(e.x, pp, precedence, associativity)
```
Uses Enosimp 79 and expimage 98a.

105a ⟨*cases for* expimage 99a⟩+≡ (98a) ◁104g 105b▷
```
  "Sstmts"    : pp.in || pp.be || "{ " || pp.cn ||
                commaseparate(maplist3(expimage, e.x, pp), "; " || pp.cn) ||
                pp.ou || pp.cn || " }" || pp.en
```
Uses expimage 98a and Sstmts 79.

105b ⟨*cases for* expimage 99a⟩+≡ (98a) ◁105a 105c▷
```
  "Einstance" : { l := [];
                  every i := inputs_of(e.cons) do
                    put(l, expimage(e.argt[i.name], pp))
                  s := e.cons.name || "(" || commaseparate(l, ", " || pp.on) || ")"
                  bracket(s, pp, "app", precedence)
                }
  "Einstance_input" : {
                 s := expimage(e.x, pp, prec["."]) || "." || e.cons.name || "." || e.name
                 bracket(s, pp, ".", precedence)
               }
  "Einstance_tagged" : {
                 s := expimage(e.x, pp, prec["="]) || " IS " || e.cons.name
                 bracket(s, pp, "=", precedence)
               }
  "Ebinding_instance" : e.name || " : " || e.type.name
```
Uses bracket 108b, Ebinding_instance 79, Einstance 79, Einstance_input 79, Einstance_tagged 79, expimage 98a, inputs_of 162a, and prec 108b.

105c ⟨*cases for* expimage 99a⟩+≡ (98a) ◁105b 106a▷
```
  "Sif"      : { s := pp.cn || "if"
                 every s ||:=
                   pp.cn || ":: " || pp.in || expimage(!e.arms, pp, prec["low"]) || pp.ou
                 bracket(s || " " || pp.cn || "fi", pp, "app", precedence)
               }
  "Sguarded" : { s := expimage(e.guard, pp, prec["Sguarded"]) || " -> " ||
                      pp.in || pp.be || pp.cn ||
                      expimage(e.x,     pp, prec["Sguarded"]) ||
                      pp.en || pp.ou
                 bracket(s, pp, "Sguarded", precedence)
               }
```
Uses bracket 108b, expimage 98a, prec 108b, Sguarded 79, and Sif 79.

106a ⟨*cases for* expimage 99a⟩+≡ (98a) ◁105c 106b▷
```
  "Stagcase" : { s := "CASE " || expimage(e.x, pp) || " OF "
                 every c := kept_constructors(e.type) do
                   s ||:= pp.nl || "| " || c.name || " => " || pp.in || pp.in || pp.be ||
                           pp.cn || expimage(e.arms[c], pp) || pp.en || pp.ou || pp.ou
                 pp.be || pp.cn || s || pp.nl || "END (* " || expimage(e.x, pp) ||
                     " *)" || pp.en || pp.on
               }
```
Uses expimage 98a, kept_constructors 31d, and Stagcase 178a.

106b ⟨*cases for* expimage 99a⟩+≡ (98a) ◁106a 106c▷
```
  "Sfail"    : { s := "FAIL(" || image(if *pp.nl > 0 then escape_dollars(e.fmt) else e.fmt)
                  every a := e.a1 | e.a2 | e.a3 do
                    s ||:= ", " || pp.on || expimage(\a, pp, prec[","])
                  s := pp.in || pp.be || s || ")" || pp.en || pp.ou
                  bracket(s, pp, "app", precedence)
                }
  "Sepsilon" : "/* skip */"
```
Uses bracket 108b, escape_dollars 109b, expimage 98a, prec 108b, Sepsilon 79, and Sfail 182c.

106c ⟨*cases for* expimage 99a⟩+≡ (98a) ◁106b 106d▷
```
  "Efail"    : "[BAD EXP: " || e.msg || "]"
```
Uses Efail 79.

106d ⟨*cases for* expimage 99a⟩+≡ (98a) ◁106c 107a▷
```
  "Glines"    : commaseparate(maplist2(expimage, e.x, pp), "\n")
  "Gresynch"  : "#line " || e.line  || " " || image(\e.file | "generated-code")
  "Gblock"    : { s := "{ " || pp.be || pp.in || pp.cn
                  every s ||:= expimage(!e.decls, pp) || "; " || pp.nl
                  every s ||:= expimage(!e.stmts, pp) || "; " || pp.cn
                  s || pp.ou || pp.cn || "}" || pp.en
                }
  "Gdecl"     : { s := pp.be || expimage(e.type, pp) || " " || e.name
                  s ||:= " = " || pp.in || pp.cn || expimage(\e.init, pp) || pp.ou
                  s || pp.en
                }
```
Uses expimage 98a, file 345a, Gblock 281a, Gdecl 281a, Glines 281a, Gresynch 281a, and line 345a.

107a ⟨*cases for* expimage 99a⟩+≡ (98a) ◁106d 107b▷

```
  "Gcase"     : { s := "CASE " || expimage(e.x, pp) || " OF "
                  every s ||:= expimage(!e.arms, pp)
                  pp.be || pp.cn || s || pp.nl || "END (* CASE " || expimage(e.x, pp) ||
                    " *)" || pp.en || pp.on
                 }
  "Gcasearm"  : { s := pp.nl || "| " || pp.in
                  every i := 1 to *e.tags by 2 do {
                    if i > 1 then s ||:= ", " || pp.on
                    s ||:= if e.tags[i] + 1 = e.tags[i+1] then e.tags[i]
                           else e.tags[i] || ".." || (e.tags[i+1]-1)
                  }
                  s ||:= " => "  || pp.in || pp.cn || expimage(e.x, pp)
                  pp.be || s || pp.ou || pp.ou || pp.en
                }
  "Ginrange"  : bracket(pp.in || e.lo || " <= " || pp.on ||
                           expimage(e.x, pp, prec["<="]) || " < " || e.hi,
                        pp, "<", precedence)
```

Uses bracket 108b, expimage 98a, Gcase 282a, Gcasearm 282a, Ginrange 282a, and prec 108b.

107b ⟨*cases for* expimage 99a⟩+≡ (98a) ◁107a 107c▷

```
  "Gasgn"     : bracket(e.lhs || " := " || expimage(e.x, pp, prec[":="]),
                        pp, ":=", precedence)
  "Gsetname"  : e.lhs || " := " || image(e.name)
  "Gnomatch"  : "<NO MATCH>"
  "Tunsigned" : { s := "unsigned"
                  s ||:= " /* " || \e.width || " bits */"
                  s
                }
```

Uses bracket 108b, expimage 98a, Gasgn 282a, Gnomatch 282a, Gsetname 282a, prec 108b, and Tunsigned 282a.

107c ⟨*cases for* expimage 99a⟩+≡ (98a) ◁107b 108a▷

```
  "Eapp" : image(e.f) || "(" || pp.in || expimage(e.args, pp, prec[","]) || pp.ou || ")"
```

Uses Eapp 79, expimage 98a, and prec 108b.

108a ⟨*cases for* `expimage` 99a⟩+≡ (98a) ◁107c

```
"Eclosure" : "CLOSURE(" || pp.in || pp.be || pp.cn ||
               "type = " || e.ty || ", " || pp.cn ||
               "fun = " ||  e.fun || ", " || pp.cn ||
               "values = {" || expimage(e.values) || "}, " || pp.cn ||
               "addresses = {" || expimage(e.addresses) || "}" || pp.en || pp.ou || ")"
"Elambda" : pp.be || "(FN " || commaseparate(e.formals) || " => " || pp.in || pp.on ||
               expimage(e.body, pp, prec[","]) || ")" || pp.ou || pp.en
"Eclosure_loc"  : "CL->loc"
"Eclosure_addr" : "CL->a" || e.n
"Eclosure_val"  : "CL->v" || e.n
```

Uses `Eclosure` 207, `Eclosure_addr` 79, `Eclosure_loc` 79, `Eclosure_val` 79, `Elambda` 206, `expimage` 98a, and `prec` 108b.

108b ⟨*** 79⟩+≡ ◁100b 109b▷

```
global prec, assoc
procedure bracket(s, pp, op, p, a)
  /a := "L"
  return pp.be || (if prec[op] > p | (prec[op] = p & assoc[p] == a) then s
                   else "(" || s || ")") || pp.en
end
```

Defines:
`assoc`, used in chunk 109a.
`bracket`, used in chunks 99–107.
`prec`, used in chunks 99–109.

109a ⟨*initialize* prec *and* assoc 109a⟩≡ (98a)
```
  ops := ["N", ["low"],
          "L", ["|", "pattern"],
          "L", [";"],
          "L", ["&", "sequent", "patlabel"],
          "R", [":="],
          "L", [","],
          "N", ["Sguarded"],
          "N", ["="],
          "L", ["ORB", "Eorb"],
          "L", ["AND", "Eand"],
          "L", ["<", "<="],
          "L", ["+"],
          "N", ["Emod", "Ediv", "*"],
          "N", ["Eshift"],
          "N", ["NOT"],
          "L", ["Eslice", "Enarrowu", "Enarrows", "Ewiden", "."],
          "N", ["app"],             # function application
          "N", ["high"]
         ]
  prec := table([])  # missed lookups break arithmetic comparisons
  assoc := table()
  every i := 1 to *ops by 2 do {
    every prec[!ops[i+1]] := i
    assoc[i] := ops[i]
  }
```
Uses assoc 108b, Eand 79, Ediv 79, Emod 79, Enarrows 79, Enarrowu 79, Eorb 79, Eshift 79, Eslice 79, Ewiden 79, patlabel 53, pattern 53, prec 108b, sequent 53, and Sguarded 79.

109b ⟨* 79⟩+≡ ◁108b 110a▷
```
  procedure escape_dollars(s)
    r := ""
    s ? {
      while r ||:= tab(upto('$')) do { ="$"; r ||:= "$$" }
      return r || tab(0)
    }
  end
```
Defines:
escape_dollars, used in chunks 106b, 318a, and 330b.

5.6 Semantic functions

binop implements the semantics of binary operators. It modifies its inputs and returns e1, so it should be used only during parsing.

110a ⟨* 79⟩+≡ ◁109b 110b▷

```
procedure binop(e1, op, e2)
  e1 := term2table(e1)
  e2 := term2table(e2)
  case op of {
    "+" : every v := key(e2) do e1[v] +:= e2[v]
    "-" : every v := key(e2) do e1[v] -:= e2[v]
    "*" : if n2 := constant(e2) then {
              every !e1 *:= n2
          } else if n1 := constant(e1) then {
              every !e2 *:= n1
              e1 := e2
          } else error("multiplication must be by constants only")
    "/" : error("division is not permitted in this language")
  }
  return e1
end
```

Defines:
binop, used in chunks 47c, 185a, 252, and 258a.
Uses constant 111b and term2table 110b.

110b ⟨* 79⟩+≡ ◁110a 111a▷

```
procedure term2table(x)
  return case type(x) of {
    "table"   : x
    "integer" : 1(e := table(0), e[1] := x)
    default   : 1(e := table(0), e[x] := 1)
  }
end
```

Defines:
term2table, used in chunks 110a, 132a, 139b, and 140c.

addconst is nondestructive.

111a ⟨* 79⟩+≡ ◁110b 111b▷

```
procedure addconst(e, n)
  if n = 0 then return e
  if type(e) == "table" then
    t := copy(e)
  else {
    t := table(0)
    if type(t) == "integer" then t[1] := e else t[e] := 1
  }
  t[1] +:= n
  return t
end
```

Defines:
addconst, used in chunks 315g and 328c.

It is necessary to identify a constant in normal form.

111b ⟨* 79⟩+≡ ◁111a 111c▷

```
procedure constant(e)
  if type(c := untable(e)) == "integer" then return c
end
```

Defines:
constant, used in chunks 75a, 102d, 110a, 113a, 140b, 208d, 258b, 271a, 313a, 327a, 335–37, and 349a.
Uses untable 111c.

111c ⟨* 79⟩+≡ ◁111b 112a▷

```
procedure untable(e)
  return if type(e) ~== "table" then e else {
    n := 0
    every k := key(e) & e[k] ~= 0 do n +:= 1
    case n of {
      0       : 0
      1       : if k := key(e) & e[k] = 1 then k
                else if e[1] ~= 0 then e[1]
                else e
      default : e
    }
  }
end
```

Defines:
untable, used in chunks 111b, 140b, 172, and 211a.

5.7 Creation

`mkslice` is used only to create slices from actual specifications, in which the bit numbering could change. ßChange #26

112a ⟨* 79⟩+≡ ◃111c 112c▹
```
  procedure mkslice(x, lo, hi)
    local n # size of range
    if lo < 0 then error("Can't take negative bit slice!!")
    if hi <= lo then error(expimage(x), "[", lo, ":", hi - 1, "] has no bits!")
    if hi > bitsizeof(x) then
      error(if type(f := symtab[x]) == "field" then "Field " || f.name
            else "Host machine",
               " has only ", bitsizeof(x), " bits")
    bit_numbering_used := 1
    ⟨if needed, flip lo and hi 112b⟩
    return Eslice(x, lo, hi - lo)
  end
```
Defines:
`mkslice`, used in chunk 48b.
Uses `bit_numbering_used` 34b, `bitsizeof` 112c, `Eslice` 79, `expimage` 98a, `field` 53, and `symtab` 30a 353a.

112b ⟨*if needed, flip* `lo` *and* `hi` 112b⟩≡ (112a)
```
  if /bit_zero_is_lsb then {
    lo := bitsizeof(x) - hi
    hi := bitsizeof(x) - lo
  }
```
Uses `bitsizeof` 112c and `bit_zero_is_lsb` 30a.

112c ⟨* 79⟩+≡ ◃112a 112d▹
```
  procedure bitsizeof(f)
    return case type(symtab[f]) of {
        "field" : fwidth(symtab[f])
        default : wordsize
      }
  end
```
Defines:
`bitsizeof`, used in chunks 112 and 149e.
Uses `field` 53, `fwidth` 112d, `symtab` 30a 353a, and `wordsize` 36c.

112d ⟨* 79⟩+≡ ◃112c 113a▹
```
  procedure fwidth(f)
    return f.hi - f.lo
  end
```
Defines:
`fwidth`, used in chunks 44b, 48b, 59, 60a, 68c, 94b, 112c, 182b, 187, 191c, 193–95, 198b, 202d, 235, 237a, 270a, 273b, 278b, 285c, 301b, 321a, 332c, 335–37, 341d, 343b, and 357b.

113a ⟨* 79⟩+≡ ◁112d 113b▷

```
procedure mkshift(x, n)
  return if n = 0 then x else
         ishift(\simp & constant(x), n) | Eshift(x, n)
end
```

Uses `constant` 111b and `Eshift` 79.

113b ⟨* 79⟩+≡ ◁113a 113c▷

```
record bogus(e) # ditch invalid field name complaint
```

Defines:
`bogus`, used in chunks 22, 24a, 32a, and 347a.
Uses `field` 53.

`flatten` is used to flatten the argument lists of binary associative operators like `Eorb` and `Eand`.

113c ⟨* 79⟩+≡ ◁113b 113d▷

```
procedure flatten(e, ty, l)
  /l := []
  return if type(e) == ty then
    flatten(e.x, ty, flatten(e.y, ty, l))
  else
    push(l, e)
end
```

Defines:
`flatten`, used in chunks 127a and 210a.

Unflatten takes the list and the operator and turns it back into a tree.

113d ⟨* 79⟩+≡ ◁113c

```
procedure unflatten(l, rator, zero)
  if *l = 0 then return zero
  else {
    x := l[1]
    every x := rator(x, l[2 to *l])
  }
  return x
end
```

Defines:
`unflatten`, used in chunk 210a.

5.8 Visiting every term of an expression

`expwalk` suspends *every* result of applying `f` to each node in a bottom-up tree walk.

114a ⟨* 114a⟩≡ 117a▷

```
procedure expwalk(e, f, closure[])
  suspend do_expwalk(e, f, closure)
end

procedure do_expwalk(e, f, closure)
  suspend case type(e) of {
          ⟨cases for do_expwalk 114b⟩
          default : impossible("expression type in expression walking")
         } | f ! ([e] ||| closure)

end
```

Defines:
`do_expwalk`, used in chunks 82a and 114–16.
`expwalk`, used in chunks 47b, 82b, 144b, and 315a.

114b ⟨*cases for* `do_expwalk` 114b⟩≡ (114a) 114c▷

```
"pattern"          : do_expwalk(!e.disjuncts, f, closure)
"disjunct"         : do_expwalk(!\e.conditions | !e.sequents, f, closure)
"adisjunct"        : do_expwalk(!\e.conditions | !e.aconstraints, f, closure)
"sequent"          : do_expwalk(!e.constraints, f, closure)
"dots_sequent"     : &fail
"patlabel"         : &fail
```

Uses `adisjunct` 256, `constraints` 132a, `disjunct` 53, `do_expwalk` 114a, `dots_sequent` 60c, `fail` 396, `patlabel` 53, `pattern` 53, and `sequent` 53.

`e.instance` can be `null` when a pattern label is "vanishing," and we wouldn't want to walk a null.

114c ⟨*cases for* `do_expwalk` 114b⟩+≡ (114a) ◁114b 114d▷

```
"latent_patlabel" : do_expwalk((vanishing_latent_patlabel ~=== e).instance, f, closure)
"constraint"      : do_expwalk(e.field, f, closure)
"fieldbinding"    : do_expwalk(e.code | e.field, f, closure)
"absolute_field"  : do_expwalk(e.field, f, closure)
"field"           : &fail
```

Uses `absolute_field` 256, `constraint` 53, `do_expwalk` 114a, `fail` 396, `field` 53, `fieldbinding` 53, `latent_patlabel` 53, and `vanishing_latent_patlabel` 30a.

114d ⟨*cases for* `do_expwalk` 114b⟩+≡ (114a) ◁114c 115a▷

```
"list" | "set"    : do_expwalk(!e, f, closure)
"eqn"             : do_expwalk(e.left | e.right, f, closure)
"table"           : do_expwalk(key(e), f, closure)
```

Uses `do_expwalk` 114a and `eqn` 81c.

115a ⟨*cases for* do_expwalk 114b⟩+≡ (114a) ◁114d 115b▷

```
  "Eorb" | "Eand"   : do_expwalk(e.x | e.y, f, closure)
  "Eslice" | "Eshift" | "Enarrowu" | "Enarrows" | "Ewiden" |
  "Ediv" | "Emod" | "Semit" | "Stoken" | "Efitsu" | "Efitss" |
    "Einstance_input" | "Einstance_tagged" :
                      do_expwalk(e.x, f, closure)
  "Epatlabel"       : do_expwalk(e.l, f, closure)
  "Eforce" | "Eforceable" | "Enot" | "Enosimp" | "Sstmts" :
                      do_expwalk(e.x, f, closure)
  "Epc" | "Epc_known" : &fail
  "Einstance"       : do_expwalk(!e.argt, f, closure)
  "Ebinding_instance" : do_expwalk(e.name, f, closure)
  "Eapp"            : do_expwalk(!e.args, f, closure)
  "Eclosure_loc"  : &fail
  "Eclosure_addr" : &fail
  "Eclosure_val"  : &fail
```

Uses do_expwalk 114a, Eand 79, Eapp 79, Ebinding_instance 79, Eclosure_addr 79, Eclosure_loc 79, Eclosure_val 79, Ediv 79, Efitss 79, Efitsu 79, Eforce 79, Eforceable 79, Einstance 79, Einstance_input 79, Einstance_tagged 79, Emod 79, Enarrows 79, Enarrowu 79, Enosimp 79, Enot 79, Eorb 79, Epatlabel 79, Epc 79, Epc_known 79, Eshift 79, Eslice 79, Ewiden 79, fail 396, Semit 79, Sstmts 79, and Stoken 79.

115b ⟨*cases for* do_expwalk 114b⟩+≡ (114a) ◁115a 115c▷

```
  "Sif"             : do_expwalk(!e.arms, f, closure)
  "Sguarded"        : do_expwalk(e.guard | e.x, f, closure)
  "Sepsilon"        : &fail
  "Stagcase"        : do_expwalk(e.x | !e.arms, f, closure)
  "Sfail"           : &fail
  "integer" | "string" | "literal" : &fail
  "Efail"           : &fail
  "Sclosure"        : do_expwalk(e.disjunct | \e.conditions | \e.creation, f, closure)
```

Uses disjunct 53, do_expwalk 114a, Efail 79, fail 396, literal 43e, Sclosure 185f, Sepsilon 79, Sfail 182c, Sguarded 79, Sif 79, and Stagcase 178a.

115c ⟨*cases for* do_expwalk 114b⟩+≡ (114a) ◁115b 116a▷

```
  "balance"         : do_expwalk(e.left | e.right, f, closure)
  "balitem"         : do_expwalk(e.v | e.value, f, closure)
```

Uses balance 137a, balitem 137a, do_expwalk 114a, and value 132a.

116a ⟨*cases for* `do_expwalk` 114b⟩+≡ (114a) ◁115c 116b▷

```
"Glines"          : &fail
"Gresynch"        : &fail
"Gblock"          : do_expwalk(!e.decls | !e.stmts, f, closure)
"Gdecl"           : do_expwalk(e.init, f, closure)
"Gcase"           : do_expwalk(e.x | !e.arms, f, closure)
"Gcasearm"        : do_expwalk(e.x, f, closure)
"Ginrange"        : do_expwalk(e.x, f, closure)
"Gsetname"        : &fail
"Gnomatch"        : &fail
"Gasgn"           : do_expwalk(e.x, f, closure)
"Tunsigned"       : &fail
```

Uses `do_expwalk` 114a, `fail` 396, `Gasgn` 282a, `Gblock` 281a, `Gcase` 282a, `Gcasearm` 282a, `Gdecl` 281a, `Ginrange` 282a, `Glines` 281a, `Gnomatch` 282a, `Gresynch` 281a, `Gsetname` 282a, and `Tunsigned` 282a.

116b ⟨*cases for* `do_expwalk` 114b⟩+≡ (114a) ◁116a 116c▷

```
"inject"          : do_expwalk(\e.pattern | \e.integer, f, closure)
```

Uses `do_expwalk` 114a, `inject` 27, and `pattern` 53.

116c ⟨*cases for* `do_expwalk` 114b⟩+≡ (114a) ◁116b

```
"Eclosure" : do_expwalk(!e.values | !e.addresses, f, closure)
"Elambda"  : do_expwalk(e.body, f, closure)
```

Uses `do_expwalk` 114a, `Eclosure` 207, and `Elambda` 206.

5.9 Generalized substitution

117a ⟨* 114a⟩+≡ ◁114a 117b▷

```
procedure gsubst(e, f, closure[])
  return do_gsubst(e, f, closure)
end

procedure do_gsubst(e, f, closure)
  local args
  return f ! ([e] ||| closure) |
         case type(e) of {
           ⟨cases for do_gsubst 118a⟩
           default : impossible("expression type in generalized substitution")
         }
end

procedure do_gsubst_children(e, f, closure)
  local args
  return case type(e) of {
           ⟨cases for do_gsubst 118a⟩
           default : impossible("expression type in generalized substitution")
         }
end
```

Defines:
`do_gsubst`, used in chunks 86–88 and 118–23.
`gsubst`, used in chunks 75a, 144b, 145a, 170b, 171b, 174a, 175, 184, 248, 257, and 300a.

117b ⟨* 114a⟩+≡ ◁117a 118b▷

```
procedure maplistn(f, l, closure)
  local result, x
  result := []
  every x := !l do
    put(result, f ! ([x] ||| closure) | fail)
  return result
end
```

Defines:
`maplistn`, used in chunks 118–20.
Uses `fail` 396.

118a ⟨*cases for* `do_gsubst` 118a⟩≡ (117a) 119a▷

```
"pattern"  : if l := maplistn(do_gsubst, e.disjuncts, [f, closure]) &
                lists_match(l, e.disjuncts) then e else {
                  ll := [];
                  every d := !l do
                    if not member(\d.conditions, 0) then put(ll, d)
                pattern(ll, e.name)
             }
"disjunct" : { fclosure := [f, closure]
               c := do_gsubst_conditions(e.conditions, f, closure)
               s := []  # this code vanishes dead latent pattern labels
               every x := do_gsubst(!e.sequents, f, closure) do
                 put(s, vanishing_latent_patlabel ~=== x)
               if lists_match(s, e.sequents) & c === e.conditions then
                 e
               else
                 disjunct(s, e.name, c)
             }
```

Uses `disjunct` 53, `do_gsubst` 117a, `do_gsubst_conditions` 118b, `lists_match` 124a, `maplistn` 117b, `pattern` 53, and `vanishing_latent_patlabel` 30a.

118b ⟨* 114a⟩+≡ ◁117b 123b▷

```
procedure do_gsubst_conditions(e, f, closure)
  if \(c := e) then {
     c := set();
     every insert_condition(c, do_gsubst(!e, f, closure))
     if sets_match(c, e) then c := e
  }
  return c
end

procedure gsubst_conditions(e, f, closure[])
  return do_gsubst_conditions(e, f, closure)
end
```

Defines:
`do_gsubst_conditions`, used in chunk 118a.
Uses `do_gsubst` 117a, `insert_condition` 123c, and `sets_match` 124b.

119a ⟨*cases for* `do_gsubst` 118a⟩+≡ (117a) ◁118a 119b▷

```
  "adisjunct" : {
                  if \(c := e.conditions) then {
                    c := set();
                    every insert_condition(c, do_gsubst(!e.conditions, f, closure))
                    if sets_match(c, e.conditions) then c := e.conditions
                  }
                  if \(p := e.patlabelbindings) then
                    p := do_gsubst_values_in_table(p, f, closure)
                  s := maplistn(do_gsubst, e.aconstraints, [f, closure])
                  if lists_match(s, e.aconstraints) & c === e.conditions &
                     p === e.patlabelbindings
                  then
                    e
                  else
                    adisjunct(s, e.name, c, e.length, p)
                }
  "sequent"  : if l := maplistn(do_gsubst, e.constraints, [f, closure]) &
                  lists_match(l, e.constraints) then e else sequent(l, e.class)
```

Uses `adisjunct` 256, `constraints` 132a, `do_gsubst` 117a, `do_gsubst_values_in_table` 88a, `insert_condition` 123c, `lists_match` 124a, `maplistn` 117b, `sequent` 53, and `sets_match` 124b.

119b ⟨*cases for* `do_gsubst` 118a⟩+≡ (117a) ◁119a 120a▷

```
  "patlabel"        : e
  "latent_patlabel" : if (x := do_gsubst(e.instance, f, closure)) === e.instance then e
                      else latent_patlabel(x)
  "dots_sequent" : e
  "constraint"   : e
  "fieldbinding" : if (x := do_gsubst(e.code, f, closure)) === e.code then e
                   else fieldbinding(e.field, x)
  "absolute_field" : e
```

Uses `absolute_field` 256, `constraint` 53, `do_gsubst` 117a, `dots_sequent` 60c, `field` 53, `fieldbinding` 53, `latent_patlabel` 53, and `patlabel` 53.

120a ⟨*cases for* do_gsubst 118a⟩+≡ (117a) ◁119b 120b▷
```
"list"       : if lists_match(l := maplistn(do_gsubst, e, [f, closure]), e) then e else l
"eqn"        : if l := do_gsubst(e.left, f, closure) &
                  r := do_gsubst(e.right, f, closure) &
                  l === e.left & r === e.right then e else eqn(l, e.op, r)
"table"      : if (t := do_gsubst_sumprod_table(e, f, closure)) === e then e else t
"set"        : {s := set()
                every insert(s, do_gsubst(!e, f, closure))
                if x := !s & not member(e, x) then s else e
               }
"string"     : e
"literal"    : e
"integer"    : e
"null"       : e
```
Uses do_gsubst 117a, do_gsubst_sumprod_table 123b, eqn 81c, lists_match 124a, literal 43e, and maplistn 117b.

120b ⟨*cases for* do_gsubst 118a⟩+≡ (117a) ◁120a 121a▷
```
"Eorb"       : if x := do_gsubst(e.x, f, closure) & y := do_gsubst(e.y, f, closure) &
                  x === e.x & y === e.y then e else Eorb(x, y)
"Eand"       : if x := do_gsubst(e.x, f, closure) & y := do_gsubst(e.y, f, closure) &
                  x === e.x & y === e.y then e else Eand(x, y)
"Eslice"     : if (x := do_gsubst(e.x, f, closure)) === e.x then e
               else Eslice(x, e.lo, e.n)
"Eshift"     : if (x := do_gsubst(e.x, f, closure)) === e.x then e else Eshift(x, e.n)
"Enarrowu"   : if (x := do_gsubst(e.x, f, closure)) === e.x then e else Enarrowu(x, e.n)
"Enarrows"   : if (x := do_gsubst(e.x, f, closure)) === e.x then e else Enarrows(x, e.n)
"Ewiden"     : if (x := do_gsubst(e.x, f, closure)) === e.x then e else Ewiden(x, e.n)
"Ediv"       : if (x := do_gsubst(e.x, f, closure)) === e.x then e else Ediv(x, e.n)
"Emod"       : if (x := do_gsubst(e.x, f, closure)) === e.x then e else Emod(x, e.n)
"Efitsu"     : if (x := do_gsubst(e.x, f, closure)) === e.x then e else Efitsu(x, e.n)
"Efitss"     : if (x := do_gsubst(e.x, f, closure)) === e.x then e else Efitss(x, e.n)
"Epatlabel"  : if (l := do_gsubst(e.l, f, closure)) === e.l then e else Epatlabel(l)
"Eforce"     : if (x := do_gsubst(e.x, f, closure)) === e.x then e else Eforce(x)
"Eforceable" : if (x := do_gsubst(e.x, f, closure)) === e.x then e else Eforceable(x)
"Epc" | "Epc_known" : e
"Enot"       : if (x := do_gsubst(e.x, f, closure)) === e.x then e else Enot(x)
"Enosimp"    : if (x := do_gsubst(e.x, f, closure)) === e.x then e else Enosimp(x)
"Sstmts"     : if (x := do_gsubst(e.x, f, closure)) === e.x then e else Sstmts(x)
"Semit"      : if (x := do_gsubst(e.x, f, closure)) === e.x then e else Semit(x)
"Stoken"     : if (x := do_gsubst(e.x, f, closure)) === e.x then e
               else Stoken(x, e.n, e.offset)
```
Uses do_gsubst 117a, Eand 79, Ediv 79, Efitss 79, Efitsu 79, Eforce 79, Eforceable 79, Emod 79, Enarrows 79, Enarrowu 79, Enosimp 79, Enot 79, Eorb 79, Epatlabel 79, Epc 79, Epc_known 79, Eshift 79, Eslice 79, Ewiden 79, Semit 79, Sstmts 79, and Stoken 79.

121a ⟨*cases for* `do_gsubst` 118a⟩+≡ (117a) ◁120b 121b▷

```
"Einstance"  :  e # suppress warning --- will confuse users
                  # --- and we do this with eliminate_instances now
                  # { warning("escaping Einstance: ", expimage(e)); e}
"Ebinding_instance" : e
"Einstance_input" :
  if (x := do_gsubst(e.x, f, closure)) === e.x then e
  else Einstance_input(x, e.cons, e.name)
"Einstance_tagged" :
  if (x := do_gsubst(e.x, f, closure)) === e.x then e
  else Einstance_tagged(x, e.cons, e.uid)
```

Uses `do_gsubst` 117a, `Ebinding_instance` 79, `Einstance` 79, `Einstance_input` 79, `Einstance_tagged` 79, `eliminate_instances` 174a, and `expimage` 98a.

121b ⟨*cases for* `do_gsubst` 118a⟩+≡ (117a) ◁121a 121c▷

```
"Eapp"  :
  if (args := do_gsubst(e.args, f, closure)) === e.args then e else Eapp(e.f, args)
"Eclosure_loc"  : e
"Eclosure_addr" : e
"Eclosure_val"  : e
```

Uses `do_gsubst` 117a, `Eapp` 79, `Eclosure_addr` 79, `Eclosure_loc` 79, and `Eclosure_val` 79.

121c ⟨*cases for* `do_gsubst` 118a⟩+≡ (117a) ◁121b 122a▷

```
"Sif"      : if (arms := do_gsubst(e.arms, f, closure)) === e.arms then e
             else Sif(arms)
"Sguarded" : if (guard := do_gsubst(e.guard, f, closure),
                 x := do_gsubst(e.x, f, closure),
                 guard === e.guard, x === e.x) then e
             else Sguarded(guard, x)
"Sepsilon" : e
"Sfail" : e
"Stagcase" : if (arms := do_gsubst_vals(e.arms, f, closure),
                 x := do_gsubst(e.x, f, closure),
                 arms === e.arms, x === e.x) then e
             else {
write(&errout, "TAGCASE: ", expimage(e.x))
write(&errout, "TAGCASE: ", expimage(e.arms))
write(&errout, "TAGCASE: ", expimage(x))
write(&errout, "TAGCASE: ", expimage(arms))
             Stagcase(x, e.type, arms)
}
"Efail"    : e
```

Uses `do_gsubst` 117a, `Efail` 79, `expimage` 98a, `Sepsilon` 79, `Sfail` 182c, `Sguarded` 79, `Sif` 79, and `Stagcase` 178a.

122a ⟨*cases for* do_gsubst 118a⟩+≡ (117a) ◁121c 122b▷
```
  "balance"  : if l := do_gsubst(e.left, f, closure) &
                  r := do_gsubst(e.right, f, closure) &
                  l === e.left & r === e.right then e else balance(l, r)
  "balitem"  : if l := do_gsubst(e.v, f, closure) &
                  r := do_gsubst(e.value, f, closure) &
                  l === e.v & r === e.value then e else balitem(l, r)
```
Uses balance 137a, balitem 137a, do_gsubst 117a, and value 132a.

122b ⟨*cases for* do_gsubst 118a⟩+≡ (117a) ◁122a 122c▷
```
  "Glines"    : e
  "Gresynch"  : e
  "Gblock"    : if (decls := do_gsubst(e.decls, f, closure),
                    stmts := do_gsubst(e.stmts, f, closure),
                    decls === e.decls, stmts === e.stmts) then e
                else Gblock(decls, stmts)
  "Gdecl"     : if (init := do_gsubst(e.init, f, closure)) === e.init then e
                else Gdecl(e.name, e.type, init)
  "Gcase"     : if (x := do_gsubst(e.x, f, closure), arms := do_gsubst(e.arms, f, closure),
                    x === e.x, arms === e.arms) then e
                else Gcase(x, arms)
  "Gcasearm"  : if (x := do_gsubst(e.x, f, closure)) === e.x then e
                else Gcasearm(e.tags, x)
  "Ginrange"  : if (x := do_gsubst(e.x, f, closure)) === e.x then e
                else Ginrange(e.lo, x, e.hi)
  "Gsetname"  : e
  "Gnomatch"  : e
  "Gasgn"     : if (x := do_gsubst(e.x, f, closure)) === e.x then e
                else Gasgn(e.lhs, x)
  "Tunsigned" : e
```
Uses do_gsubst 117a, Gasgn 282a, Gblock 281a, Gcase 282a, Gcasearm 282a, Gdecl 281a, Ginrange 282a, Glines 281a, Gnomatch 282a, Gresynch 281a, Gsetname 282a, and Tunsigned 282a.

122c ⟨*cases for* do_gsubst 118a⟩+≡ (117a) ◁122b 123a▷
```
  "inject"    : if (p := do_gsubst(\e.pattern, f, closure) | &null,
                    i := do_gsubst(\e.integer, f, closure) | &null,
                    p === e.pattern, i === e.integer) then e
                else inject(p, i, e.consop)
```
Uses do_gsubst 117a, inject 27, and pattern 53.

123a ⟨*cases for* `do_gsubst` 118a⟩+≡ (117a) ◁122c

```
"Eclosure" : if (v := do_gsubst(e.values, f, closure),
                 a := do_gsubst(e.addresses, f, closure),
                 v === e.values, a === e.addresses)
             then e
             else Eclosure(e.ty, e.fun, v, a)
"Elambda"  : if (x := do_gsubst(e.body, f, closure)) === e.body then e
             else Elambda(e.formals, x)
```

Uses `do_gsubst` 117a, `Eclosure` 207, and `Elambda` 206.

Making a new "sum of products" table involves substituting all the keys. The only trick comes in when one of the new keys is itself a table. In that case, we have a sum of terms in which one of the terms is itself a sum of terms, and by distributing the outer coefficient over the inner terms, we can wind up with a single sum (that is, a single table). Using `+:=` to assign the coefficients makes it all work like magic, even when substitution causes duplicate keys.

123b ⟨* 114a⟩+≡ ◁118b 123c▷

```
procedure do_gsubst_sumprod_table(e, f, closure)
  t := table(0)
  every k := key(e) & kk := do_gsubst(k, f, closure) do
    add_to_table(t, kk, e[k])
  every k := key(t) & t[k] = 0 do delete(t, k) # delete zeroes
  return if tables_match(e, t) then e else t
end
```

Defines:
`do_gsubst_sumprod_table`, used in chunk 120a.
Uses `add_to_table` 123d, `do_gsubst` 117a, `tables_match` 124c, and `zeroes` 132a.

123c ⟨* 114a⟩+≡ ◁123b 123d▷

```
procedure insert_condition(s, c)
  if c ~=== 1 & not exps_eq(c, !s) then insert(s, c)
  return s
end
```

Defines:
`insert_condition`, used in chunks 68c, 78a, 118b, 119a, 189c, 191a, 192b, 252, and 287b.
Uses `exps_eq` 82c.

123d ⟨* 114a⟩+≡ ◁123c 124a▷

```
procedure add_to_table(t, k, multiplier)
  case type(k) of {
    "table"   : every v := key(k) do t[v] +:= k[v] * multiplier
    "integer" : t[1] +:= k * multiplier
    default   : t[k] +:= multiplier
  }
  return t
end
```

Defines:
`add_to_table`, used in chunks 92d and 123b.

124a ⟨* 114a⟩+≡ ◃123d 124b▹
```
procedure lists_match(l1, l2)
  if n := *l1 = *l2 then
    if l1[i := 1 to n] ~=== l2[i] then fail
    else return l2
  else fail
end
```
Defines:
lists_match, used in chunks 90, 91, 93b, and 118–20.
Uses fail 396.

124b ⟨* 114a⟩+≡ ◃124a 124c▹
```
procedure sets_match(s1, s2)
  if n := *s1 = *s2 then {
    s := set()
    every insert(s, expimage(!s1 | !s2))
    if *s = n then return s1 else fail
  } else fail
end
```
Defines:
sets_match, used in chunks 92a, 118b, and 119a.
Uses expimage 98a and fail 396.

124c ⟨* 114a⟩+≡ ◃124b
```
procedure tables_match(t1, t2)
  if k := key(t1 | t2) & t1[k] ~= t2[k] then fail else return t1
end
```
Defines:
tables_match, used in chunks 92d and 123b.
Uses fail 396.

5.10 Mapping expressions to postfix strings

The idea is to get a postfix, executable representation of function bodies. As with `expuniq`, we try to ensure two representations of the same function map to the same string.

125a ⟨* 125a⟩≡ 126b ▷

```
procedure expps(e)
  local leadingsign, prefix
  static psopstab
  initial {⟨init 125c⟩}
  return case type(e) of {
    ⟨cases for expps 125b⟩
    default    : &fail
  } | impossible("postfix expression ", image(e), " : ", type(e))
end
```

Defines:
`expps`, used in chunks 125–30 and 215b.
Uses `fail` 396.

125b ⟨*cases for* `expps` 125b⟩≡ (125a) 125d ▷

```
"list"     : "[" || commaseparate(maplist(expps, e), " ") || "]"
"set"      : "[" || commaseparate(sort(maplist(expps, sort(e))), ",") || "] set"
"eqn"      : expps(e.left) || " " || expps(e.right) || " " || \psopstab[e.op]
```

Uses `eqn` 81c and `expps` 125a.

125c ⟨*init* 125c⟩≡ (125a)

```
psopstab := table()
psopstab["<"]  := "lt"
psopstab["<="] := "le"
psopstab[">"]  := "gt"
psopstab[">="] := "ge"
psopstab["!="] := "ne"
psopstab["="]  := "eq"
```

125d ⟨*cases for* `expps` 125b⟩+≡ (125a) ◁125b 127a ▷

```
"table"    : {                      # standard normal form
    ⟨make s represent table e 126a⟩
    s
  }
"string"   : e
"literal"  : image(e.s) || " lit"
"integer"  : string(e)
"null"     : "null"
```

Uses `literal` 43e.

Be sure to omit multiplicative and additive units. Otherwise, it doesn't matter.

126a ⟨*make* `s` *represent table* `e` 126a⟩≡ (125d)

```
{
  s := ""; firstterm := 1
  t := table(0)
  l := []
  every e[k := key(e)] do {
    put(l, x := expps(k))
    t[x] +:= e[k]
  }
  l := sort(l)
  every x := !l & t[x] ~= 0 do {
    if \firstterm then
      s ||:= psx(x, t[x])
    else
      s ||:= " " || psabs(x, t[x]) || " " || if t[x] > 0 then "add" else "sub"
    firstterm := &null
  }
  if s == "" then s := "0"
}
```

Uses `expps` 125a, `psabs` 126b, and `psx` 126b.

126b ⟨* 125a⟩+≡ ◁125a 129a▷

```
procedure psabs(x, mul)
  return if x == "1" then mul
         else if abs(mul) = 1 then x
         else x || " " || abs(mul) || " mul"
end
procedure psx(x, mul)
  return if x == "1" then mul
         else case mul of {
            1 : x
           -1 : x || " neg"
           default : x || " " || mul || " mul"
         }
end
```

Defines:
`psabs`, used in chunk 126a.
`psx`, used in chunk 126a.

127a ⟨*cases for* expps 125b⟩+≡ (125a) ◁125d 127b▷
```
"Eorb"     : { l := sort(maplist(expps, flatten(e, type(e))))
               s := l[1]
               every s ||:= " " || l[2 to *l] || " orb"
               s
             }
"Eand"     : { l := sort(maplist(expps, flatten(e, type(e))))
               s := l[1]
               every s ||:= " " || l[2 to *l] || " and"
               s
             }
```
Uses Eand 79, Eorb 79, expps 125a, and flatten 113c.

127b ⟨*cases for* expps 125b⟩+≡ (125a) ◁127a 127c▷
```
"Eslice"   : expps(e.x) || " " || e.lo || " " || (e.lo + e.n - 1) || " bitslice"
"Eshift"   : expps(e.x) || " " || e.n || " bitshift"
```
Uses Eshift 79, Eslice 79, and expps 125a.

127c ⟨*cases for* expps 125b⟩+≡ (125a) ◁127b 127d▷
```
"Enarrowu" : expps(e.x) || " " || e.n || " narrowu"
"Enarrows" : expps(e.x) || " " || e.n || " narrows"
"Ewiden"   : expps(e.x) || " " || e.n || " widen"
```
Uses Enarrows 79, Enarrowu 79, Ewiden 79, and expps 125a.

127d ⟨*cases for* expps 125b⟩+≡ (125a) ◁127c 127e▷
```
"Ediv"     : expps(e.x) || " " || e.n || " idiv"
"Emod"     : expps(e.x) || " " || e.n || " mod"
```
Uses Ediv 79, Emod 79, and expps 125a.

127e ⟨*cases for* expps 125b⟩+≡ (125a) ◁127d 127f▷
```
"Semit"    : commaseparate(maplist(expps, e.x), "\n") || "\n"
"Stoken"   : expps(e.x) || " cl-loc force " ||
             (if e.offset ~= 0 then e.offset || " add " else "") ||
             e.n || " emit-at"
```
Uses expps 125a, Semit 79, and Stoken 79.

127f ⟨*cases for* expps 125b⟩+≡ (125a) ◁127e 127g▷
```
"Efitsu"   : expps(e.x) || " " || e.n || " fitsu"
"Efitss"   : expps(e.x) || " " || e.n || " fitss"
```
Uses Efitss 79, Efitsu 79, and expps 125a.

127g ⟨*cases for* expps 125b⟩+≡ (125a) ◁127f 127h▷
```
"Epatlabel" : &fail
```
Uses Epatlabel 79 and fail 396.

127h ⟨*cases for* expps 125b⟩+≡ (125a) ◁127g 128a▷
```
"Eforce"    : expps(e.x) || " force"
"Eforceable": expps(e.x) || " known"
```
Uses Eforce 79, Eforceable 79, and expps 125a.

128a ⟨*cases for* expps 125b⟩+≡ (125a) ◁127h 128b▷
```
"Epc"       : &fail
"Epc_known" : &fail
```
Uses Epc 79, Epc_known 79, and fail 396.

128b ⟨*cases for* expps 125b⟩+≡ (125a) ◁128a 128c▷
```
"Enot"      : expps(e.x) || " not"
```
Uses Enot 79 and expps 125a.

128c ⟨*cases for* expps 125b⟩+≡ (125a) ◁128b 128d▷
```
"Enosimp"   : expps(e.x)
```
Uses Enosimp 79 and expps 125a.

128d ⟨*cases for* expps 125b⟩+≡ (125a) ◁128c 128e▷
```
"Sstmts"    : commaseparate(maplist(expps, e.x), "\n") || "\n"
```
Uses expps 125a and Sstmts 79.

128e ⟨*cases for* expps 125b⟩+≡ (125a) ◁128d 128f▷
```
"Einstance" : { s := "<< /instance-of " || e.cons.name
                every i := inputs_of(e.cons) do
                  s ||:= " /" || i.name || " " || expps(e.argt[i.name])
                s || " >>"
              }
"Einstance_input" :
        expps(e.x) || " /" || e.cons.name || " /" || e.name || " get_instance"
"Einstance_tagged" :
        expps(e.x) || " /" || e.cons.name || " is_instance"
"Ebinding_instance" : &fail
```
Uses Ebinding_instance 79, Einstance 79, Einstance_input 79, Einstance_tagged 79, expps 125a, fail 396, and inputs_of 162a.

128f ⟨*cases for* expps 125b⟩+≡ (125a) ◁128e 129b▷
```
"Sif"      : { s := "["
               every s ||:= "\n" || expps(!e.arms)
               s || "\n] if-guard"
             }
"Sguarded" : psguard(e.guard) || "\n\t{" || expps(e.x) || "}"
```
Uses expps 125a, psguard 129a, Sguarded 79, and Sif 79.

129a ⟨* 125a⟩+≡ ◁126b

```
procedure psguard(e)
  return case type(e) of {
    "integer" : case e of { 0 : "false"; 1 : "true"; default: impossible("condition") }
    "null" : "true"
    "set" : {
       l := sort(maplist(expps, sort(e)))
       s := l[1]
       every s ||:= " " || l[2 to *l] || " and"
       if *l = 0 then "true" else s
    }
    default : expps(e)
  }
end
```

Defines:
psguard, used in chunk 128f.
Uses expps 125a.

129b ⟨*cases for* expps 125b⟩+≡ (125a) ◁128f 129c▷

```
"Stagcase" : { s := "["
               every c := kept_constructors(e.type) do
                 s ||:= "\n/" || c.name || " {" || expps(e.arms[c]) || "}"
               s ||:= "\n]" || expps(e.x) || " tag-case"
             }
```

Uses expps 125a, kept_constructors 31d, and Stagcase 178a.

129c ⟨*cases for* expps 125b⟩+≡ (125a) ◁129b 129d▷

```
"Sfail"    : { s := image(e.fmt) || " ["
               every a := e.a1 | e.a2 | e.a3 do
                 s ||:= " " || expps(\a)
               s := s || " ] failmsg"
             }
"Sepsilon" : ""
```

Uses expps 125a, Sepsilon 79, and Sfail 182c.

129d ⟨*cases for* expps 125b⟩+≡ (125a) ◁129c 130a▷

```
"Efail"    : &fail
```

Uses Efail 79 and fail 396.

130a ⟨*cases for* expps 125b⟩+≡ (125a) ◁129d 130b▷
```
  "Glines"    : commaseparate(maplist2(expps, e.x), "\n")
  "Gresynch"  : ""
  "Gblock"    : { s := "enterscope "
                  every s ||:= expps(!e.decls) || "\n"
                  every s ||:= expps(!e.stmts) || "\n"
                  s || "exitscope"
                }
  "Gdecl"     : { s := "/" || e.name || " " || expps(e.type)
                  s ||:= expps(\e.init) | "&uninitialized"
                  s || " decl"
                }
```
Uses expps 125a, Gblock 281a, Gdecl 281a, Glines 281a, and Gresynch 281a.

130b ⟨*cases for* expps 125b⟩+≡ (125a) ◁130a 130c▷
```
  "Gcase"     : &fail
  "Gcasearm"  : &fail
  "Ginrange"  : &fail
```
Uses fail 396, Gcase 282a, Gcasearm 282a, and Ginrange 282a.

130c ⟨*cases for* expps 125b⟩+≡ (125a) ◁130b 130d▷
```
  "Gasgn"     : &fail
  "Gsetname"  : &fail
  "Gnomatch"  : &fail
  "Tunsigned" : (\e.width | "wordsize") || " unsigned"
```
Uses fail 396, Gasgn 282a, Gnomatch 282a, Gsetname 282a, Tunsigned 282a, and wordsize 36c.

130d ⟨*cases for* expps 125b⟩+≡ (125a) ◁130c 130e▷
```
  "Eapp" : expps(e.args) || " " || image(e.f) || " apply"
```
Uses apply 266, Eapp 79, and expps 125a.

130e ⟨*cases for* expps 125b⟩+≡ (125a) ◁130d 130f▷
```
  "Eclosure_loc"  : "cl-loc"
  "Eclosure_addr" : e.n || " cla"
  "Eclosure_val"  : e.n || " clv"
```
Uses Eclosure_addr 79, Eclosure_loc 79, and Eclosure_val 79.

130f ⟨*cases for* expps 125b⟩+≡ (125a) ◁130e
```
  "Eclosure" : &fail
  "Elambda" :  {
    s := "["
    every s ||:= " /" || !e.formals
    s ||:= " ] { " || expps(e.body) || " } fn"
  }
```
Uses Eclosure 207, Elambda 206, expps 125a, and fail 396.

Chapter 6

Solving

6.1 Basic linear solving

Variables may be *unknown*, *dependent*, *known*, or *inputs*. A value is *computable* if it is a function of inputs, which appear in the set `inputs`. Both dependent variables and inputs appear as keys in the table `value`. Known variables are dependent variables whose values are computable. Unknown variables appear in neither `value` nor `inputs`.

The new solve works in two steps, first finding a solution in the form of a `value` table, then applying it.

132a ⟨*132a⟩≡ 132b▷

```
procedure solve(be, inputspec)
  local value, constraints, fieldsknown, zeroes, balances
  if *be.eqns = 0 then ⟨return vacuous solution 133b⟩ # common short cut

  value  := table()     # values of dependent variables
  constraints := []     # constraints to check
  zeroes      := []     # expressions equal to zero
  pending     := []     # pending equations we couldn't solve earlier
  every defined | used := set()        # sets of idents defined and used

  debugs("# Inputs:"); every debugs(" ", expimage(!inputspec)); debug()
  inputs := copy(inputspec)
  insert(inputs, 1)     # 1 is hack for finding dependent vars
  every x := !inputs do value[x] := term2table(x)
  every eq := !be.eqns do
    if eq.op == "=" then put(zeroes, subtract(eq.left, eq.right))
    else put(constraints, eq)
  ⟨initialize balance machinery and balance out inputs 138a⟩
  ⟨empty zeroes 134⟩
  ⟨dump value and constraints 141d⟩
  ⟨make sure all dependent variables are known 141c⟩
  ⟨build and return solution 133a⟩
end
```

Defines:
`constraints`, used in chunks 20b, 21a, 53, 63c, 65a, 68, 69, 71b, 90c, 91b, 101a, 114b, 119a, 132–34, 139c, 141d, 166a, 187, 252, 254a, 256, 257, 260b, 266, 271, 287b, 300d, 356b, 358a, 360, 361a, and 363c.
`defined`, used in chunks 26a, 30a, 35a, 70b, 132b, 133a, 136, 141e, 161d, 298, and 363c.
`solve`, used in chunks 141f, 157a, 165, 168a, 248, 252, and 253a.
`used`, used in chunks 9c, 31c, 34c, 45e, 55, 71b, 75b, 79, 132b, 133a, 136, 139, 141e, 155, 157c, 161, 168a, 173, 227b, 230c, 256, 258–60, 279a, 280a, 287b, and 363c.
`value`, used in chunks 14i, 39a, 49a, 50d, 53, 59, 74a, 76b, 79, 86b, 88, 115c, 122a, 133a, 134, 136–41, 152b, 173a, 187, 235, 259b, 281a, 282a, 318e, 336, 337a, 339, 341d, 343b, and 363a.
`zeroes`, used in chunks 123b, 134–36, and 139.

Uses `debug` 362a, `debugs` 362a, `expimage` 98a, `subtract` 140c, and `term2table` 110b.

In a solution the `defined` and `used` fields are sets of strings, which may name variables or fields. `answers` is a table mapping string variables to expression values.

132b ⟨*132a⟩+≡ ◁132a 135c▷

```
record solution(answers, constraints, defined, used)
```

Defines:
`solution`, used in chunks 133, 141c, and 252.

Uses `constraints` 132a, `defined` 132a, and `used` 132a.

133a ⟨*build and return solution* 133a⟩≡ (132a)
```
answers := table()
non_answer := inputs ## ++ fresh_vars (delete! matcher uses these!)
every ident := key(value) & type(ident) == "string" & not member(non_answer, ident) do
  answers[ident] := simplify(value[ident])
    # we think simplify is OK -- super definitely not OK!
every insert(used, free_variables(!answers | !constraints))
every debug("# ===> answers[", expimage(k := key(answers)), "] = ", expimage(answers[k]))
⟨dump def-use info 141e⟩
return solution(answers, simplify(constraints), defined, used)
```
Uses `constraints` 132a, `debug` 362a, `defined` 132a, `expimage` 98a, `free_variables` 82b, `simplify` 89b, `solution` 132b, `used` 132a, and `value` 132a.

133b ⟨*return vacuous solution* 133b⟩≡ (132a)
```
return solution(table(), [], emptyset, emptyset)
```
Uses `emptyset` 16a and `solution` 132b.

Invariants:

1. If variable `v` is known, than all of its ranges and extensions are known. *what has this to do with balances?*
2. Ranges and extensions of expressions (as opposed to variables or fields) appear only when the expressions are computable.
3. No key in `zeroes`, `pending`, or `constraints` is dependent.
4. Nothing in the range of `value` is dependent.
5. Values in `zeroes`, `pending`, `constraints`, and `value` are all in table form (so destructive substitution works).

134 ⟨*empty* `zeroes` 134⟩≡ (132a)

```
pending := []           # equations with unknowns but no unit coefficients
while *zeroes > 0 do {
  while z := get(zeroes) do {
    every v := key(z) & z[v] = 0 do delete(z, v)
    debug("# new equation: ", expimage(z), " = 0")
    ⟨normalize z by dividing out the greatest common divisor 142a⟩
    if v := key(z) & type(v) == "string" & not member(inputs, v) & z[v] = (1 | -1) then {
      ⟨make v in z a new dependent variable 136a⟩
      if computable(inputs, value[v]) then push(newlyknown, v)
      ⟨use newlyknown to make more variables known 138c⟩
      while put(zeroes, get(pending))
    } else if v := key(z) & not computable(inputs, v) then {
      debug("# no new dependent variable in ", expimage(z), " = 0")
      put(pending, z)
    } else if not values_known_equal(z, 0) then {
      debug("# new constraint ", expimage(z), " = 0")
      ⟨make sure z is satisfiable 141b⟩
      put(constraints, zero_to_constraint(z))
    }
  }
  ⟨if a pending equation can be fixed with div or mod, drain pending 135a⟩
}
⟨make sure there are no pending equations 141f⟩
```

Uses `computable` 135c, `constraint` 53, `constraints` 132a, `debug` 362a, `expimage` 98a, `value` 132a, `values_known_equal` 140b, `zeroes` 132a, and `zero_to_constraint` 141a.

135a ⟨*if a* pending *equation can be fixed with* div *or* mod*, drain* pending 135a⟩≡ (134)
```
pendingpending := []
while z := get(pending) do
  if v := key(z) & type(v) == "string" & not member(inputs, v) then {
    m := -z[v]
    delete(z, v)
    if m < 0 then { m := -m; z := subtract(0, z) }      # make m positive
    m ~= 1 | impossible("coefficient")
    ⟨create fresh variables x, q = x div m, r = x mod m 135b⟩
    # now force x = z, v = q, r = 0
    every put(zeroes, subtract(x, z) | subtract(v, q) | subtract(r, 0))
    while put(zeroes, get(pendingpending) | get(pending))
  } else
    put(pendingpending, z)
```
Uses subtract 140c and zeroes 132a.

135b ⟨*create fresh variables* x*,* q = x div m*,* r = x mod m 135b⟩≡ (135a)
```
every x | q | r := fresh_variable(v)
put(balances, b := balance([balitem(x, n_times_q_plus_r(m, q, r))],
                           [balitem(q, Ediv(x, m)), balitem(r, Emod(x, m))]))
every /baltab[x | q | r] := []; every put(baltab[x | q | r], b)
debug("New balance: ", balimage(b))
```
Uses balance 137a, balimage 152b, balitem 137a, debug 362a, Ediv 79, Emod 79, fresh_variable 151b, and n_times_q_plus_r 146b.

Fundamental concept: computability.

135c ⟨* 132a⟩+≡ ◁132b 137a▷
```
procedure computable(inputs, val)
  if member(inputs, val) then return
  else if v := free_variables(val) & not member(inputs, v) then fail
  else return
end
```
Defines:
computable, used in chunks 134, 138, 139b, and 141c.
Uses fail 396 and free_variables 82b.

By (2), `v` must be a variable, field, or range or extension thereof. By (3), (4) is maintained. (2) is maintained because `subst` substitutes for terms only, not inside of ranges or extensions.

136a ⟨*make* `v` *in* `z` *a new dependent variable* 136a⟩≡ (134)
```
debug("# new dependent variable: ", expimage(v))
m := - z[v]             # multiplier of 1 or -1
delete(z, v)
every !z *:= m
value[v] := z
⟨update def-use info for v 136b⟩
every dsubst(!zeroes | !pending | !value, v, z)
debug("# value[", expimage(v), "] := ", expimage(z))
```
Uses `debug` 362a, `dsubst` 88b, `expimage` 98a, `value` 132a, and `zeroes` 132a.

136b ⟨*update def-use info for* `v` 136b⟩≡ (136a)
```
insert(defined, v)
if v  == free_variables(!zeroes | !pending | !value) then
  insert(used, v)
```
Uses `defined` 132a, `free_variables` 82b, `used` 132a, `value` 132a, and `zeroes` 132a.

136c ⟨*update def-use info for* `vv` 136c⟩≡ (140a)
```
insert(defined, vv)
if vv == free_variables(!zeroes | !pending | !value) then
  insert(used, vv)
```
Uses `defined` 132a, `free_variables` 82b, `used` 132a, `value` 132a, and `zeroes` 132a.

6.2 Balances

Balances are a hack to deal with sign extension, slicing, DIV and MOD, and other computations that don't fit the simple linear-equation model. The idea is to put a set of variables on each side of the balance, such that each variable on one side is a function of all the variables on the other side, and vice versa. If all the variables on one side become known, the ones on the other side are also known. The reason this is useful is that we can substitute such variables in for slices, extensions, and other computations. For example, the slices of a variable balance that variable:

$$\begin{array}{rcl} a & = & x[0:9] \\ b & = & x[10:15] \\ c & = & x[16:31] \end{array} \quad \Longleftrightarrow \quad x = a \mid b \ll 10 \mid c \ll 16$$

The representation of a balance is:

137a ⟨* 132a⟩+≡ ◃135c 137b▹

```
record balance(left, right)      # lists of balitem
record balitem(v, value)         # v is string, value is exp
```

Defines:
balance, used in chunks 115c, 122a, 135b, 139a, 146a, 148, and 150.
balitem, used in chunks 115c, 122a, 135b, 137b, 146a, 148, and 150.
Uses value 132a.

where the only free variables in the `value` fields are the variables listed on the opposite side of the balance.

The solver assumes that variable names used in balances don't collide with other names; it's the responsibility of whatever agent provides the balances to make it so.

Once a balance becomes known, it's useful to identify the side on which all variables are known:

137b ⟨* 132a⟩+≡ ◃137a 138b▹

```
record kbalance(known, unknown) # lists of balitem
```

Defines:
kbalance, used in chunk 138b.
Uses balitem 137a.

To discover when a variable completes a balance, I create a table that lists the balances associated with each variable. `balsused` ensures I don't use the same balance more than once.

138a ⟨*initialize balance machinery and balance out inputs* 138a⟩≡ (132a)
```
balances := copy(be.balances)
baltab := table()
every b := !balances & v := (!b.left | !b.right).v do {
  /baltab[v] := []
  put(baltab[v], b)
}
balsused := set()
newlyknown := sort(inputs)
oldknown := set()
⟨use newlyknown to make more variables known 138c⟩
```

138b ⟨* 132a⟩+≡ ◁137b 140b▷
```
procedure balance_completed(bal, value, inputs)
  local vl, vr
  if vl := (!bal.left).v & not computable(inputs, \value[vl]) then
    if vr := (!bal.right).v & not computable(inputs, \value[vr]) then
{
debug("Balance not completed; ",
        vl, " = ", expimage(\value[vl]) | "???", " unknown on left and ",
        vr, " = ", expimage(\value[vr]) | "???", " unknown on right")
      fail
}
    else
      return kbalance(bal.right, bal.left)
  else
    return kbalance(bal.left, bal.right)
end
```
Defines:
`balance_completed`, used in chunk 139b.
Uses `computable` 135c, `debug` 362a, `expimage` 98a, `fail` 396, `kbalance` 137b, and `value` 132a.

138c ⟨*use* `newlyknown` *to make more variables known* 138c⟩≡ (134 138a)
```
while v := get(newlyknown) do
  if not member(oldknown, v) then {
    insert(oldknown, v)
    ⟨debug balancing act 139a⟩
    ⟨if v completes half of a balance, make the other half known 139b⟩
    every v := key(value) & computable(inputs, value[v]) do
      push(newlyknown, v)
  }
```
Uses `computable` 135c and `value` 132a.

139a ⟨*debug balancing act* 139a⟩≡ (138c)
```
debug(image(v), " is newly known ")
every vb := !\baltab[v] do
  if member(balsused, vb) then
    debug("%%%%% appears in (already used) balance ", balimage(vb))
  else
    debug("!!!!! appears in another balance ", balimage(vb))
if not !\baltab[v] then
  debug (":-( doesn't appear in any balances")
```
Uses balance 137a, balimage 152b, debug 362a, and used 132a.

139b ⟨*if* v *completes half of a balance, make the other half known* 139b⟩≡ (138c)
```
every vb := !\baltab[v] & not member(balsused, vb) &
      b := balance_completed(vb, value, inputs) do {
  insert(balsused, vb)
  tt := table() # used to substitute all balanced goodies at once
  every vv := (ii := !b.unknown).v & zz := term2table(subst_tab(ii.value, value, 1)) do
{ debug("=> balancing tells us ", vv, " = ", expimage(zz))
    if computable(inputs, \value[vv]) then {
      ⟨make new constraint from value[vv]=zz 139c⟩
    } else {
      if \value[vv] then
        {⟨make new equation value[vv]=zz, then update value[vv] 139d⟩}
      else
        {⟨make vv a new dependent variable with value[vv]=zz 140a⟩}
      tt[vv] := zz
      if computable(inputs, zz) then push(newlyknown, vv)
    }
}
  every dsubst_tab(!zeroes | !pending | !value, tt)
}
```
Uses balance_completed 138b, computable 135c, debug 362a, dsubst_tab 88b, expimage 98a, subst_tab 87a, term2table 110b, used 132a, value 132a, and zeroes 132a.

139c ⟨*make new constraint from* value[vv]=zz 139c⟩≡ (139b)
```
if not values_known_equal(value[vv], zz) then {
  put(constraints, eqn(value[vv], "=", zz))
  debug("# new constraint ", expimage(constraints[-1]))
}
```
Uses constraint 53, constraints 132a, debug 362a, eqn 81c, expimage 98a, value 132a, and values_known_equal 140b.

139d ⟨*make new equation* value[vv]=zz*, then update* value[vv] 139d⟩≡ (139b)
```
put(zeroes, subtract(value[vv], zz))
value[vv] := zz
```
Uses subtract 140c, value 132a, and zeroes 132a.

140a ⟨*make* `vv` *a new dependent variable with* `value[vv]=zz` 140a⟩≡ (139b)
```
value[vv] := zz
⟨update def-use info for vv 136c⟩
```
Uses `value` 132a.

6.3 Utility goo

This hack is just a heuristic that tries to avoid creating unnecessary constraints, so false negatives are OK.

140b ⟨* 132a⟩+≡ ◁138b 140c▷
```
procedure values_known_equal(e1, e2)
  if e1 === e2 then return e1
  e1 := untable(e1)
  e2 := untable(e2)
debug("vals ", expimage(e1), " ?= ", expimage(e2))
  return case type(e1) == type(e2) of {
    "Ewiden"   : e1.n = e2.n & exps_eq(e1.x, e2.x)
    "Eslice"   : e1.n == e2.n & e1.lo == e2.lo & exps_eq(e1.x, e2.x)
    "table"    : constant(subtract(e1, e2)) = 0
    "integer"  : e1 = e2
    "string"   : e1 == e2
    default    : e1 === e2
  }
end
```
Defines:
`values_known_equal`, used in chunks 134 and 139c.
Uses `constant` 111b, `debug` 362a, `Eslice` 79, `Ewiden` 79, `expimage` 98a, `exps_eq` 82c, `subtract` 140c, and `untable` 111c.

Unlike `binop`, `subtract` is non-destructive. It would not be safe to use `binop` within the solver.

140c ⟨* 132a⟩+≡ ◁140b 141a▷
```
procedure subtract(l, r)
  z := table(0)
  l := term2table(l)
  r := term2table(r)
  every v := key(l) do z[v]  := l[v]
  every v := key(r) do z[v] -:= r[v]
  return z
end
```
Defines:
`subtract`, used in chunks 85b, 132a, 135a, 139d, and 140b.
Uses `term2table` 110b.

When converting a redundent zero to a constraint, I make all coefficients positive. It's not necessary for correctness, but it leads to more readable constraints (and therefore to more readable code).

141a ⟨* 132a⟩+≡ ◁140c 142b▷
```
procedure zero_to_constraint(z)
  e := eqn(table(0), "=", table(0))
  every k := key(z) do
    if      z[k] > 0 then e.left[k]  +:= z[k]
    else if z[k] < 0 then e.right[k] -:= z[k]
  return e
end
```
Defines:
zero_to_constraint, used in chunk 134.
Uses eqn 81c.

141b ⟨*make sure* z *is satisfiable* 141b⟩≡ (134)
```
debug("#### sure hope ", expimage(z), " is satisfiable!!!")
```
Uses debug 362a and expimage 98a.

141c ⟨*make sure all dependent variables are known* 141c⟩≡ (132a)
```
if v := key(value) & not computable(inputs, value[v]) then {
  write(&errout, "Error! Incomplete solution for value[", expimage(v), "] = ",
                 expimage(value[v]))
  every write(&errout, "  value[", expimage(k := key(value)), "] = ", expimage(value[k]))
  error()
}
if v := key(value) & type(v) == "Eslice" & not member(value, v.x) then
  error("Solved for ", expimage(v), " but not for ", expimage(v.x))
if v := key(value) & type(v) == "Ewiden" & type(v.x) == "Eslice" &
   not member(value, v.x.x) then
  error("Solved for ", expimage(v), " but not for ", expimage(v.x.x))
```
Uses computable 135c, Eslice 79, Ewiden 79, expimage 98a, solution 132b, and value 132a.

141d ⟨*dump* value *and* constraints 141d⟩≡ (132a)
```
every debug("# ===> value[", expimage(k := key(value)), "] = ", expimage(value[k]))
every debug("# ===> constrain  ", expimage(!constraints))
```
Uses constraints 132a, debug 362a, expimage 98a, and value 132a.

141e ⟨*dump def-use info* 141e⟩≡ (133a)
```
debugs("# defined:"); every debugs(" ", !defined); debug()
debugs("# used:");    every debugs(" ", !used);    debug()
```
Uses debug 362a, debugs 362a, defined 132a, and used 132a.

141f ⟨*make sure there are no* pending *equations* 141f⟩≡ (134)
```
if *pending > 0 then {
  every write(&errout, "error: equation ", expimage(!pending), " = 0  is unusable")
  error("Can't solve equations; some are useless")
}
```
Uses expimage 98a and solve 132a.

142a ⟨*normalize* **z** *by dividing out the greatest common divisor* 142a⟩≡ (134)
```
g := &null
every g := gcd(g, !z)
if \g > 1 then every !z /:= g
```
Uses gcd 142b.

142b ⟨* 132a⟩+≡ ◁141a
```
procedure gcd(m, n) # Knuth vol 1, p 4
  if n < 0 then n := -n
  if /m then return n
  if m < n then m :=: n
  while r := 0 < (m % n) do {m := n; n := r}
  return n
end
```
Defines:
gcd, used in chunk 142a.

Chapter 7

Balancing out nonlinear expressions

This code uses the "balance" mechanism defined in the solver to define the relationships among extensions and slices in a set of equations.

This was the old story:

> *Slices make things tricky, since we needn't always have a complete set of them. We therefore have to find the slices first, so that we substitute variables in only when we have a complete set. This isn't too hard, because the original source of equations is the parser, so slices can be applied to variables only.*
>
> *Extensions are simpler since there's no such thing as an incomplete set, so they're computed on the fly by* `balexp`.

In the new world, things are harder because the parser is no longer the sole source of equations. Instead, we may be asked to take a solution and invert it (for decoding). Instead of using only variables that appear in the spec, we have to discover new opportunities and introduce balancing variables. We do this with a procedure called `var_for`, which has the property that if `exps_eq(e1, e2)` then `var_for(e1) == var_for(e2)`. `var_for` takes a couple of extra arguments: `varaux` to keep track of its state, and `varmap` to provide an inverting map (from variables to expressions).

We create balances in three passes:

1. Find all slices, extensions, narrows, divs, mods, and their arguments. Introduce variables in opportune cases, and salt everything away in preliminary sets and tables.

2. Go through the accumulated sets and tables, and create balances whenever we have accumulated enough pieces to be able to do so. For example, if we see `e MOD 4`, we don't have enough to create a balance, but if we see `e MOD 4` and `e DIV 4`, we do. When we do introduce a balance, add

entries to `balmap`, which maps expressions to variables of the balance. Sometimes introducing balances also requires introducing new equations.

3. Substitute the variables of the balance in the equations. This requires an odd form of top-down substitution; see `balsub_f` for more information.

7.1 Superstructure

This code shows the implementation of the three-pass method given above. The first pass is made using the expression-walking function `balpass1`.

144a ⟨* 144a⟩≡ 144b▷

```
procedure balpass1(e, varaux, varmap ⟨, aux tables 145c⟩)
  case type(e) of {
    ⟨cases for identifying candidates 145d⟩
  }
end
```

Defines:
`balpass1`, used in chunk 144b.

The second pass we do right in line, and the third pass relies on the recursive-substition procedure `balsub_f`, about which more below.

144b ⟨* 144a⟩+≡ ◁144a 145a▷

```
procedure balance_eqns(eqns)
  local width
  if *eqns = 0 then return balanced_eqns(eqns, []) # common short cut
  debug ("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$")
  pushtrace("BALANCE")
  every ⟨aux tables | 145b⟩ varaux | varmap := table()
  expwalk(eqns, balpass1, varaux, varmap ⟨, aux tables 145c⟩)
  balances := []
  neweqns := copy(eqns)
  balmap := table()
  ⟨use auxiliaries to add to balances, neweqns, and balmap 146a⟩
  neweqns := gsubst(neweqns, balsub_f, balmap)
  poptrace()
  ⟨brain dump 153⟩
  return balanced_eqns(neweqns, balances)
end
record balanced_eqns(eqns, balances)
```

Defines:
`balance_eqns`, used in chunks 157a, 248, and 252.
Uses `balpass1` 144a, `balsub_f` 145a, `debug` 362a, `expwalk` 114a, and `gsubst` 117a.

145a ⟨* 144a⟩+≡ ◁144b 145e▷
```
procedure balsub_f(e, balmap)
  return gsubst(\balmap[e], balsub_f, balmap)
end
```
Defines:
`balsub_f`, used in chunk 144b.
Uses `gsubst` 117a.

7.2 Balancing out DIV and MOD

Each balancing act has some substitutions, a balance, and some equations. For DIV and MOD we have

$$\begin{array}{rcl} e\,\mathtt{DIV}\,n & \mapsto & q \\ e\,\mathtt{MOD}\,n & \mapsto & m \\ d = q \times n + m & \bowtie & q = d\,\mathtt{DIV}\,n, m = d\,\mathtt{MOD}\,n \\ d & = & e \end{array}$$

I use two auxiliary tables, one to save DIV expressions and one to save MOD expressions.

145b ⟨*aux tables* | 145b⟩≡ (144b) 147a▷
```
divs | mods |
```

145c ⟨, *aux tables* 145c⟩≡ (144) 147b▷
```
, divs, mods
```

The `divs` table represents a mapping; when `d = e MOD n`, we have

```
divs: var_for(e) -> n -> e
```

and analogously for `mods`.

145d ⟨*cases for identifying candidates* 145d⟩≡ (144a) 147c▷
```
"Ediv" : addnset(divs, var_for(e.x, varaux, varmap), e)
"Emod" : addnset(mods, var_for(e.x, varaux, varmap), e)
```
Uses `addnset` 145e, `Ediv` 79, `Emod` 79, and `var_for` 152a.

(Actually, to tell the truth, the result of the mapping is the set of expressions equivalent to `e`, not just `e` itself.)

145e ⟨* 144a⟩+≡ ◁145a 146b▷
```
procedure addnset(t, vx, e)
  /t[vx] := table()
  /t[vx, e.n] := set()
  return insert(t[vx, e.n], e)
end
```
Defines:
`addnset`, used in chunks 145d and 147c.

To create balances, we look for dividends vd that also appear in mods with the same n, and we create a balance for each one. If needed, we also introduce the equation making the dividend equal to the variable we introduced.

146a ⟨*use auxiliaries to add to* balances, neweqns, *and* balmap 146a⟩≡ (144b) 148a▷

```
every vd := key(divs) & n := key(divs[vd]) do
  if ms := \(\mods[vd])[n] & qs := divs[vd, n] then {
    vq := var_for(!qs, varaux, varmap)
    vm := var_for(!ms, varaux, varmap)
    # vq = vd div n, vm = vd mod n, vd = vq * n + vm
    put(balances, balance([balitem(vd, n_times_q_plus_r(n, vq, vm))],
                          [balitem(vq, Ediv(vd, n)), balitem(vm, Emod(vd, n))]))
    debug("==> New balance:", balimage(balances[-1]))
    put(neweqns, eqn(vd, "=", vd ~=== varmap[vd])) &
debug("New balancing equation ", vd, " = ", expimage(varmap[vd]))
    every balmap[!qs] := vq
    every balmap[!ms] := vm
  }
```

Uses balance 137a, balimage 152b, balitem 137a, debug 362a, Ediv 79, Emod 79, eqn 81c, expimage 98a, n_times_q_plus_r 146b, and var_for 152a.

Here's a convenient function to compute the sum that forms one side of balances introduced for DIV and MOD:

146b ⟨* 144a⟩+≡ ◁145e 147d▷

```
procedure n_times_q_plus_r(d, q, r)
  local xtab
  xtab := table(0); xtab[q] := d; xtab[r] := 1
  return xtab
end
```

Defines:
n_times_q_plus_r, used in chunks 135b and 146a.

7.3 Balancing out narrows and widens

Here we have two schemata that could introduce isomorphic balances. They are:

$$\begin{array}{rcl} e\uparrow_n & \mapsto & w \\ w = n\uparrow_n & \bowtie & n = w\downarrow_n \\ n & = & e \end{array}$$

and

$$\begin{array}{rcl} e\downarrow_n & \mapsto & n \\ w = n\uparrow_n & \bowtie & n = w\downarrow_n \\ w & = & e \end{array}$$

The trick here is to avoid introducing the same balance twice for the case where the relevant stuff appears in a seperate narrow and widen.

147a ⟨*aux tables* | 145b⟩+≡ (144b) ◁145b 149b▷

```
extensions | narrows |
```

147b ⟨, *aux tables* 145c⟩+≡ (144) ◁145c 149c▷

```
, extensions, narrows
```

147c ⟨*cases for identifying candidates* 145d⟩+≡ (144a) ◁145d 149d▷

```
"Ewiden"   : addnset(extensions, var_for(e.x, varaux, varmap), e)
"Enarrows" : addnset(narrows,    var_for(e.x, varaux, varmap), e)
```

Uses `addnset` 145e, `Enarrows` 79, `Ewiden` 79, and `var_for` 152a.

147d ⟨* 144a⟩+≡ ◁146b 150▷

Here we have the special problem that, if we weren't careful, we could introduce the same balance from both an extension and a narrow. `baln2w` maps `vn` to the list of `vw`'s in every balance using a narrow `vn`, so we don't ever create the same balance twice.

148a ⟨*use auxiliaries to add to* `balances`, `neweqns`, *and* `balmap` 146a⟩+≡ (144b) ◁146a 148b▷

```
  baln2w := table()
  every vn := key(extensions) & width := key(extensions[vn]) do {
    ws := extensions[vn, width]
    vw := var_for(!ws, varaux, varmap) | impossible("no extensions")
    # vw := vn! >< vn := vw[]
    if member(\baln2w[vn], vw) then {
      debug("DUPLICATE balance ", vn, " |><| ", vw)
    } else {
      ⟨add new balance to baln2w 149a⟩
      put(balances, balance([balitem(vw, Ewiden  (vn, width))],
                            [balitem(vn, Enarrows(vw, width))]))
      debug("==> New balance:", balimage(balances[-1]))
    }
    put(neweqns, eqn(vn, "=", vn ~=== varmap[vn])) # could tighten (1-element table)
  #####  *ws # seems to be needed to work around bug in icont!
    every balmap[!ws] := vw
  }
```

Uses `balance` 137a, `balimage` 152b, `balitem` 137a, `debug` 362a, `Enarrows` 79, `eqn` 81c, `Ewiden` 79, and `var_for` 152a.

148b ⟨*use auxiliaries to add to* `balances`, `neweqns`, *and* `balmap` 146a⟩+≡ (144b) ◁148a 149e▷

```
  every vw := key(narrows) & width := key(narrows[vw]) do {
    ns := narrows[vw, width]
    vn := var_for(!ns, varaux, varmap) | impossible("no narrows")
    # vw := vn! >< vn := vw[]
    if member(\baln2w[vn], vw) then {
      debug("DUPLICATE balance ", vn, " |><| ", vw)
    } else {
      ⟨add new balance to baln2w 149a⟩
      put(balances, balance([balitem(vw, Ewiden  (vn, width))],
                            [balitem(vn, Enarrows(vw, width))]))
      debug("==> New balance:", balimage(balances[-1]))
    }
    put(neweqns, eqn(vw, "=", vw ~=== varmap[vw])) # could tighten (1-element table)
  #####  *ns # seems to be needed to work around bug in icont!
    every balmap[!ns] := vn
  }
  baln2w := &null # make it possible to garbage-collect the memory
```

Uses `balance` 137a, `balimage` 152b, `balitem` 137a, `debug` 362a, `Enarrows` 79, `eqn` 81c, `Ewiden` 79, and `var_for` 152a.

149a ⟨*add new balance to* `baln2w` 149a⟩≡ (148)
```
/baln2w[vn] := set()
insert(baln2w[vn], vw)
```

7.4 Balancing out slices

Here things are a bit odd, because we can have slices nested within slices. I ignore that problem for the time being (although I have some code); instead I describe the simple case:

$$\begin{array}{rcl} e[r_1] & \mapsto & s_1 \\ & \vdots & \\ e[r_n] & \mapsto & s_n \\ w = \sum_i 2^{l_i} \times s_i & \bowtie & s_1 = w[r_1], \ldots, s_n = w[r_n] \\ w & = & e \end{array}$$

149b ⟨*aux tables* `|` 145b⟩+≡ (144b) ◁147a
```
slices |
```

149c ⟨, *aux tables* 145c⟩+≡ (144) ◁147b
```
, slices
```

149d ⟨*cases for identifying candidates* 145d⟩+≡ (144a) ◁147c
```
"Eslice" : { vx := var_for(e.x, varaux, varmap)
             /slices[vx] := set()
             insert(slices[vx], e)
           }
```
Uses `Eslice` 79 and `var_for` 152a.

149e ⟨*use auxiliaries to add to* `balances`, `neweqns`, *and* `balmap` 146a⟩+≡ (144b) ◁148b
```
every vx := key(slices) & xslices := slices[vx] do
  slicetree(vx, xslices, varaux, varmap, 0, bitsizeof(varmap[vx]),
            neweqns, balances, balmap)
```
Uses `bitsizeof` 112c and `slicetree` 150.

I'm not sure I'll be introducing equations properly when it gets to be time to slice a tree...

150 ⟨* 144a⟩+≡ ◁147d 151b▷

```
procedure slicetree(vw, xslices, varaux, varmap, lo, hi, neweqns, balances, balmap)
  return do_slicetree(vw, copy(xslices), varaux, varmap, lo, hi,
                      neweqns, balances, balmap)
end

procedure do_slicetree(vw, xslices, varaux, varmap, lo, hi, neweqns, balances, balmap)
  sibs := set()
  sum := table(0)
  leftbal := []
  debug("Slicing ", expimage(sort(xslices)), " from ", lo, " to ", hi)
  while lo < hi do {
    kids := set()
    ⟨make first the largest slice at lo (fail if not there and leftmost) 151a⟩
    insert(sibs, first)
    delete(xslices, first)
    firstv := var_for(first, varaux, varmap)
    sum[if lo = 0 then firstv else Eshift(firstv, lo)] +:= 1
    put(leftbal, balitem(firstv, Eslice(vw, first.lo, first.n)))
    every slice := !xslices do
      if slice.lo + slice.n <= first.lo + first.n then {
        insert(kids, slice)
        delete(xslices, slice)
      } else if slice.lo < first.lo + first.n then
        error("Overlapping slices: ", expimage(subst_tab(first, varmap)), " and ",
                                      expimage(subst_tab(slice, varmap)))
    if *kids > 0 then
      do_slicetree(firstv, kids, varaux, varmap, first.lo, first.lo + first.n,
                   neweqns, balances, balmap)
    lo +:= first.n
  }
  *xslices = 0 |
      impossible("leftover slices: ", expimage(subst_tab(sort(xslices), varmap)))
  put(balances, balance(leftbal, [balitem(vw, sum)]))
  debug("==> New balance:", balimage(balances[-1]))
  every slice := !sibs do
    balmap[slice] := var_for(slice, varaux, varmap)
  put(neweqns, eqn(vw, "=", vw ~=== varmap[vw])) &
debug("New balancing equation ", vw, " = ", expimage(varmap[vw]))
end
```

Defines:
slicetree, used in chunk 149e.

Uses balance 137a, balimage 152b, balitem 137a, debug 362a, eqn 81c, Eshift 79, Eslice 79, expimage 98a, subst_tab 87a, and var_for 152a.

151a ⟨*make* `first` *the largest slice at* `lo` *(*`fail` *if not there and leftmost)* 151a⟩≡ (150)
```
first := &null
every slice := !xslices do
  if slice.lo < lo then fail
  else if slice.lo = lo & not ((\first).n > slice.n) then
    first := slice
\first | fail
```
Uses `fail` 396.

7.5 Utilities

There are now all sorts of places we introduce fresh variables. Maintaining the global set of fresh variables might be a bit of a waste of memory, but it's a convenient way to avoid putting these auxiliary variables where they aren't needed.

151b ⟨* 144a⟩+≡ ◁150 151c▷
```
global fresh_variables
procedure fresh_variable(name)
  static n
  initial n := 0
  insert(fresh_variables, s := fresh_base(name) || "#" || (n +:= 1))
  return s
end
```
Defines:
`fresh_variable`, used in chunks 47c, 135b, 152a, 177b, 206, 207, and 253c.
`fresh_variables`, used in chunks 17a, 161d, and 289b.
Uses `fresh_base` 151c.

151c ⟨* 144a⟩+≡ ◁151b 152a▷
```
procedure fresh_base(name)
  static tail
  initial tail := '#' ++ &digits
  name ? return 1(tab(upto('#')|0), tab(many(tail)) | "", pos(0))
end
```
Defines:
`fresh_base`, used in chunk 151b.

The procedure `var_for` produces a fresh variable to stand in for the expression `e`. Its special property is that `var_for(e1) == var_for(e2)` whenever `exps_eq(e1, e2)`.

152a ⟨* 144a⟩+≡ ◁151c 152b▷

```
procedure var_for(e, aux, var2exp)
  e := untableexp(e)
  if type(e) == "string" then
    return var2exp[e] := e
  /aux[type(e)] := table()
  t := aux[type(e)]
  return \t[e] | {
    every k := key(t) do
      if exps_eq(e, k) then {
        t[e] := t[k]
        break
      }
    var2exp[/t[e] := fresh_variable(free_variables(e) | "???")] := e
    t[e]
  }
end
```

Defines:
`var_for`, used in chunks 145–50.
Uses `exps_eq` 82c, `free_variables` 82b, `fresh_variable` 151b, and `untableexp` 86a.

7.6 Debugging support

152b ⟨* 144a⟩+≡ ◁152a

```
procedure balimage(b)
  s := ""
  every ii := !b.left do
    s ||:= "\n  <<< " || ii.v || " = " || expimage(ii.value)
  s ||:= "\n  -----------------------"
  every ii := !b.right do
    s ||:= "\n  >>> " || ii.v || " = " || expimage(ii.value)
  return s
end
```

Defines:
`balimage`, used in chunks 135b, 139a, 146a, 148, and 150.
Uses `expimage` 98a and `value` 132a.

153 ⟨*brain dump* 153⟩≡ (144b)
```
debug("Old equations:")
every debug(" ", expimage(!eqns))
write(\baldebug, "After substituting:", envimage(balmap, "balmap"))
debug("New equations:")
every debug(" ", expimage(!neweqns))
debug()
```
Uses `baldebug` 11c, `debug` 362a, `envimage` 25a, and `expimage` 98a.

Chapter 8

Constructors

8.1 Creating and checking constructors

8.1.1 Basic types

Most of the elements of a constructor are straight out of the concrete syntax. Operands may be `literal` or `input`. `type` is a `constype`. `rho` is a bit odd; it contains the initial bindings used to evaluate the patterns in the branches, including any bindings introduced as part of exploding the constructor's opcode. The constructor's `tag` is used to identify instances of that constructor. Tags are unique only within constructor types.

To each branch we add the solution to its equations.

The constructor type keeps track of its members, the number of tags used, and the line on which the type itself was used (to enforce definition before use). The `members` field is initially a set. When the type is first used, it must be complete, so the `members` are turned into a list (in alphabetical order by constructor name).

155 ⟨* 155⟩≡ 156▷

```
record constructor(name, opcode, operands, type, branches, rho, tag)
record branch(eqns, soln, pat)
record constype(name, members, used, ntags)
```

Defines:

`branch`, used in chunks 157a, 161d, and 185b.

`constructor`, used in chunks 13m, 30, 39a, 44b, 47c, 79, 101b, 156–58, 161–63, 167a, 172, 176b, 184, 185, 189b, 199–201, 205a, 213a, 220a, and 231.

`constype`, used in chunks 30b, 31d, 43d, 44b, 101b, 160a, 165, 167, 168a, 170a, 183a, 194b, 195a, 198b, 201a, 230b, 235, 238a, 244, 250a, 297a, and 303.

Uses `used` 132a.

The solution (`soln`) in a branch has appropriate expressions substituted for the free variables representing the inputs.

8.1.2 Interpreting and checking constructor specifications

The parser calls `note_constructor` when it sees a constructor specification. One specification may create many constructors; `note_constructor` calls `explode` to generate them all. Its effects are:

- add the constructor to the members of its type
- make sure the type hasn't been used an an argument to another constructor (thus avoiding possible recursion)
- add the constructor to the `constructors` name space

156 ⟨* 155⟩+≡ ◁155 157a▷

```
procedure note_constructor(opcode, operands, type, branches)
    local cons, template

    ⟨to avoid circularity, make sure type has never been used 161b⟩
    template := constemplate(type, opcode, operands, branches)
    every cons := explode(opcode, template, globals) do {
        if /constructors[cons.name] := cons then {
            verbose("New constructor ", cons.name)
            put(conslist, cons)
            insert(type.members, cons)
            ⟨if first time thru, check for unused definitions in cons 161c⟩
        } else if *crhs(constructors[cons.name]).disjuncts = 0 then {
            verbose("Replacing vacuous constructor ", cons.name)
            constructors[cons.name] := cons
            put(conslist, cons)
            insert(type.members, cons)
            ⟨if first time thru, check for unused definitions in cons 161c⟩
        } else if *crhs(cons).disjuncts = 0 then {
            verbose("Ignoring extra, vacuous constructor ", cons.name)
        } else {
            warning("Ignoring duplicate definition of constructor ", cons.name)
#           PPxwrite(PPnew(&errout),
#               "Keeping $t$o", ppexpimage(crhs(constructors[cons.name])), "$b$n",
#               "Discarding $t$o", ppexpimage(crhs(cons)), "$b")
        }
    }
    return
end
```

Defines:
`note_constructor`, used in chunk 42a.

Uses `conslist` 30a, `constemplate` 157a, `constructor` 155, `constructors` 30a, `crhs` 164, `explode` 158b, `globals` 30a, `ppexpimage` 98d, and `verbose` 16a.

A constructor specification defines one or more constructors. (Multiple constructors are specified by enumerating disjuncts of patterns that appear in the opcode, or by enumerating values of fields that appear in the opcode.) `constemplate` returns a template that is filled in differently for each constructor (by `explode`). `instantiate_template` is called by `explode` to create a complete `Stype` from a template and environment `rho`. Differences between a template and a full-fledged constructor:

- A template has no name, initial environment, or tag

157a ⟨* 155⟩+≡ ◁156 157b▷

```
procedure constemplate(type, opcode, operands, branches)
  local inputs, inputs_labs
  ⟨make inputs the set of input names, barfing if duplicates exist 157c⟩
  B := []
  every b := !branches do { # b === [eqns, pat]
    ⟨make inputs_labs inputs plus labels defined in b[2], barfing on duplicates 158a⟩
    put(B, branch(b[1], inject_soln(solve(balance_eqns(b[1]), inputs_labs)),
                 \b[2] | implicit_pattern(opcode, operands)))
  }
  return constructor(&null, &null, operands, type, B, &null, &null)
end
```

Defines:
  `constemplate`, used in chunk 156.
Uses `balance_eqns` 144b, `branch` 155, `constructor` 155, `implicit_pattern` 160a, `inject_soln` 166b, and `solve` 132a.

`instantiate_template` instantiates the templates by assigning name, environment, and branch tags. Both the list of branches and the branches themselves have to be copied.

157b ⟨* 155⟩+≡ ◁157a 158b▷

```
procedure instantiate_template(op, t, rho)
  /t.type.ntags := 0
  t.type.ntags +:= 1
  ⟨check that t.type.ntags doesn't overflow tag bits 161e⟩
  return constructor(iname(op), op, t.operands, t.type, t.branches, rho, t.type.ntags)
end
```

Defines:
  `instantiate_template`, used in chunk 158b.
Uses `constructor` 155 and `iname` 160b.

157c ⟨*make* `inputs` *the set of input names, barfing if duplicates exist* 157c⟩≡ (157a)

```
inputs := set()
every i := inputs_of_operands(operands) do
  if member(inputs, i.name) then
    error("Input named ", i.name, " is used twice in one constructor")
  else
    insert(inputs, i.name)
```

Uses `constructor` 155, `inputs_of_operands` 162a, and `used` 132a.

Note that duplicate labels are OK, because they could appear in different disjuncts. We have to sort that stuff out later. (Also note that implicit patterns never define labels.)

158a ⟨*make* `inputs_labs` `inputs` *plus labels defined in* `b[2]`*, barfing on duplicates* 158a⟩≡ (157a)

```
if \b[2] then {
  inputs_labs := copy(inputs)
  every labname := pattern_label_names(b[2]) do
    if member(inputs, labname) then
      error("label name ", labname, ": conflicts with constructor input name")
    else
      insert(inputs_labs, labname)
} else {
  inputs_labs := inputs
}
```

Uses `constructor` 155, `input` 44a, `label` 369a, and `pattern_label_names` 77a.

`explode` generates all those constructors denoted by a single constructor specification by enumerating disjuncts and field values for disjuncts and fields that appear in the constructor's opcode. Those disjuncts and fields are bound in the initial environment for the constructor.

158b ⟨* 155⟩+≡ ◁157b 159b▷

```
procedure explode(opcode, template, rho, frame)
    /frame := table()
    l := copy(opcode)
    if type(p := l[i := 1 to *l]) == "pattern" then {
        every l[i] := !p.disjuncts do {
            add_to_frame(p.name, pattern([l[i]], \l[i].name | p.name), frame)
            suspend explode(l, template, rho, frame)
            delete(frame, p.name)
        }
    } else if type(f := l[i := 1 to *l]) == "field" then {
        t := fieldname_table(f); ⟨insist on *\t > 0 159a⟩
        every x := t[l[i] := key(t)] do {
            add_to_frame(f.name, inject(conspat(f, "=", x), x, &null), frame)
            suspend explode(l, template, rho, frame)
            delete(frame, f.name)
        }
    } else
        suspend instantiate_template(l, template, extendscope(rho, copy(frame)))
end
```

Defines:
`explode`, used in chunk 156.

Uses `add_to_frame` 24a, `conspat` 59, `extendscope` 24c, `field` 53, `fieldname_table` 342a, `inject` 27, `instantiate_template` 157b, and `pattern` 53.

159a ⟨*insist on* `*\t > 0` 159a⟩≡ (158b 159b)

```
  *\t > 0 | error("Can't use field '", f.name, "' in opcode without supplying field names")
```

Uses `field` 53.

For constructor applications, we explode the applied constructor's name, first using the constructor's "local" environment(s) then the global environment to determine the meaning of the applied constructor's name. For example, in the definition of the constructors generated by `P^Y`, the pattern `P` is bound to `A` in the environment created for the constructor `A^Y` *or* is bound to `B` in `B^Y`'s environment. In the definition of the constructor `R`, however, `P` is not bound in `R`'s environment, therefore its meaning in the global enviroment is used, i.e., `(A | B)(s)` which is `A(s) | B(s)`.

```
patterns     P is A | B
constructors
  P r is r
  P^Y s is P(s) & Y
  R s is P(s)
```

`explode_names` uses a pattern's name if it is bound in `rho`, otherwise it explodes its disjunct's names.

159b ⟨* 155⟩+≡ ◁158b 160a▷

```
  procedure explode_names(opcode, rho)
      l := copy(opcode)
      if type(p := l[i := 1 to *l]) == "pattern" then {
          if l[i] := \lookup(p.name, \rho) then
              suspend explode_names(l)
          else every l[i] := !p.disjuncts do {
              suspend explode_names(l)
          }
      } else if type(f := l[i := 1 to *l]) == "field" then {
          t := fieldname_table(f); ⟨insist on *\t > 0 159a⟩
          every x := t[l[i] := key(t)] do
              suspend explode_names(l)
      } else
          suspend iname(l)
  end
```

Defines:
`explode_names`, used in chunks 39b, 46b, and 50c.

Uses `field` 53, `fieldname_table` 342a, `iname` 160b, `lookup` 23a, and `pattern` 53.

An implicit pattern conjoins all the pattern and field names appearing in the opcode and operands.

160a ⟨* 155⟩+≡ ◁159b 160b▷

```
procedure implicit_pattern(opcode, operands)
  l := []
  every op := !opcode do
    case type(op) of {
      "pattern" : put(l, Pident(\op.name)) | impossible("unnamed opcode pattern")
      "field"   : put(l, Pident(op.name))
    }
  every ipt := inputs_of_operands(operands) do
    if type(ipt.meaning) == ("field"|"integer"|"constype") then
      put(l, Pident(ipt.name))
  *l > 0 |
    error("Cannot use implicit pattern with no patterns or fields on lhs (",
          expimage(opcode), ")")
  return Pand(l)
end
```

Defines:
`implicit_pattern`, used in chunk 157a.
Uses `constype` 155, `expimage` 98a, `field` 53, `inputs_of_operands` 162a, `Pand` 39a, `pattern` 53, and `Pident` 39a.

These auxiliary functions help choose names for constructors.

160b ⟨* 155⟩+≡ ◁160a 161a▷

```
procedure iname(opcode)
  local name
  name := ""
  every name ||:= opcode_component_name(!opcode)
  if \lowercons then name := map(name)
  return mapoutbadchars(name)
end

procedure opcode_component_name(op)
  return case type(op) of {
           "string"   : op
           "disjunct" : {
             if *\op.name = 0 then impossible("disjunct with empty name")
             \op.name | "???unnamed disjunct???"
             }
           default : impossible("opcode component", image(op))
         }
end
```

Defines:
`iname`, used in chunks 97a, 157b, 159b, and 163.
`opcode_component_name`, used in chunk 226a.
Uses `disjunct` 53, `lowercons` 16a, and `mapoutbadchars` 161a.

161a ⟨* 155⟩+≡ ◁160b 162a▷
```
procedure mapoutbadchars(name)
  static nonalnum, underscores
  initial {
    nonalnum := string(&ascii -- &letters -- &digits -- '_')
    underscores := repl("_", *nonalnum)
  }
  return map(name, nonalnum, underscores)
end
```
Defines:
`mapoutbadchars`, used in chunks 160b and 215a.

161b ⟨*to avoid circularity, make sure* `type` *has never been used* 161b⟩≡ (156)
```
if \type.used then
  error("You can't create new constructors of type ", type.name,
        "!\n\tThat name has already been used (on line", type.used, ")")
```
Uses `constructors` 30a, `line` 345a, and `used` 132a.

161c ⟨*if first time thru, check for unused definitions in* `cons` 161c⟩≡ (156)
```
if /checked then {
  checked := 1
  ⟨check for unused definitions in cons 161d⟩
}
```

161d ⟨*check for unused definitions in* `cons` 161d⟩≡ (161c)
```
every i := 1 to *cons.branches & b := cons.branches[i] do {
  u := copy(b.soln.used)
  every insert(u, pattern_free_variables(b.pat))
  d := copy(b.soln.defined)
  every insert(d, key(cons.rho[1 to *cons.rho - 1]) | inputs_of(cons).name)
  every x := !(d--u--fresh_variables) do
    warning(
      if \cons.rho[1 to *cons.rho-1][x] then "opcode part"
      else if member(b.soln.defined, x) then "equation result"
      else "operand",
      " ", image(x), " not used in constructor ", cons.name,
           if (*cons.branches = 1) then "."
           else " in " || ordinal(i) || " branch.")
}
```
Uses `branch` 155, `constructor` 155, `defined` 132a, `fresh_variables` 151b, `inputs_of` 162a, `ordinal` 163c, `pattern_free_variables` 76b, and `used` 132a.

161e ⟨*check that* `t.type.ntags` *doesn't overflow tag bits* 161e⟩≡ (157b)
```
if t.type.ntags >= 2^11 then
  impossible("Too many type tags --- change mclib.nw (struct instance), constructors.nw")
```
Uses `constructors` 30a.

8.2 Utility functions

8.2.1 Input generation

`inputs_of(cons, t)` generates the inputs with meanings of type `t`, or all inputs if `t` is omitted.

162a ⟨* 155⟩+≡ ◁161a 162b▷

```
procedure inputs_of(cons, t)
  suspend inputs_of_operands(cons.operands, t)
end
procedure inputs_of_operands(ops, t)
  type(ops) == "list" | impossible("inputs_of")
  suspend if \t then (type(i := !ops) == "input", type(i.meaning) == t, i)
          else       (type(i := !ops) == "input",                       i)
end
```

Defines:
`inputs_of`, used in chunks 94b, 105b, 128e, 161d, 162b, 165, 168, 169, 182b, 183a, 193–95, 201–3, 205a, 231, 234, 238b, 242, 301, 303, and 304c.
`inputs_of_operands`, used in chunks 157c, 160a, and 231.
Uses `input` 44a.

`input_named(cons, n)` returns the input named `n` if there is one.

162b ⟨* 155⟩+≡ ◁162a 162c▷

```
procedure input_named(cons, n)
  return if i := inputs_of(cons) & i.name == n then i
end
```

Defines:
`input_named`, used in chunks 200a and 208b.
Uses `inputs_of` 162a.

8.2.2 Checking constructor types

The two `enforce` functions distinguish between typed and untyped constructors by comparing their argument (a constructor type) to the anonymous type.

162c ⟨* 155⟩+≡ ◁162b 162d▷

```
procedure enforce_instance(ct)
  return instructionctype ~=== ct | impossible("instance of untyped constructor")
end
```

Defines:
`enforce_instance`, used in chunk 194a.
Uses `constructor` 155 and `instructionctype` 30a.

162d ⟨* 155⟩+≡ ◁162c 163a▷

```
procedure enforce_closure(ct)
  return instructionctype  === ct | impossible("closure of typed constructor")
end
```

Uses `constructor` 155 and `instructionctype` 30a.

8.2.3 Miscellany

`cons_named` produces the constructor named `s` and issues an error if no such constructor exists. `is_constructor` tests if `s` is a constructor and prints an optional message if it is not.

163a ⟨* 155⟩+≡ ◁162d 163b▷

```
  procedure cons_named(s)
    return is_constructor(s, error)
  end
  procedure is_constructor(s, p)
    return \constructors[s | iname([s])] | (\p)(image(s), " is not a constructor name")
  end
```

Defines:
`cons_named`, used in chunks 39b and 172.
`is_constructor`, used in chunks 46b and 50c.
Uses `constructor` 155, `constructors` 30a, and `iname` 160b.

163b ⟨* 155⟩+≡ ◁163a 163c▷

```
  procedure discard_cons_named(s)
    if member(constructors, s <- (s | iname([s]))) then
      delete(constructors, s)
    else
      warning("There is no constructor named ", s)
    return
  end
```

Defines:
`discard_cons_named`, used in chunk 46b.
Uses `constructor` 155, `constructors` 30a, and `iname` 160b.

163c ⟨* 155⟩+≡ ◁163b 164▷

```
  procedure ordinal(n)
    return case n of {
      1 : "1st"
      2 : "2nd"
      3 : "3rd"
      default : n || "th"
    }
  end
```

Defines:
`ordinal`, used in chunks 161d and 173c.

8.3 Applying constructors

8.3.1 Right-hand-side patterns

For every constructor, I compute a right-hand side, which is a pattern with free variables for the constructor's inputs. I then can apply the constructor by substituting for the inputs.

`chrs` computes the stuff. Every input makes a contribution to the environment. I don't try to make constructor inputs visible as labels. This is perhaps inconsistent with the treatment of free constructor types in matching statements, but I need to hear an argument in favor before I'll be convinced to work out a proper semantics.

164 ⟨* 155⟩+≡ ◁163c 166b▷

```
procedure crhs(cons)
  local rho, labrho
  static cache
  initial cache := table()
  if /cache[cons] then {⟨compute and cache pattern for cons 165⟩
PPxwrite(PPnew(\mdebug), "crhs for ", cons.name, " is ", ppexpimage(cache[cons]))
}
  return cache[cons]
end
```

Defines:
`crhs`, used in chunks 156, 168a, 169a, 182b, 221a, and 294.

Uses `mdebug` 16a and `ppexpimage` 98d.

165 ⟨*compute and cache pattern for* `cons` 165⟩≡ (164)

```
rho := newscope(cons.rho)
every ipt := inputs_of(cons) do
  case type(ipt.meaning) of {
    "constype" :
        add_to_rho(ipt.name, inject(consinput_pattern(ipt), &null, ipt), rho)
    "field" | "integer" :
        add_to_rho(ipt.name, inject(fieldinput_pattern(ipt), ipt.name, &null), rho)
    "string"| "null" :
        add_to_rho(ipt.name, inject(&null, ipt.name, &null), rho)
    default    : impossible("input type")
  }
p := &null
every b := !cons.branches do {
  ⟨convert branch b to a pattern q 166a⟩
  p := orp(\p, q) | q
}
cache[cons] := subst(eliminate_contradictions(p), "nonexistent variable", 0)
                # can't afford to simplify -- makes it too hard to solve eqns
               # subst eliminates bad tag conditions
```

Uses `add_to_rho` 24a, `consinput_pattern` 167a, `constype` 155, `eliminate_contradictions` 69b, `field` 53, `fieldinput_pattern` 168b, `inject` 27, `input` 44a, `inputs_of` 162a, `newscope` 24b, `orp` 61a, `simplify` 89b, `solve` 132a, and `subst` 86b.

OK, the treatment of labels is a dreadful hack. The problem is that with the pattern in syntactic form, we don't know which disjuncts labels actually belong to. The depressing solution is to make multiple passes:

1. Build an environment mapping label names to themselves (make sure there are no conflicts with inputs).
2. Convert the pattern to normal form, which will propagate the free-variable names into the field bindings.
3. Make a separate pass over each disjunct, converting the label names to suitable expressions of type `Epatlabel`. That last pass is done by `bind_and_remove_patlabel_names`—it also nulls out the `patlabel` stuff.

166a ⟨*convert branch* `b` *to a pattern* `q` 166a⟩≡ (165)

```
push(rho, b.soln.answers) # answers already injected by inject_soln
t := table()
every n := pattern_label_names(b.pat) do
  t[n] := n
push(rho, t)
q := freshen_disjuncts(pnf(b.pat, rho))
pop(rho)
pop(rho)
every bind_condition(q, !b.soln.constraints)
q := bind_and_remove_patlabel_names(q)
```

Uses `bind_and_remove_patlabel_names` 55, `bind_condition` 78a, `constraints` 132a, `freshen_disjuncts` 78b, `inject_soln` 166b, `pattern_label_names` 77a, and `pnf` 72.

166b ⟨* 155⟩+≡ ◁164 167a▷

```
procedure inject_soln(soln)
  every k := key(soln.answers) do
    if type(symtab[k]) == "field" then
      soln.answers[k] :=
        inject(constraints2pattern(fieldbinding(symtab[k], soln.answers[k])),
               soln.answers[k],
               &null)
  return soln
end
```

Defines:
`inject_soln`, used in chunks 157a and 166a.

Uses `constraints2pattern` 59, `field` 53, `fieldbinding` 53, `inject` 27, and `symtab` 30a 353a.

Making an instance means instantiating the precomputed one.

167a ⟨* 155⟩+≡ ◁166b 167b▷

```
procedure consinput_pattern(ipt)
  type(ipt.meaning) == "constype" | impossible("non-constructor input")
  return subst(constype_pattern(ipt.meaning), ipt.meaning.name, ipt.name)
end
```

Defines:
consinput_pattern, used in chunks 165 and 250a.
Uses constructor 155, constype 155, constype_pattern 167b, input 44a, and subst 86b.

We get the precomputed instance by using the right-hand sides of all the constructors.

167b ⟨* 155⟩+≡ ◁167a 168b▷

```
procedure constype_pattern(constype)
  local cons, luid
  static cache, uid
  initial { cache := table(); uid := 0 }
  if /cache[constype] then {⟨compute and cache pattern for constype 168a⟩}
  return cache[constype]
end
```

Defines:
constype_pattern, used in chunks 167a and 221b.
Uses constype 155.

Static local variables, like `uid` are evil. `constype_pattern` is called recursively via `crhs`. Each instance of `constype` is tagged with `uid`, but `uid` can be updated by the recursive call to `crhs`! So `luid` preserves its value across recursive calls. Yuckorama. You wouldn't believe what tripped this bug — Icon's generation of record values in alphabetical order.

168a ⟨*compute and cache pattern for* `constype` 168a⟩≡ (167b)

```
p := &null
uid +:= 1 # one uid per constype??? will fail with multiple args of some constype
luid := uid
every cons := kept_constructors(constype) do {
  t := table()   # substitution table to get inputs from instance
  every ipt := inputs_of(cons) do {
    t[ipt.name] := Einstance_input(constype.name, cons, ipt.name)
    if type(ipt.meaning) == "string" then
      t[ipt.name] := Eforce(t[ipt.name])  # ???? could this be right?
  }
  q := freshen_disjuncts(subst_tab(crhs(cons), t, 1))
  bind_condition(q, Einstance_tagged(constype.name, cons, luid))
  p := orp(\p, q) | q
}
p := seqpx(latent_label2pattern(constype.name), p)
       # label will be used for binding instances later on!
cache[constype] := p   # can't afford to simplify -- makes it too hard to solve eqns
```

Uses `bind_condition` 78a, `constype` 155, `crhs` 164, `Eforce` 79, `Einstance_input` 79, `Einstance_tagged` 79, `fail` 396, `freshen_disjuncts` 78b, `inputs_of` 162a, `kept_constructors` 31d, `label` 369a, `latent_label2pattern` 58b, `orp` 61a, `seqpx` 62a, `simplify` 89b, `solve` 132a, `subst_tab` 87a, and `used` 132a.

The pattern corresponding to a field input is a field binding.

168b ⟨* 155⟩+≡ ◁167b 169a▷

```
procedure fieldinput_pattern(ipt)
  type(symtab[ipt.name]) == "field" | impossible("input name is not field name")
  type(ipt.meaning) == ("field"|"integer") | impossible("non-field input")
  return constraints2pattern(fieldbinding(symtab[ipt.name], ipt.name))
end
```

Defines:
`fieldinput_pattern`, used in chunk 165.
Uses `constraints2pattern` 59, `field` 53, `fieldbinding` 53, `input` 44a, and `symtab` 30a 353a.

8.3.2 Applying the patterns

apply_constructor produces the pattern value and constraints for a constructor when applied to args in an output pattern in the context of an environment rho. We have to freshen pattern labels to maintain the nonduplication invariant.

169a ⟨* 155⟩+≡ ◁168b 169b▷

```
procedure apply_constructor(cons, args, rho, free_env)
  local inputs, c, l
pushtrace("APPCONS")
  inputs := []; every put(inputs, inputs_of(cons))
  ⟨insist input lengths match 173d⟩
  t := argtable(inputs, args, rho, free_env)
  p := freshen_patlabels(subst_tab(crhs(cons), t, 1)) # can't afford to simplify
  every p.name | /(!p.disjuncts).name := cons.name
        # overwrite pattern name always, but only default disjuncts
PPxwrite(PPnew(\mdebug), "applied ", cons.name, " to get ",ppexpimage(p))
poptrace()
  return p
end
```

Defines:
apply_constructor, used in chunk 73.
Uses argtable 170a, crhs 164, freshen_patlabels 57d, inputs_of 162a, mdebug 16a, pattern 53, ppexpimage 98d, simplify 89b, and subst_tab 87a.

app_to_instance is almost the same, but it produces an instance, not the pattern. You might wonder why we need it. Well, crhs produces a pattern in which constructor operands are expected to be *instances*, not patterns. That means that apply_constructor expects its constructor-typed arguments to be instances, not patterns. But apply_constructor itself produces a pattern (which it must do, because that pattern is then used to produce an emitter or whatever). So then, what are we to do when the argument to a constructor is itself a constructor application? We need a procedure that will apply a constructor and produce an instance, therefore app_to_instance. Note that the call to subst_tab in apply_constructor above actually eliminates the instances. (That's a sneaky hack.)

I wish I knew whether freshening pattern variables was needed here, or how to do it.

169b ⟨* 155⟩+≡ ◁169a 170a▷

```
procedure app_to_instance(cons, args, rho, free_env)
  local inputs
  inputs := []; every put(inputs, inputs_of(cons))
  ⟨insist input lengths match 173d⟩
  return Einstance(cons, argtable(inputs, args, rho, free_env))
end
```

Defines:
app_to_instance, used in chunk 172.
Uses argtable 170a, Einstance 79, and inputs_of 162a.

`argtable` builds a table mapping input names to expresssions. It does all of the type checking.

170a ⟨* 155⟩+≡ ◁169b 171a▷

```
procedure argtable(inputs, args, rho, free_env)
  t := table()
  every ipt := inputs[i := 1 to *args] do
    t[ipt.name] := case type(ipt.meaning) of {
      "constype" : ⟨make args[i] a constructor input 172⟩
      "integer"  : ⟨make args[i] a signed integer input 171b⟩
      "field" | "string" | "null" :
                   ⟨make args[i] an unsigned integer input 170b⟩
      default    : impossible("input type")
    }
  return t
end
```

Defines:
`argtable`, used in chunk 169.
Uses `constype` 155, `field` 53, and `input` 44a.

Field inputs (typically registers) may have special names associated with particular values. `fieldname_env_for_ipt(ipt)` returns an environment that is empty for non-field inputs, but that contains the special names for field inputs. We can afford to put it first because we check elsewhere that these special names never collide with operand names or identifiers used in equations, and therefore they can't collide with names in `rho`. They have to go first because the decoding code blindly puts all free identifiers into `rho`, even if they are one of these special names. Extended fields don't use the special names. If we look up a name and get a field, we treat it just as if it weren't defined, i.e., we can use it as a free variable. This change lets us use the name of a field as a binding instance in a matching statement. God only knows what other consequences it may have.

ge #29

170b ⟨*make* `args[i]` *an unsigned integer input* 170b⟩≡ (170a)

```
super_simplify(gsubst(args[i], unsigned_arg_f, rho, free_env, args, i, ipt))
```

Uses `gsubst` 117a, `super_simplify` 90a, and `unsigned_arg_f` 171a.

171a ⟨* 155⟩+≡ ◁170a 171c▷
```
procedure unsigned_arg_f(e, rho, free_env, args, i, ipt)
  local fieldrho
  return case type(e) of {
    "string" : {
       fieldrho := fieldname_env_for_ipt(ipt) ||| rho
       (if is_defined(e, fieldrho) & type(lookup(e, fieldrho)) ~== "field" then
          project(lookup(e, fieldrho), "integer")
        else
          new_binding_instance(e, e, "integer", \free_env)
       ) | badarg(args, i, ipt, "integer or field")
    }
    "literal" : project(lookup(e.s, fieldname_env_for_ipt(ipt)), "integer") |
                                           badarg(args, i, ipt, "integer or field")
    "Eapp" : error("Constructor application not allowed; expected integer or field")
  }
end
```
Defines:
`unsigned_arg_f`, used in chunk 170b.
Uses `badarg` 173c, `Eapp` 79, `field` 53, `fieldname_env_for_ipt` 342b, `is_defined` 23a, `literal` 43e, `lookup` 23a, `new_binding_instance` 75b, and `project` 27.

171b ⟨*make* `args[i]` *a signed integer input* 171b⟩≡ (170a)
```
Enarrows(super_simplify(gsubst(args[i], signed_arg_f, rho, free_env, args, i, ipt)),
         ipt.meaning)
```
Uses `Enarrows` 79, `gsubst` 117a, `signed_arg_f` 171c, and `super_simplify` 90a.

171c ⟨* 155⟩+≡ ◁171a 173c▷
```
procedure signed_arg_f(e, rho, free_env, args, i, ipt)
  return case type(e) of {
    "string" : (if is_defined(e, rho) & type(lookup(e, rho)) ~== "field" then
                  project(lookup(e, rho), "integer")
                else
                  new_binding_instance(e, e, "integer", \free_env)
               ) | badarg(args, i, ipt, "integer or field")
    "literal" : badarg(args, i, ipt, "integer or field")
    "Eapp" : error("Constructor application not allowed; expected integer or field")
  }
end
```
Defines:
`signed_arg_f`, used in chunk 171b.
Uses `badarg` 173c, `Eapp` 79, `field` 53, `is_defined` 23a, `literal` 43e, `lookup` 23a, `new_binding_instance` 75b, and `project` 27.

172 ⟨*make* `args[i]` *a constructor input* 172⟩≡ (170a)

```
  case type(x := untable(args[i])) of {
    "Papp" : impossible("Papp as constructor arg")
    "Eapp" :
      if c := cons_named(x.f) & c.type === ipt.meaning then
        app_to_instance(c, x.args, rho, free_env)
      else
        badarg(args, i, ipt, "constructor of type " || ipt.meaning.name ||
                             "; denotes " || c.type.name || ")")
    "string" :
        (if is_defined(x, rho) then
           if x := project(lookup(x, rho), "consop") then
             if type(x) == "input" then
                if x.meaning === ipt.meaning then
                  ipt.name  # stands for an instance, not a pattern
                else
                  badarg(args, i, ipt, " constructor of type " || ipt.meaning.name ||
                                       " (not " || x.meaning.name || ")")
             else
               impossible("consop projected into non-input")
           else
             if lookup(x, rho) === ipt.meaning then
               ⟨possible binding instance x of constructor type ipt.meaning 173a⟩
             else
               &fail
         else
           ⟨possible binding instance x of constructor type ipt.meaning 173a⟩
       ) | badarg(args, i, ipt, " constructor of type " || ipt.meaning.name)
    default : impossible("argument to constructor")
  }
```

Uses `app_to_instance` 169b, `badarg` 173c, `cons_named` 163a, `constructor` 155, `Eapp` 79, `fail` 396, `input` 44a, `is_defined` 23a, `lookup` 23a, `Papp` 39a, `pattern` 53, `project` 27, and `untable` 111c.

173a ⟨*possible binding instance* `x` *of constructor type* `ipt.meaning` 173a⟩≡ (172)
```
if \free_env then
  1(y := Ebinding_instance(x, ipt.meaning, table()),
    if (/free_env[x] := binding_instance(y, ipt.meaning)) |
       (type(free_env[x]) == "binding_instance",
        type(free_env[x].val) == "Ebinding_instance",
        free_env[x].type === free_env[x].val.type === y.type)
    then
      &null
    else
      error("Can't re-use ", x, "; already used as ", type(free_env[x]), "(",
            expimage(free_env[x]), ")",
            "\n --- new value as ", type(y), "(", expimage(y), ") no good"))
else
  &fail
```
Uses `binding_instance` 75b, `Ebinding_instance` 79, `expimage` 98a, `fail` 396, `used` 132a, and `value` 132a.

This was the old code. I don't understand it, but it was causing errors that I didn't want, so I changed it.

173b ⟨*old possible binding instance* `x` *of constructor type* `ipt.meaning` 173b⟩≡
```
if \free_env then
  1(y := Ebinding_instance(x, ipt.meaning, table()),
    (/free_env[x] := binding_instance(y, ipt.meaning)) |
     error("Can't re-use ", x, "; already used as ", expimage(free_env[x])))
else
  &fail
```
Uses `binding_instance` 75b, `Ebinding_instance` 79, `expimage` 98a, `fail` 396, and `used` 132a.

173c ⟨* 155⟩+≡ ◁171c 174a▷
```
procedure badarg(args, i, ipt, expected)
  error(expimage(args[i]), " [", image(args[i]), "]",
        " (", ordinal(i), " arg ", ipt.name, ") ",
        "does not denote ", if any('aeiou', expected) then "an " else "a ", expected)
end
```
Defines:
`badarg`, used in chunks 171 and 172.
Uses `expimage` 98a and `ordinal` 163c.

173d ⟨*insist input lengths match* 173d⟩≡ (169)
```
*inputs = *args |
   error(cons.name, " expects ", *inputs, " arguments, but you gave ", *args)
```

8.3.3 Eliminating instances

This code is applied to every pattern as the last step in conversion to normal form. We don't remember why it's the *last* step, or if it really has to be the last.

174a ⟨* 155⟩+≡ ◁173c 175▷

```
procedure eliminate_instances(e)
  return gsubst(e, eliminate_instances_f)
end
```

Defines:
`eliminate_instances`, used in chunks 72, 121a, and 175.
Uses `eliminate_instances_f` 175 and `gsubst` 117a.

`eliminate_instances_f` simplifies a pattern by removing constructor instances. For some forgotten reason,[1] these transformations are not specified by rewrite rules in `exp.nw`.

The basic idea is that if we have an insance, we can simplify away some predicates on the instance and selection of elements of the instance.

174b ⟨*what we might write if we were doing this with rewrite rules* 174b⟩≡ 174c▷

```
Einstance_tagged(Einstance(c, a), c2, uid) -> if c === c2 then 1 else 0
Einstance_input(Einstance(c, a), c2, name) -> if c === c2 then a[name]
                                              else Efail(expimage(e))
Einstance_tagged(Ebinding_instance(_, _, _),_, _) -> 1 # why?
```

Uses `Ebinding_instance` 79, `Efail` 79, `Einstance` 79, `Einstance_input` 79, `Einstance_tagged` 79, and `expimage` 98a.

The real kicker is dealing with names. If we have a binding instance of a constructor type, we have a name for it. Now, if we want to select an element of that constructor type, we need a name for the element, too, since that element itself becomes a binding instance. The general idea is

174c ⟨*what we might write if we were doing this with rewrite rules* 174b⟩+≡ ◁174b 174d▷

```
Einstance_input(Ebinding_instance(name, _, _), c, fname) ->
   Ebinding_instance(name || "." || c.name || "." || fname)
```

Uses `Ebinding_instance` 79 and `Einstance_input` 79.

but the real truth is in `unwind_instance_inputs`.

We also take this opportunity to make some decisions about latent pattern labels. Latent labels associated with binding instances of constructor types become real, but labels associated with literal instances (`Einstance`) are discarded.

174d ⟨*what we might write if we were doing this with rewrite rules* 174b⟩+≡ ◁174c

```
latent_patlabel(Einstance(_, _)) -> vanishing_latent_patlabel
latent_patlabel(Ewildcard(nam)) -> patlabel(nam, nam)
```

Uses `Einstance` 79, `latent_patlabel` 53, `patlabel` 53, and `vanishing_latent_patlabel` 30a.

[1] One might suspect the reason has something to do with the complicated logic of `do_unwind_instance_inputs`, which doesn't look like it could be implemented with mere rewrite rules.

With rewrite rules, we would enjoy the great convenience that the rewrite engine would simplify the elements of every expression before simplifying the expression itself, since it works bottom-up. Unfortunately, we're using **gsubst**, which works top-down, and the code is therefore more labored than one might like.

Even after all these soothing explanations, Mary may still be afraid of this code. She has good judgment.

175 ⟨* 155⟩+≡ ◁174a 176c▷

```
procedure eliminate_instances_f(e)
  static issued_warning
  return case type(e) of {
    "Einstance_input" : unwind_instance_inputs(e)
    "Einstance_tagged" : {
        e.x := eliminate_instances(e.x)  # simplify children bottom-up
        case type(e.x) of {
          "Ebinding_instance" : 1
          "Einstance" : if e.cons === e.x.cons then 1 else 0
          default : e    # don't let gsubst continue; we did it already
        }
      }
    "latent_patlabel" : {
       e.instance := eliminate_instances(e.instance) # simplify bottom-up
       case type(e.instance) of {
         "Ebinding_instance" : patlabel(e.instance.name, e.instance.name)
         "string"            : e  # no change
         "Einstance_input"   : {⟨latent pattern label of an instance input 176a⟩}
         "Einstance"         : vanishing_latent_patlabel
         "Efail"             : Efail("latent pattern label of " || e.instance.msg)
        default              : impossible("type of latent pattern label")
      }
    }
  }
end
```

Defines:
`eliminate_instances_f`, used in chunks 174a and 176b.

Uses `Ebinding_instance` 79, `Efail` 79, `Einstance` 79, `Einstance_input` 79, `Einstance_tagged` 79, `eliminate_instances` 174a, `gsubst` 117a, `label` 369a, `latent_patlabel` 53, `patlabel` 53, `pattern` 53, `simplify` 89b, `unwind_instance_inputs` 176c, and `vanishing_latent_patlabel` 30a.

That last case has caused much consternation. Originally we weren't sure if a latent pattern label of an instance input should vanish or just stay unchanged. For a long time we just let it stay unchanged. I do believe that's right, since I think the recursive call to `eliminate_instances` handles the cases that need to be handled.

176a ⟨*latent pattern label of an instance input* 176a⟩≡ (175)

```
/issued_warning := 1 &
  warning("Yes, Virginia, there are latent labels of instance inputs: ", expimage(e))
e # change nothing
```

Uses `expimage` 98a.

Let it be noted that at one point we had the following Band-Aid:

176b ⟨*Band-Aid that once covered latent pattern label of an instance input* 176b⟩≡

```
# fail or vanishing?  or other? not sure!
# Should this arm should be like the one above? i.e.:
# This gets us part of the way to binding an a "constructor-typed" operand
# in a constructor application to a location in an instruction stream,
# but it's not quite right.
# I have to think about this some more.
eliminate_instances_f(latent_patlabel(unwind_instance_inputs(e.instance, "")))
```

Uses `arm` 259b, `constructor` 155, `eliminate_instances_f` 175, `fail` 396, `latent_patlabel` 53, `location`, and `unwind_instance_inputs` 176c.

This was before we introduced bottom-up simplification.

Unwinding the instance inputs may mean using an argument table, or it may mean creating a fresh name for a field of a binding instance.

176c ⟨* 155⟩+≡ ◁175 177a▷

```
record binding_instance_var(s)

procedure unwind_instance_inputs(e)
  pushtrace("UNWIND")
  y := case type(x := do_unwind_instance_inputs(e, "")) of {
    "binding_instance_var" : x.s
    default                : x
  } | {poptrace(); fail}
  poptrace()
  return y
end
```

Defines:
`binding_instance_var`, used in chunk 177a.
`unwind_instance_inputs`, used in chunks 175–77.

Uses `do_unwind_instance_inputs` 177a and `fail` 396.

If do_unwind_instance_inputs succeeds, we're guaranteed to have made progress, so we can safely re-apply it.

In the arm for Einstance below, the "false" branch of the conditional returns Efail, because nested constructor applications can produce nonsense (Efail) guarded by conditions that are statically false. For example, in the following pattern, the first disjunct is always true, and the second is always false.

```
  {Y(x) IS Y} => (?Y(x):): <pattern>
| {Y(x) IS C, ... } => (?Y(x):): (?Y(x).C.B:): <pattern>
```

The binding instance (?Y(x).C.B:) is meaningless because Y(x) is C is false.

177a ⟨* 155⟩+≡ ◁176c 177b▷

```
procedure do_unwind_instance_inputs(e, postfix_name)
  type(e) == "Einstance_input" | impossible("unwinding ", expimage(e), " : ", type(e))
  postfix_name := "." || e.cons.name || "." || e.name || postfix_name
  return case type(e.x) of {
    "Ebinding_instance" :
      binding_instance_var(
         binding_instance_input_name(e.x.name || postfix_name, e.x.vart))
    "Einstance" :
         if e.x.cons === e.cons then e.x.argt[e.name] else Efail(expimage(e))
    "Einstance_input" :
      case type(x := unwind_instance_inputs(e.x, postfix_name)) of {
        "binding_instance_var" : x
        default : {
          e := Einstance_input(x, e.cons, e.name)
          do_unwind_instance_inputs(e, postfix_name) | e  # guaranteed to terminate
        }
      }
  }
end
```

Defines:
do_unwind_instance_inputs, used in chunk 176c.
Uses binding_instance_input_name 177b, binding_instance_var 176c, Ebinding_instance 79, Efail 79, Einstance 79, Einstance_input 79, expimage 98a, and unwind_instance_inputs 176c.

177b ⟨* 155⟩+≡ ◁177a 178a▷

```
procedure binding_instance_input_name(name, vart)
  /vart[name] := fresh_variable(name)
  return vart[name]
end
```

Defines:
binding_instance_input_name, used in chunk 177a.
Uses fresh_variable 151b.

8.4 Machine-independent support for encoding procedures

The deal here is to transform a pattern with tag conditions into a nested case statement.

178a ⟨* 155⟩+≡ ◁177b 178b▷
```
record Stagcase(x, type, arms)          # CASE x : type OF arms END
```
Defines:
`Stagcase`, used in chunks 81a, 84b, 90c, 106a, 115b, 121c, 129b, 179a, 183b, 238a, 299, 317e, and 330a.

`x` is the instance from which we will check the tag, `type` is the constructor type (of `x`), and `arms` is a table mapping each constructor of the type to a pattern (or another case statement).

We're going to destroy the pattern, so we use a fresh one.

178b ⟨* 155⟩+≡ ◁178a 179a▷
```
procedure pattern_to_case(p)
  ⟨error if p has no disjuncts 185c⟩
  return do_pattern_to_case(freshen_disjuncts(p))
end
```
Defines:
`pattern_to_case`, used in chunks 182b, 221, and 294.
Uses `do_pattern_to_case` 179a and `freshen_disjuncts` 78b.

To produce a case statement, pull out the tags, then recurse. It's possible that some constructors have been discarded between the time we defined pattern `p` and the time we're actually doing the pattern-to-case conversion. For example, on the Pentium, we specify all addressing modes, but when generating instructions for the 32-bit instructions, we discard unused constructors (e.g., `Index8`):

178c ⟨*example* 178c⟩≡
```
constructors
  Index8  base,index: Eaddr ...
  Index32 base,index: Eaddr ...
  Add Eaddr, r is ...
discard Index8
```
Uses `constructors` 30a.

We drop disjuncts with conditions that say an instance is tagged with a discarded constructor. Since `s.arms[cons]` is non-null only for kept constructors `cons`, we can use that test to drop such disjuncts.

179a ⟨* 155⟩+≡ ◁178b 179b▷

```
procedure do_pattern_to_case(p)
  local rep             # representative tag condition
  if rep := tag_test_in_every_disjunct(p) then {
    s := Stagcase(rep.x, rep.cons.type, table())
    every s.arms[kept_constructors(s.type)] := pattern([])
    every d := !p.disjuncts do {
      if type(c := !d.conditions) == "Einstance_tagged" & c.uid = rep.uid &
         exps_eq(rep.x, c.x)
      then {
        delete(d.conditions, c)
        put((\s.arms[c.cons]).disjuncts, d)
      } else
        impossible("Mislaid a tag condition on ", expimage(rep.x))
    }
    every c := key(s.arms) do s.arms[c] := do_pattern_to_case(s.arms[c])
    return s
  } else if rep := tag_test_in_any_disjunct(p) then {
     PPxwrite(PPnew(&errout), "Can't eliminate tag condition on ", ppexpimage(rep.x),
                       " $t$ofrom pattern $c", ppexpimage(p))
     impossible("Report a bug in the toolkit")
  } else
    return simplify(p)
end
```

Defines:
`do_pattern_to_case`, used in chunk 178b.

Uses `Einstance_tagged` 79, `expimage` 98a, `exps_eq` 82c, `kept_constructors` 31d, `pattern` 53, `ppexpimage` 98d, `simplify` 89b, `Stagcase` 178a, `tag_test_in_any_disjunct` 179b, and `tag_test_in_every_disjunct` 180a.

179b ⟨* 155⟩+≡ ◁179a 180a▷

```
procedure tag_test_in_any_disjunct(p)
  suspend type(rep := !\(!p.disjuncts).conditions) == "Einstance_tagged" & rep
end
```

Defines:
`tag_test_in_any_disjunct`, used in chunk 179a.

Uses `Einstance_tagged` 79.

180a ⟨* 155⟩+≡ ◁179b 180b▷

```
procedure tag_test_in_every_disjunct(p)
  every type(rep := !\p.disjuncts[1].conditions) == "Einstance_tagged" do
    if tag_test_not_in_disjunct(rep, !p.disjuncts) then &null else return rep
  fail
end

procedure tag_test_not_in_disjunct(rep, d)
  if type(c := !\d.conditions) == "Einstance_tagged" & c.uid = rep.uid &
     exps_eq(rep.x, c.x) then fail
  else return
end
```

Defines:
  tag_test_in_every_disjunct, used in chunk 179a.
Uses Einstance_tagged 79, exps_eq 82c, and fail 396.

The structure of an encoding procedure is complex because of the number of conditions to be checked. Here is a sketch of the conditions:

`iwidth`	input-width conditions for the constructor
`conds`	conditions for a branch to be taken
`cknown`	conditions that must hold before `conds` can be evaluated (e.g., relocatable addresses are known)
`fits`	field-width conditions for fields to be emitted, and conditions that attempts to narrow actually fit
`fknown`	conditions that must hold before fields can be emitted or `fits` can be evaluated (again, relocatable addresses). `fknown` never repeats any conditions already in `cknown`, because its use is always guarded by `cknown`

The `*known` conditions influence closure creation, the `*width` conditions are error checks only, and `conds` is used for branch selection.

Rather than use C templates, we use a language-independent representation of control flow.

180b ⟨* 155⟩+≡ ◁180a 181b▷

```
record Sstms(stmts)                          # statement sequence
```

A constructor branch is chosen conservatively, except for the last branch, which gets special treatment. While an early branch is taken iff its conditions are known to be true, a late branch is taken iff its conditions are not known to be false.

181a ⟨*early-branch.s* 181a⟩≡
```
if (cknown && conds) {
  if (fknown) {
    fits_aborts;        # abort if fits not satisfied
    emit_fields;
  } else
    create_closure;
}
```

The Icon branch procedures add their branches to a list of arms.

181b ⟨* 155⟩+≡ ◁180b 182a▷
```
procedure early_branch(arms, cknown, conds, fknown, fits_and_emit, closure)
  return put(arms, Sguarded(Eand(cknown, conds),
                     Sif([Sguarded(fknown, fits_and_emit), Sguarded(1, closure)])))
end
```
Defines:
early_branch, used in chunk 184.
Uses Eand 79, Sguarded 79, and Sif 79.

181c ⟨*late-branch.s* 181c⟩≡
```
if (cknown) {
  if (conds) {
    if (fknown) {
      fits_aborts;
      emit_fields;
    } else {
      unchecked_closure;
    }
  } else {
    condition_failure;
  }
} else {
  checked_closure;
}
```

182a ⟨* 155⟩+≡ ◁181b 182b▷

```
procedure last_branch(arms, cknown, conds, fknown, fits_and_emit, uclosure,
                      condfail, cclosure)
  every put(arms,
            Sguarded(cknown, Sif(
                [Sguarded(conds,
                       Sif([Sguarded(fknown, fits_and_emit), Sguarded(1, uclosure)])),
                 Sguarded(1, condfail)])) |
            Sguarded(1, cclosure))
  return arms
end
```

Defines:
  last_branch, used in chunk 184.
Uses Sguarded 79 and Sif 79.

182b ⟨* 155⟩+≡ ◁182a 182c▷

```
procedure emitter_body(cons)
  s := []
  every f := inputs_of(cons, "field").meaning &
        not member(unchecked_fields, f) & fwidth(f) < wordsize do
    put(s, Sguarded(Enot(Efitsu(literal(f.name), fwidth(f))),
                    Sfail("field " || f.name || " does not fit in " ||
                          fwidth(f) || " unsigned bits")))
  every f := symtab[inputs_of(cons, "integer").name] &
        not member(unchecked_fields, f) & fwidth(f) < wordsize do
    put(s, Sguarded(Enot(Efitss(literal(f.name), fwidth(f))),
                    Sfail("field " || f.name || " does not fit in " ||
                          fwidth(f) || " signed bits")))
  put(s, Sguarded(1, case_to_emitter(pattern_to_case(
              subst_tab(crhs(cons), parmtab(cons), 1)), cons)))
  return super_simplify(Sif(s))
end
```

Defines:
  emitter_body, used in chunk 204a.
Uses case_to_emitter 183b, crhs 164, Efitss 79, Efitsu 79, Enot 79, field 53, fwidth 112d, inputs_of 162a, literal 43e, parmtab 183a, pattern_to_case 178b, Sfail 182c, Sguarded 79, Sif 79, subst_tab 87a, super_simplify 90a, symtab 30a 353a, unchecked_fields 340f, and wordsize 36c.

Sfail is like printf

182c ⟨* 155⟩+≡ ◁182b 183a▷

```
record Sfail(fmt, a1, a2, a3)
```

Defines:
  Sfail, used in chunks 81a, 106b, 115b, 121c, 129c, 182b, 184, 190a, 199b, 205a, 211c, 318a, and 330b.

parmtab produces a substitution table that does the right thing with the parameters to a C emission procedure.

183a ⟨* 155⟩+≡ ◁182c 183b▷

```
procedure parmtab(cons)
  t := table()
  every ipt := inputs_of(cons) do
    t[ipt.name] := case type(ipt.meaning) of {
      "constype" : ipt.name # name stands for an instance
      "field"    : ipt.name
      "integer"  : Enarrows(ipt.name, ipt.meaning)
      "string"   : Eforce(ipt.name)
      "null"     : ipt.name
      default    : impossible("input type")
    }
  return t
end
```

Defines:
parmtab, used in chunks 182b and 294.
Uses constype 155, Eforce 79, Enarrows 79, field 53, input 44a, and inputs_of 162a.

8.4.1 Emission

183b ⟨* 155⟩+≡ ◁183a 185a▷

```
procedure case_to_emitter(p, cons)
  local cknown, conds, fknown, fits, condition_failure_msg
  case type(p) of {
    "Stagcase"   : {
       every c := kept_constructors(p.type) do
         p.arms[c] := case_to_emitter(p.arms[c], cons)
       return p
    }
    "pattern" : {⟨turn pattern p into emitter and return it 184⟩}
  }
end
```

Defines:
case_to_emitter, used in chunk 182b.
Uses kept_constructors 31d, pattern 53, and Stagcase 178a.

We go ahead and permit encoding of patterns with no disjuncts, because they might represent one case in a statement where other cases are valid. This situation holds on machines like the 68k, which forbid certain addressing modes for certain instructions. We issue a warning message, and if sonmebody tries the mode at encoding time they get slapped with an error message.

Note that we just set the `patlabel` offsets, not worrying about whether they're shared (although by the invariant they shouldn't be). It shouldn't matter because the offsets are then used immediately.

184 ⟨*turn pattern* `p` *into emitter and return it* 184⟩≡ (183b)
```
  remove_duplicate_conditions(p, cons)
  if *p.disjuncts = 0 then {
    ⟨possibly warn about bad encoding for cons 185d⟩
    return Sfail("impossible encoding (no disjuncts) --- perhaps a bad address mode?")
  }
  s := Sif([])
  ⟨make condition_failure_msg complain of undecided branch or unsatisfied conditions 185b⟩
  while d := get(p.disjuncts) do {
    set_patlabel_offsets(d)
    d := gsubst(d, Epatlabel_to_Epc)
    ds_conds := extract_conditions(d, cons)
    cknown := ds_conds[1]; conds := ds_conds[2]
    fknown := ds_conds[3]; fits :=  ds_conds[4]
    sanitize_sequents(d, fits)
    fwe := Sif([])
    every c := !\fits do
      put(fwe.arms, Sguarded(Enot(c), widthfailure(c)))
    put(fwe.arms, Sguarded(1, disjunct_to_emission(d)))
    if *p.disjuncts > 0 then
      early_branch(s.arms, cknown, conds, fknown, fwe, Sclosure(d))
    else
      last_branch (s.arms, cknown, conds, fknown, fwe, Sclosure(d),
                   Sfail(condition_failure_msg || " for constructor " || cons.name),
                   Sclosure(d, conds))
  }
  return s
```
Uses `address` 369a, `constructor` 155, `disjunct_to_emission` 186b, `early_branch` 181b, `Enot` 79, `Epatlabel_to_Epc` 185a, `extract_conditions` 190b, `gsubst` 117a, `last_branch` 182a, `remove_duplicate_conditions` 189a, `sanitize_sequents` 186a, `Sclosure` 185f, `set_patlabel_offsets` 57c, `Sfail` 182c, `Sguarded` 79, `Sif` 79, and `widthfailure` 190a.

185a ⟨* 155⟩+≡ ◁183b 185e▷
```
procedure Epatlabel_to_Epc(x)
  if type(x) == "Epatlabel" then
    return binop(the_global_pc, "+", x.l.offset)
end
```
Defines:
Epatlabel_to_Epc, used in chunks 184, 258a, and 300a.
Uses binop 110a, Epatlabel 79, and the_global_pc 17d.

185b ⟨*make* condition_failure_msg *complain of undecided branch or unsatisfied conditions* 185b⟩≡ (184)
```
condition_failure_msg :=
  if *p.disjuncts > 1 then "Can't decide on branch"
  else "Conditions not satisfied"
```
Uses branch 155.

185c ⟨*error if* p *has no disjuncts* 185c⟩≡ (178b)
```
if *p.disjuncts = 0 then
  error("Output pattern for constructor ", cons.name, " can never match anything.\n",
        "\tCould you have written a bad conjunction?")
```
Uses constructor 155 and pattern 53.

185d ⟨*possibly warn about bad encoding for* cons 185d⟩≡ (184)
```
{ /warned_no_disjuncts := table()
  if /warned_no_disjuncts[cons] := 1 then
    warning("constructor ", cons.name,
            " has an encoding with no matches -- maybe a bad address mode?")
}
```
Uses address 369a, constructor 155, and warned_no_disjuncts 185e.

185e ⟨* 155⟩+≡ ◁185a 185f▷
```
global warned_no_disjuncts
```
Defines:
warned_no_disjuncts, used in chunk 185d.

185f ⟨* 155⟩+≡ ◁185e 186a▷
```
record Sclosure(disjunct, conditions, creation)
```
Defines:
Sclosure, used in chunks 115b, 184, 195b, 204c, 318b, and 330c.
Uses disjunct 53.

185g ⟨*argument descriptions* 185g⟩≡
```
expargs disjunct conditions creation.
```
Uses disjunct 53.

Sanitize d's sequents, possibly adding conditions to fits.

186a ⟨* 155⟩+≡ ◁185f 186b▷
```
procedure sanitize_sequents(d, fits)
  l := []
  every put(l, sanitize_for_output(!d.sequents, fits))
  d.sequents := l
end
```
Defines:
sanitize_sequents, used in chunks 184 and 300a.
Uses sanitize_for_output 68a.

sequent_to_stoken can't know the offsets, so disjunct_to_emission fills them in.

186b ⟨* 155⟩+≡ ◁186a 187▷
```
procedure disjunct_to_emission(d)
  s := Semit([])
  every seq := !d.sequents & type(seq) == "sequent" do
    put(s.x, sequent_to_Stoken(seq))
  ⟨fill in offsets in s 186c⟩
  return s
end
```
Defines:
disjunct_to_emission, used in chunks 184, 196b, 199b, and 205.
Uses Semit 79, sequent 53, and sequent_to_Stoken 187.

186c ⟨*fill in offsets in* s 186c⟩≡ (186b)
```
n := 0
every ss := !s.x do { ss.offset := n; n +:= ss.n }
```

187 ⟨* 155⟩+≡ ◁186b 188a▷

```
procedure sequent_to_Stoken(s)
  v := &null
  o := start_overlap_check()
  every c := !s.constraints & x := case type(c) of {
    "constraint":
        if c.lo + 1 = c.hi then {
            add_overlap_field(o, c.field)
            if 0 <= c.lo < 2^fwidth(c.field) then
               c.lo
            else {
               warning("Field value ", c.lo, " exceeds width of field ", c.field.name);
               Eslice(c.lo, 0, fwidth(c.field))
            }
        } else if c.lo < c.hi then {
              warning("Field ", c.field.name, " is underconstrained by ",
                     constraintimage(c), "; no value output")
              &fail
        }
    "fieldbinding": {
       add_overlap_field(o, c.field)
       if member(guaranteed_fields, c.field) # | not member(unchecked_fields, c.field)
       then
         c.code
       else
         Eslice(c.code, 0, fwidth(c.field))
    }
  } & y := emitshift(x, c.field.lo) do
    v := Eorb(\v, y) | y
  return Stoken(\v | 0,
     if s.class.size % emit_unit_bits = 0 then s.class.size / emit_unit_bits
     else error("tokens are emitted in units of ", token_unit_bits,
                ", but some pattern is ", s.class.size, " bits wide"))
end
```

Defines:
sequent_to_Stoken, used in chunk 186b.

Uses add_overlap_field 188a, constraint 53, constraintimage 357a 357a, constraints 132a, emitshift 188c, emit_unit_bits 16a, Eorb 79, Eslice 79, fail 396, field 53, fieldbinding 53, fwidth 112d, guaranteed_fields 340f, pattern 53, start_overlap_check 188a, Stoken 79, unchecked_fields 340f, and value 132a.

188a ⟨* 155⟩+≡ ◁187 188c▷

```
record overlap_check(fields, loset, hiset)
procedure start_overlap_check()
  return overlap_check(set(), set(), set())
end

procedure add_overlap_field(o, f)
  if overlaps(o.loset, o.hiset, f.lo, f.hi) then {
    ⟨issue error message about overlapping fields 188b⟩
  } else {
    insert(o.fields, f)
    addinterval(o.loset, o.hiset, f.lo, f.hi)
  }
  return o
end
```

Defines:
  `add_overlap_field`, used in chunk 187.
  `start_overlap_check`, used in chunk 187.
Uses `addinterval` 276a and `overlaps` 276b.

188b ⟨*issue error message about overlapping fields* 188b⟩≡ (188a)

```
every g := !o.fields do
  if f.hi <= g.lo | g.hi <= f.lo then
    &null
  else
    error("Cannot use overlapping fields ", f.name, " and ", g.name, " in the same token")
impossible("some fields overlap, but I can't tell which ones")
```

Uses `token` 345a.

188c ⟨* 155⟩+≡ ◁188a 189a▷

```
procedure emitshift(x, n)
  return if n = 0 then x
         else if \simplify_emits then Eshift(x, n)
         else Eshift(Enosimp(super_simplify(x)), n)
end
```

Defines:
  `emitshift`, used in chunk 187.
Uses `Enosimp` 79, `Eshift` 79, `simplify_emits` 16a, and `super_simplify` 90a.

Delete p's disjuncts with duplicate conditions, issuing suitable warnings.

189a ⟨* 155⟩+≡ ◁188c 189c▷

```
procedure remove_duplicate_conditions(p, cons)
  local l
  l := []
  every i := *p.disjuncts to 1 by -1 do
    if j := 1 to i-1 &
       same_conditions(p.disjuncts[i].conditions, p.disjuncts[j].conditions) then
      ⟨warn of redundant disjunct 189b⟩
    else
      push(l, p.disjuncts[i])
  if *l < *p.disjuncts then p.disjuncts := l
  return
end
```

Defines:
remove_duplicate_conditions, used in chunks 184 and 299.
Uses same_conditions 189c.

189b ⟨*warn of redundant disjunct* 189b⟩≡ (189a)

```
warning("Pattern on right-hand side of constructor ", cons.name,
        " has redundant disjuncts ", expimage(p.disjuncts[j]), " and ",
        expimage(p.disjuncts[i]), ".\tI'll use the first one for encoding")
```

Uses constructor 155 and expimage 98a.

189c ⟨* 155⟩+≡ ◁189a 190a▷

```
procedure same_conditions(c1, c2)
  if /c1 & /c2 then return
  else if /c1 | /c2 then fail
  else if *c1 = *c2 then {
    c := copy(c1)
    every insert_condition(c, !c2)
    if *c > *c1 then fail
    else return
  } else fail
end
```

Defines:
same_conditions, used in chunk 189a.
Uses fail 396 and insert_condition 123c.

190a ⟨* 155⟩+≡ ◁189c 190b▷

```
procedure widthfailure(c)
  c.n < wordsize | impossible("test to fit in word: ", expimage(c))
  case type(c) of {
    "Efitsu" : {f := "0x%x"; s := "unsigned"}
    "Efitss" : {f := "%d"; s := "signed"}
    default  : impossible("width condition")
  }
  return Sfail("`" || expimage(c.x) || "' = " || f || " won't fit in " ||
               c.n || " " || s || " bits.", c.x)
end
```

Defines:
widthfailure, used in chunks 184 and 192b.
Uses Efitss 79, Efitsu 79, expimage 98a, Sfail 182c, and wordsize 36c.

8.4.2 Conditions

extract_conditions returns the list [cknown, conds, fknown, fits] ; the four elements contain the conditions associated with disjunct d.

190b ⟨* 155⟩+≡ ◁190a 191a▷

```
procedure extract_conditions(d, cons)
  if *\d.conditions > 0 then {
    conds := d.conditions
    ⟨from conds, compute conds_narrows_ok 192a⟩
    if *conds_narrows_ok > 0 then
      conds ++:= conds_narrows_ok
    cknown := known_conditions(d.conditions)
  } else
    conds := cknown := 1
  fknown := known_conditions(d.sequents)
  ⟨strip fknown of conditions already in cknown 191b⟩
  ⟨from d, compute fits 191c⟩
  return [ cknown, conds, fknown, fits ]
end
```

Defines:
extract_conditions, used in chunks 184 and 300a.
Uses known_conditions 191a.

191a ⟨* 155⟩+≡ ◁190b 192b▷
```
procedure known_conditions(e)
  local known
  known := set()
  every ff := subterms_matching(e, "Eforce") &
        cc := super_simplify(Eforceable(ff.x)) do
    insert_condition(known, cc)
  if subterms_matching(e, "Epc") then
    insert_condition(known, Epc_known())
  delete(known, 1)
  return if *known = 0 then 1 else known
end
```
Defines:
  known_conditions, used in chunk 190b.
Uses Eforce 79, Eforceable 79, Epc 79, Epc_known 79, insert_condition 123c, subterms_matching 82a, and super_simplify 90a.

191b ⟨*strip* fknown *of conditions already in* cknown 191b⟩≡ (190b)
```
if type(cknown) == "set" then
  every ff := !fknown do
    if exps_eq(ff, !cknown) then
      delete(fknown, ff)
if *fknown = 0 then fknown := 1
```
Uses exps_eq 82c.

At this point, we know that field and extended inputs satisfy the appropriate width conditions.

191c ⟨*from* d*, compute* fits 191c⟩≡ (190b)
```
fits := set()
every ff := subterms_matching(d.sequents, "fieldbinding") &
      not member(unchecked_fields, ff.field) do
  if not known_to_fit(input_fitsu, ff.code, cons, fwidth(ff.field)) then
    insert_width_condition(fits, Efitsu(ff.code, fwidth(ff.field)))
every ee := subterms_matching(d.sequents, "Enarrowu") do
  if not known_to_fit(input_fitsu, ee.x, cons, ee.n) then
    insert_width_condition(fits, Efitsu(ee.x, ee.n))
every ee := subterms_matching(d.sequents, "Enarrows") do
  if not known_to_fit(input_fitss, ee.x, cons, ee.n) then
    insert_width_condition(fits, Efitss(ee.x, ee.n))
```
Uses Efitss 79, Efitsu 79, Enarrows 79, Enarrowu 79, field 53, fieldbinding 53, fwidth 112d, input_fitss 193a, input_fitsu 193a, insert_width_condition 192b, known_to_fit 192c, subterms_matching 82a, and unchecked_fields 340f.

We lost earlier because the solver converted the condition: `v = v[0:15]!` to

Enarrows(v, 16) = v[0:15].

Without a check for the success of the narrow, this condition becomes a tautology, with the result that we were always taking an incorrect branch of the SPARC `set` constructor. By seeking out narrows in conditions, we make sure to include the correct check (although we continue to emit C code for the tautology, which is annoying).

192a ⟨*from* `conds`*, compute* `conds_narrows_ok` 192a⟩≡ (190b)
```
conds_narrows_ok := set()
every ee := subterms_matching(conds, "Enarrowu") do
  if not known_to_fit(input_fitsu, ee.x, cons, ee.n) then
    insert_width_condition(conds_narrows_ok, Efitsu(ee.x, ee.n))
every ee := subterms_matching(conds, "Enarrows") do
  if not known_to_fit(input_fitss, ee.x, cons, ee.n) then
    insert_width_condition(conds_narrows_ok, Efitss(ee.x, ee.n))
```
Uses `Efitss` 79, `Efitsu` 79, `Enarrows` 79, `Enarrowu` 79, `input_fitss` 193a, `input_fitsu` 193a, `insert_width_condition` 192b, `known_to_fit` 192c, and `subterms_matching` 82a.

192b ⟨* 155⟩+≡ ◃191a 192c▹
```
procedure insert_width_condition(fits, c)
  if (x := super_simplify(c)) === 0 then
    error(widthfailure(c).fmt)
  else
    insert_condition(fits, x)
  return
end
```
Defines:
`insert_width_condition`, used in chunks 191c and 192a.
Uses `insert_condition` 123c, `super_simplify` 90a, and `widthfailure` 190a.

Either inputs or instances can be known to fit. The test on strings is valid *only when all free variables represent inputs of constructor* `cons`. In particular, it is valid when patterns are prepared for emission. The test on instance inputs is, of course, always valid. `test` is `input_fitsu` or `input_fitss`.

192c ⟨* 155⟩+≡ ◃192b 193a▹
```
procedure known_to_fit(test, code, cons, width)
  return case type(code) of {
    "string"          : test(code, cons, width)
    "Einstance_input" : known_to_fit(test, code.name, code.cons, width)
  }
end
```
Defines:
`known_to_fit`, used in chunks 96b, 191c, 192a, and 200b.
Uses `Einstance_input` 79.

193a ⟨* 155⟩+≡ ◁192c 194a▷

```
procedure input_fitsu(name, cons, width)
  return width >= wordsize |
    ((ipt := inputs_of(cons)).name == name & case type(ipt.meaning) of {
      "field"   : fwidth(ipt.meaning) <= width
      "integer" : ipt.meaning <= width
    })
end

procedure input_fitss(name, cons, width)
  return width >= wordsize |
    ((ipt := inputs_of(cons)).name == name & case type(ipt.meaning) of {
      "integer" : ipt.meaning <= width
      "field"   : fwidth(ipt.meaning) < width
    })
end
```

Defines:
`input_fitss`, used in chunks 96b, 191c, and 192a.
`input_fitsu`, used in chunks 96b, 191c, and 192a.
Uses `field` 53, `fwidth` 112d, `inputs_of` 162a, and `wordsize` 36c.

8.5 Support for encoding procedures (in C)

8.5.1 C type declarations for instances

A user-defined constructor type is represented by T`Instance`, a C type that stores an instance of a constructor type T. It contains a union of all possible constructors of type T; for each one it stores the instance. The header `h` points to a statically allocated record that contains, among other information, a tag that identifies not just the constructor, but its branch. Here's the template:

193b ⟨*instance-type.t* 193b⟩≡

```
typedef struct %{name}_instance {$t
int tag;
union {$t%constructors$b
} u;$b
} %{name}_Instance;
```

Uses `constructors` 30a.

And here's the code the does the emission.

194a ⟨* 155⟩+≡ ◁193a 194b▷
```
procedure emit_instance_type(pp, ct)
  local constructors
  enforce_instance(ct)
  constructors := []
  every put(constructors, input_record_for(kept_constructors(ct)))
  emit_template(pp, "instance-type.t", "name", ct.name, "constructors", constructors)
  return
end
```
Defines:
emit_instance_type, used in chunk 218b.
Uses constructors 30a, enforce_instance 162c, input_record_for 194b, and kept_constructors 31d.

There's a trick to the layout of an input record; we use bit fields to hold field inputs, and we want to make them consecutive so they have a decent chance of packing. To achieve this effect, we run through the inputs twice.

194b ⟨* 155⟩+≡ ◁194a 195a▷
```
procedure input_record_for(cons, struct_name)
  local pp, ipt
  pp := []
  put(pp, "$nstruct {$t");
  every ipt := inputs_of(cons) do       ## emit bit fields
    case type(ipt.meaning) of {
      "field"   : put(pp, "$nunsigned " || ipt.name || ":" || fwidth(ipt.meaning)||";")
      "integer" : put(pp, "$nint "      || ipt.name || ":" || ipt.meaning       ||";")
    }
  every ipt := inputs_of(cons) do       ## emit other inputs
    case type(ipt.meaning) of {
      "null"     : put(pp, "$nint "   ||                              ipt.name ||";")
      "string"   : put(pp, "$nRAddr " ||                              ipt.name ||";")
      "constype" : put(pp, "$n" || ipt.meaning.name || "_Instance  " || ipt.name ||";")
      "field" | "integer" : &fail
      default : impossible("input meaning")
    }
  put(pp, "$b$n} " || (\struct_name | cons.name) || ";")
  return pp
end
```
Defines:
input_record_for, used in chunk 194a.
Uses constype 155, fail 396, field 53, fwidth 112d, input 44a, and inputs_of 162a.

While we're playing the input game, we use a similar technique to compute a list of argument declarations.

195a ⟨* 155⟩+≡ ◁194b 195b▷

```
procedure arg_decls(cons)
  l := []
  every ipt := inputs_of(cons) do
    put(l, case type(ipt.meaning) of {
            "null"     : "int"
            "string"   : "RAddr"
            "constype" : ipt.meaning.name || "_Instance"
            "field"    : unsigned_type(fwidth(ipt.meaning))
            "integer"  : "int"
            default    : impossible("arg_decls input")
          } || " " || ipt.name)
  return if *l = 0 then "void" else commaseparate(l)
end
```

Defines:
`arg_decls`, used in chunks 203, 204a, and 234.
Uses `constype` 155, `field` 53, `fwidth` 112d, `input` 44a, `inputs_of` 162a, `RAddr` 397a, and `unsigned_type` 320c.

8.6 C support for closures

Closures are numbered within a particular constructor, except for the first.

195b ⟨* 155⟩+≡ ◁195a 196a▷

```
procedure emit_original_closure_functions(pp, cons, b)
  local suffix
  every cl := subterms_matching(b, "Sclosure") do
    emit_original_closure_function(pp, cons, cons.name || ⟨suffix with update 195c⟩, cl)
  return
end
```

Defines:
`emit_original_closure_functions`, used in chunk 15e.
Uses `emit_original_closure_function` 196a, `Sclosure` 185f, and `subterms_matching` 82a.

195c ⟨*suffix with update* 195c⟩≡ (195b)

```
(if /suffix then (suffix := 1, "") else "_" || (suffix +:= 1))
```

The idea with a closure is to ferret out all the free variables and stuff them into a custom closure. The trick is, we don't want to use *instance*-valued variables; we want the components of the instance. Step 1, then, is to grab all the instance selections, noting the instances that we're not interested in. Step 2 will add all the free variables. The things to save, are the selections and free variables that aren't selected from.

Everything to save is put in the closure. Equivalent expressions are to be saved only once, so we use a table mapping things to be saved to their names in the closure. `closurenametab` computes that table.

The definition of a closure type requires the inverse of that table.

196a ⟨* 155⟩+≡ ◁195b 197c▷

```
procedure emit_original_closure_function(pp, cons, name, cl)
  local selections, selected, free, save, upc
  ⟨make selections, selected, free hold selections, things selected from, and free variables 197b⟩
  nt := closurenametab(selections ++ free -- selected)
  mt := meaningtab(nt, cons)
  emit_original_closure_typedef(pp, name, cons, mt)
  emit_original_closure_relocfn(pp, name, cons, mt)
  emit_original_closure_function_def(pp, name, cl, nt)
  ⟨make cl.creation to create closure and emit placeholder 196b⟩
  return
end
```

Defines:
`emit_original_closure_function`, used in chunk 195b.

Uses `closurenametab` 197c, `emit_original_closure_function_def` 199b, `emit_original_closure_relocfn` 199a, `emit_original_closure_typedef` 198b, and `meaningtab` 200b.

196b ⟨*make* `cl.creation` *to create closure and emit placeholder* 196b⟩≡ (196a)

```
⟨make l a list of assignments by using the name table nt 197a⟩
upc := if subterms_matching(\cl.conditions | cl.disjunct, "Epc", "Epc_known") then 1
       else 0
cl.creation := Sstmts([
  literal(template_to_list("create-closure.t", "name", name, "clofun", name || "_app",
                           "uses-pc", upc, "save", l)),
  disjunct_to_emission(place_holder(cl.disjunct))])
```

Uses `disjunct` 53, `disjunct_to_emission` 186b, `Epc` 79, `Epc_known` 79, `literal` 43e, `place_holder` 70a, `Sstmts` 79, and `subterms_matching` 82a.

196c ⟨*create-closure.t* 196c⟩≡

```
$t{ %{name}_Closure _c;
static struct closure_header _h = $t
{ %{clofun}, %{name}_relocfn, %{uses-pc}, sizeof *_c };$b
_c = (%{name}_Closure) mc_create_closure_here(sizeof *_c, &_h);
%{save}/* this line intentionally left blank */$b
}
```

Uses `line` 345a and `mc_create_closure_here` 394a.

197a ⟨*make* `l` *a list of assignments by using the name table* `nt` 197a⟩≡ (196b)

```
l := []
s := set()
every e := key(nt) & not member(s, nt[e]) do {
  insert(s, nt[e])
  put(l, "_c->v." || nt[e] || " = " || pretty(e) || ";$n")
}
s := &null # enable garbage collection
```

Uses `pretty` 16a.

197b ⟨*make* `selections`, `selected`, `free` *hold selections, things selected from, and free variables* 197b⟩≡ (196a)

```
every selections | selected | free := set()
every s := subterms_matching(cl.disjunct | \cl.conditions, "Einstance_input") do {
  insert(selections, s)
  insert(selected, s.x)
}
every insert(free, free_variables(cl.disjunct | \cl.conditions))
```

Uses `disjunct` 53, `Einstance_input` 79, `free_variables` 82b, and `subterms_matching` 82a.

197c ⟨* 155⟩+≡ ◁196a 198b▷

```
procedure closurenametab(save)
  local namecounts, saved, name
  every namecounts | t := table()
  saved := set()
  every e := !save do
    if eprime := !saved & exps_eq(e, eprime) then
      t[e] := t[eprime]
    else {
      insert(saved, e)
      ⟨set name to the proper name of the field for e 198a⟩
      t[e] := name
    }
  return t
end
```

Defines:
`closurenametab`, used in chunk 196a.

Uses `exps_eq` 82c.

198a ⟨*set* `name` *to the proper name of the field for* `e` 198a⟩≡ (197c)
```
name := case type(e) of {
  "string" : e
  "Einstance_input" : e.name
  default : impossible("type of saved exp")
}
if /namecounts[name] then
  namecounts[name] := 1
else
  name ||:= "__" || (namecounts[name] +:= 1)
```
Uses `Einstance_input` 79.

To compute the closure type, we pack the names; big ones first, then little ones. First we invert the table, so that the names are the unique keys in `u`.

198b ⟨* 155⟩+≡ ◁197c 199a▷
```
procedure emit_original_closure_typedef(pp, name, cons, t)
  l := []
  every fname := key(t) do
    put(l, case type(t[fname]) of {
             "null"     : "$cint "   || fname || ";"
             "string"   : "$cRAddr " || fname || ";"
             "constype" : impossible("failed to eliminate an instance")
           })
  every fname := key(t) do
    put(l, case type(t[fname]) of {
             "field"   : "$cunsigned " || fname || ":" || fwidth(t[fname]) || ";"
             "integer" : "$cint "      || fname || ":" || t[fname]         || ";"
           })
  emit_template(pp, "closure-type.t", "name", name, "decls", l)
end
```
Defines:
`emit_original_closure_typedef`, used in chunk 196a.
Uses `constype` 155, `field` 53, and `fwidth` 112d.

198c ⟨*closure-type.t* 198c⟩≡
```
typedef struct %{name}_closure {$t
ClosureHeader h;
ClosureLocation loc;
struct { $t${%decls $b$c$}} v;$b
} *%{name}_Closure;
```
Uses `ClosureHeader` 392a and `ClosureLocation` 392a.

199a ⟨* 155⟩+≡ ◁198b 199b▷
```
procedure emit_original_closure_relocfn(pp, name, cons, t)
  local calls
  calls := []
  every fname := key(t) & type(t[fname]) == "string" do
     put(calls, template_to_list("reloc-call.t", "irec", "v", "input", fname))
  emit_template(pp, "constructor-labels.t", "ptrtype", name || "_Closure",
                               "name", name, "calls", calls)
  return
end
```
Defines:
emit_original_closure_relocfn, used in chunk 196a.
Uses constructor 155 and input 44a.

The emitter code in Cexp assumes that the closure is stored in _c.

199b ⟨* 155⟩+≡ ◁199a 200a▷
```
procedure emit_original_closure_function_def(pp, name, cl, t)
  local es
  tt := copy(t)
  every k := key(t) do tt[k] := literal("_c->v." || tt[k])
  tt[the_global_pc] := Eforce(Eclosure_loc())
  PPxwrite(pp, "static void ", name,
               "_app (RClosure c,$o Emitter emitter,$o FailCont fail) {$t$n",
               name, "_Closure _c = (", name, "_Closure) c;$n")
  es := emitterstyle
  emitterstyle := "closure"
  PPxwrite(pp, pretty(
     super_simplify(Sif([Sguarded(subst_table_elements(cl.conditions, tt),
                                disjunct_to_emission(
                                    subst_table_elements(cl.disjunct, tt))),
        ### disjunct_to_emission should be changed to include width conditions &c
                    Sguarded(1,
                       Sfail("Conditions not satisfied for constructor " || name))]))))
  emitterstyle := es
  PPxwrite(pp, "$b$n}")
end
```
Defines:
emit_original_closure_function_def, used in chunk 196a.
Uses constructor 155, disjunct 53, disjunct_to_emission 186b, Eclosure_loc 79, Eforce 79, Emitter, emitterstyle 17b, fail 396, FailCont, literal 43e, pretty 16a, RClosure, Sfail 182c, Sguarded 79, Sif 79, subst_table_elements 87b, super_simplify 90a, and the_global_pc 17d.

199c ⟨*apply-fun.t* 199c⟩≡
```
static void %{name}_app (RClosure c, Emitter emitter, FailCont fail) {
```
Uses Emitter, fail 396, FailCont, and RClosure.

200a ⟨* 155⟩+≡ ◁199b 200b▷
```
procedure find_input(name, cons)
  return input_named(cons, name)
end
```
Defines:
find_input, used in chunk 200b.
Uses input_named 162b.

200b ⟨* 155⟩+≡ ◁200a 200c▷
```
procedure meaningtab(t, cons)
  u := table()
  every k := key(t) do
    (u[t[k]] := known_to_fit(find_input, k, cons).meaning) |
        impossible("unknown free variable ", k)
  return u
end
```
Defines:
meaningtab, used in chunk 196a.
Uses find_input 200a and known_to_fit 192c.

8.6.1 Extracting relocatable addresses from instances and closures

There are three possible sources of relocatable addresses: the program counter, relocatable inputs, or constructors (input or applied).

uses_pc determines if a constructor *directly* uses the program counter. Conservatively, we estimate that a constructor uses the program counter if any of its branches have labels, or if some constructor used in *any* of the constructor's output patterns uses the program counter. This is overly conservative but easy to implement.

200c ⟨* 155⟩+≡ ◁200b 201a▷
```
procedure uses_pc(x)
  static pccache
  initial pccache := table()
  if not member(pccache, x) then {
    pccache[x] := 0
    type(x) == "constructor" | impossible("uses_pc")
    if pattern_label_names((!x.branches).pat) |
       uses_pc(constructors_applied_in((!x.branches).pat)) then
      return pccache[x] := 1
  }
  return 0 < pccache[x]
end
```
Defines:
uses_pc, used in chunks 201a, 203b, and 392a.
Uses constructor 155, constructors_applied_in 77b, and pattern_label_names 77a.

uses_reloc determines if the given constructor can depend on the label of some relocatable address. uses_reloc holds if uses_pc holds or if a relocatable address is in the constructor's inputs, or if uses_reloc holds for some constructor-typed input or for any constructor used in an output pattern.

201a ⟨* 155⟩+≡ ◁200c 202d▷
```
procedure uses_reloc(x)
   static cache
   initial cache := table()
   if not member(cache, x) then {
     cache[x] := 0
     case type(x) of {
       "constype"    : if uses_reloc(kept_constructors(x)) then return cache[x] := 1
       "constructor" :
         if inputs_of(x, "string") | uses_pc(x) |
            uses_reloc(inputs_of(x, "constype").meaning |
                       constructors_applied_in((!x.branches).pat))
         then
           return cache[x] := 1
       default : impossible("uses_reloc")
     }
   }
   return 0 < cache[x]
end
```
Uses constructor 155, constructors_applied_in 77b, constype 155, inputs_of 162a, kept_constructors 31d, and uses_pc 200c.

ptrtype is the exact pointer type of this instance or closure. irec is the record containing the inputs.

201b ⟨*constructor-labels.t* 201b⟩≡
```
static void %{name}_relocfn(RClosure c, RelocCallback f, void *closure) {$t
%ptrtype _c = (%ptrtype) c;
%calls$b}
```
Uses RClosure and RelocCallback 391 391.

201c ⟨*reloc-call.t* 201c⟩≡
```
(*f)(closure, _c->%irec.%input);
```
Uses input 44a.

201d ⟨*cons-call.t* 201d⟩≡
```
$t${(*(_c->%irec.%input.h->relocfn))$o((Instance*)(&_c->%irec.%input), f, closure);$}$b
```
Uses input 44a and Instance 391 391.

201e ⟨*no-labels.t* 201e⟩≡
```
static void %{name}_relocfn(RClosure c, RelocCallback f, void *closure) {
  return;
}
```
Uses RClosure and RelocCallback 391 391.

8.6.2 Creating instances

Instance creation is simple-minded—check the widths of the field inputs, and stuff all the inputs into the instance.

202a ⟨*create-instance-body.t* 202a⟩≡
```
%{class}%{type}_Instance %safename(%args) {$t
%{type}_Instance _i = { %{name}_TAG };
%{input-tests}%{assignments}return _i;$b
}
```
Uses `input` 44a.

202b ⟨*input-test.t* 202b⟩≡
```
$t${if (!(${%condition$})) $c(*fail) ("%name = %%d won't fit in %width %signed bits");$}$b
```
Uses `fail` 396.

202c ⟨*instance-assignment.t* 202c⟩≡
```
_i.u.%name.%l = %r;
```

202d ⟨* 155⟩+≡ ◁201a 202e▷
```
procedure input_width_tests(cons)
  t := []
  every i := inputs_of(cons) & case type(i.meaning) of {
    "integer" : { w := i.meaning; s := "signed"; c := Efitss(literal(i.name), w) }
    "field"   : if member(unchecked_fields, i.meaning) then &fail
                else { w := fwidth(i.meaning); s := "unsigned"
                       c := Efitsu(literal(i.name), w) }
  } do put(t, template_to_list("input-test.t", "name", i.name, "width", w, "signed", s,
                                                "condition", pretty(c)))
  return t
end
```
Defines:
`input_width_tests`, used in chunk 203b.
Uses `Efitss` 79, `Efitsu` 79, `fail` 396, `field` 53, `fwidth` 112d, `input` 44a, `inputs_of` 162a, `literal` 43e, `pretty` 16a, and `unchecked_fields` 340f.

202e ⟨* 155⟩+≡ ◁202d 203a▷
```
procedure emit_proc_declaration(pp, cons)
  if cons.type === instructiontype then
    emit_emitter_proto(pp, cons)
  else
    emit_create_instance_proto(pp, cons)
  return
end
```
Defines:
`emit_proc_declaration`, used in chunks 218c and 219d.
Uses `emit_create_instance_proto` 203a, `emit_emitter_proto` 204a, and `instructiontype` 30a.

203a ⟨* 155⟩+≡ ◁202e 203b▷

```
procedure emit_create_instance_proto(pp, cons)
  c_function_declaration(pp, cons.type.name || "_Instance", Cnoreserve(cons.name),
                         arg_decls(cons))
end
```

Defines:
emit_create_instance_proto, used in chunk 202e.
Uses arg_decls 195a, c_function_declaration 323b, and Cnoreserve 322b.

203b ⟨* 155⟩+≡ ◁203a 204a▷

```
procedure emit_create_instance_body(pp, cons)
  a := []
  every i := inputs_of(cons).name do
     put(a, template_to_list("instance-assignment.t", "name", cons.name, "l", i, "r", i))
  emit_template(pp, "create-instance-body.t", "safename", Cnoreserve(cons.name),
                "name", cons.name, "type", cons.type.name, "args", arg_decls(cons),
                "class", if \indirectname then "static " else "",
                "uses_pc", if uses_pc(cons) then 1 else 0,
                "input-tests", input_width_tests(cons), "assignments", a)
  return
end
```

Defines:
emit_create_instance_body, used in chunks 220a and 224a.
Uses arg_decls 195a, Cnoreserve 322b, indirectname 16a, input 44a, inputs_of 162a,
input_width_tests 202d, and uses_pc 200c.

8.6.3 Emitting

204a ⟨* 155⟩+≡ ◁203b 204c▷

```
procedure emit_emitter_proto(pp, cons)
  c_function_declaration(pp, "void", Cnoreserve(cons.name), arg_decls(cons))
  return
end

procedure emit_emitter_body(pp, cons)
  b := emitter_body(cons)
  emit_closure_functions(pp, cons, b)
  emit_template(pp, "emitter-body.t",
                "safename", Cnoreserve(cons.name), "args", arg_decls(cons),
                "class", if \indirectname then "static " else "")
  PPxwrites(pp, pretty(b))
  if \gen_counters then
    PPxwrites(pp, "$n", cons.name, "_ctr++;")
  PPxwrite(pp, "$b$n}")
  return
end
```

Defines:
emit_emitter_body, used in chunk 220a.
emit_emitter_proto, used in chunk 202e.
Uses arg_decls 195a, c_function_declaration 323b, Cnoreserve 322b, emit_closure_functions 16a, emitter_body 182b, gen_counters 16a, indirectname 16a, and pretty 16a.

204b ⟨*emitter-body.t* 204b⟩≡

```
%{class}void %safename(%args) {$t
```

8.7 Optimizing closures

We don't have to number optimized closures; their naming is arbitrary.

204c ⟨* 155⟩+≡ ◁204a 205a▷

```
procedure emit_optimized_closure_functions(pp, cons, b)
  pushtrace("CLO")
  every cl := subterms_matching(b, "Sclosure") do
    emit_optimized_closure_function(pp, cons, cons.name, cl)
  poptrace()
  return
end
```

Defines:
emit_optimized_closure_functions, used in chunk 17a.
Uses emit_optimized_closure_function 205a, Sclosure 185f, and subterms_matching 82a.

We really ought to add a string type to the closure, so that we could give the name of the constructor when conditions aren't satisfied in the closure. As it is, I use the same error message for each constructor—otherwise they won't all use the same closure function!

205a ⟨* 155⟩+≡ ◁204c 206▷

```
procedure emit_optimized_closure_function(pp, cons, name, cl)
  local selections, selected, free, save, upc, latevars, clo, subst, body
  latevars := set()
  every insert(latevars, inputs_of(cons, "string").name)
  body :=
     super_simplify(Sif([Sguarded(cl.conditions, disjunct_to_emission(cl.disjunct)),
        ### disjunct_to_emission should be changed to include width conditions &c
                         Sguarded(1,
                            Sfail("Conditions not satisfied for unnamed constructor "))
                        ]))
  p := hoist(pp, Elambda(sort(latevars), body), latevars)
  clo := p.e # is a closure
  clo := apply_subst(clo, p.sigma)
  free := set(); every insert(free, free_variables(clo))
  free := sort(free)
PPwrite(pp, "/****************")
PPxwrite(pp, "CLOSURE IS:$t $o", ppexpimage(clo), "$b")
PPwrite(pp, "****************/")
  ⟨make cl.creation to create optimized closure clo and emit placeholder 205b⟩
  return
end
```

Defines:
emit_optimized_closure_function, used in chunk 204c.
Uses apply_subst 214a, constructor 155, disjunct 53, disjunct_to_emission 186b, Elambda 206, free_variables 82b, hoist 207, inputs_of 162a, ppexpimage 98d, Sfail 182c, Sguarded 79, Sif 79, subst 86b, and super_simplify 90a.

205b ⟨*make* `cl.creation` *to create optimized closure* `clo` *and emit placeholder* 205b⟩≡ (205a)

```
l := []
every i := 1 to *clo.values do
  put(l, pretty(Gasgn(Eclosure_val(i),  clo.values[i])) || "$n")
every i := 1 to *clo.addresses do
  put(l, pretty(Gasgn(Eclosure_addr(i), clo.addresses[i])) || "$n")
upc := if subterms_matching(\cl.conditions | cl.disjunct, "Epc", "Epc_known") then 1
       else 0
cl.creation := Sstmts([
  literal(template_to_list("create-closure.t", "name", clo.ty, "clofun", clo.fun,
                           "uses-pc", upc, "save", l)),
  disjunct_to_emission(place_holder(cl.disjunct))])
```

Uses disjunct 53, disjunct_to_emission 186b, Eclosure_addr 79, Eclosure_val 79, Epc 79, Epc_known 79, Gasgn 282a, literal 43e, place_holder 70a, pretty 16a, Sstmts 79, and subterms_matching 82a.

206 ⟨* 155⟩+≡ ◁205a 207▷

```
record Elambda(formals, body)
record hoisted(e, sigma)        # pair containing exp, substitution
record arrow(v, e)              # part of a substitution
procedure make_early(v, e, sigma)
  v := fresh_variable(v)
  return hoisted(v, push(sigma, arrow(v, e)))
end

procedure make_late(v, e, sigma)
  return hoisted(e, sigma)
end

procedure make_time(e, latevars)
  return if islate(e, latevars) then make_late else make_early
end

procedure islate(e, latevars)
  return case type(e) of {
    "string" : member(latevars, e)
    "list"   : islate(!e, latevars)
    default  : 1
  }
end
```

Defines:
- `arrow`, used in chunks 208d, 209a, and 214.
- `Elambda`, used in chunks 108a, 116c, 123a, 130f, 205a, and 207.
- `hoisted`, used in chunks 207 and 210–12.
- `islate`, used in chunks 209–11.
- `make_early`, used in chunks 207 and 209–11.
- `make_late`, used in chunks 207 and 209–11.
- `make_time`, used in chunk 207.

Uses `fresh_variable` 151b.

207 ⟨* 155⟩+≡ ◁206 208b▷

```
record Eclosure(ty, fun, values, addresses)

procedure hoist(pp, e, latevars)
  local body, sigma, sigma1, p, free, freeset, clo, closubst, early, late
  local values, addresses
  x := case type(e) of {
    "string"  : hoisted(e, [])
    "integer" : make_early("lit", e, [])
    "list"    : hoistlist(pp, e, latevars)
    "Elambda" : {
       p := hoist(pp, e.body, set(e.formals))
       body := p.e
       sigma := p.sigma
       ⟨make addresses and values free vars of body (in order of appearance) 208c⟩
       if \lateconst then {
         ⟨using sigma, push constants back into body and out of values 208d⟩
       }

       clo := fresh_variable("clo")
       ⟨make closubst change addresses and values to select from clo 209a⟩
       body := apply_subst(body, closubst)

       clotype := closure_type(pp, values, addresses)
       clofun := closure_function(pp, clotype, addresses, body)
       ⟨make cloargs the list of closure args, using values and sigma 209b⟩
       p := hoistlist(pp, cloargs, latevars)
       make_time(p.e, latevars)("closure",
              Eclosure(clotype, clofun, p.e, apply_subst(addresses, sigma)), p.sigma)
    }
    "Epc" | "Epc_known" : make_late("pc", e, [])
    ⟨other cases for hoisting 209c⟩
    ⟨generated cases for hoisting 208a⟩
    default : impossible("hoisting ", image(type(e)))
  }
### PPxwrites(pp, "Hoisting ", ppexpimage(e), "$t$ngot $t${$o", ppexpimage(x.e),
###                      "$}$b$nwith")
### showsigma(pp, x.sigma)
### PPxwrite(pp, "$nso, when applied, have $t$o${",
###             ppexpimage(apply_subst(x.e, x.sigma)),
###                 "$}$b$b$n")
  return x
end
```

Defines:
`Eclosure`, used in chunks 108a, 116c, 123a, and 130f.
`hoist`, used in chunks 81a, 205a, 210–12, and 278b.

Uses apply_subst 214a, closure_function 215b, closure_type 213a, Elambda 206, Epc 79, Epc_known 79, fresh_variable 151b, hoisted 206, hoistlist 212c, lateconst 16a, make_early 206, make_late 206, make_time 206, ppexpimage 98d, and showsigma 214b.

208a ⟨*generated cases for hoisting* 208a⟩≡ (207)

A value is an address if it's a formal parameter or if it's a suitable element of an instance input:

208b ⟨* 155⟩+≡ ◁207 212c▷

```
procedure is_address(e, addressparms)
  return e === !addressparms |
         (type(e) == "Einstance_input",
          type(input_named(e.cons, e.name).meaning) == "string")
end
```

Defines:
is_address, used in chunk 208c.
Uses Einstance_input 79 and input_named 162b.

Preserving order here helps keep us from creating closure functions that differ only by a permutation of arguments. We have to apply `sigma`, and see what sort of value the free variable stands for, in order to classify it as a value or an address.

208c ⟨*make* `addresses` *and* `values` *free vars of body (in order of appearance)* 208c⟩≡ (207)

```
every addresses | values := []
freeset := set()
every v := free_variables(body) & not member(freeset, v) &
      x := apply_subst(v, sigma)
do {
  insert(freeset, v)
  put(if is_address(x, e.formals) then addresses else values, v)
}
```

Uses apply_subst 214a, free_variables 82b, and is_address 208b.

208d ⟨*using* `sigma`*, push constants back into* `body` *and out of* `values` 208d⟩≡ (207)

```
l := []
every f := !values do
  if x := constant(apply_subst(f, sigma)) then
    body := apply_subst(body, arrow(f, x))
  else
    put(l, f)
values := l
```

Uses apply_subst 214a, arrow 206, and constant 111b.

209a ⟨*make* `closubst` *change addresses and values to select from* `clo` 209a⟩≡ (207)
```
  closubst := []
  every i := 1 to *values do
    put(closubst, arrow(values[i], Eclosure_val(i)))
  every i := 1 to *addresses do
    put(closubst, arrow(addresses[i], Eclosure_addr(i)))
```
Uses `arrow` 206, `Eclosure_addr` 79, and `Eclosure_val` 79.

209b ⟨*make* `cloargs` *the list of closure args, using* `values` *and* `sigma` 209b⟩≡ (207)
```
  cloargs := []
  every put(cloargs, apply_subst(!values, sigma))
```
Uses `apply_subst` 214a.

209c ⟨*other cases for hoisting* 209c⟩≡ (207) 210a ▷
```
  "table" : {
    kl := []; every k := key(e) & e[k] ~= 0 do put(kl, k)
    hl := hoistlist(pp, kl, latevars)
    sigma := hl.sigma
    hl := hl.e
    if x := !hl & not islate(x, latevars) then {
      t := table(0)
      every k := kl[i := 1 to *kl] & x := hl[i] & not islate(x, latevars) do
        t[x] +:= e[k]
      early := make_early("sum", t, sigma)
      sigma := early.sigma
      early := early.e
    } else
      early := &null
    if islate(!hl, latevars) then {
      t := table(0)
      every k := kl[i := 1 to *kl] & x := hl[i] & islate(x, latevars) do
        t[x] +:= e[k]
      t[\early] +:= 1
      make_late("sum", t, sigma)
    } else {
      make_early("sum", \early | 0, sigma)
    }
  }
```
Uses `hoistlist` 212c, `islate` 206, `make_early` 206, and `make_late` 206.

210a ⟨*other cases for hoisting* 209c⟩+≡ (207) ◁209c 210b▷

```
"Eorb" : {
  l := hoistlist(pp, flatten(e, "Eorb"), latevars)
  every early | late := []
  every x := !l.e do put(if islate(x, latevars) then late else early, x)
  if *late > 0 then
    if *early = 0 then
      make_late("or", unflatten(late, Eorb, 0), l.sigma)
    else {
      early := make_early("or", unflatten(early, Eorb, 0), l.sigma)
      push(late, early.e)
      make_late("or", unflatten(late, Eorb, 0), early.sigma)
    }
  else
    make_early("or", unflatten(early, Eorb, 0), l.sigma)
}
```

Uses `Eorb` 79, `flatten` 113c, `hoistlist` 212c, `islate` 206, `make_early` 206, `make_late` 206, and `unflatten` 113d.

210b ⟨*other cases for hoisting* 209c⟩+≡ (207) ◁210a 211a▷

```
"set" : {
  sigma := []
  s := set()
  every x := !e do {
    p := hoist(pp, x, latevars)
    insert(s, p.e)
    sigma := compose(sigma, p.sigma)
  }
  hoisted(s, sigma)
}
```

Uses `compose` 213b, `hoist` 207, and `hoisted` 206.

For equations, we try a funky heuristic, namely, if we see a zero on the right-hand side, *don't* hoist it.

211a ⟨*other cases for hoisting* 209c⟩+≡ (207) ◁210b 211b▷
```
  "eqn" : {
    sigma := []
    m := make_early
    _a0 := hoist(pp, e.left, latevars)
    sigma := compose(sigma, _a0.sigma)
    _a0 := _a0.e
    if islate(_a0, latevars) then m := make_late
    if \latezero & untable(e.right) === 0 then {
      _a2 := 0
      m := make_late
    } else {
      _a2 := hoist(pp, e.right, latevars)
      sigma := compose(sigma, _a2.sigma)
      _a2 := _a2.e
      if islate(_a2, latevars) then m := make_late
    }
    if not (_a0 === e.left, _a2 === e.right) then
      e := eqn(_a0, e.op, _a2)
    m("eqn", e, sigma)
  }
```
Uses `compose` 213b, `eqn` 81c, `hoist` 207, `islate` 206, `latezero` 16a, `make_early` 206, `make_late` 206, and `untable` 111c.

We can't hoist a guard! And we might as well not hoist 1 when it's just the guard that's always satisfied.

211b ⟨*other cases for hoisting* 209c⟩+≡ (207) ◁211a 211c▷
```
  "Sguarded" : {
    p := if guard_always_satisfied(e.guard) then hoisted(1, [])
         else hoist(pp, e.guard, latevars)
    q := hoist(pp, e.x, latevars)
    hoisted(Sguarded(p.e, q.e), compose(p.sigma, q.sigma))
  }
  "Sepsilon" : hoisted(e, [])
```
Uses `compose` 213b, `guard_always_satisfied` 93c, `hoist` 207, `hoisted` 206, `Sepsilon` 79, and `Sguarded` 79.

For `Sfail`, we ought to be able to keep track of the types of variables that are hoisted, so we could hoist both the message and, if need be, the arguments.

211c ⟨*other cases for hoisting* 209c⟩+≡ (207) ◁211b 212a▷
```
  "Sfail"    : hoisted(e, [])
```
Uses `hoisted` 206 and `Sfail` 182c.

For emission, we ought to do something extra special. We ought *always* to hoist emission, converting unknown tokens into placeholders as we go. Then we can build closures for the remaining tokesn. Perhaps if we have a type mechanism, we can do that one day.

For now, we prevent hoisting by keeping `Stoken`s late.

212a ⟨*other cases for hoisting* 209c⟩+≡ (207) ◁211c 212b▷
```
"Stoken" : {
  p := hoist(pp, e.x, latevars)
  hoisted(Stoken(p.e, e.n, e.offset), p.sigma)
}
```
Uses `hoist` 207, `hoisted` 206, and `Stoken` 79.

For a particular instance input, we have to check its type to know whether it is early or late.

212b ⟨*other cases for hoisting* 209c⟩+≡ (207) ◁212a
```
"Eforce" : {
  p := hoist(pp, e.x, latevars)
  hoisted(Eforce(p.e), p.sigma)
}
```
Uses `Eforce` 79, `hoist` 207, and `hoisted` 206.

212c ⟨* 155⟩+≡ ◁208b 213a▷
```
procedure hoistlist(pp, l, latevars)
  local sigma, newl, e
  sigma := []
  newl := []
  every e := !l do {
    p := hoist(pp, e, latevars)
    put(newl, p.e)
    sigma := compose(p.sigma, sigma)
  }
  return hoisted(newl, sigma)
end
```
Defines:
`hoistlist`, used in chunks 207, 209c, and 210a.
Uses `compose` 213b, `hoist` 207, and `hoisted` 206.

The function closure_type looks at the number of free and bound variables and finds (or emits) a type suitable for use as the closure. If it has to emit a new type, it also emits a "reloc function" to go with it.

213a ⟨* 155⟩+≡ ◁212c 213b▷
```
procedure closure_type(pp, values, addresses)
  static closure_types
  local clname, calls
  initial closure_types := set()
  clname := "O" || *addresses || "_" || *values
  if not member(closure_types, clname) then {
    insert(closure_types, clname)
    every calls | l := []
    every i := 1 to *addresses do {
      put(l, " $cRAddr a" || i || ";")
      put(calls, template_to_list("reloc-call.t", "irec", "v", "input", "a" || i))
    }
    every i := 1 to *values do
      put(l, " $cunsigned u" || i || ";")
    emit_template(pp, "closure-type.t", "name", clname, "decls", l)
    emit_template(pp, "constructor-labels.t", "ptrtype", clname || "_Closure",
                              "name", clname, "calls", calls)
  }
  return clname
end
```
Defines:
closure_type, used in chunk 207.
Uses constructor 155 and input 44a.

For compose, the right-hand substitution is applied first.

213b ⟨* 155⟩+≡ ◁213a 214a▷
```
procedure compose(sigma1, sigma2)
  return if *sigma1 = 0 then sigma2
         else if *sigma2 = 0 then sigma1
         else push(sigma1, sigma2)
end
```
Defines:
compose, used in chunks 210–12.

214a ⟨* 155⟩+≡ ◁213b 214b▷
```
procedure apply_subst(e, sigma)
  return case type(sigma) of {
           "list" : {
               every s := !sigma do e := apply_subst(e, s)
               e
             }
           "arrow": subst(e, sigma.v, sigma.e)
           default :  impossible("substitution")
         }
end
procedure apply_subst_list(e, sigma)
  every i :=
  return case type(sigma) of {
           "list" : apply_subst_list(e, sigma, 1)
           "arrow": subst(e, s.v, s.e)
           default :  impossible("substitution")
         }
end
```
Defines:
  `apply_subst`, used in chunks 205a and 207–9.
Uses `arrow` 206 and `subst` 86b.

214b ⟨* 155⟩+≡ ◁214a 215a▷
```
procedure showsigma(pp, sigma)
  case type(sigma) of {
    "arrow" : PPxwrites(pp, "$t$n${", sigma.v, " --> ",
                        ppexpimage(sigma.e), "$}$b")
    "list"  : every showsigma(pp, !sigma)
    default : impossible("substitution")
  }
  return
end
```
Defines:
  `showsigma`, used in chunk 207.
Uses `arrow` 206 and `ppexpimage` 98d.

215a ⟨* 155⟩+≡ ◁214b 215b▷

```
global closure_functions_postfix
procedure emit_closure_functions_postfix(pp, interfacebasename)
  if *\closure_functions_postfix > 0 then {
    PPxwrite(pp, "/****************************$t")
    every k := key(\closure_functions_postfix) do
      PPxwrite(pp, "${", k, " = $t$c", closure_functions_postfix[k], "$b$}")
    PPxwrite(pp, "$b$n****************/")
  }
  PPxwrites(pp, "ClosurePostfix ",
              mapoutbadchars(interfacebasename), "_clofuns[] = {$t")
  every k := key(\closure_functions_postfix) do
    PPxwrites(pp, "$n{ ", k, ", ", image(closure_functions_postfix[k]), " }, ")
  PPxwrites(pp, "$n{ (ApplyMethod) 0, (char *) 0 }")
  PPxwrite(pp, "$b$n};")
  return
end
```

Defines:
`closure_functions_postfix`, used in chunk 215b.
`emit_closure_functions_postfix`, used in chunk 220a.
Uses `ApplyMethod` 392a, `ClosurePostfix` 392b, and `mapoutbadchars` 161a.

215b ⟨* 155⟩+≡ ◁215a

```
procedure closure_function(pp, cloty, addresses, body)
  static cache, count
  local bodyimage, es, orig_body
  initial { every closure_functions_postfix | cache := table(); count := 0 }
  tt := table()
  every i := 1 to *addresses do tt[addresses[i]] := Eclosure_addr(i)
  tt[the_global_pc] := Eforce(Eclosure_loc())
  orig_body := body
  body := subst_table_elements(body, tt)
  bodyimage := expps(body)
  if /cache[bodyimage] then {
    cache[bodyimage] := "_clofun_" || (count +:= 1)
    closure_functions_postfix[cache[bodyimage]] := bodyimage
    verbose("New closure function for \n", bodyimage)
    PPxwrite(pp, "/* CLOSURE FUNCTION _clofun_", count, " is $t$n", bodyimage, "$b$n */")
    ⟨emit new closure function numbered count and put it in cache 216⟩
  }
  return cache[bodyimage]
end
```

Defines:
`closure_function`, used in chunk 207.
Uses `closure_functions_postfix` 215a, `Eclosure_addr` 79, `Eclosure_loc` 79, `Eforce` 79, `expps` 125a, `subst_table_elements` 87b, `the_global_pc` 17d, and `verbose` 16a.

216 ⟨*emit new closure function numbered* `count` *and put it in cache* 216⟩≡ (215b)

```
es := emitterstyle
emitterstyle := "closure"
PPxwrite(pp, "static void _clofun_", count,
             "(RClosure c,$o Emitter emitter,$o FailCont fail) {$t$n",
             cloty, "_Closure _c = (", cloty, "_Closure) c;$n")
PPxwrite(pp, pretty(body))
emitterstyle := es
PPxwrite(pp, "$b$n}")
```

Uses `Emitter`, `emitterstyle` 17b, `fail` 396, `FailCont`, `pretty` 16a, and `RClosure`.

Chapter 9

Managing generation of binary encoders

217 ⟨* 217⟩≡ 218a ▷

```
link pretty
procedure emit_encoding(outfilename)
  local i, f
  emit_encoding_interface(open(i := outfilename || interface_extension, "w")) |
    error("Could not open ", image(i), " for writing")
  emit_encoding_implementation(
         open(f := outfilename || implementation_extension, "w"), i, outfilename) |
    error("Could not open ", image(f), " for writing")
end
```

Defines:
  emit_encoding, used in chunk 18.
Uses emit_encoding_implementation 220a, emit_encoding_interface 218a,
  implementation_extension 16a, interface_extension 16a, and pretty 16a.

218a ⟨*217⟩+≡ ◁217 219a▷

```
procedure emit_encoding_interface(outfile)
  local cons, t, u
  verbose("Emitting encoding interface")
  pp := PPnew(outfile)
  PPwrite(pp, "/* must #include <mclib.h> before this file */")
  ⟨write definitions of all instance types 218b⟩
  if \gen_counters then declare_counter_types(pp)
  if \indirectname then
    PPxwrite(pp, "struct ", indirecttype, " {$t")
  ⟨emit declarations for all of the kept constructors 218c⟩
  if \gen_counters then declare_counter_funs(pp)
  if \indirectname then
    PPxwrite(pp, "$b$n};")
  return
end
```

Defines:
emit_encoding_interface, used in chunks 217 and 223a.
Uses declare_counter_funs 222a, declare_counter_types 222a, file 345a, gen_counters 16a, indirectname 16a, indirecttype 16a, and verbose 16a.

218b ⟨*write definitions of all instance types* 218b⟩≡ (218a)

```
s := set([instructionctype])  # don't try to emit instance def for untyped constructors
every cons := kept_constructors() & not member(s, cons.type) do {
  insert(s, cons.type)
  emit_instance_type(pp, cons.type)
}
s := &null # enable garbage collection
```

Uses constructors 30a, emit_instance_type 194a, instructionctype 30a, and kept_constructors 31d.

If an indirect encoder is emitted, the order of the constructors has to match exactly the order used in `emit_encoding_interface`. At one time we rearranged things to put typed constructors first, but on March 3, 1996, we decided there was no good reason to rearrange the users' constructor declarations.

218c ⟨*emit declarations for all of the kept constructors* 218c⟩≡ (218a)

```
every emit_proc_declaration(pp, kept_constructors())
```

Uses emit_proc_declaration 202e and kept_constructors 31d.

219a ⟨* 217⟩+≡ ◁218a 220a▷

```
procedure emit_indirect_encoder(pp)
  e := []
  every cons := kept_constructors() do
    put(e, Cnoreserve(cons.name) || ", $o")
  c := if \gen_counters then counter_names() else ""
  emit_template(pp, "proc-structure.t",
                    "indirectname", indirectname, "indirecttype", indirecttype,
                    "encoders", e, "counters", c)
  return
end
```

Defines:
  `emit_indirect_encoder`, used in chunks 220a and 224a.
Uses `Cnoreserve` 322b, `counter_names` 222a, `gen_counters` 16a, `indirectname` 16a, `indirecttype` 16a, and `kept_constructors` 31d.

219b ⟨*proc-structure.t* 219b⟩≡

```
static struct %indirecttype encoding_procs = {$t%encoders%counters$b
};
struct %indirecttype *%indirectname = &encoding_procs;
```

Uses `indirectname` 16a and `indirecttype` 16a.

Here's some old code

219c ⟨*proc-structure.t (segregated version)* 219c⟩≡

```
static struct %indirecttype encoding_procs = {$t%typed%untyped%counters$b
};
struct %indirecttype *%indirectname = &encoding_procs;
```

Uses `indirectname` 16a and `indirecttype` 16a.

219d ⟨*emit declarations for all of the kept constructors (segregated version)* 219d⟩≡

```
every t | u := []
every cons := kept_constructors() do
  put(if cons.type === instructionctype then u else t, cons)
    every emit_proc_declaration(pp, !t | !u)
```

Uses `emit_proc_declaration` 202e, `instructionctype` 30a, and `kept_constructors` 31d.

220a ⟨* 217⟩+≡ ◁219a 221a▷

```
procedure emit_encoding_implementation(outfile, interfacename, basename)
  verbose("Emitting encoding implementation")
  pp := PPnew(outfile)
  PPwrite(pp, "#include <mclib.h>")
  PPwrite(pp, "#include ", image(interfacename))
  PPwrite(pp, "#define sign_extend(N,SIZE) \\")
  PPwrite(pp, "  (((int)((N) << (sizeof(unsigned)*8-(SIZE)))) ",
              ">> (sizeof(unsigned)*8-(SIZE)))")

  pushtrace("ASS")
  if \gen_counters then declare_counters(pp)
  ⟨write tag definitions for typed kept_constructors 220b⟩
  every cons := kept_constructors() do {
    verbose("Encoder for constructor ", cons.name)
    PPwrite(pp, "/**************")
    show_constructor(pp, cons)
    PPwrite(pp, "***********/")
    if cons.type ~=== instructionctype then
      emit_create_instance_body(pp, cons)
    else
      emit_emitter_body(pp, cons)
  }
  PPwrite(pp)   # flush prettyprinter
  poptrace()
  if \gen_counters then define_counter_funs(pp)
  if \indirectname then emit_indirect_encoder(pp)
  emit_closure_functions_postfix(pp, basename)
  return
end
```

Defines:
emit_encoding_implementation, used in chunk 217.

Uses constructor 155, declare_counters 222a, define_counter_funs 222a, emit_closure_functions_postfix 215a, emit_create_instance_body 203b, emit_emitter_body 204a, emit_indirect_encoder 219a, gen_counters 16a, indirectname 16a, instructionctype 30a, kept_constructors 31d, show_constructor 221a, and verbose 16a.

220b ⟨*write tag definitions for typed* kept_constructors 220b⟩≡ (220a)

```
every cons := kept_constructors() & cons.type ~=== instructionctype do
  PPwrite(pp, "#define ", cons.name, "_TAG ", cons.tag)
```

Uses instructionctype 30a and kept_constructors 31d.

221a ⟨* 217⟩+≡ ◁220a 221b▷
```
procedure show_constructor(pp, cons)
  PPwrite(pp)
  PPwrites(pp, cons.name, " ")
  every o := !cons.operands do
    case type(o) of {
      "literal" : PPwrites(pp, o.s)
      "input"   : PPwrites(pp, o.name,
                           if type(o.meaning) == "integer" then "!" else "")
      default   : impossible("operand")
    }
  PPxwrites(pp, " is $t${$c")
  PPxwrite(pp, ppexpimage(pattern_to_case(crhs(cons))), "$}$b$n")
  return
end
```
Defines:
show_constructor, used in chunk 220a.
Uses crhs 164, input 44a, literal 43e, pattern_to_case 178b, and ppexpimage 98d.

221b ⟨* 217⟩+≡ ◁221a 222a▷
```
procedure show_constype(outfile, type)
  write(outfile)
  l := []; every put(l, kept_constructors(type).name)
  PPwrite(pp, type.name, " (", commaseparate(l, " | "), "):")
  show_case(outfile, pattern_to_case(constype_pattern(type)))
  write(outfile)
  return
end
```
Uses constype_pattern 167b, kept_constructors 31d, and pattern_to_case 178b.

222a ⟨* 217⟩+≡ ◁221b

```
procedure declare_counters(pp)
  every PPwrite(pp, "static int ", kept_constructors().name, "_ctr = 0;")
  return
end
procedure declare_counter_types(pp)
  PPwrite(pp, "typedef void (*Printer)(void *closure, char *fmt, ...);")
  return
end
procedure declare_counter_funs(pp)
  c_function_declaration(pp, "void", "reset_cons_counters", "(void)")
  c_function_declaration(pp, "void", "dump_cons_counters","(Printer p, void *closure)")
  return
end
procedure counter_names()
  return "reset_cons_counters, dump_cons_counters, $o"
end
procedure define_counter_funs(pp)
  PPxwrites(pp, if \indirectname then "static " else "",
                "void reset_cons_counters(void) {$t")
  every PPxwrites(pp, "$n", kept_constructors().name, "_ctr = 0;")
  PPxwrite(pp, "$b$n}")
  PPxwrites(pp, if \indirectname then "static " else "",
                 "void dump_cons_counters(Printer p, void *closure) {$t")
  every cons := kept_constructors() do
    PPxwrites(pp, "$np(closure, \"%d ", cons.name, " : ", ⟨type of cons 222b⟩,
           "\\n\", ", cons.name, "_ctr);")
  PPxwrite(pp, "$b$n}")
end
```

Defines:
- `counter_names`, used in chunk 219a.
- `declare_counter_funs`, used in chunk 218a.
- `declare_counters`, used in chunk 220a.
- `declare_counter_types`, used in chunk 218a.
- `define_counter_funs`, used in chunk 220a.

Uses `c_function_declaration` 323b, `indirectname` 16a, and `kept_constructors` 31d.

222b ⟨*type of* cons 222b⟩≡ (222a)

```
(if cons.type === instructiontype then "<instruction>" else cons.type.name)
```

Uses `instructiontype` 30a.

Chapter 10

Managing generation of assembly encoders

We can reuse the encoding interface, since the assembly stuff is an implementation of it.

223a ⟨* 223a⟩≡ 224a ▷

```
link pretty
procedure emit_assembly(outfilename)
  local i, f
  verbose("Emitting encoding interface")
  emit_encoding_interface(openfile(i := outfilename || interface_extension, "w")) |
    error("Could not open ", image(i), " for writing")
  verbose("Emitting assembly procedures")
  emit_assembly_implementation(
         openfile(f := outfilename || implementation_extension, "w"), i) |
    error("Could not open ", image(f), " for writing")
end
```

Defines:
`emit_assembly`, used in chunk 18.

Uses `emit_assembly_implementation` 224a, `emit_encoding_interface` 218a, `implementation_extension` 16a, `interface_extension` 16a, `openfile` 22, `pretty` 16a, and `verbose` 16a.

Here's the template that begins all assembly implementations:

223b ⟨*assembly-impl.t* 223b⟩≡

```
#include <mclib.h>
#include %interface

#define sign_extend(N,SIZE) \
  (((int)((N) << (sizeof(unsigned)*8-(SIZE)))) >> (sizeof(unsigned)*8-(SIZE)))
```

This next should probably be turned into something higher-order at some point. We reuse the same implementations of typed constructors to bundle up their arguments. We also create a local procedure we can call for each typed constructor to emit is assembly code.

224a ⟨* 223a⟩+≡ ◁223a 226a▷

```
procedure emit_assembly_implementation(outfile, interfacename)
  pp := PPnew(outfile)
  emit_template(pp, "assembly-impl.t", "interface", image(interfacename))

  pushtrace("ASS")
  ⟨write tag definitions for typed kept_constructors 224b⟩
  every cons := kept_constructors() do {
    verbose("Assembler for ", cons.name)
    if cons.type ~=== instructionctype then
      emit_create_instance_body(pp, cons)
    emit_assembler_body(pp, cons)
  }
  PPwrite(pp)   # flush prettyprinter
  if \gen_counters then
    emit_counter_funs(outfile)
  poptrace()
  if \indirectname then emit_indirect_encoder(pp)
  return
end
```

Defines:
emit_assembly_implementation, used in chunk 223a.
Uses emit_assembler_body 234, emit_create_instance_body 203b, emit_indirect_encoder 219a, gen_counters 16a, indirectname 16a, instructionctype 30a, kept_constructors 31d, and verbose 16a.

224b ⟨*write tag definitions for typed* kept_constructors 224b⟩≡ (224a)

```
every cons := kept_constructors() & cons.type ~=== instructionctype do
  PPwrite(pp, "#define ", cons.name, "_TAG ", cons.tag)
```

Uses instructionctype 30a and kept_constructors 31d.

10.1 Name mapping

And now we start in with name mapping o-rama. The game here is to use separate mappings for every possible component of a constructor. By postulating these mappings now, we separate concerns, so that we can worry elsewhere about how they are defined.

As part of planning the mappings, we need to think about how we want the code organized at run time. In particular, we need to think about where to use specialized print procedures and where to use simple format strings. We assume that if people want speed, they'll be emitting binary, not assembly language, so for simplicity we use lots and lots of procedures. We have a procedure for each:

- field
- constructor
- constructor type

We try to reuse field procedures where possible. The application must supply a print procedure `asmprintf` in the style of `fprintf`, and also a procedure `asmprintreloc`, which prints a relocatable address.

We're going to create these procedures lazily. The field-printing procedure is off in the `fieldinfo.nw` module.

The mappings are:	`consname2asm`	constructor names
	`field_asm_syntax`	field operands
	`intarg_asm_syntax`	non-field integer operands
	???	what else?

10.1.1 Mapping constructor names

Use the mapping for constructor's complete name, if it exists. Otherwise, we break the constructor name into its components and match the individual components. We never use full-name mapping and componentwise mapping on the same constructor name.

226a ⟨* 223a⟩+≡ ◁224a 226b▷

```
  procedure consname2asm(cons)
    local n, x, cname
    n := ""
    cname := ""
    every cname ||:= opcode_component_name(!cons.opcode)
    x := map_fullname(cname)
    pushtrace("GLOB")
    if (\x) then
      n := if (\lowercons) then map(x) else x
    else
      every x := map_component(opcode_component_name(!cons.opcode), cons.type) do
        n ||:= if (\lowercons) then map(x) else x
    poptrace()
    return if cons.type === instructionctype | *n > 0 then n || " " else ""
  end
```

Defines:
`consname2asm`, used in chunks 234 and 242.
Uses `instructionctype` 30a, `lowercons` 16a, `map_component` 226b, `map_fullname` 226b, and `opcode_component_name` 160b.

We map the individual components by opcode globbing.

226b ⟨* 223a⟩+≡ ◁226a 227a▷

```
  global opcode_globs, opcode_full
  procedure map_fullname(name)
    initial /opcode_full := []
    return globmap(name, !opcode_full) | &null
  end
  procedure map_component(name, type)
    initial /opcode_globs := []
    return globmap(name, !opcode_globs) | if type === instructionctype then name else ""
  end
```

Defines:
`map_component`, used in chunk 226a.
`map_fullname`, used in chunk 226a.
`opcode_full`, used in chunk 227a.
`opcode_globs`, used in chunk 227a.
Uses `globmap` 227b and `instructionctype` 30a.

And finally, here's where we install the globs.

227a ⟨* 223a⟩+≡ ◁226b 227b▷

```
record globpair(lhs, rhs)
procedure asmopcode(lhs, rhs, full)
  initial { /opcode_globs := []
            /opcode_full := []
          }
  put(if (\full) then opcode_full else opcode_globs, globpair(lhs, rhs))
  return
end
```

Defines:
asmopcode, used in chunk 50a.
Uses opcode_full 226b and opcode_globs 226b.

And now, the implementation of globbing.

227b ⟨* 223a⟩+≡ ◁227a 227c▷

```
procedure globmap(name, glob)
  l := []   # used for $n
  name ?
    if consume_globs(glob.lhs, l) & pos(0) then
      return insert_glob_matches(glob.rhs, l)
end
```

Defines:
globmap, used in chunk 226b.
Uses consume_globs 227c, insert_glob_matches 228c, and used 132a.

227c ⟨* 223a⟩+≡ ◁227b 228b▷

```
procedure consume_globs(pat, dollars, i)
  suspend case type(pat) of {
    "glob_any" : {
       while *dollars < pat.number do put(dollars, &null)
       dollars[pat.number] <- consume_globs(!pat.alternatives, dollars)
    }
    "glob_wildcard" : tab(&pos to *&subject + 1)
    "list"          : {⟨glob a concatenation 228a⟩}
    "string"        : =pat
    default         : impossible("glob pattern")
  }
end
```

Defines:
consume_globs, used in chunks 227b and 228a.
Uses glob_any 50d, glob_wildcard 50d, and pattern 53.

228a ⟨*glob a concatenation* 228a⟩≡ (227c)

```
/i := 1
if pat[i] then
  consume_globs(pat[i], dollars) || consume_globs(pat, dollars, i+1)
else
  ""
```

Uses `consume_globs` 227c.

228b ⟨* 223a⟩+≡ ◁227c 228c▷

```
procedure number_braces(pat)
  do_number_braces(pat, 0)
  return pat
end

procedure do_number_braces(pat, bracecount)
  case type(pat) of {
    "glob_any"      : { pat.number := bracecount +:= 1
                        bracecount := do_number_braces(pat.alternatives, bracecount)
                      }
    "glob_wildcard" : &null
    "list"          : every i := 1 to *pat do
                        bracecount := do_number_braces(pat[i], bracecount)
    "string"        : &null
    default         : impossible("glob pattern: ", image(pat))
  }
  return bracecount
end
```

Defines:
`number_braces`, used in chunk 51a.
Uses `glob_any` 50d, `glob_wildcard` 50d, and `pattern` 53.

228c ⟨* 223a⟩+≡ ◁228b 229a▷

```
procedure insert_glob_matches(rhs, dollars)
  s := ""
  every x := !rhs do
    s ||:= case type(x) of {
      "string"      : x
      "glob_dollar" : \dollars[x.number] |
                        error("No match for $", x.number, " in `", grhsimage(rhs), "'")
      default : impossible("glob rhs")
    }
  return s
end
```

Defines:
`insert_glob_matches`, used in chunk 227b.
Uses `glob_dollar` 50d and `grhsimage` 229a.

229a ⟨* 223a⟩+≡ ◁228c 229b▷
```
procedure grhsimage(rhs)
  s := ""
  every x := !rhs do
    s ||:= case type(x) of {
      "string"      : x
      "glob_dollar" : "$" || x.number
      default : impossible("glob rhs")
    }
  return s
end
```
Defines:
grhsimage, used in chunk 228c.
Uses glob_dollar 50d.

10.1.2 Mapping operands

First, we're only allowed to map integers and fields, not relocatables or constructor types.

229b ⟨* 223a⟩+≡ ◁229a 230d▷
```
record operand_syntax(syntax, nametable)
  # syntax string, optional name table, optional bound on values (max+1)

global operand_syntax_tab

procedure asmoperand(name, syntax, names)
  local nametable
  initial /operand_syntax_tab := table()
  ⟨insist that name not refer to a constructor type or relocatable 230b⟩
  ⟨set nametable to the name table, checking for consistency 230a⟩
  ⟨insist on names if %s is used 230c⟩
  (return /operand_syntax_tab[name] := operand_syntax(syntax, nametable)) |
     error("Operand syntax for ", name, " already specified")
end
```
Defines:
asmoperand, used in chunk 50a.
operand_syntax, used in chunks 230d and 238a.
operand_syntax_tab, used in chunk 230d.
Uses max 396.

230a ⟨*set* nametable *to the name table, checking for consistency* 230a⟩≡ (229b)
```
if type(names) == "field" then
  names := namespec(\fieldname_table(names)) |
              error("No names supplied for field ", ii2)
nametable :=
  if type(f := symtab[name]) == "field" then {
    /names := namespec(\fieldname_table(f))  # use default names if none given
    check_namespec(\names, f)
  } else
    (\names).nametable
if \names & /nametable then impossible("name table: ", image(names))
```
Uses check_namespec 341d, field 53, fieldname_table 342a, namespec 341a, and symtab 30a 353a.

230b ⟨*insist that* name *not refer to a constructor type or relocatable* 230b⟩≡ (229b)
```
if type(symtab[name]) == ("constype" | "relocatable") then
  error("You may specify an operand syntax only for integer or field operands")
```
Uses constype 155, field 53, relocatable 45a, and symtab 30a 353a.

230c ⟨*insist on names if* %s *is used* 230c⟩≡ (229b)
```
if /nametable & !percent_split(syntax) == "%s" then
  error("You used %s to format operand ", name, ", but you gave no name specifier")
```
Uses percent_split 239a and used 132a.

To implement lookup, we must provide suitable defaults. The only tricky case is that we use a field's name if it exists, otherwise not.

230d ⟨* 223a⟩+≡ ◁229b 231a▷
```
procedure operand_syntax_of(ipt)
  initial /operand_syntax_tab := table()
  return \operand_syntax_tab[ipt.name] |
    case type(ipt.meaning) of {
      "null"    : operand_syntax("%d")   # integer
      "integer" : operand_syntax("%d")   # signed field
      "field"   : { n := fieldname_table(symtab[ipt.name])
                    operand_syntax(if \n then "%s" else "%u", n)
                  }
      default   : impossible("violated default syntax invariant")
    }
end
```
Defines:
operand_syntax_of, used in chunk 235.

Uses field 53, fieldname_table 342a, operand_syntax 229b, operand_syntax_tab 229b, and symtab 30a 353a.

10.2 Assembly syntax of constructors

Next, the syntax may be given by an **asmsyntax** directive, or it may be taken from the original declaration.

231a ⟨* 223a⟩+≡ ◃230d 232b▹
```
global asmsyntax
procedure set_asmsyntax(cons, operands)
  local oldoperands
  initial /asmsyntax := table()
  ⟨insist that operands and operands of cons have the same names 231b⟩
  ⟨insist that operands and operands of cons have the same signed operands 231c⟩
  (/asmsyntax[cons] := operands) |
    {⟨operands matches existing assembly syntax for cons 232a⟩} |
    error("Conflicting assembly syntax already given for constructor ", cons.name)
  return
end

procedure asmoperands(cons)
  initial /asmsyntax := table()
  return \asmsyntax[cons] | cons.operands
end
```
Defines:
asmoperands, used in chunks 234 and 242.
asmsyntax, used in chunk 232a.
set_asmsyntax, used in chunk 50c.
Uses constructor 155.

231b ⟨*insist that* operands *and operands of* cons *have the same names* 231b⟩≡ (231a)
```
s1 := set(); every insert(s1, inputs_of_operands(operands).name)
s2 := set(); every insert(s2, inputs_of(cons).name)
every x := !s1 & not member(s2, x) do
  error("Operand ", x, " given in assembly syntax for ", cons.name,
        " is not in the original constructor specification")
every x := !s2 & not member(s1, x) do
  warning("Operand ", x, " of constructor ", cons.name,
          " is not shown in the assembly syntax")
```
Uses constructor 155, inputs_of 162a, and inputs_of_operands 162a.

231c ⟨*insist that* operands *and operands of* cons *have the same signed operands* 231c⟩≡ (231a)
```
s1 := set(); every insert(s1, inputs_of_operands(operands, "integer").name)
s2 := set(); every insert(s2, inputs_of(cons, "integer").name)
every x := !s1 & not member(s2, x) do
  error("Operand ", x, " of constructor ", cons.name,
        " is signed in assembly syntax but unsigned in the constructor specification")
every x := !s2 & not member(s1, x) do
  error("Operand ", x, " of constructor ", cons.name,
        " is unsigned in assembly syntax but signed in the constructor specification")
```
Uses constructor 155, inputs_of 162a, and inputs_of_operands 162a.

Because some constructors explode with vacuous disjuncts, we make it OK to define duplicate syntax, but we do insist the duplicates be identical.

232a ⟨operands *matches existing assembly syntax for* cons 232a⟩≡ (231a)

```
oldoperands := asmsyntax[cons]
*operands = *oldoperands &
  if i := 1 to *operands & not operands_match(operands[i], oldoperands[i]) then
    {dump_em("new", operands); dump_em("old", oldoperands); &fail}
  else
    &null
```

Uses asmsyntax 231a, dump_em 232c, fail 396, and operands_match 232b.

232b ⟨* 223a⟩+≡ ◁231a 232c▷

```
procedure operands_match(op1, op2)
  return case type(op1) == type(op2) of {
    "literal" : op1.s == op2.s
    "input"   : op1.name == op2.name & op1.meaning === op2.meaning
  }
end
```

Defines:
operands_match, used in chunk 232a.
Uses input 44a and literal 43e.

232c ⟨* 223a⟩+≡ ◁232b 233▷

```
procedure dump_em(ty, ops)
  writes(&errout, ty, " syntax:")
  every writes(&errout, x := !ops &
               case type(x) of {
                    "literal" : x.s
                    "input"   : x.name ||
                                if type(x.meaning) == "integer" then "!"
                                else ""
                    default : impossible("operand type")
               })
  write(&errout)
  return
end
```

Defines:
dump_em, used in chunk 232a.
Uses input 44a and literal 43e.

10.3 Emitting print procedures for constructors

The first issue is figuring out what to call the print procedures and what storage class they should be. Untyped constructors print directly and are visible (if not doing indirect), whereas typed constructors create instances, so we have to give them different names and we have to make them always hidden (static).

233 ⟨* 223a⟩+≡ ◁232c 234▷

```
procedure assembler_proc_name(cons)
  return if cons.type === instructionctype then cons.name
         else cons.name || "_asm"  # unsafe, but what do you want?
end
procedure assembler_proc_class(cons)
  return if \indirectname |  cons.type ~=== instructionctype then "static " else ""
end
```

Defines:
assembler_proc_class, used in chunk 234.
assembler_proc_name, used in chunks 234 and 238b.
Uses indirectname 16a and instructionctype 30a.

`emit_assembler_body` creates an assembly-emitting procedure for the constructor `cons`. Its first act is to make sure there are print procedures for all the operands. After that, it just prints all the syntax, then possibly a newline.

234 ⟨* 223a⟩+≡ ◁233 235▷

```
procedure emit_assembler_body(pp, cons)
  local asmname
  every create_input_print_proc(pp, inputs_of(cons))
  emit_template(pp, "emitter-body.t",
                    "safename", Cnoreserve(assembler_proc_name(cons)),
                    "args", arg_decls(cons),
                    "class", assembler_proc_class(cons))
  asmname := consname2asm(cons)
  if *asmname > 0 then
    emit_asm_printf(pp, "%s", image(asmname))
  every o := !asmoperands(cons) do
    case type(o) of {
      "literal" : emit_asm_printf(pp, "%s", image(o.s))
      "input"  :
          PPxwrites(pp, "$n",
                        create_input_print_proc(pp, o), "(", Cnoreserve(o.name), ");")
      default : impossible("operand type")
    }
  if cons.type === instructionctype then
    emit_asm_printf(pp, "\n")
  PPxwrite(pp, "$b$n}")
end
```

Defines:
`emit_assembler_body`, used in chunk 224a.

Uses `arg_decls` 195a, `asmoperands` 231a, `assembler_proc_class` 233, `assembler_proc_name` 233, `Cnoreserve` 322b, `consname2asm` 226a, `create_input_print_proc` 235, `emit_asm_printf` 239b, `input` 44a, `inputs_of` 162a, `instructionctype` 30a, and `literal` 43e.

10.4 Generating print procedures for operands

Next we show how to create a print procedure for an operand. We do it lazily, and we return the name of the procedure.

We're allowed to print any integer argument using any syntax we like. Relocatable addresses constructor types have constrained syntaxes.

235 ⟨* 223a⟩+≡ ◁234 236▷

```
procedure create_input_print_proc(pp, ipt)
  return case type(ipt.meaning) of {
    "string"   : "asmprintreloc"
    "constype" : create_constype_print_proc(pp, ipt.name, ipt.meaning)
    "integer"  : create_integer_print_proc(pp, "signed",   ipt,
                    signed_type(fwidth(f := symtab[ipt.name])), operand_syntax_of(ipt))
                        # maybe this should be forced to %d?
    "field"    : create_integer_print_proc(pp, "unsigned", ipt,
                    unsigned_type(fwidth(f := ipt.meaning)), operand_syntax_of(ipt))
    "null"     : create_integer_print_proc(pp, "signed",   ipt,
                    signed_type(), operand_syntax_of(ipt))
    default    : impossible("Bad operand value", type(ipt.meaning))
  }
end
```

Defines:
create_input_print_proc, used in chunk 234.

Uses asmprintreloc 394d, constype 155, create_constype_print_proc 238a, create_integer_print_proc 236, field 53, fwidth 112d, operand_syntax_of 230d, signed_type 320c, symtab 30a 353a, unsigned_type 320c, and value 132a.

There's no science here.

236 ⟨* 223a⟩+≡ ◁235 237d▷

```
procedure create_integer_print_proc(pp, sign, ipt, argtype, syntax)
  local namearray, argname, limit
  if p := lookup_printproc(syntax, argtype) then return p
  argname := ipt.name
  emit_template(pp, "int-print-header.t",
                   "name", argname, "type", argtype, "sign", sign)
  l := percent_split(syntax.syntax)
  if !l == "%s" then {
    \syntax.nametable | ⟨complain no names specified 237c⟩
    ⟨set limit based on ipt and the name table 237a⟩
    namearray := name_array_from_table(syntax.nametable, limit, argname)
    namearray.storageclass := "static"
    PPxwrite(pp, pretty(Gdeclnamearray(namearray)), ";") # local name array
    if type(ipt.meaning) == "null" then
      PPxwrite(pp, pretty(Sif([
          Sguarded(set([literal("0 <= " || argname),
                         literal(argname || " < " || limit)]),
                   Sepsilon()),
          Sguarded(set(),
                   literal("{asmprintf(asmprintfd, " ||
                            image(bad_operand_name(argname, "%d")) || ", " ||
                            argname || "); return;}"))])))
  }
  every x := !l do
    case x of {
      "%s" : emit_asm_printf(pp, "%s", namearray.codename || "[" || argname || "]")
      "%d" | "%u" | "%x" : emit_asm_printf(pp, x, argname)
      "%%" : emit_asm_printf(pp, "%%")
      default :
        if match("%", x) then
          error("Unknown escape ", x, " in syntax string for argument ", argname)
        else
          emit_asm_printf(pp, "%s", image(x))
    }
  PPxwrite(pp, "$b$n}")
  p := "print_" || sign || "_" || argname
  save_printproc(p, syntax, argtype)
  return p
end
```

Defines:
create_integer_print_proc, used in chunk 235.

Uses asmprintf 394d, asmprintfd 394d, bad_operand_name 343b, emit_asm_printf 239b, Gdeclnamearray 281b, literal 43e, lookup_printproc 237d, namearray 279a, name_array_from_table 343b, percent_split 239a, pretty 16a, save_printproc 237d, Sepsilon 79, Sguarded 79, and Sif 79.

237a ⟨*set* limit *based on* ipt *and the name table* 237a⟩≡ (236)

```
case type(ipt.meaning) of {
  "field"   : limit := 2^fwidth(ipt.meaning)
  "integer" : limit := 2^ipt.meaning
  default   : { limit := 0; every limit <:= !syntax.nametable; limit +:= 1 }
}
```

Uses field 53 and fwidth 112d.

237b ⟨*int-print-header.t* 237b⟩≡

```
static void print_%sign_%name(%type %name) {$t
```

237c ⟨*complain no names specified* 237c⟩≡ (236)

```
error("Tried to use field names to format argument ", argname,
      ", but no name specifier was given")
```

Uses field 53.

To keep track of print procedures, the easy case is one in which no field names are used: we just look up a procedure by its syntax and argument type. Otherwise we need a key that captures all the field names. I use "\0" to separate parts of a key. This trick isn't perfectly safe, but names are most unlikely to contain the null character, and if they do, the C code is going to break anyway.

237d ⟨* 223a⟩+≡ ◁236 238a▷

```
global printproctab

procedure printproc_key(syntax, argtype)
  k := syntax.syntax || "\0" || argtype
  if find("%s", syntax.syntax) then
    k ||:= nametablekey(syntax.nametable)
  return k
end

procedure lookup_printproc(syntax, argtype)
  initial /printproctab := table()
  return \printproctab[printproc_key(syntax, argtype)]
end

procedure save_printproc(procname, syntax, argtype)
  initial /printproctab := table()
  (/printproctab[printproc_key(syntax, argtype)] := procname) |
  impossible("duplicated print procedures")
  return procname
end
```

Defines:
lookup_printproc, used in chunks 236 and 238a.
save_printproc, used in chunks 236 and 238a.
Uses nametablekey 344a.

Here's the print procedure for a constructor type. Note that we don't have to worry about whether the print procedures for the individual constructors are out, because the global ordering on constructors guarantees they've all been emitted.[1]

238a ⟨* 223a⟩+≡ ◁237d 239a▷

```
procedure create_constype_print_proc(pp, argname, constype)
  local call, prefix
  static constype_syntax
  initial constype_syntax := operand_syntax("")
  if p := lookup_printproc(constype_syntax, constype.name) then return p
  emit_template(pp, "constype-print-header.t", "type", constype.name)
  s := Stagcase(argname, constype, table())
  every cons := kept_constructors(constype) do {
    ⟨let call represent a call to the assembly proc for cons 238b⟩
    s.arms[cons] := literal(call)
  }
  PPxwrite(pp, pretty(s), "$b$n}")
  p := "print_" || constype.name
  save_printproc(p, operand_syntax(""), constype.name)
  return p
end
```

Defines:
create_constype_print_proc, used in chunk 235.
Uses constype 155, kept_constructors 31d, literal 43e, lookup_printproc 237d, operand_syntax 229b, pretty 16a, save_printproc 237d, and Stagcase 178a.

Just blast out the arguments...

238b ⟨*let* call *represent a call to the assembly proc for* cons 238b⟩≡ (238a)

```
call := [assembler_proc_name(cons)]
prefix := "("
every ipt := inputs_of(cons) do {
  put(call, prefix); prefix := ", "
  put(call, pretty(Einstance_input(argname, cons, ipt.name)))
}
put(call, ");")
```

Uses assembler_proc_name 233, Einstance_input 79, inputs_of 162a, and pretty 16a.

238c ⟨*constype-print-header.t* 238c⟩≡

```
static void print_%type(%type_Instance %type) {$t
```

[1]There are probably several lurking bugs if somebody discards a typed constructor.

10.5 Little utility procedures

`percent_split` divides up a syntax string such that the escapes are obvious.

239a ⟨* 223a⟩+≡ ◁238a 239b▷
```
procedure percent_split(s)
  l := []
  s ?
    while not pos(0) do {
      if not match("%") then
        put(l, tab(upto('%') | 0))
      while match("%") do
        put(l, move(2)) | error("unescaped % at end of syntax string")
    }
  return l
end
```
Defines:
`percent_split`, used in chunks 230c and 236.

`emit_asm_printf` emits a call to `asmprintf`[2].

239b ⟨* 223a⟩+≡ ◁239a 241a▷
```
procedure emit_asm_printf(pp, fmt, args[])
  PPxwrites(pp, "$nasmprintf(asmprintfd, ")
  PPwrites(pp, image(fmt))
  every PPwrites(pp, ", ", !args)
  PPwrites(pp, ");")
  return
end
```
Defines:
`emit_asm_printf`, used in chunks 234 and 236.
Uses `asmprintfd` 394d.

[2]Note the complete lack of consistency in my use of the underscore

Chapter 11

Emitting assembly-language grammars

241a ⟨* 223a⟩+≡ ◁239b 241b▷

```
procedure emit_asm_grammar(outfilename)
  local outfile
  verbose("Emitting assembly-language grammar")
  (outfile := openfile(outfilename, "w")) |
                error("could not open ", image(outfilename), " for writing")
  pushtrace("ASMGRAMMAR")
  every emit_cons_production(outfile, kept_constructors())
  poptrace()
  return
end
```

Defines:
  emit_asm_grammar, used in chunk 18.
Uses emit_cons_production 242, kept_constructors 31d, openfile 22, and verbose 16a.

We use the nonterminal Instruction for untyped constructors. For typed constructors we use the name of the type plus the word Operand.

241b ⟨* 223a⟩+≡ ◁241a 242▷

```
procedure nonterminal_name(cons)
  return if cons.type === instructionctype then "Instruction"
         else cons.type.name || "Operand"
end
```

Defines:
  nonterminal_name, used in chunk 242.
Uses instructionctype 30a.

To emit a production, we use the operands and the concrete syntax. We share `consname2asm` to determine the syntax for the constructor name (and whether to use it at all!). The semantic action calls an encoding procedure.

242 ⟨* 223a⟩+≡ ◁241b 243▷

```
procedure emit_cons_production(outfile, cons)
  local asmname, i, operand_positions, procname
  every create_field_nonterminal(outfile, inputs_of(cons, "field"))
  writes(outfile, nonterminal_name(cons), " :")
  asmname := consname2asm(cons)
  i := 1
  if *asmname > 0 then
    i +:= emit_literal_syntax(outfile, asmname)
  operand_positions := []
  every o := !asmoperands(cons) do
    case type(o) of {
      "literal" : i +:= emit_literal_syntax(outfile, o.s)
      "input"   :  {
          put(operand_positions, "$" || i)
          i +:= emit_operand_syntax(outfile, o)
        }
      default : impossible("operand type")
    }
  procname := Cnoreserve(cons.name)
  write(outfile, " /* ",
           if cons.type === instructionctype then "" else "$0 = ",
           (\indirectname || "->" | "") || procname,
                "(", commaseparate(operand_positions), "); */;")
  return
end
```

Defines:
`emit_cons_production`, used in chunk 241a.

Uses `asmoperands` 231a, `Cnoreserve` 322b, `consname2asm` 226a, `create_field_nonterminal` 245, `emit_literal_syntax` 243, `emit_operand_syntax` 244, `field` 53, `indirectname` 16a, `input` 44a, `inputs_of` 162a, `instructionctype` 30a, `literal` 43e, and `nonterminal_name` 241b.

This is a first cut and will have to be refined to tokenize things properly.

243 ⟨* 223a⟩+≡ ◁242 244▷

```
procedure emit_literal_syntax(outfile, s)
  static alphanum, letters, multichar
  initial { letters := &letters ++ '_.'
            alphanum := letters ++ &digits
            multichar := ["<=", ">=", "!="]
           }
  n := 0
  s ? {
    optwhite()
    while not pos(0) do {
      if any(letters) then
        writes(outfile, " ", image(tab(many(alphanum))))
      else if any(&digits) then {
        warning("Can't put digits in assembly syntax")
        tab(many(&digits))
      } else
        writes(outfile, " ", image(=!multichar | move(1)))
      n +:= 1
      optwhite()
    }
  }
  return n
end
```

Defines:
emit_literal_syntax, used in chunks 242 and 245.
Uses optwhite 350a.

This, too, is temporary and bogus. We need to take the same directives used to create `operand_syntax_tab` and use them to create a sensible input syntax.

244 ⟨* 223a⟩+≡ ◁243 245▷

```
global operand_input_syntax_tab
procedure emit_operand_syntax(outfile, ipt)
  initial /operand_input_syntax_tab := table()
  s := \operand_input_syntax_tab[ipt.name] |
    case type(ipt.meaning) of {
      "null"     : "Integer"
      "integer"  : "Integer"
      "field"    : field_nonterminal[ipt.meaning]
      "string"   : "Relocatable"
      "constype" : ipt.meaning.name || "Operand"
      default    : impossible("type of operand")
    }
  writes(outfile, " ", s)
  return 1
end
```

Defines:
`emit_operand_syntax`, used in chunk 242.
Uses `constype` 155, `field` 53, and `field_nonterminal` 245.

245 ⟨* 223a⟩+≡ ◁244

```
global field_nonterminal # maps fields to nonterminal names

procedure create_field_nonterminal(outfile, ipt)
  local nt, prefix
  /field_nonterminal := table()
  f := ipt.meaning
  if member(field_nonterminal, f) then return field_nonterminal[f]
  t := fieldname_table(f := ipt.meaning)
  return field_nonterminal[f] :=
    if /t then
      "Integer"
    else if fprime := key(field_nonterminal) & t === fieldname_table(fprime) then
      field_nonterminal[fprime] # reuse
    else {
      nt := f.name || "Field"
      write(outfile, nt)
      prefix := ":"
      every p := !sort(t, 2) do {
        writes(outfile, "  ", prefix)
        emit_literal_syntax(outfile, p[1])
        write(outfile, " /* $0 = ", p[2], "; */")
        prefix := "|"
      }
      write(outfile, "  ;\n")
      nt
    }
end
```

Defines:
`create_field_nonterminal`, used in chunk 242.
`field_nonterminal`, used in chunk 244.
Uses `emit_literal_syntax` 243 and `fieldname_table` 342a.

Chapter 12

Support for pattern-matching statements

Much of what happens here is the dual of what is done in **constructors.nw**. Abstract syntax is converted to patterns, but this time the free variables must be discovered, and they become the inputs to, not the outputs from, the equation-solving process.

12.1 Pattern syntax in decoding

Free variables induce field or constructor bindings (although we have no support yet for constructor bindings). What we really want is for all the constructor stuff to DWIM. Whatever that is. All patterns are resolved into absolute normal form (q.v.).

247 ⟨* 247⟩≡ 248 ▷

```
procedure resolve_case_patterns(cs)
  a := []
  every put(a, new_matching_arms(!cs.arms, cs.succptr))
  cs.arms := a
  return cs
end
```

Defines:
`resolve_case_patterns`, used in chunk 19a.
Uses `new_matching_arms` 248 and `succptr` 346b.

In resolving an arm, the first step is to find all the free identifiers, so we can do some suitable injection with them. We'll also identify the subset of those identifiers that become the binding instances that might be used in equations. We can then put the pattern first into normal form, then into absolute normal form.

At that point, we have our usual encoding-type pattern, with fields bound to expressions, which is emphatically no good for decoding. The next step is to introduce fresh variables to the field bindings for which we need expressions, creating a set of equations used solely to invert the field bindings. We'll solve that set of equations, giving us our goo solely in terms of the absolute fields. We'll then be able to apply it to any conditions.

Eventually we'll also have to deal with any explicit equations, including identifying any additional binding instances of identifiers that appear only in such equations.

Only identifiers that are bound in every disjunct can reliably be used in code or in equations.

old: In principle, an identifier could be bound to different fields in different disjuncts. I resolve that problem by forbidding the construct; a single identifier must always be bound to the same field, although it may appear in different positions and therefore in different absolute fields.

248 ⟨* 247⟩+≡ ◁247 250b▷

```
procedure new_matching_arms(a, succptr)
  local idents, Ii, id, patlabeltab, freevars
  pp := PPnew(\mdebug)
###  idents := set(); every insert(idents, pattern_free_variables(a.pattern))
###  rho := newscope(globals)
###  <inject [[idents]] into [[rho]]>
###  patlabeltab := table()
###  every n := pattern_label_names(a.pattern) do
###    patlabeltab[n] := n
###  push(rho, patlabeltab)
  freevars := table()
  x := pnf(a.pattern, globals, freevars)
  PPxwrite(\pp, "after pnf we have case arm $t$o", ppexpimage(x), "$b")
  x := bind_patlabel_names(x, 1)
  PPxwrite(\pp, "after binding patlabel names we have case arm $t$o", ppexpimage(x), "$b")
  every set_patlabel_offsets(!x.disjuncts)
  PPxwrite(\pp, "after setting offsets we have case arm $t$o", ppexpimage(x), "$b")
  a.pattern := anf(x)
  a.pattern := gsubst(a.pattern, always_fail)   # remove wildcards and instances,
                                                # turn latent to real labels
  PPxwrite(\pp, "after anf+subst we have case arm $t$o", ppexpimage(a.pattern), "$b")
  ⟨remove from freevars names of pattern labels in a.pattern 249a⟩
  if *freevars > 0 then
    write(\mdebug, "caught free variables:", envimage(freevars, "freevars"))
```

```
    ⟨dump image of case arm before simplification 249b⟩
    x := simplify(a.pattern)
    if not exps_eq(x, a.pattern) then {
      a.pattern := x
      ⟨dump image of case arm after simplification 249c⟩
    } else
      write(\mdebug, "Case arm doesn't simplify")
    idents := all_disjuncts_ids(a.pattern, free_var_or_patlabelbinding_name,
                                "Ignoring bound identifier ",
                                " because it doesn't appear in every disjunct")
    if *idents > 0 then
      write(\mdebug, "Saw binding instances of ", commafy(sort(idents)))
    else
      write(\mdebug, "No binding instances")
    a.soln := solve(balance_eqns(\a.eqns), idents)
    ⟨convert field bindings to answers, split arm as needed, and suspend results 251a⟩
  end
```

Defines:
  `new_matching_arms`, used in chunk 247.

Uses `all_disjuncts_ids` 56, `always_fail` 250b, `anf` 257, `arm` 259b, `balance_eqns` 144b, `bind_patlabel_names` 55, `disjunct` 53, `envimage` 25a, `exps_eq` 82c, `free_var_or_patlabelbinding_name` 250c, `globals` 30a, `gsubst` 117a, `inject` 27, `mdebug` 16a, `newscope` 24b, `patlabel` 53, `pattern` 53, `pattern_free_variables` 76b, `pattern_label_names` 77a, `pnf` 72, `ppexpimage` 98d, `set_patlabel_offsets` 57c, `simplify` 89b, `solve` 132a, `subst` 86b, and `succptr` 346b.

249a ⟨*remove from* `freevars` *names of pattern labels in* `a.pattern` 249a⟩≡ (248)

```
  every delete(freevars, key(\(!a.pattern.disjuncts).patlabelbindings))
```

Uses `pattern` 53.

249b ⟨*dump image of case arm before simplification* 249b⟩≡ (248)

```
  PPxwrites(\pp, "Before simplification, case arm is $t${$c", ppexpimage(a.pattern))
  PPxwrites(\pp, "$o{ ", ppexpimage(\a.eqns), " }")
  PPxwrite(\pp, "$}$b")
```

Uses `arm` 259b, `pattern` 53, and `ppexpimage` 98d.

249c ⟨*dump image of case arm after simplification* 249c⟩≡ (248)

```
  PPxwrites(\pp, "After simplification, case arm is $t${$c", ppexpimage(a.pattern))
  PPxwrites(\pp, "$o{ ", ppexpimage(\a.eqns), " }")
  PPxwrite(\pp, "$}$b")
```

Uses `arm` 259b, `pattern` 53, and `ppexpimage` 98d.

Note that we're letting field names be used either to stand for patterns or to be binding instances of integers (just like predefined identifiers). We think it's OK to allow both, but we're not sure. In any case, it was too annoying not to allow filed names as binding instances in constructor applications.

250a ⟨*inject* idents *into* rho 250a⟩≡

```
every i := !idents do
  case type(symtab[i]) of {
    "field"    : add_to_rho(i, inject(constraints2pattern(fieldbinding(symtab[i], i)),
                                      i, &null), rho)
    "pattern"  : add_to_rho(i, inject(symtab[i], &null, &null), rho)
    "constype" :
       add_to_rho(
          i,
          inject(seqpx(label2pattern(i),
                       consinput_pattern(ipt := input(Ewildcard(i), symtab[i]))),
                 &null,
                 ipt),
          rho)
    "null"     : add_to_rho(i, inject(&null, i, &null), rho)
    default    : error("I can't make sense of ", image(i), " (a ", type(symtab[i]),
                       ") within a pattern")
  }
```

Uses add_to_rho 24a, consinput_pattern 167a, constraints2pattern 59, constype 155, field 53, fieldbinding 53, inject 27, input 44a, label2pattern 58a, pattern 53, seqpx 62a, and symtab 30a 353a.

250b ⟨* 247⟩+≡ ◃248 250c▹

```
procedure always_fail()
  fail
end
```

Defines:
always_fail, used in chunk 248.
Uses fail 396.

250c ⟨* 247⟩+≡ ◃250b 251c▹

```
procedure free_var_or_patlabelbinding_name(d)
  suspend free_variables(d) | key(\d.patlabelbindings)
end
```

Defines:
free_var_or_patlabelbinding_name, used in chunk 248.
Uses free_variables 82b.

251a ⟨*convert field bindings to answers, split arm as needed, and suspend results* 251a⟩≡ (248)
```
  acount := 0
  lasta := &null
  every d := !a.pattern.disjuncts do {
    s := fieldbindings_to_soln(d, idents)
    l := if \succptr then d.length else &null
    if (\lasta).imp_soln === s & lasta.patlen === l &
       patlabelbindings_match(lasta.pattern.disjuncts[1].patlabelbindings,
                              d.patlabelbindings)
    then
      put(lasta.pattern.disjuncts, d)
    else {
  ⟨show last answers and conditions 251b⟩
  write(\mdebug, "Answers and conditions:", solnimage((\lasta).imp_soln))
      suspend \lasta
      lasta := copy(a)
      lasta.pattern := copy(a.pattern)
      lasta.pattern.disjuncts := [d]
      lasta.imp_soln := s
      lasta.patlen := l
      acount +:= 1
    }
  }
  if acount > 1 then
    write(\mdebug, "split case arm at ", image(a.file), ", line ", a.line,
                 " because of varying implicit solutions or pattern lengths")
  ⟨show last answers and conditions 251b⟩
  suspend \lasta
```
Uses `arm` 259b, `fieldbindings_to_soln` 252, `file` 345a, `line` 345a, `mdebug` 16a, `patlabelbindings_match` 251c, `pattern` 53, `solnimage` 363c, `split` 265a, and `succptr` 346b.

251b ⟨*show last answers and conditions* 251b⟩≡ (251a)
```
  write(\mdebug, "Answers and conditions (", image((\lasta).imp_soln),
        "):", solnimage((\lasta).imp_soln))
```
Uses `mdebug` 16a and `solnimage` 363c.

251c ⟨* 247⟩+≡ ◁250c 252▷

```
procedure patlabelbindings_match(t1, t2)
  if      /t1 then return (/t2 | *t2 = 0, t1)
  else if /t2 then return (/t1 | *t1 = 0, t1)
  else if k := key(t1) & (/t2[k] | t1[k] ~= t2[k]) then fail
  else if k := key(t2) & (/t1[k] | t1[k] ~= t2[k]) then fail
  else return t1
end
```

Defines:
`patlabelbindings_match`, used in chunk 251a.
Uses `fail` 396.

For each disjunct, we turn field bindings into answers and conditions, using the following steps:

1. Remove all the field bindings from the constraints, making a new equation for each field binding. Because the solve works with variables, use the function `afield_var` to map each absolute field to its corresponding variable. The table `afield_vars_inverse` is used later to perform the inverse mapping. The variables introduced by `afield_var` stand for the absolute fields in the bindings, and they are exactly the variables that are the inputs to the equations.

2. Solve the equations, and build the answers table by using the values they give to the identifiers in `idents`. At this point we inverse-map the variables, so the results are free in absolute fields, not in these auxiliary variables.

3. Add any constraints produced by the equations to the conditions of the disjunct, performing the same inverse mapping.

252 ⟨* 247⟩+≡ ◁251c 253c▷

```
procedure fieldbindings_to_soln(d, idents)
  local newconstraints, newconditions, eqns, soln, inputs, x, result
  every newconstraints | eqns := []
  inputs := set()
  every insert(inputs, key(\d.patlabelbindings))
  afield_vars := table()
  every c := !d.aconstraints do
    case type(c) of {
      "constraint"   : put(newconstraints, c)
      "fieldbinding" : {
         insert(inputs, x := afield_var(c.field))
         put(eqns, eqn(x, "=", c.code))
      }
      default : impossible("constraint type")
    }
  d.aconstraints := newconstraints
  newconditions := set()
  every e := !\d.conditions do
    case type(e) of {
      "eqn"   : put(eqns, e)
      default : insert(newconditions, e)
    }
  d.conditions := if *newconditions > 0 then newconditions else &null
  soln := solve(balance_eqns(eqns), inputs)
    # soln.answers : bound [[idents]] -> exps in afield_vars
  result := solution(table(), set())
  ⟨make sure there's an answer for every non-label in idents 253a⟩
```

```
    every id := !idents do
      result.answers[id] :=
        if member(\d.patlabelbindings, id) then
          binop(the_global_pc, "+", d.patlabelbindings[id])
        else
          super_simplify(subst_tab(soln.answers[id], afield_vars_inverse, 1))
              # is super_simplify really safe here?  dunno...
    every insert_condition(result.constraints,
            super_simplify(subst_tab(!soln.constraints, afield_vars_inverse, 1)))
    return unique_soln(result)
  end
```

Defines:
fieldbindings_to_soln, used in chunk 251a.
Uses afield_var 253c, afield_vars 253c, afield_vars_inverse 253c, balance_eqns 144b, binop 110a, constraint 53, constraints 132a, eqn 81c, field 53, fieldbinding 53, insert_condition 123c, solution 132b, solve 132a, subst_tab 87a, super_simplify 90a, the_global_pc 17d, and unique_soln 254a.

253a ⟨*make sure there's an answer for every non-label in* idents 253a⟩≡ (252)

```
  if (id := !idents, not member(\d.patlabelbindings, id), /soln.answers[id]) then {
    PPxwrite(PPnew(&errout),"Error: trouble with disjunct ${$t$o", ppexpimage(d), "$b$}")
    ⟨make l list of idents not in soln.answers 253b⟩
    error("Can't solve for ", commafy(sort(l)), envimage(soln.answers, "soln.answers"))
  }
```

Uses disjunct 53, envimage 25a, ppexpimage 98d, and solve 132a.

ge #15 Better error message:

253b ⟨*make* l *list of* idents *not in* soln.answers 253b⟩≡ (253a)

```
  l := []
  every /soln.answers[x := !idents] & not member(\d.patlabelbindings, x) do
    put(l, x)
```

253c ⟨* 247⟩+≡ ◁252 254a▷

```
  global afield_vars, afield_vars_inverse
  procedure afield_var(af)
    initial {
      every afield_vars | afield_vars_inverse := table()
    }
    afield_vars_inverse[/afield_vars[af] := fresh_variable(af.field.name)] := af
    return afield_vars[af]
  end
```

Defines:
afield_var, used in chunk 252.
afield_vars, used in chunk 252.
afield_vars_inverse, used in chunk 252.
Uses field 53 and fresh_variable 151b.

254a ⟨* 247⟩+≡ ◁253c 254b▷

```
procedure unique_soln(soln)
  static cache
  initial cache := table()
  /cache[*soln.constraints] := table()
  t := cache[*soln.constraints]
  l := sort(soln.answers)
  k := "";
  every p := !sort(soln.answers) do k ||:= p[1] || " " || type(p[2]) || " "
  /t[k] := set()
  if s := !t[k] & answers_eq(s.answers, soln.answers) &
     exps_eq(s.constraints, soln.constraints) then
    return s
  else {
    insert(t[k], soln)
    return soln
  }
end
```

Defines:
`unique_soln`, used in chunk 252.
Uses `answers_eq` 254b, `constraints` 132a, and `exps_eq` 82c.

254b ⟨* 247⟩+≡ ◁254a

```
procedure answers_eq(a1, a2)
  if a1 === a2 then return
  *a1 = *a2 | fail
  if id := key(a1) & not exps_eq(a1[id], \a2[id]) then fail
  return
end
```

Defines:
`answers_eq`, used in chunk 254a.
Uses `exps_eq` 82c and `fail` 396.

Chapter 13

Generating decision trees

The crux of the problem is to transform a *matching statement* into a *decision tree*. A matching statement has a *value*, a sequence of *arms*, and a *trailer*. Each arm has a pattern, and code to be executed. When the matching statement is executed, it chooses the first arm whose pattern matches the value, then executes the corresponding code, then executes the trailer. I generate a *decision tree* to do the job. Each internal node of the decision tree tests a field of a word. It then chooses an edge (child) based on what range constraints can be satisfied by the value of that field, and it continues testing fields until it reaches a leaf, at which time it executes the code associated with that leaf.

The goal of tree generation is not to generate just any tree, but the tree with the fewest nodes. This problem is NP-complete, so I apply a few heuristics. The results, at least for the machine descriptions I use, seem to be as good as what I would come up with by hand.

When a pattern is in normal form, it is not obvious what word is tested by a particular range constraint; one needs to know the position of the sequent containing the range constraint. To make the problem simpler, I put the patterns into a new *absolute normal form*, which is described by the following rules:

1. Each disjunction contains not a list of sequents but a set of range constraints and field bindings.
2. The range constraints and field bindings are made "absolute" by using an `absolute_field` in place of a `field`. The `absolute_field` gives the bit offset of the word containing the field (its size is available from the field's class).

In support of this scheme, we use "absolute disjuncts."

256 ⟨* 256⟩≡ 257▷

```
  record adisjunct(aconstraints, name, conditions, length, patlabelbindings)
                                        # list of absolute constraints, name, conds
  record absolute_field(field, offset)    # used to make absolute constraints
```

Defines:
`absolute_field`, used in chunks 83a, 90c, 101a, 114c, 119b, 259a, 288a, 321a, 332c, 357b, and 358a.
`adisjunct`, used in chunks 83a, 90c, 91c, 100a, 114b, 119a, 257, 267a, 355d, and 358a.
Uses `constraints` 132a, `field` 53, and `used` 132a.

We have to store the length explicitly in an `adisjunct`, because sequents that constrain no fields are lost. The `patlabelbindings` binds label names to offsets, which are expressed in PC units, not bits.

13.1 Transformation to absolute normal form

The transformation is simply a matter of adding up word sizes to compute offsets. I cache absolute fields to avoid allocating gazillions of them.

257 ⟨* 256⟩+≡ ◁256 258a▷

```
procedure anf(p)
  return pattern(maplist(anfd, p.disjuncts), p.name)
end

procedure anfd(d)
  local offset
  offset := 0
  l := []
  t := table()
  every s := !d.sequents do
    case type(s) of {
      "sequent"  : { every put(l, aconstraint(!s.constraints, offset))
                     offset +:= s.class.size
                   }
      "patlabel" : t[\s.name] := bits_to_pcunits(offset)
      "latent_patlabel" : &null
      default    : impossible("sequent type")
    }
  a := adisjunct(l, d.name, d.conditions, offset, if *t > 0 then t else &null)
  return gsubst(a, Epatlabel_to_Epc_by_table, t, a)
end
```

Defines:
`anf`, used in chunk 248.

Uses `aconstraint` 258b, `adisjunct` 256, `bits_to_pcunits` 62b, `constraints` 132a, `Epatlabel_to_Epc_by_table` 258a, `gsubst` 117a, `latent_patlabel` 53, `patlabel` 53, `pattern` 53, and `sequent` 53.

During decoding we eliminate the pattern label offsets by using a table of bindings. If the label is already bound, of course, we need do nothing.

258a ⟨* 256⟩+≡ ◁257 258b▷
```
procedure Epatlabel_to_Epc_by_table(x, t, a)
  if type(x) == "Epatlabel" then
    return if /x.l.name then Epatlabel_to_Epc(x)
           else {
  write(\mdebug, "====> RESORTED TO TABLE in ", expimage(x))
  binop(the_global_pc, "+", \t[\x.l.name]) |
               impossible("in ", expimage(a), "---Label ", x.l.name,
                          " not used yet, but is not in table:",
                          envimage(t, "pattern_table"))
            }
end
```
Defines:
`Epatlabel_to_Epc_by_table`, used in chunk 257.
Uses `binop` 110a, `envimage` 25a, `Epatlabel` 79, `Epatlabel_to_Epc` 185a, `expimage` 98a, `mdebug` 16a, `the_global_pc` 17d, and `used` 132a.

I don't cache constraints, but I do cache fields. I have absolutely no measurements to justify either decision, but it simplifies the code to make absolute fields unique (as fields are) because they can be inserted into sets.

258b ⟨* 256⟩+≡ ◁258a 259a▷
```
procedure aconstraint(c, offset)
  return case type(c) of {
    "constraint"   : constraint(afield(c.field, offset), c.lo, c.hi)
    "fieldbinding" :
       if x := constant(super_simplify(c.code)) then
         constraint(afield(c.field, offset), x, x+1)
       else
         fieldbinding(afield(c.field, offset), c.code)
    default : impossible("constraint type")
  }
end
```
Defines:
`aconstraint`, used in chunk 257.
Uses `afield` 259a, `constant` 111b, `constraint` 53, `field` 53, `fieldbinding` 53, and `super_simplify` 90a.

259a ⟨* 256⟩+≡ ◁258b 259b▷

```
procedure afield(f, offset)
  static tables
  initial tables := table()
  /tables[offset] := table()
  /tables[offset][f] := absolute_field(f, offset)
  return tables[offset][f]
end
```

Defines:
`afield`, used in chunk 258b.
Uses `absolute_field` 256.

13.2 Structure of matching statements and tree nodes

The arms of the matching statement have some extra information. The file and line number help with error message and make it possible to generate `#line` statements that identify the source of the code. The original arm gives the arm from which the current arm is derived, and is useful for many of the heuristics.

259b ⟨* 256⟩+≡ ◁259a 260a▷

```
record matching_stmt(arms,valcode,succptr,trailer)
                # case arms, code to compute value, id to set to end of p, trailing code
record arm(file, line, pattern, eqns, soln, imp_soln, patlen, name, code, original)
                # pattern (in absolute normal form), eqns, name, and code are the content
                # line, file, original(pattern) are used for error reporting
```

Defines:
`arm`, used in chunks 49a, 93b, 99a, 176b, 248, 249, 251a, 260b, 262, 263, 268b, 288, and 358a.
`matching_stmt`, used in chunks 49a and 358a.
Uses `file` 345a, `line` 345a, `pattern` 53, `succptr` 346b, `used` 132a, and `value` 132a.

`imp_soln` gives answers and conditions associated with identifiers that appear as field bindings or constructor operands in the pattern. These identifiers are the *inputs* to the equations. This construct is a little odd, because the meanings of bound identifiers and the conditions that need to be satisfied are more naturally associated with disjuncts, not arms. We "raise the differences" by splitting arms until each arm as a unique such "implicit solution." We further guarantee the uniqueness of the `imp_soln` field. The reason for going to all this trouble is to simplify the task of dagging the eventual decision tree: we'll be able to unify nodes just by taking the `image()` of the `imp_soln` field (along with a few other goodies, of course).

If `succptr` was requested in the corresponding case statment, `patlen` gives the length of the pattern in the arm. We split arms as needed to make lengths unique. If `succptr` wasn't requiested, `patlen` is null. `patlen` is assigned by `resolve_case_arms`.

Each node of the decision tree is associated with a particular matching statement. Internal nodes have children, and a `field` and `offset` that say which field of which word we decided to test on. The edges that point to the children record the interval of values for the particular child. Leaf nodes have a `name` that records the name of the pattern known to match at that leaf node.

260a ⟨* 256⟩+≡ ◁259b 260b▷

```
record node(cs, children, field, offset, name, parent)
        # matching statement, list of edges to children, field chosen, pattern name
        #       (name field used to support name operator, assigned only to leaves)
record edge(node, lo, hi)
        # node pointed to and lo and hi interval of field for this edge
```

Defines:
`edge`, used in chunks 262, 265a, and 279b.
`node`, used in chunks 15f, 20b, 21a, 262, 264a, 265a, 268, 269, 272, 273a, 278–80, 284–87, 319c, 320a, 332a, 359b, and 360.
Uses `field` 53, `pattern` 53, and `used` 132a.

To create a decision tree, I begin with a node containing the full, original matching statement. I then use a "work queue" approach to check each node and see if it needs to be split. If no pattern matches the node, or if the first pattern always matches (with a unique name), no further splitting needs to be done, and I assign a name to the leaf.[1] Otherwise, I split the node.

260b ⟨* 256⟩+≡ ◁260a 261▷

```
procedure needs_splitting(n)
    local name
    if *n.cs.arms = 0 then fail
    if not guard_always_satisfied(n.cs.arms[1].imp_soln.constraints) then
        return # first arm can't always match.
    p := n.cs.arms[1].pattern
    name := \p.disjuncts[1].name | p.name
    every d := !p.disjuncts do {
        n := \d.name | p.name
        if n ~=== name then
          return   # needs splitting if names or answers are different
        else if adalwaysmatches(d) then
          fail  # always matches, needn't split
    }
    return                      # pattern doesn't always match -> split
end
```

Defines:
`needs_splitting`, used in chunks 262 and 268b.
Uses `adalwaysmatches` 261, `arm` 259b, `constraints` 132a, `fail` 396, `guard_always_satisfied` 93c, `pattern` 53, and `split` 265a.

[1] If the name isn't used, I assign the name `"-unused-"`, because that will make it easier to combine nodes in the dagging phase.

I need different procedures to check matching because the patterns are in absolute normal form.

261 ⟨* 256⟩+≡ ◃260b 262▹

```
procedure aalwaysmatches(p)
    return adalwaysmatches(!p.disjuncts)
end
procedure adalwaysmatches(d)
  if type(!d.aconstraints) == "constraint" then fail
  else return guard_always_satisfied(d.conditions)
end
```

Defines:
`aalwaysmatches`, used in chunk 266.
`adalwaysmatches`, used in chunks 260b and 355d.

Uses `constraint` 53, `fail` 396, and `guard_always_satisfied` 93c.

tree converts a matching statement into a decision tree.

262 ⟨* 256⟩+≡ ◁261 263a▷

```
procedure tree(cs)
    local armcount, arm, armname, nodename
    static heuristics
    initial {
       heuristics := [leafarms, childarms, nomatch, childdisjuncts, branchfactor]
    }
    if verbose ~=== nop then writes(&errout, "Decision tree")
    root := node(copy(cs), [])  # need empty children in case root not split
    work := [edge(root)]        # work queue of edges (nodes) to be expanded
    while n := get(work).node do {
        if verbose ~=== nop then writes(&errout, ".")
        if (needs_splitting(n) & *(afields := mentions(n.cs)) > 0) then {
            ⟨split node n and add children to work queue 264a⟩
        } else {
            write(\sdebug, "Not splitting ",
                  commaseparate(maplist(expimage, n.cs.arms), "\n"))
            armcount := *n.cs.arms
            trim_impossible_arms(n.cs)
            n.name := case *n.cs.arms of {
                        0 : "-NOMATCH-"
                        default: get_nodename(n)
                      }
            if \lc_pat_names then n.name := map(\n.name)
            if armcount > *n.cs.arms then
              write(\sdebug, "Trimmed node is ",
                  commaseparate(maplist(expimage, n.cs.arms), "\n"))
       }
    }
    if verbose ~=== nop then write(&errout, "built.")
    return root
end
```

Defines:
tree, used in chunks 9–11, 19a, and 283b.

Uses arm 259b, branchfactor 269d, childarms 269a, childdisjuncts 269c, edge 260a, expimage 98a, get_nodename 263a, lc_pat_names 16a, leafarms 268b, mentions 270b, needs_splitting 260b, node 260a, nomatch 269b, nop 362a, sdebug 16a, split 265a, trim_impossible_arms 271a, and verbose 16a.

We want to assign each tree node a name, which is derived from the names of the pattern arms that the node matches. If all pattern arms in the node have the same name N or are the null string, i.e., they do not specify a name, then the node's name is simply N. This case always holds when the node matches exactly one arm; one arm and a default (wildcard) arm; or multiple arms that all match the same constructor (possibly applied to different arguments). If the names of the pattern arms in the node are not the same, then the node's name is ambiguous, because no single name exists for all possible matches. An ambiguous node name will cause an error in `genarm`, if any of the node's pattern arms attempts to bind a `[name]` .

263a ⟨* 256⟩+≡ ◁262 264b▷

```
procedure get_nodename(n)
local nodename, armname
    nodename := armname := &null
    every arm := !n.cs.arms do
      if (armname := \(⟨Get name from pattern arm 263b⟩)) then {
        write(\sdebug, "[", image(arm.name),"] = ",
             image(armname), " for ",expimage(arm.pattern))
        if (\nodename ~== armname) then
          nodename := ⟨Ambiguous name warning 263c⟩
        else nodename := armname
      }
    return nodename
end
```

Defines:
`get_nodename`, used in chunk 262.
Uses `arm` 259b, `expimage` 98a, `pattern` 53, and `sdebug` 16a.

263b ⟨*Get name from pattern* `arm` 263b⟩≡ (263a)

```
if \arm.name then {
  \arm.pattern.disjuncts[1].name | \arm.pattern.name | &null
# "-unnamed-"
} else &null
```

Uses `arm` 259b and `pattern` 53.

263c ⟨*Ambiguous name warning* 263c⟩≡ (263a)

```
(warning("ambiguous name for pattern arm at ", arm.original.file, ", line ",
         arm.original.line, ": ", commaseparate(maplist(expimage, n.cs.arms),
         "\nAre you trying to decode a synthetic instruction?\n")),
&null)
```

Uses `arm` 259b, `expimage` 98a, `file` 345a, `line` 345a, and `pattern` 53.

Splitting a node involves choosing a field, finding out which intervals of values of that field are interesting, and creating a child node for each such interval of values. The patterns in the matching statement of the child node reflect the knowledge of the value interval of the tested field.

I make the decision by splitting the node on *each* field mentioned in the matching statement. I then compute some heuristic functions of the children from each splitting and use the best-scoring field.

Some debugging information may be written to `hdebug` or `sdebug`.

264a ⟨*split node* `n` *and add children to work queue* 264a⟩≡ (262)

```
afields := mentions(n.cs)
*afields > 0 | impossible("internal node mentions no fields")
candidates := table()
every f := !afields do
    candidates[f] := split(n, f)
⟨if debugging, split all and report 264c⟩
*afields > 1 & write(\hdebug, "Choosing one of ", patimage(afields))
every h := !heuristics do {
    if *afields = 1 then break
    afields := findmaxima(h, candidates, afields)
write(\hdebug, image(h), " chose ", patimage(afields))
}
*afields > 0 | impossible("no fields")
*afields = 1 | write(\hdebug, "tie among fields", patimage(afields), " near ",
                      image(n.cs.arms[1].original.file), ", line ",
                      n.cs.arms[1].original.line)
work |||:= n.children := candidates[n.field := ?afields]
```

Uses `field` 53, `file` 345a, `findmaxima` 268a, `hdebug` 16a, `line` 345a, `mentions` 270b, `node` 260a, `patimage` 358a, and `split` 265a.

264b ⟨* 256⟩+≡ ◁263a 265a▷

```
procedure parentchoices(n)
  l := []
  n := n.parent
  while \n do { push(l, n.field); n := n.parent }
  return l
end
```

Uses `field` 53.

264c ⟨*if debugging, split all and report* 264c⟩≡ (264a)

```
if \tryall & \hdebug & *afields > 1 then     {
  write(\hdebug, repl("=",10), " Splitting ", repl("=", 10))
  every findmaxima(!heuristics, candidates, afields) do write(\hdebug)
  write(\hdebug, repl("=", 30), "\n")
}
```

Uses `findmaxima` 268a, `hdebug` 16a, and `tryall` 16a.

To split a node, I look at each interval of values that might be interesting. I apply that interval to the matching statement, and if there can be any match, I create and add a new child node. `f` is an absolute field.

265a ⟨* 256⟩+≡ ◁264b 266▷

```
procedure split(n, f)
    local vals,v,d,val,c,p,j,i,newd,cst,child,newp, xxx

    patterns := []
    children := []
    every put(patterns, (!n.cs.arms).pattern)
    r := intervals(patterns, f)
    ⟨if debugging, write about splitting this node 265b⟩

    every i := 1 to *r - 1 do
        put(children, edge(node(apply(n.cs, f, r[i], r[i+1]),[]), r[i], r[i+1]))

    write(\sdebug, "Done splitting.\n")
    every (!children).node.parent := n
    return children
end
```

Defines:
`split`, used in chunks 11l, 12a, 251a, 260b, 262, and 264a.

Uses `apply` 266, `edge` 260a, `intervals` 270a, `node` 260a, `pattern` 53, and `sdebug` 16a.

265b ⟨*if debugging, write about splitting this node* 265b⟩≡ (265a)

```
writes(\sdebug, "Splitting ")
outpattern(\sdebug, patterns[1])
every i := 2 to *patterns do { writes(\sdebug, " | "); outpattern(\sdebug, patterns[i])}
write(\sdebug, " on ", f.field.name, " at ", f.offset)
```

Uses `field` 53, `outpattern` 354c, and `sdebug` 16a.

What is the new matching statement that results from applying lo $\leq$ f $<$ hi to cs? For each arm, I match the pattern against the interval. If it succeeds, I create a new arm for the new matching statement, containing the reduced pattern. f is an absolute field.

266 ⟨* 256⟩+≡ ◃265a 267a▹

```
procedure apply(cs, f, lo, hi)
    local newarm
    result := copy(cs)
    result.arms := []
    write(\sdebug, "    Applying ", stringininterval(patimage(f), lo, hi))
    every a := !cs.arms do {
        newarm := copy(a)
        put(result.arms, if newarm.pattern := pmatch(a.pattern, f, lo, hi) then newarm)
    }
    if *result.arms > 1 & aalwaysmatches(result.arms[1].pattern) &
       guard_always_satisfied(result.arms[1].imp_soln.constraints) then { # change 21
        write(\sdebug, "    Trimming results of apply to ", expimage(result.arms[1]))
        result.arms := [result.arms[1]]
    }
    return result
end
```

Defines:
apply, used in chunks 130d, 265a, 392, and 393.
Uses aalwaysmatches 261, constraints 132a, expimage 98a, guard_always_satisfied 93c, patimage 358a, pattern 53, pmatch 267a, sdebug 16a, and stringininterval 358b.

pmatch both tests to see whether lo $\leq$ f $<$ hi and, if so, returns the new p. f is an absolute field.

267a ⟨*256⟩+≡ ◁266 268a▷

```
procedure pmatch(p, f, lo, hi)
    result := pattern([], p.name)
    every d := !p.disjuncts do
        if c := !d.aconstraints & c.field === f & type(c) == "constraint" then
                                                        # disjunct mentions f
            if c.lo <= lo & hi <= c.hi then {           # this constraint is matched
                newd := adisjunct([], d.name, d.conditions, d.length,d.patlabelbindings)
                every c := !d.aconstraints & c.field ~=== f do
                    put(newd.aconstraints, c)
                put(result.disjuncts, newd)
            } else
                c.hi <= lo | c.lo >= hi | impossible("bad intervals")
        else                                             # disjunct does not mention f
            put(result.disjuncts, d)
    ⟨if debugging, write about results of pmatch 267b⟩
    if *result.disjuncts > 0 then return result
end
```

Defines:
pmatch, used in chunk 266.
Uses adisjunct 256, constraint 53, disjunct 53, field 53, intervals 270a, mentions 270b, and pattern 53.

267b ⟨*if debugging, write about results of* pmatch 267b⟩≡ (267a)

```
if *result.disjuncts > 0 then writes(\sdebug, "        ===> ") & outpattern(\sdebug, p)
# else writes(\sdebug, "             ") & outpattern(\sdebug, p)

if *result.disjuncts > 0 then write(\sdebug, " matches")
# else write(\sdebug, " does not match")
```

Uses outpattern 354c and sdebug 16a.

13.3 Tree-minimization heuristics

First, the boilerplate that takes a heuristic h, candidate splittings, and a set of fields, and returns the set of fields with the largest score on h.

268a ⟨* 256⟩+≡ ◁267a 268b▷

```
procedure findmaxima(h, candidates, afields)
    local max
    S := []
    every f := !afields do {
        score := h(candidates[f], f)
        write(\hdebug,"Field ", patimage(f), " scores ", score, " on ", image(h))
        /max := score - 1
        if score > max then {
            max := score
            S := [f]
        } else if score = max then
            put(S, f)
    }
    return set(S)
end
```

Defines:
findmaxima, used in chunk 264.
Uses hdebug 16a, max 396, and patimage 358a.

Here's a big pile of heuristics. I'm not sure I've ever needed more than the first two, but they're amusing and easy enough to write.

268b ⟨* 256⟩+≡ ◁268a 269a▷

```
# leafarms: prefer candidate with most arms that appear at leaf
#           nodes.  Each original arm counted only once.
#           Not matching is also counted as an arm.

procedure leafarms(children, f)
    arms := set()
    every n := (!children).node & *n.cs.arms > 0 do
       if not needs_splitting(n) then
           insert(arms, n.cs.arms[1].original)
    return *arms + if *(!children).node.cs.arms = 0 then 1 else 0
end
```

Defines:
leafarms, used in chunks 262 and 269b.
Uses arm 259b, needs_splitting 260b, and node 260a.

269a ⟨* 256⟩+≡ ◁268b 269b▷
```
# childarms: prefer the candidate with the fewest arms in children

procedure childarms(children, f)
    sum := 0
    every sum -:= *(!children).node.cs.arms
    return sum
end
```
Defines:
  childarms, used in chunks 262 and 269b.
Uses node 260a.

269b ⟨* 256⟩+≡ ◁269a 269c▷
```
# nomatch: if tied on leafarms and childarms, take candidate
#          with real leaf in preference to nomatch leaf

procedure nomatch(children, f)
    return if *(!children).node.cs.arms = 0 then -1 else 0
end
```
Defines:
  nomatch, used in chunk 262.
Uses childarms 269a, leafarms 268b, and node 260a.

269c ⟨* 256⟩+≡ ◁269b 269d▷
```
# childdisjuncts: prefer the candidate with the fewest disjuncts in children

procedure childdisjuncts(children, f)
    sum := 0
    every sum -:= *(!(!children).node.cs.arms).pattern.disjuncts
    return sum
end
```
Defines:
  childdisjuncts, used in chunk 262.
Uses node 260a and pattern 53.

269d ⟨* 256⟩+≡ ◁269c 270a▷
```
# branchfactor:  prefer the candidate with the fewest children

procedure branchfactor(children, f)
    return - *children
end
```
Defines:
  branchfactor, used in chunk 262.

13.4 Utility functions

If an absolute field `f` is to be used to split patterns, `intervals` returns a sorted list defining the intervals that need to be considered.

270a ⟨* 256⟩+≡ ◁269d 270b▷

```
procedure intervals(patterns, f)
    cuts := set([0, 2^fwidth(f.field)])
    every p := !patterns & d := !p.disjuncts & c := !d.aconstraints & c.field === f &
          type(c) == "constraint"
    do
        every insert(cuts, c.lo | c.hi)
    return sort(cuts)
end
```

Defines:
`intervals`, used in chunks 265a and 267a.
Uses `constraint` 53, `field` 53, and `fwidth` 112d.

`mentions` produces the set containing all absolute fields mentioned in a matching statement. Mentions in field bindings *don't* count; this information is for building decision trees only.[2]

270b ⟨* 256⟩+≡ ◁270a 271a▷

```
procedure mentions(cs)
    result := set()
    every a := !cs.arms & d := !a.pattern.disjuncts & c := !d.aconstraints &
          type(c) == "constraint"
    do
       insert(result, c.field)
    return result
end
```

Defines:
`mentions`, used in chunks 262, 264a, and 267a.
Uses `constraint` 53, `field` 53, and `pattern` 53.

[2] The original design had no field bindings and omitting them seems to be the best migration path.

271a ⟨* 256⟩+≡ ◁270b 271b▷

```
procedure trim_impossible_arms(cs)
  l := []
  every a := !cs.arms do
    if arm_conditions_always_satisfied(a) then {
      put(l, a)
      if *l < *cs.arms then cs.arms := l
      return cs
    } else if member(a.imp_soln.constraints, 0) |
          constant(!(\a.soln).constraints) = 0 then {
      cs.arms := l
      return cs
    } else {
      put(l, a)
    }
  return cs
end
```

Defines:
trim_impossible_arms, used in chunk 262.
Uses arm_conditions_always_satisfied 271b, constant 111b, and constraints 132a.

271b ⟨* 256⟩+≡ ◁271a 271c▷

```
procedure arm_conditions_always_satisfied(a)
  return guard_always_satisfied(a.imp_soln.constraints) &
         /a.soln | guard_always_satisfied(a.soln.constraints)
end
```

Defines:
arm_conditions_always_satisfied, used in chunk 271a.
Uses constraints 132a and guard_always_satisfied 93c.

271c ⟨* 256⟩+≡ ◁271b 272a▷

```
# find_id: tab to and past identifier id, returning its position
# ignores quotes, comment brackets

procedure find_id(id)
    static notlnum
    initial notlnum := ~ (&letters ++ &digits ++ '_')
    tab(p := find(id)) & p = 1 | (move(-1) & any(notlnum) & move(1)) &
              =id & pos(0) | any(notlnum) & suspend p
end
```

13.5 Tree checking

Once the tree is generated, it's useful to check it for redundant arms and for arms that never match. These checks will help users catch mistakes in their specifications. Note that I must check the "original" arms; that's why they're there.

272a ⟨* 256⟩+≡ ◁271c 272b▷
```
procedure checktree(n, cs)
    originals := set()
    every insert(originals, (!cs.arms).original)
    deletematching(n, originals)
    every show_unmatched(n, !originals)
    if hasnomatch(n) then
        warning("Matching statement at ", image(cs.arms[1].file), ", line ",
                n.cs.arms[1].line - 1, " doesn't cover all cases")
    return n
end
```
Defines:
checktree, used in chunk 20a.
Uses deletematching 272b, file 345a, hasnomatch 272c, line 345a, and show_unmatched 273a.

272b ⟨* 256⟩+≡ ◁272a 272c▷
```
procedure deletematching(n, originals)
    if *originals = 0 then return
    else if *n.children > 0 then every deletematching((!n.children).node, originals)
    else every delete(originals, (!n.cs.arms).original)
end
```
Defines:
deletematching, used in chunk 272a.
Uses node 260a.

272c ⟨* 256⟩+≡ ◁272b 273a▷
```
procedure hasnomatch(n)
    if *n.children > 0 then return hasnomatch((!n.children).node)
    else if *n.cs.arms = 0 then return  # found it
end
```
Defines:
hasnomatch, used in chunk 272a.
Uses node 260a.

If an arm never matches, I push its pattern through the tree and find out combinations of arms do match that pattern.

ge #33

273a ⟨* 256⟩+≡ ◃272c 273b▹

```
procedure show_unmatched(n, a)
  warning("No word matches pattern at ", image(a.file), ", line ", a.line, ".")
  write(&errout,"    Covered by patterns at")
  every find_covering_arms(n, a, !a.pattern.disjuncts)
  return
end

procedure find_covering_arms(n, a, ad)
  if *n.children = 0 then
    every a := !n.cs.arms do
      write(&errout, "\t", image(a.file), ", line ", a.line)
  else {
    c := find_or_invent_constraint(n.field, ad)
    every e := !n.children & intervals_intersect(c.lo, c.hi, e.lo, e.hi) do
      find_covering_arms(e.node, a, ad)
  }
  return
end
```

Defines:
`show_unmatched`, used in chunk 272a.
Uses `field` 53, `file` 345a, `find_or_invent_constraint` 273b, `intervals_intersect` 273b, `line` 345a, `node` 260a, and `pattern` 53.

273b ⟨* 256⟩+≡ ◃273a

```
procedure intervals_intersect(lo1, hi1, lo2, hi2)
  if hi1 <= lo2 | hi2 <= lo1 then fail else return
end

# absolute disjuncts!
procedure find_or_invent_constraint(f, d)
  return if type(c := !d.aconstraints) == "constraint" & c.field === f then c
         else constraint(f, 0, 2^fwidth(f.field))
end
```

Defines:
`find_or_invent_constraint`, used in chunk 273a.
`intervals_intersect`, used in chunk 273a.
Uses `constraint` 53, `fail` 396, `field` 53, and `fwidth` 112d.

Chapter 14

Converting trees to dags

The problem with the trees generated in the previous section is that there's a different edge, and therefore a different child, for each possible interval of the field tested, even if those children both execute exactly the same "original" arm of the case statement. The code in this section converts the trees to dags, and as part of the process it combines edges pointing to the same node. This can reduce the size of the tree by huge factors.

To make the transformation work, I have to represent a *set of intervals* on each edge, not just a single interval. Because no two intervals overlap, I can use a wonderful dirty trick, detailed below. I also *may* convert a node's name string to a `namearray` mapping field values to strings. The goal is for children of the same parent to share a single name array; that way the edges can be merged and the name operator can be implemented with an array reference. If I don't convert a node's name, the only penalty is that the tree might be bigger. (Code generation will be different for the two cases.)

Now, the dirty representation trick: I can represent a set of numbers S (a union of intervals) as two sets, lo and hi, such that

$lo \cap hi = \emptyset$

if $\texttt{sort}(lo \cup hi) = a, b, c, d, \ldots$, then $S = [a, b-1] \cup [c, d-1] \cup \ldots$.

The procedure `addinterval` adds a new interval to such a set S, relying on the fact that no two intervals overlap. The implementation may be treated as "black magic" or "mathematics;" take your pick.[1]

276a ⟨* 276a⟩≡ 276b▷

```
procedure addinterval(loset, hiset, lonum, hinum)
    if member(loset, hinum) then delete(loset, hinum) else insert(hiset, hinum)
    if member(hiset, lonum) then delete(hiset, lonum) else insert(loset, lonum)
    return
end
```

Defines:
`addinterval`, used in chunks 188a and 279b.

A new interval overlaps unless it falls between two existing intervals. The local variable `leftcount` is the number of intervals completely to the left of the new interval. `rightcount` is the number of intervals completely to the right of the new interval. If these total to the number of intervals, there is no overlap. Otherwise, there is an overlap.

276b ⟨* 276a⟩+≡ ◁276a 277a▷

```
procedure overlaps(loset, hiset, lonum, hinum)
  local leftcount, rightcount
  leftcount  := 0; every lonum >= !hiset do leftcount  +:= 1
  rightcount := 0; every hinum <= !loset do rightcount +:= 1
  return leftcount + rightcount < *loset
end
```

Defines:
`overlaps`, used in chunk 188a.

[1] Extra credit for proofs of correctness.

To show a bit set, give the number of bits.

277a ⟨* 276a⟩+≡ ◁276b 277b▷

```
procedure showbitset(loset, hiset, width)
  l := sort(loset ++ hiset)
  b := 0
  i := 0
  s := ""
  while i < width do {
    if i = l[1] then {
      get(l)
      b := 1 - b
    }
    s ||:= b
    i +:= 1
  }
  return reverse(s)
end
```

To convert trees to dags I need to be able to compare two nodes for structural identity, and the easiest way is to compute a canonical representation as a string:

```
node : [fname:patimage(list of edges)]
     | (image(node.name):image(node.cs.arms.(original,imp_soln))
edge : patimage(list of sort(loset ++ hiset)):node
```

277b ⟨* 276a⟩+≡ ◁277a 278a▷

```
procedure nodetostring(n, depth)
    static cache
    initial cache := table()
    /depth := 0
    if /cache[n] then
        if *n.children > 0 then {
            result := "[" || n.field.field.name || "@" || n.field.offset || ":"
            every result ||:= edgetostring(!n.children, depth+2)
            cache[n] := result || "]"
        } else {
            cache[n] := "(" || image(n.name)
            every a := !n.cs.arms do
              cache[n] ||:= ":" || image(a.original) || ":" || image(a.imp_soln)
            cache[n] ||:= ")"
        }
    return \cache[n]
end
```

Defines:
`nodetostring`, used in chunk 278.
Uses `edgetostring` 278a and `field` 53.

278a ⟨* 276a⟩+≡ ◁277b 278b▷
```
procedure edgetostring(e,depth)
    return left("\n", depth) ||
          "{" || patimage(sort(e.lo ++ e.hi)) || ":" || nodetostring(e.node,depth) || "}"
end
```
Defines:
edgetostring, used in chunk 277b.
Uses node 260a, nodetostring 277b, and patimage 358a.

Conversion to dag is the usual bottom-up hashing; here I compute the string and then use the string to index into a table. The real work of merging edges is done by `combinechildren`. If edge merging results in a single each, the node is replaced by its child, provided the edge really covers all possible values of the field.

278b ⟨* 276a⟩+≡ ◁278a 278c▷
```
procedure tree2dag(n, nodetable, depth)
    outtree(\ascii_tree, n)
    /nodetable := table()
    /depth := 0
    if *n.children > 0 then
        combinechildren(n, nodetable, depth+2)  # converts edges to set form
    if *n.children = 1 then {
        e := n.children[1]
        if covers(n.children[1], fwidth(n.field.field)) then
            n := n.children[1].node     # all roads to child: hoist it
        else
            warning("node with one child doesn't match all cases")
    }
    s := nodetostring(n, depth)
    outtree(\ascii_dag, n)
    /nodetable[s] := n
    return nodetable[s]
end
```
Defines:
tree2dag, used in chunks 20a and 279b.
Uses combinechildren 279b, covers 278c, field 53, fwidth 112d, hoist 207, node 260a, nodetostring 277b, and outtree 359b.

Here's where I check coverage. Only success or failure of `covers` is meaningful, not the value returned.

278c ⟨* 276a⟩+≡ ◁278b 279a▷
```
procedure covers(e, width)
    l := sort(e.lo ++ e.hi)
    return *l = 2 & l[1] = 0 & l[2] = 2^width
end
```
Defines:
covers, used in chunk 278b.

The complicated stuff here is identifying a name array. At each node, either all edges go in an exiting name array or a new name array is used. If not, I create a new one.

279a ⟨* 276a⟩+≡ ◁278c 279b▷
```
record namearray(field, tbl, hi, codename, storageclass)
# field used as index, table[integer] of name, bound on table, name of this array
global natable
```
Defines:
`namearray`, used in chunks 236, 280–82, 319c, 320a, 332a, and 343b.
`natable`, used in chunks 279b and 280a.

Uses `field` 53 and `used` 132a.

The fields `codename` and `storageclass` can be defaulted (to a gensym'ed name and to `static`, respectively).

279b ⟨* 276a⟩+≡ ◁279a 280b▷
```
procedure arraycandidates(n)
    initial MAXRANGE := 32
    suspend e := !n.children & type(e.node.name) == "string" &
            e.hi - e.lo <= MAXRANGE & e
end

procedure combinechildren(n, nodetable, depth)
    initial natable := table()

    if arraycandidates(n).node.name ~== arraycandidates(n).node.name then {
        ⟨change names of children from strings to namearrays when possible 280a⟩
    }

    lotable := table()
    hitable := table()
    every e := !n.children & child := tree2dag(e.node, nodetable, depth) do {
        /lotable[child] := set()
        /hitable[child] := set()
        addinterval(lotable[child], hitable[child], e.lo, e.hi)
    }
    n.children := []
    every child := key(lotable) do
        put(n.children, edge(child, lotable[child], hitable[child]))
    return
end
```
Defines:
`arraycandidates`, used in chunk 280a.
`combinechildren`, used in chunk 278b.

Uses `addinterval` 276a, `edge` 260a, `MAXRANGE` 286d, `natable` 279a, `node` 260a, and `tree2dag` 278b.

280a ⟨*change names of children from strings to namearrays when possible* 280a⟩≡ (279b)
```
mightuse := set()           # name arrays we might use must have right field
every na := !\natable[n.field] do
    insert(mightuse, na)
every e := arraycandidates(n) & na := !mightuse do
    if \na.tbl[e.lo to e.hi - 1] ~== e.node.name then  # slot used with wrong name
        delete(mightuse, na)
if *mightuse > 0 then
    willuse := ?mightuse
else {
    /natable[n.field] := set()
    insert(natable[n.field], willuse := namearray(n.field, table(), 0))
}
every e := arraycandidates(n) &
      e.lo - willuse.hi <= MAXRANGE do {
          every willuse.tbl[e.lo to e.hi - 1] := e.node.name;
          e.node.name := willuse
          willuse.hi <:= e.hi
      }
```
Uses `arraycandidates` 279b, `field` 53, `MAXRANGE` 286d, `namearray` 279a, `natable` 279a, `node` 260a, and `used` 132a.

280b ⟨* 276a⟩+≡ ◁279b
```
procedure namesused(n, result)
    /result := set()
    if type(n.name) == "namearray" then insert(result, n.name)
    every namesused((!n.children).node, result)
    return result
end
```
Defines:
`namesused`, used in chunk 283b.
Uses `namearray` 279a and `node` 260a.

Chapter 15

Code generation for matching

This module contains the language-independent part of code generation for a matching statement. The records beginning with G define the structure, which is to be instantiated in a language-dependent way.

281a ⟨* 281a⟩≡ 281b ▷

```
record Glines(x)                # list of lines to be written out
record Gresynch(line, file)     # C #line or m3 <* LINE ... *>
record Gblock(decls, stmts)     # block with local variables
record Gdecl(name, type, init)  # variable with name, type, optional initial value
record Gcall(name, args)        # call to named function
```

Defines:
Gblock, used in chunks 84c, 106d, 116a, 122b, 130a, 283–85, 288a, 317d, 318d, and 330e.
Gcall, used in chunks 295, 297a, 316e, and 329b.
Gdecl, used in chunks 84c, 106d, 116a, 122b, 130a, 283b, 285c, 288a, 318d, and 330e.
Glines, used in chunks 84c, 106d, 116a, 122b, 130a, 282d, 283b, 289a, 318c, and 330d.
Gresynch, used in chunks 84c, 106d, 116a, 122b, 130a, 282d, 283b, 289a, 318c, and 330d.
Uses file 345a, line 345a, and value 132a.

For Gdecl, we require only that a type or an initial value be specified, as in Modula-3. Emitters for C may have to infer a type when an initial value is given.

281b ⟨* 281a⟩+≡ ◁ 281a 282a ▷

```
record Gdeclnamearray(na)       # declares CONST ARRAY OF TEXT specified by
                                # namearray na.
```

Defines:
Gdeclnamearray, used in chunks 236, 283b, 319a, 331a, and 342c.
Uses namearray 279a.

282a ⟨* 281a⟩+≡ ◁281b 282d▷
```
record Gcase(x, arms)           # CASE x OF arms END;
record Gcasearm(tags, x)        # tags is sorted list of lo, hi s.t. lo <= x < hi

record Ginrange(lo, x, hi)      # predicate lo <= x < hi
record Gsetname(lhs, name)      # lhs := name (where name is string or namearray)

record Gnomatch()               # what happens when there's no match

record Tunsigned(width)         # unsigned value of width bits (word size if /width)
record Tsigned(width)           # signed value of width bits (word size if /width)
record Gasgn(lhs, x)            # set lhs := x
```
Defines:
`Gasgn`, used in chunks 84c, 107b, 116a, 122b, 130c, 205b, 288a, 320a, and 332a.
`Gcase`, used in chunks 84c, 107a, 116a, 122b, 130b, 284b, 319d, and 331c.
`Gcasearm`, used in chunks 84c, 107a, 116a, 122b, 130b, 286d, 319d, and 331c.
`Ginrange`, used in chunks 84c, 107a, 116a, 122b, 130b, 282c, 285a, 287a, 320a, and 332a.
`Gnomatch`, used in chunks 84c, 107b, 116a, 122b, 130c, 287b, 320b, and 332b.
`Gsetname`, used in chunks 84c, 107b, 116a, 122b, 130c, 288a, 319c, 320a, and 332a.
`Tsigned`, used in chunks 318e and 320b.
`Tunsigned`, used in chunks 84c, 107b, 116a, 122b, 130c, 285c, 318e, 320b, and 332b.
Uses `namearray` 279a and `value` 132a.

282b ⟨*argument descriptions* 282b⟩≡
```
expargs decls stmts init.
```

282c ⟨*rewrite rules* 282c⟩≡
```
Ginrange(lo, N, hi) -> if lo <= N < hi then 1 else 0
```
Uses `Ginrange` 282a.

282d ⟨* 281a⟩+≡ ◁282a 283a▷
```
global gen_file_header, gen_outer_decls
procedure genheader(header)
  return Glines([
          Gresynch(codeline := 1),
          literal(gen_file_header),
          Gresynch(header.line, header.file),
          Glines(header.code),
          Gresynch(codeline +:= 100)])
end
```
Defines:
`gen_file_header`, used in chunks 323a and 334.
`genheader`, used in chunk 19a.
`gen_outer_decls`, used in chunks 283b, 323a, and 334.
Uses `codeline` 283a, `file` 345a, `Glines` 281a, `Gresynch` 281a, `line` 345a, and `literal` 43e.

283a ⟨* 281a⟩+≡ ◁282d 283b▷

```
global fetchtab                # code to use to fetch words: size -> fetch string
global codeline
```

Defines:
`codeline`, used in chunks 282d, 283b, and 289a.
`fetchtab`, used in chunks 17a, 49b, 283b, 289, 290a, 323a, and 334.

283b ⟨* 281a⟩+≡ ◁283a 284a▷

```
procedure gencase(cs, root)
  static label
  local decls
  initial label := 0
  /root := tree(cs)
  outtree(\ascii_tree, root)
  decls := [Gdecl("MATCH_p", fetchtab["type"], cs.valcode)]
  every put(decls, !gen_outer_decls | Gdeclnamearray(!namesused(root)))
  return Glines([Gblock(decls, [Gresynch(codeline +:= 100), gennode(root, set())])] |||
                armscode(cs.trailer))
end
```

Defines:
`gencase`, used in chunk 20a.

Uses `armscode` 289a, `codeline` 283a, `fetchtab` 283a, `Gblock` 281a, `Gdecl` 281a, `Gdeclnamearray` 281b, `gennode` 284a, `gen_outer_decls` 282d, `Glines` 281a, `Gresynch` 281a, `label` 369a, `namesused` 280b, `outtree` 359b, and `tree` 262.

I special-case the case of two children where one has a single range; the generated code is cleaner if I just use one `if` statement. Actually, I could do this any time I have only two children... ßChange #28

284a ⟨* 281a⟩+≡ ◁283b 285c▷

```
procedure gennode(n, wordsmatched)
    local firstif, decls, added, single_range, other_child, answer
    while *n.children = 1 do
        n := n.children[1].node
    if *n.children > 0 then {
        decls := []
pushtrace("DECLS")
        added := add_decls(decls, wordsmatched, n.field)
poptrace()
        if *n.children = 2 & single_range := !n.children & *single_range.lo = 1 then {
          ⟨set answer to single if statement 285a⟩
        } else {
          ⟨split children into if and case tests, and make combination answer 284b⟩
        }
        every delete(wordsmatched, !added)
        return answer
    } else {
        return genarms(n.cs.arms, n, wordsmatched)
    }
end
```

Defines:
  `gennode`, used in chunks 283b and 285–87.
Uses `add_decls` 285c, `field` 53, `genarms` 287b, and `node` 260a.

284b ⟨*split children into* `if` *and case tests, and make combination* `answer` 284b⟩≡ (284a)

```
edges := table()
every e := !n.children do edges[sort(e.lo)[1]] := e
edges := sort(edges)
ifarms := []
every addarms(ifarms, n.field, (!edges)[2], wordsmatched)
casearms := []
every genedge(casearms, n.field, (!edges)[2], wordsmatched)
put(ifarms, Sguarded(1, Gcase(literal(afieldname(n.field)), casearms)))
answer := subst_for_pc(Gblock(decls, [Sif(ifarms)]), address_to_integer("MATCH_p"))
```

Uses `addarms` 287a, `address_to_integer` 290a, `afieldname` 286c, `field` 53, `Gblock` 281a, `Gcase` 282a, `genedge` 286d, `literal` 43e, `Sguarded` 79, `Sif` 79, and `subst_for_pc` 87c.

285a ⟨*set* answer *to single* if *statement* 285a⟩≡ (284a)

```
  other_child := single_range ~=== !n.children
  ⟨if other_child's range has one element, exchange it with single_range 285b⟩
  f := literal(afieldname(n.field))
  s := Sif([Sguarded(Ginrange(!single_range.lo, f, !single_range.hi),
                     gennode(single_range.node, wordsmatched)),
            Sguarded(1, gennode(other_child.node, wordsmatched))])
  answer := subst_for_pc(Gblock(decls, [s]), address_to_integer("MATCH_p"))
```

Uses address_to_integer 290a, afieldname 286c, field 53, Gblock 281a, gennode 284a, Ginrange 282a, literal 43e, node 260a, Sguarded 79, Sif 79, and subst_for_pc 87c.

This little goodie makes it a bit more likely to use just one equality test. It's not clear whether it's the right heuristic when both ranges contain more than one element.

285b ⟨*if* other_child*'s range has one element, exchange it with* single_range 285b⟩≡ (285a)

```
  if *other_child.lo = 1 & (!other_child.lo + 1 = !other_child.hi) then {
    *other_child.hi = 1 | impossible("ranges")
    other_child :=: single_range
  }
```

wordsmatched is slightly misnamed. It contains the *names* of all the words that have been fetched, and it also contains the absolute fields that have been fetched.

285c ⟨* 281a⟩+≡ ◁284a 286b▷

```
  procedure add_decls(decls, wordsmatched, f)
    local added, fields
    ⟨make fields a set from field, set, or list f 286a⟩
    added := set()
    every w := wordname(f := !fields) & not member(wordsmatched, w) do {
      put(decls, Gdecl(wordname(f), Tunsigned(f.field.class.size),
                       fetchcode(address_add("MATCH_p", f.offset), f.field.class.size)))
      every insert(wordsmatched | added, w)
    }
    every f := !fields & not member(wordsmatched, f) do {
      put(decls, Gdecl(afieldname(f), Tunsigned(fwidth(f.field)),
                         Eslice(literal(wordname(f)), f.field.lo, fwidth(f.field))))
      every insert(wordsmatched | added, f)
    }
    return added
  end
```

Defines:
add_decls, used in chunks 284a and 288a.

Uses address_add 289c, afieldname 286c, Eslice 79, fetchcode 289d, field 53, fwidth 112d, Gdecl 281a, literal 43e, Tunsigned 282a, and wordname 286b.

286a ⟨*make* `fields` *a set from field, set, or list* `f` 286a⟩≡ (285c)
```
fields := case type(f) of {
  "set"   : f
  "list"  : set(f)
  default : set([f])
}
```
We use a stylized `wordname` to refer to the word containing a particular field. It's uniquely determined by its size and offset.

286b ⟨* 281a⟩+≡ ◁285c 286c▷
```
procedure wordname(f)
  return "MATCH_w_" || f.field.class.size || "_" || f.offset;
end
```
Defines:
`wordname`, used in chunks 285c, 321a, and 332c.
Uses `field` 53.

A similar trick gives us a unique name for each absolute field.

286c ⟨* 281a⟩+≡ ◁286b 286d▷
```
procedure afieldname(f)
  return "MATCH_f_" || f.field.name || "_" || f.offset
end
```
Defines:
`afieldname`, used in chunks 284, 285, and 287a.
Uses `field` 53.

286d ⟨* 281a⟩+≡ ◁286c 287a▷
```
global MAXRANGE

procedure genedge(casearms, f, e, wordsmatched)
    local tags
    tags := []
    r := sort(e.lo ++ e.hi)
    while lo := get(r) & hi := get(r) do
        if hi - lo <= MAXRANGE then
            every put(tags, lo | hi)
    if *tags > 0 then
        put(casearms, Gcasearm(tags, gennode(e.node, wordsmatched)))
    return
end
```
Defines:
`genedge`, used in chunk 284b.
`MAXRANGE`, used in chunks 279b, 280a, and 287a.
Uses `Gcasearm` 282a, `gennode` 284a, and `node` 260a.

287a ⟨* 281a⟩+≡ ◁286d 287b▷

```
procedure addarms(ifarms, f, e, wordsmatched)
    r := sort(e.lo ++ e.hi)
    while lo := get(r) & hi := get(r) do
        if hi - lo > MAXRANGE then
            put(ifarms, Sguarded(Ginrange(lo, literal(afieldname(f)), hi),
                                 gennode(e.node, wordsmatched)))
    return
end
```

Defines:
addarms, used in chunk 284b.
Uses afieldname 286c, gennode 284a, Ginrange 282a, literal 43e, MAXRANGE 286d, node 260a, and Sguarded 79.

287b ⟨* 281a⟩+≡ ◁287a 288a▷

```
procedure genarms(arms, thenode, wordsmatched)
    local ifarms
    ifarms := []
    every a := !arms do {
      c := copy((\a.imp_soln).constraints) | set()
# introduce fields used in conditions????
      every insert_condition(c,
                subst_tab(!(\a.soln).constraints, (\a.imp_soln).answers, 1))
      put(ifarms, Sguarded(c, genarm(a, thenode, wordsmatched)))
    }
    put(ifarms, Sguarded(1, Gnomatch()))
    return Sif(ifarms)
end
```

Defines:
genarms, used in chunk 284a.
Uses constraints 132a, genarm 288a, Gnomatch 282a, insert_condition 123c, Sguarded 79, Sif 79, subst_tab 87a, and used 132a.

It might be surprising to see us declaring identifiers with the null type (using `Gdecl` below), but that's what we use when the exact type is unknown or unimportant. Emitters for languages like SML or Modula-3 can then ignore type entirely.

Something is going awry in the equation solver. Pattern labels are inputs to the solver, and therefore should not appear in `answers`; however, they do appear in `answers`, which means they were declared twice. Omitting the code to declare pattern labels seems to solve the problem, but why? Notice also that the omitted code could never have been right to begin with, since it bound the label to an *offset*, not to its true value.

288a ⟨* 281a⟩+≡ ◃287b 289a▹
```
  procedure genarm(a, thenode, wordsmatched)
      local bindings, fused, block, decls, stmts, patlabels
      /continue := 0
      ⟨insist there are no field bindings in arm a 288b⟩
      fused := set();
      every insert(fused, subterms_matching(!(\a.imp_soln).answers, "absolute_field"))
      decls := []
      added := add_decls(decls, wordsmatched, fused)
  #    patlabels := a.pattern.disjuncts[1].patlabelbindings
  #    every id := key(\patlabels) do
  #      put(decls, Gdecl(id, unsigned_type(), patlabels[id]))
      every id := key((\a.imp_soln).answers) & not is_wildcard(id) do
        put(decls, Gdecl(id, &null, a.imp_soln.answers[id]))
      every id := key((\a.soln).answers) & not is_wildcard(id)  do
        put(decls, Gdecl(id, &null, a.soln.answers[id]))
      if (\a.name & /thenode.name) then
        warning("Name `" || a.name || "' in pattern arm is unbound.\n")
      put(decls, Gsetname(\a.name, \thenode.name))
      every delete(wordsmatched, !added)
      stmts := armscode(a)
      push(stmts, Gasgn(\thenode.cs.succptr, address_add("MATCH_p", \a.patlen)))
      return subst_for_pc(Gblock(decls, stmts), address_to_integer("MATCH_p"))
  end
```
Defines:
  `genarm`, used in chunk 287b.
Uses `absolute_field` 256, `add_decls` 285c, `address_add` 289c, `address_to_integer` 290a, `arm` 259b, `armscode` 289a, `Gasgn` 282a, `Gblock` 281a, `Gdecl` 281a, `Gsetname` 282a, `is_wildcard` 289b, `pattern` 53, `subst_for_pc` 87c, `subterms_matching` 82a, `succptr` 346b, and `unsigned_type` 320c.

288b ⟨*insist there are no field bindings in arm* a 288b⟩≡ (288a)
```
  if type(c := !(!(\a.pattern).disjuncts).aconstraints) == "fieldbinding"
    then impossible("field binding in arm: ", expimage(c))
```
Uses `arm` 259b, `expimage` 98a, `field` 53, `fieldbinding` 53, and `pattern` 53.

289a ⟨* 281a⟩+≡ ◃288a 289b▹
```
procedure armscode(a)
  return [Gresynch(a.line, a.file), Glines(a.code), Gresynch(codeline +:= 100)]
end
```
Defines:
armscode, used in chunks 283b and 288a.
Uses codeline 283a, file 345a, Glines 281a, Gresynch 281a, and line 345a.

We don't create declarations for the _ wildcard. At the moment, its implementation is as a fresh variable, and since fresh variables should not otherwise leak out (aren't valid C names anyway), this should be enough: ßChange #30

289b ⟨* 281a⟩+≡ ◃289a 289c▹
```
procedure is_wildcard(v)
  return member(fresh_variables, v)
end
```
Defines:
is_wildcard, used in chunk 288a.
Uses fresh_variables 151b.

15.1 Handling fetch and address templates

fetchcode does the expansion of the fetch table.

289c ⟨* 281a⟩+≡ ◃289b 289d▹
```
procedure address_add(address, offset)
  if offset % pc_unit_bits ~= 0 then
    error("Tried to fetch at offset ", offset, ", but pc_unit_bits = ",
          pc_unit_bits, " doesn't divide ", offset)
  offset /:= pc_unit_bits;
  return interpret_fetchtab(\fetchtab["add"], address, offset, &null, "address add") |
    error("No template given for 'address add'")
end
```
Defines:
address_add, used in chunks 285c and 288a.
Uses address 369a, fetchtab 283a, interpret_fetchtab 290b, and pc_unit_bits 16a.

289d ⟨* 281a⟩+≡ ◃289c 290a▹
```
procedure fetchcode(address, width)
  return interpret_fetchtab(\fetchtab[width | "any"], address, &null, width, "fetch") |
    error("No template given to fetch ", width, "-bit word")
end
```
Defines:
fetchcode, used in chunk 285c.
Uses address 369a, fetchtab 283a, and interpret_fetchtab 290b.

290a ⟨* 281a⟩+≡ ◁289d 290b▷

```
procedure address_to_integer(address)
  return interpret_fetchtab(\fetchtab["integer"], address, &null, &null,
                            "address to integer") |
    error("No template given for 'address to integer'")
end
```

Defines:
`address_to_integer`, used in chunks 284b, 285a, and 288a.
Uses `address` 369a, `fetchtab` 283a, and `interpret_fetchtab` 290b.

290b ⟨* 281a⟩+≡ ◁290a

```
procedure interpret_fetchtab(s, address, offset, width, msg)
  r := "("
  s ? {
    while r ||:= tab(upto('%')) do {
      ="%"
      r ||:= case move(1) of {
               "a" : \address | error("%a illegal in template for ", msg)
               "o" : \offset  | error("%o illegal in template for ", msg)
               "w" : \width   | error("%w illegal in template for ", msg)
               "%" : "%"
               default : error("Bad escape in fetch string for ",
                               width, "-bit word: %", move(-1), "; try %a, %o, or %w")
             }
    }
    return r || tab(0) || ")"
  }
end
```

Defines:
`interpret_fetchtab`, used in chunks 289 and 290a.
Uses `address` 369a.

Chapter 16

Specification Checker

The toolkit can generate a program, which with some simple shell scripts, can be used to check the consistency of an architecture's toolkit specification against the machine/assembly language recognized by existing assemblers and disassemblers for that architecture. The `-checker`*file* option writes the C source for the checker program in *file*. Debugging information is written into *file* using the `-debug-checker`*file* option. See the reference manual for a complete description of the checker's functionality.

The toolkit-generated program "checks" a specification by applying pairs of constructors for each instruction. The first constructor emits the instruction's binary code. For convenience, a constructor's binary representation is emitted using assembly directives (e.g., `.byte, .word, .long`), which allows the checker to emit both representations in one assembly file. The second constructor in a pair emits the the assembly-language representation of the instruction. When executed, the checker program emits its output into an assembly file, which is then assembled by the native assembler. This technique assumes that the native assembler permits both instructions and data in the text segment; the SGI-Irix assembler is the only one that we have found that prohibits this mix.

The resulting object file is disassembled using a native disassembler or debugger. We use `gdb -batch` to disassemble the object file. `gdb` reads the file `gdb-{asm,bin}output`, which disassemble the instructions generated by the checker.

291a ⟨*gdb-binoutput* 291a⟩≡

```
disas main _asmoutput
```

Uses `main` 18.

291b ⟨*gdb-asmoutput* 291b⟩≡

```
disas _asmoutput _endoutput
```

If the specification's assembly-to-binary mappings are correct, the assembly output should contain pairs of matching instructions; a matched pair indicates that the assembler's binary encoding of an instruction matches the toolkit's encoding. A short script compares pairs of instructions and flags any mismatched pairs. Because the checker emits mismatched pairs in assembly, the user can easily identify the source of the discrepancy in the original specification.

292 ⟨*chkrfilter* 292⟩≡

```
#!/bin/sh
#
# Usage: chckfilter <gawk or nawk> <object file name>
#
#   Compare pairs of disassembled instructions
GNAWK=$1
shift
gdb -batch -x gdb-asmoutput $1 | \
  sed -e 's/^[^:]*://g' >$1.asmoutput
gdb -batch -x gdb-binoutput $1 | \
  sed -e 's/^[^:]*://g' >$1.binoutput
echo "Differences"
# Display diffs in 2 column format
diff $1.asmoutput $1.binoutput | expand | \
  $GNAWK '/^---$/ { next }
/^< / { left[nextleft++] = substr($0,3); next }
/^> / { right[nextright++] = substr($0,3); next }
function dump () {
  if (nextright==1 && right[0]=="# ERROR") {
        nextleft = 1; left[0] = ">>> ERROR <<<"
  }
  while (nextleft < nextright) left[nextleft++]=""
  while (nextright < nextleft) right[nextright++]=""
  for (i=0; i<nextleft; i++) {
      printf "%-38.38s | %-38.38s\n", left[i], right[i]
  }
  nextleft = nextright = 0
}

{ dump(); print }
END {dump()}
'
rm -f $1.asmoutput $1.binoutput
```

Uses `file` 345a and `next` 385c.

16.1 How to build a checker

The toolkit distribution includes the directory `base-checker`, which contains all the files necessary to build checkers for the MIPS, SPARC, and Pentium targets. It also contains a makefile that will build and run a checker. We describe the steps necessary to build and run a checker here.

A checker requires an assembly emitter (e.g., *target*`-asm.c`), a binary emitter (e.g., *target*`-bin.c`), target-specific code to emit bytes in assembly pseudo-ops, the checker program and the toolkit library. The checker program accesses the binary and assembly emitters indirectly via the interfaces *target*`_asm` and *target*`_bin`.

To check instructions that take relocatable operands, we use a trick to guarantee that the address assigned by the assembler matches that used by the binary emitter. The address in the file `chkraddr` is the entry point for `main`; we get that value before building the checker program, then use it to assign known addresses to relocatable operands.

293 ⟨*rules to build check address* 293⟩≡

```
chkraddr:
        echo "main(){}" >_boot.c
        $(CC) -g _boot.c -o _boot
        nm _boot | egrep "T _main|T main" | \
                   awk '{print "0x"$$1}' | cpif $@
        rm -f _boot.c _boot
```

Uses `main` 18.

A specification is checked in several steps. All of these steps appear in the `base-checker`'s makefile. First, the checker program emits assembly code in the file *_target.s*. The assembly file is assembled into the binary file *_target.out*. Then the `chkrfilter` shell script disassembles the binary file and compares the two representations. Any differences are printed to standard output.

16.2 Building a checker

`emit_checker` produces the checker. For each constructor, `generate_input_values` produces a table that associates operands' names with sets of test inputs that satisfy the constructor's constraints. For each instruction constructor, `generate_checker` enumerates all the test values of the operands associated with that constructor, and it emits assembly and binary forms by applying the constructor to each combination. If an operand is of a constructor type (i.e., not a field or integer) the checker repeats the enumeration process recursively on constructors of that type. This recursion explores the entire state space defined by the constructors and the sets of operand values.

`generate_checker` emits a C program that applies the pairs of constructors. The assembly constructors are accessed indirectly through the interface *name*`_asm` and the binary constructors via *name*`_bin`. An interface name must be specified using the `-indirect` command-line option.

294 ⟨* 294⟩≡ 295▷

```
global chkr_input_tbl
global TESTLIMIT
global zerotab
procedure emit_checker(outfilename)
  local outfile, altoutfile, pp

  TESTLIMIT := 1024
  zerotab := table(0)
  zerotab[1] := 0
  verbose("Emitting checker program")
  if (/chkr_input_tbl) then chkr_input_tbl := table()
  if (/indirectname) then
    error("Interface name undefined. Checker requires ‘‘-indirect’’ option")
  outfile := openfile(outfilename || implementation_extension, "cw") |
    error("Could not open ", outfilename, " for writing")
  pp := PPnew(outfile)
  emit_template(pp, "checker.t", "interface",
                image(outfilename || interface_extension), "irname", indirectname)
  every cons := kept_constructors() do {
    write(\cdebug, "\nChecker for " || expimage(cons.name))
    p := pattern_to_case(subst_tab(crhs(cons), parmtab(cons), 1))
    chkr_input_tbl[cons.name] := generate_input_values(p, cons)
  }
  generate_checker(pp, outfilename, indirectname)
  PPxwrites(pp, "$b}$n")
  PPwrite(pp)   # flush prettyprinter
  return
end
```

Defines:
`chkr_input_tbl`, used in chunks 295 and 297a.

emit_checker, used in chunk 18.
TESTLIMIT, used in chunk 303.
zerotab, used in chunk 301a.
Uses cdebug 16a, crhs 164, expimage 98a, generate_checker 295, generate_input_values 299, implementation_extension 16a, indirectname 16a, interface_extension 16a, kept_constructors 31d, openfile 22, parmtab 183a, pattern_to_case 178b, subst_tab 87a, and verbose 16a.

generate_checker emits the ASCII-encoded binary instructions in the text section main and then the assembly instructions in _asmoutput . UNIX-style assembler pseudo-ops are interleaved between the two sections so that the disassembler can find the entry points of each section.

295 ⟨* 294⟩+≡ ◃294 296▹

```
procedure generate_checker(pp, outfilename, indirectname)
  local cons, test, asmoutput, binoutput, outfile, app
  count := 0
  asmoutput := []
  binoutput := ""
  asmir := indirectname || "_asm" || "->"
  binir := indirectname || "_bin" || "->"

  outfile := openfile(outfilename || "_tmp_", "cw") |
    error("Could not open ", outfilename, " for writing")
  app := PPnew(outfile)
  every cons := kept_constructors() do
    if cons.type === instructionctype then
      every test := generate_tests(cons, 1, !chkr_input_tbl[cons.name], asmir) do {
        pexp := pretty(Gcall(cons.name, test))
        asmoutput := "/* Pair " || string(count +:= 1) || ":" || pexp || " */\n"
        asmoutput ||:= asmir || pexp || ";\n"
        PPxwrite(app, asmoutput)
        binoutput := "/* Pair " || string(count) || ":" || pexp || " */\n"
        binoutput ||:= binir || pexp || ";\n"
        PPxwrite(pp, binoutput)
      }
  emit_template(pp, "inchecker.t")
  PPwrite(app)  # flush assembly prettyprinter
  outfile := openfile(outfilename || "_tmp_", "r") |
    error("Could not open ", outfilename, " for reading")
  every line :=  !outfile do
    PPxwrite(pp, line)
  emit_template(pp, "afterchecker.t")
  remove(outfilename || "_tmp_")
end
```

Defines:
generate_checker, used in chunk 294.
Uses chkr_input_tbl 294, Gcall 281a, generate_tests 296, indirectname 16a, instructionctype 30a, kept_constructors 31d, line 345a, openfile 22, and pretty 16a.

generate_tests generates the test inputs for arguments idx through the last argument to constructor cons. It suspends a list of test values for the operands from idx to *cons.operands.

296 ⟨* 294⟩+≡ ◁295 297a▷

```
procedure generate_tests(cons, idx, test_values, asmir)
  local i, rest
  write(\cdebug, "Generate tests for ", cons.name, " ",
                 idx, " ", expimage(cons.operands[idx]))
  ⟨print test_values 300e⟩
  if idx > *cons.operands then
    suspend []
  else if type(cons.operands[idx]) == "literal" then
    suspend generate_tests(cons, idx+1, test_values, asmir)
  else
    every rest := generate_tests(cons, idx+1, test_values, asmir) &
          i := tests_of(cons.operands[idx], test_values, asmir)
    do {
      push(rest, i)
      suspend rest
      pop(rest)
    }
end
```

Defines:
generate_tests, used in chunks 295 and 297a.
Uses cdebug 16a, expimage 98a, literal 43e, and tests_of 297a.

297a ⟨* 294⟩+≡ ◁296 298▷

```
procedure tests_of(input, tbl, asmir)
  local i, m, n
  if type(input) == "literal" then suspend ""
  else if type(input) == "input" then {
    case type(input.meaning) of {
      "integer" : {
        input := lookuptype(input.name, "field")
        ⟨check that test values exist for input 297b⟩
        suspend tbl[input.name]
      }
      "string" : {
        ⟨check that test values exist for input 297b⟩
        suspend if (input.meaning == "reloc" & integer(tbl[input.name])) then
            Gcall("unsigned_to_raddr", [tbl[input.name]])
        else tbl[input.name]
      }
      "field" | "null" | "string" : {
        ⟨check that test values exist for input 297b⟩
        suspend tbl[input.name]
      }
      "constype" :
         every m := kept_constructors(input.meaning) &
               i := generate_tests(m, 1, !chkr_input_tbl[m.name], asmir)
         do
           suspend Gcall(asmir || m.name, i)
      default : typeerror(input.meaning, "field, input, or constype", input.name)
    }
  }
end
```

Defines:
`tests_of`, used in chunk 296.

Uses `chkr_input_tbl` 294, `constype` 155, `field` 53, `Gcall` 281a, `generate_tests` 296, `input` 44a, `kept_constructors` 31d, `literal` 43e, `lookuptype` 23a, `typeerror` 26b, and `unsigned_to_raddr`.

297b ⟨*check that test values exist for input* 297b⟩≡ (297a)

```
inputs := \tbl[input.name] | \
error("No test values specified for `", input.name, "'")
```

Uses `input` 44a.

merge_chkr_tables merges two input tables and complains if test values have already been supplied for an input.

298 ⟨* 294⟩+≡ ◁297a 299▷

```
procedure merge_chkr_tables(L)
  local t, k
  chkr_tbl := table()

  every t := !L do {
    every k := key(t) do {
      if /chkr_tbl[k] then chkr_tbl[k] := t[k]
      else warning("Checker inputs for ", k, " already defined. Using first set.")
    }
  }
  return chkr_tbl
end
```

Uses `defined` 132a.

16.3 Generating Inputs for Checker

To test each constructor, we generate one or more sets of input values. For each of the constructor's branches (i.e., sets of equations), the goal is to generate one input set that satisfies only that branch; this guarantees that each branch of each constructor is tested at least once. If the constructor is unconditional, only one set of input values is generated; if it is conditional, one set is generated for each branch. In the general case, determining that a set of conditions is satisfiable is undecideable, because *why?. what's the reduction?* For example, a branch's equations may include mutually unsatisfiable constraints, e.g. `rd = 1, rd = 2`, in which case, no set of input values exists that satisfies these equations.

Instead, we use heuristics to generate the test values for a particular branch. First, we find equality conditions that constrain an input to a constant value, e.g., the condition `rd = 1` determines `rd`'s test value. That initial set is used to simplify the branch's conditions, which may determine other inputs' values. Test values are added until no new inputs are constrained. If some inputs remain unconstrained, they are assigned a random test value within the input's width constraints.

299 ⟨* 294⟩+≡ ◁298 300c▷

```
procedure generate_input_values(p, cons)
  local d, conds, fits, test_values, test_sets, f, tmp, names
  test_sets := []
  case type(p) of {
    "Stagcase"   : {
       every c := kept_constructors(p.type) do
         test_sets |||:= generate_input_values(p.arms[c], cons)
    }
    "pattern" : {
      remove_duplicate_conditions(p, cons)
      while d := get(p.disjuncts) do {
        test_values := table()
        test_values[Epc] := Epc
        ⟨extract d's conditions 300a⟩
        ⟨create initial set of test_values 304c⟩
        ⟨generate input values 303⟩
        put(test_sets, test_values)
      }
    }
  }
  return test_sets
end
```

Defines:
`generate_input_values`, used in chunk 294.
Uses `Epc` 79, `kept_constructors` 31d, `pattern` 53, `remove_duplicate_conditions` 189a, and `Stagcase` 178a.

This code is derived from the code in `constructors.nw` that extracts a disjunct's conditions before emitting its encoding code. To generate input values, we only need the field-width constraints in `fits` and the general conditions in `conds`.

300a ⟨*extract* `d`*'s conditions* 300a⟩≡ (299)
```
set_patlabel_offsets(d)
d := gsubst(d, Epatlabel_to_Epc)
ds_conds := extract_conditions(d, cons)
conds := ds_conds[2]
fits :=  ds_conds[4]
sanitize_sequents(d, fits)
⟨eliminate Eforce on relocatable inputs 300b⟩
⟨add constraints on unconstrained inputs 301a⟩
⟨add constraints on field inputs 301b⟩
⟨omit constraints and conditions on constructor-typed operands 301c⟩
⟨print fits and conds 300d⟩
```
Uses `Epatlabel_to_Epc` 185a, `extract_conditions` 190b, `gsubst` 117a, `sanitize_sequents` 186a, and `set_patlabel_offsets` 57c.

Relocatable addresses are always "forceable", because the checker determines their values.

300b ⟨*eliminate* `Eforce` *on relocatable inputs* 300b⟩≡ (300a)
```
every f := !fits do f.x := unforce(f.x)
```
Uses `unforce` 300c.

300c ⟨* 294⟩+≡ ◁299 304a▷
```
procedure unforce(x)
  return if (type(x) == "Eforce") then x.x else x
end
```
Defines:
`unforce`, used in chunks 300b and 304a.
Uses `Eforce` 79.

300d ⟨*print* `fits` *and* `conds` 300d⟩≡ (300a)
```
if (*fits > 0) then write(\cdebug, "Width constraints for " || cons.name)
write(\cdebug, "   " || expimage(fits))
write(\cdebug, print_conditions(conds))
```
Uses `cdebug` 16a, `constraints` 132a, `expimage` 98a, and `print_conditions` 305c.

300e ⟨*print* `test_values` 300e⟩≡ (296)
```
every k := key(test_values) do
  write(\cdebug, "test_values[", expimage(k), "] = ", expimage(test_values[k]))
```
Uses `cdebug` 16a and `expimage` 98a.

301a ⟨*add constraints on unconstrained inputs* 301a⟩≡ (300a)
```
every i := inputs_of(cons) & type(i.meaning) == "null" do
{
  write(\cdebug, "adding Efitsu(" || i.name || ", "|| wordsize || "), " || type(i.meaning))
  constrained := &null
  every f := !fits & string(f.x) == i.name do
    constrained := 1
  /constrained & insert(fits, Efitsu(i.name, wordsize))
#  if (string(i.meaning) == "reloc") then {
#    if (type(conds) == "integer") then conds := set()
#    insert(conds, eqn(Emod(i.name, 4), "=", zerotab))
#  }
}
```
Uses cdebug 16a, Efitsu 79, Emod 79, eqn 81c, inputs_of 162a, wordsize 36c, and zerotab 294.

We insert constraints on field operands into `fits` so we can generate test values for them.

301b ⟨*add constraints on field inputs* 301b⟩≡ (300a)
```
every i := inputs_of(cons, "field") do
  insert(fits, Efitsu(i.name, fwidth(i.meaning)))
every i := inputs_of(cons, "integer") do
  insert(fits, Efitss(i.name, i.meaning))
```
Uses Efitss 79, Efitsu 79, field 53, fwidth 112d, and inputs_of 162a.

A constructor's pattern may include constraints and conditions on its integer, field, and relocatable operands, and on the inputs to its constructor-typed operands. We can omit the constraints on constructor-typed operands from `fits` and `conds`, because these constraints are satisfied when test values are generated for instances of the operands' constructor types.

301c ⟨*omit constraints and conditions on constructor-typed operands* 301c⟩≡ (300a)
```
tmp := copy(fits)
every f := !\fits & (type(f.x) == "Einstance_input") do
  delete(tmp, f)
fits := tmp
tmp := conds
every c := !conds & subterms_matching(c, "Einstance_input") do
  delete(tmp, c)
conds := tmp
```
Uses Einstance_input 79 and subterms_matching 82a.

302a ⟨*insert equality conditions in* `test_values` 302a⟩≡ (304c)
```
every c := !conds  do {
  if ((equality condition 302b)) then {
    verbose("inserting test_values[" || expimage(x) || "] = " || v)
    test_values[x] := v
  }
}
```
Uses `expimage` 98a and `verbose` 16a.

302b ⟨*equality condition* 302b⟩≡ (302a)
```
type(c) == "eqn" & c.op == "=" &
 ((v := integer(untableexp(c.right)) & (not integer(c.left)) & x := c.left) |
  (v := integer(untableexp(c.left))  & (not integer(c.right)) & x := c.right))
```
Uses `eqn` 81c and `untableexp` 86a.

The initial set of test values includes equality conditions in `conds`, e.g. `rd = 1`, and equality conditions derived from this initial set, e.g., `rd = 1, rs = rd => rs = 1`.

We try to generate random test values for any operand without an initial value until all conditions dependent on the operand are satisfied. For most operands, we choose a random value that falls within its field-width constraints. If a field operand, however, is sparsely populated, i.e., a set of meaningful values are specified by the user, we choose a random value from this set.

302c ⟨*sparsely populated* 302c⟩≡ (303)
```
names := \fieldname_table(i.meaning)
```
Uses `fieldname_table` 342a.

303 ⟨*generate input values* 303⟩≡ (299)
```
every (i := inputs_of(cons) &
       type(i.meaning) ~== "constype" &
       /test_values[i.name]) do {
  every f := !fits & string(f.x) == i.name do {
    unsatisfied := 1
    count := 0
    if (type(i.meaning) == "field" & ⟨sparsely populated 302c⟩) then
      test_values[f.x] := ?names
    else
      test_values[f.x] := random_nbit_value(f)
    while (\unsatisfied & count < TESTLIMIT) do {
      verbose("Try " || expimage(f.x) || " = " || expimage(test_values[f.x]))
      ⟨check conditions 305b⟩
      if (\unsatisfied) then {
        slices := slices_of(f.x, unsatisfied)
        verbose("Slices are: ")
        every verbose(expimage(!slices))
        test_values[f.x] := interleave_slices(random_nbit_value(f), slices)
      }
      count +:= 1
    }
    if (count == TESTLIMIT) then {
      warning("Checker can't generate test values for " || i.name ||
              print_conditions(unsatisfied, "not satisfied."))
    }
  }
}
if (*test_values > 1) then
  write(\cdebug, "Tests " ||
    (if (*p.disjuncts > 1) then " for disjunct:\n" || expimage(d)
    else ":"))
every i := inputs_of(cons) & \test_values[i.name] do
  write(\cdebug, expimage(i.name) || " = " || expimage(test_values[i.name]))

⟨check conditions 305b⟩
if (\unsatisfied) then {
  warning("Checker can't generate test values for disjunct:",
          "\n", expimage(d),  print_conditions(unsatisfied, "not satisfied."))
}
if (\unknown) then {
  warning("Test values for disjunct:",
          "\n", expimage(d),
          print_conditions(unknown, "may not be satisfied by test inputs."))
}
```
Uses `cdebug` 16a, `constype` 155, `disjunct` 53, `expimage` 98a, `field` 53, `inputs_of` 162a,

`interleave_slices` 304b, `print_conditions` 305c, `random_nbit_value` 306a, `slices_of` 304a, `TESTLIMIT` 294, and `verbose` 16a.

`slices_of` projects conditions dependent on slices of `v`.

304a ⟨* 294⟩+≡ ◁300c 304b▷
```
procedure slices_of(v, conds)
  local s
  s := set()
  every c := !conds do
    if (type(c.left) == "Eslice" & (c.left.x := unforce(c.left.x)) == v) then
      insert(s,eqn(c.left, c.op, integer(untableexp(c.right))))
    else if (type(c.right) == "Eslice" & (c.right.x := unforce(c.right.x)) == v) then
      insert(s, eqn(c.right, c.op, integer(untableexp(c.left))))
  return s
end
```
Defines:
  `slices_of`, used in chunk 303.
Uses `eqn` 81c, `Eslice` 79, `unforce` 300c, and `untableexp` 86a.

Now try to interleave the slices, generating random bits for the unspecified parts. No checking for overlapping slices! The solver should do that.

304b ⟨* 294⟩+≡ ◁304a 305c▷
```
procedure interleave_slices(result, slices)
  local e, mask
  every e := !slices & e.op == "=" do {
    mask := icom(ishift(2^(e.left.n)-1, e.left.lo))
    result := ior(iand(result, mask), ishift(iand(e.right, 2^(e.left.n)-1), e.left.lo))
  }
  return result
end
```
Defines:
  `interleave_slices`, used in chunk 303.
Uses `mask` 338a.

304c ⟨*create initial set of* `test_values` 304c⟩≡ (299)
```
every i := inputs_of(cons, "string") &
      i.meaning == "reloc" & test_values[i.name] := "reloc"
n_tests := 0
⟨insert equality conditions in test_values 302a⟩
while (*test_values > n_tests) do {
  n_tests := *test_values
  ⟨simplify conditions 305a⟩
  ⟨insert equality conditions in test_values 302a⟩
}
```
Uses `inputs_of` 162a.

305a ⟨*simplify conditions* 305a⟩≡ (304c)
```
new_conds := set()
every insert(new_conds, (simplify_conditions(conds, test_values))[2])
conds := new_conds
```

Uses simplify_conditions 306b.

Simplifying conditions can produce satisfied conditions, unsatisfied conditions, or conditions with unknown satisfiability.

305b ⟨*check conditions* 305b⟩≡ (303)
```
unsatisfied := &null
unknown := &null
every l := simplify_conditions(conds, test_values) do {
  c := l[1]
  newc := l[2]
  if (type(newc) == "eqn") then {
    if (reduceable(newc)) then
      /unknown := [c] | push(unknown, c)
    else if (not tautology(newc)) then
      /unsatisfied := [c] | push(unsatisfied, c)
  }
}
```
Uses eqn 81c, reduceable 307, simplify_conditions 306b, and tautology 307.

305c ⟨* 294⟩+≡ ◁304b 306a▷
```
procedure print_conditions(conds, msg)
  local l
  l := []
  (*conds == 0) &  return ""
  every put(l, expimage(!conds))
  return  "Conditions " || (\msg | " ") || ":"  || "    " || commaseparate(l)

end
```
Defines:
print_conditions, used in chunks 300d and 303.
Uses expimage 98a.

random_nbit_value produces a random value in the range $0...2^n - 1$ for unsigned, n-bit values and values in the range $-2^(n-1)...2^(n-1) - 1$ for signed, n-bit values.

306a ⟨* 294⟩+≡ ◁305c 306b▷

```
procedure random_nbit_value(f)
  local v
  return case type(f) of {
    "Efitsu": ?(2^(f.n)-1)
    "Efitss": { v := ?(2^(f.n-1))
                if (?2 = 1) then  -v
                else if (v = 2^(f.n-1)) then v - 1
                else v
              }
    default : error("Invalid field-width constraint: ", expimage(f))
  }
end
```

Defines:
random_nbit_value, used in chunk 303.
Uses constraint 53, Efitss 79, Efitsu 79, expimage 98a, and field 53.

simplify_conditions simplifies each condition in a set of conditions and then suspends a list pair which contains the original and new condition.

306b ⟨* 294⟩+≡ ◁306a 307▷

```
procedure simplify_conditions(conditions, values)
  local c, cp
  every c := !\conditions do {
#    write("==> " || image(type(c)) || "," || expimage(c))
    cp := simplify(subst_tab(c, values))
#    write("<== " || image(type(cp)) || "," || expimage(cp))
    suspend [c, cp]
  }
end
```

Defines:
simplify_conditions, used in chunk 305.
Uses expimage 98a, simplify 89b, and subst_tab 87a.

An equation is reduceable if it contains a variable expression.

307 ⟨* 294⟩+≡ ◁306b

```
  procedure reduceable(c)
    return not integer(untableexp(c.left)) | not integer(untableexp(c.right))
  end
  procedure tautology(c)
    local l,r
    if (l := integer(untableexp(c.left)) &
        r := integer(untableexp(c.right))) then
      return case c.op of {
        "="  : l = r
        ">=" : l >= r
        "<=" : l <= r
        "!=" : l ~= r
        "<"  : l > r
        ">"  : l < r
      default: error("Invalid operator " || c.op)
      }
    return
  end
```

Defines:
  reduceable, used in chunk 305b.
  tautology, used in chunk 305b.
Uses untableexp 86a.

308 ⟨*checker.t* 308⟩≡

```
#include <mclib.h>
#include %interface
#include <stdio.h>
#include <stdarg.h>
extern struct %{irname}_asm *%{irname}_asm;
extern struct %{irname}_asm *%{irname}_bin;
static void checkerfail(char *, ...);
static void reloc_print(RAddr r);

static void reloc_print(RAddr r) {$t
assert(location_known(r));
asmprintf(asmprintfd, "%%s", r->label->name);
/* asmprintf(asmprintfd, "0x%%08x", location(r)); */$b
}
static void checkerfail(char *fmt, ...) {$t
va_list ap;

va_start(ap, fmt);
fprintf(stderr, "Error in checker: ");
vfprintf(stderr, fmt, ap);
fprintf(stderr, "\n");
va_end(ap);
exit(1);
$b}$n
void  *mc_alloc(int size, Mc_alloc_pool pool) {$t
char *p = (char *)malloc(size);
return (void *)p;
$b}$n
RClosure mc_alloc_closure(size_in_bytes, dest_block, dest_lc)
unsigned size_in_bytes; RBlock dest_block; unsigned dest_lc; {$t
RClosure cl;
checkerfail("attempt to call mc_alloc_closure!");
return cl;
$b}$n
main(){$t
RBlock rb;
RAddr  reloc;
fail = &checkerfail;
asmprintreloc = &reloc_print;
rb = block_new(0);
block_label(rb)->name = "reloc";
set_block(rb);
set_address(rb, (unsigned)CHKRADDR);
reloc = addr_new(block_label(rb), 0x0);
```

```
  printf(".text\n");
  printf(".globl reloc\n");
  printf(".globl main\n");
  printf("reloc:\n");
  printf("main:\n");
```

Uses addr_new, asmprintf 394d, asmprintfd 394d, asmprintreloc 394d, block_label 385a, block_new 384d, fail 396, label 369a, location, location_known, main 18, mc_alloc_closure, Mc_alloc_pool, RAddr 397a, RBlock, RClosure, set_address 385g, and set_block 371b.

309a ⟨*inchecker.t* 309a⟩≡

```
  printf(".text\n");
  printf(".globl _asmoutput\n");
  printf("_asmoutput:\n");
```

309b ⟨*afterchecker.t* 309b⟩≡

```
  printf(".text\n");
  printf(".globl _endoutput\n");
  printf("_endoutput:\n");
  exit(0);
```

Chapter 17

Converting Expressions to C

`prettyC` returns C code for an expression (or statement) with embedded prettyprinting escapes. `nohex` is a purely internal flag. If it is non-null, constants are in hex; otherwise they're in decimal.

311 ⟨* 311⟩≡ 315f▷

```
procedure prettyC(e, precedence, associativity, nohex)
  ⟨local declarations for prettyC 313b⟩
  initial {⟨initialize Cprec and Cassoc 321c⟩}
  /precedence := 0
  /associativity := "L"
  return case type(e) of {
    ⟨cases for prettyC 312a⟩
    default    : impossible("Bad code to prettyC")
  }
end
```

Defines:
`prettyC`, used in chunks 17a, 312–21, and 323a.

312a ⟨*cases for* prettyC 312a⟩≡ (311) 313c ▷

```
"list"     : commaseparate(maplist2(prettyC, e, Cprec[","]), ", $o")
"set"      : commaseparate(maplist2(prettyC, sort(e), Cprec["&&"]), " && $o")
"eqn"      : { o := Cop(e.op)
               Cbracket(prettyC(e.left,  Cprec[o]) || " " || o || " $o" ||
                        prettyC(e.right, Cprec[o]), "==", precedence)
             }
"table"    : {                       # standard normal form
    ⟨make s represent table e, but elide additive and multiplicative units 312b⟩
    "$t${" || s || "$}$b"
  }
"string"   : e
"literal"  : list_to_string(e.s)
"integer"  : if e > max_decimal then "0x" || hexstring(e) else string(e)
```

Uses `Cbracket` 321b, `Cop` 322a, `Cprec` 321b, `eqn` 81c, `hexstring` 322c, `literal` 43e, `max_decimal` 14k, and `prettyC` 311.

ge #23

The old version printed in any old order, but the new version prints positive terms first, then negative terms, with the constant term last in its group.

312b ⟨*make* s *represent table* e*, but elide additive and multiplicative units* 312b⟩≡ (312a)

```
{
  s := ""; leadingsign := ""
  ⟨add positive terms to s 312c⟩
  leadingsign := " - $o"
  ⟨add negative terms to s 312d⟩
  s := if s == "" then "0" else Cbracket(s, "+", precedence)
}
```

Uses `Cbracket` 321b.

312c ⟨*add positive terms to* s 312c⟩≡ (312b)

```
every e[k := 1 ~=== key(e)] > 0 do {
    s ||:= leadingsign
    s ||:= (1 ~= abs(e[k])) || "*"    # print coefficient if not 1
    s ||:= prettyC(k, Cprec["*"])
    leadingsign := " + $o"
}
if e[1] > 0 then s ||:= leadingsign || string(e[1])
```

Uses `Cprec` 321b and `prettyC` 311.

312d ⟨*add negative terms to* s 312d⟩≡ (312b)

```
every e[k := 1 ~=== key(e)] < 0 do {
    s ||:= leadingsign
    s ||:= (1 ~= abs(e[k])) || "*"    # print coefficient if not 1
    s ||:= prettyC(k, Cprec["*"])
}
if e[1] < 0 then s ||:= leadingsign || string(-e[1])
```

Uses `Cprec` 321b and `prettyC` 311.

313a ⟨*old make* s *represent table* e*, but elide additive and multiplicative units* 313a⟩≡
```
{
  # print constant term if nonzero
  if e[1] ~= 0 then { s := string(e[1]) ; leadingsign := " + $o" }
  else s := leadingsign := ""
  # print every nonconstant term k
  every k := 1 ~=== key(e) do {
      s ||:= if e[k] < 0 then " - " else leadingsign
      leadingsign := " + $o"
      s ||:= (1 ~= abs(e[k])) || "*"    # print coefficient if not 1
      s ||:= prettyC(k, Cprec["*"])
  }
  s := if s == "" then "0" else Cbracket(s, "+", precedence)
}
```
Uses Cbracket 321b, constant 111b, Cprec 321b, and prettyC 311.

313b ⟨*local declarations for* prettyC 313b⟩≡ (311) 315c▷
```
local leadingsign
```

313c ⟨*cases for* prettyC 312a⟩+≡ (311) ◁312a 313d▷
```
"Eorb"     : { s := prettyC(e.x, Cprec["|"], "L")
               t := prettyC(e.y, Cprec["|"], "R")
               Cbracket(s || " | $o" || t, "|", precedence)
             }
"Eand"     : { s := prettyC(e.x, Cprec["&&"], "L")
               t := prettyC(e.y, Cprec["&&"], "R")
               Cbracket(s || " && $o" || t, "&&", precedence)
             }
```
Uses Cbracket 321b, Cprec 321b, Eand 79, Eorb 79, and prettyC 311.

313d ⟨*cases for* prettyC 312a⟩+≡ (311) ◁313c 313e▷
```
"Eslice"   : { s := prettyC(super_simplify(Eshift(e.x, -e.lo)), Cprec["&"], "L")
               Cbracket(s || " & " || mask(e.n), "&", precedence)
             }
"Eshift"   : { s := prettyC(e.x, Cprec["<<"])
               s ||:= if e.n < 0 then (" >> " || -e.n)
                      else            (" << " ||  e.n)
               Cbracket(s, "<<", precedence)
             }
```
Uses Cbracket 321b, Cprec 321b, Eshift 79, Eslice 79, mask 338a, prettyC 311, and super_simplify 90a.

313e ⟨*cases for* prettyC 312a⟩+≡ (311) ◁313d 314a▷
```
"Enarrowu" : prettyC(e.x, precedence, associativity) # checked, so must be identity
"Enarrows" : prettyC(super_simplify(Eslice(e.x, 0, e.n)), precedence, associativity)
"Ewiden"   : "sign_extend($t${$o" || prettyC(e.x) || ", $o" || e.n || "$}$b)"
```
Uses Enarrows 79, Enarrowu 79, Eslice 79, Ewiden 79, prettyC 311, and super_simplify 90a.

314a ⟨*cases for* prettyC 312a⟩+≡ (311) ◁313e 314b▷
```
  "Ediv"     : { s := prettyC(e.x, Cprec["/"])
                 s := "((int)" || s || ")"   # temporary hack to fix a bug
                 s ||:= " / " || e.n
                 Cbracket(s, "/", precedence)
               }
  "Emod"     : { s := prettyC(e.x, Cprec["%"])
                 s ||:= " % " || e.n
                 Cbracket(s, "%", precedence)
               }
```
Uses Cbracket 321b, Cprec 321b, Ediv 79, Emod 79, and prettyC 311.

314b ⟨*cases for* prettyC 312a⟩+≡ (311) ◁314a 315d▷
```
  "Semit"    : {
    s := if *e.x > 1 then "$t${{ $c" else ""
    ⟨if direct style and late pc, add ENCODE_pc declaration to s and set pc_override 315a⟩
    s ||:= commaseparate(maplist(prettyC, e.x), " $c")
    ⟨restore pc_override 315b⟩
    s || if *e.x > 1 then " $b$c}$}" else ""
  }
  "Stoken" :
    case emitterstyle of {
      "direct" : {⟨direct call through emittername 314c⟩}
      "closure": {⟨closure call to emitter 314d⟩}
    } | impossible("emitter style")
```
Uses emitterstyle 17b, prettyC 311, Semit 79, and Stoken 79.

314c ⟨*direct call through* emittername 314c⟩≡ (314b)
```
  s := emittername || "$t(" || prettyC(e.x, Cprec[","]) || ", $o" || e.n || ")"
  Cbracket("$c" || s, "app", precedence) || ";$b"
```
Uses Cbracket 321b, Cprec 321b, emittername 16a, and prettyC 311.

For the closure call, we assume that the closure is stored in local variable _c.

314d ⟨*closure call to* emitter 314d⟩≡ (314b)
```
  o := if e.offset > 0 then " + " || e.offset else ""
  s := "emitter$t(_c->loc.dest_block, $o_c->loc.dest_lc" || o || ", $o" ||
       prettyC(e.x, Cprec[","]) || ", $o" || e.n || ")"
  Cbracket("$c" || s, "app", precedence) || ";$b"
```
Uses Cbracket 321b, Cprec 321b, and prettyC 311.

This code fixes bug 5 by using `ENCODE_pc` instead of `cur_pc` when an emitted token other than the first requires the program counter. It's needed because `cur_pc()` changes as a side effect of `emit`, but "the program counter" always refers to the location of the *first* token emitted. ge #8

315a ⟨*if direct style and late pc, add* `ENCODE_pc` *declaration to* `s` *and set* `pc_override` 315a⟩≡ (314b)

```
if *e.x > 1 & emitterstyle == "direct" & expwalk(e.x[2:0], equals_pc) then {
  s ||:= "unsigned ENCODE_pc = cur_pc(); $c"
  save_pc_override := pc_override
  pc_override := "ENCODE_pc"
}
```

Uses `cur_pc` 389c, `emitterstyle` 17b, `equals_pc` 76a, `expwalk` 114a, and `pc_override` 315f.

315b ⟨*restore* `pc_override` 315b⟩≡ (314b)

```
pc_override := save_pc_override
```

Uses `pc_override` 315f.

315c ⟨*local declarations for* `prettyC` 313b⟩+≡ (311) ◁313b 319b▷

```
local save_pc_override
```

315d ⟨*cases for* `prettyC` 312a⟩+≡ (311) ◁314b 315e▷

```
"Epc" : \pc_override | "cur_pc()"
"Epc_known" : "cur_pc_known()"
```

Uses `cur_pc` 389c, `cur_pc_known` 389c, `Epc` 79, `Epc_known` 79, and `pc_override` 315f.

315e ⟨*cases for* `prettyC` 312a⟩+≡ (311) ◁315d 315g▷

```
"Eclosure_loc"  : "_c->loc"
"Eclosure_addr" : "_c->v.a" || e.n
"Eclosure_val"  : "_c->v.u" || e.n
```

Uses `Eclosure_addr` 79, `Eclosure_loc` 79, and `Eclosure_val` 79.

315f ⟨* 311⟩+≡ ◁311 317c▷

```
global pc_override
```

Defines:
`pc_override`, used in chunk 315.

315g ⟨*cases for* `prettyC` 312a⟩+≡ (311) ◁315e 316a▷

```
"Efitsu"   : { s := "(unsigned)(" || prettyC(e.x) || ") < " || tworaised(e.n)
               Cbracket(s, "<", precedence)
             }
"Efitss"   : { s := "(unsigned)(" || prettyC(addconst(e.x, 2^(e.n-1))) ||
                                         ") < " || tworaised(e.n)
               Cbracket(s, "<", precedence)
             }
```

Uses `addconst` 111a, `Cbracket` 321b, `Efitss` 79, `Efitsu` 79, `prettyC` 311, and `tworaised` 338a.

316a ⟨*cases for* prettyC 312a⟩+≡ (311) ◁315g 316b▷
```
  "Eforce" | "Eforceable"  : {
                        (if type(e.x) == "Eclosure_loc" then "pc_" else "") ||
                        "location" ||
                        (if type(e) == "Eforceable" then "_known" else "") ||
                        "$t${(" || prettyC(e.x) || ")$}$b"
                }
```
Uses Eclosure_loc 79, Eforce 79, Eforceable 79, location, and prettyC 311.

316b ⟨*cases for* prettyC 312a⟩+≡ (311) ◁316a 316c▷
```
  "Enot"      : { s := "!" || prettyC(e.x, Cprec["!"])
                    Cbracket(s, "!", precedence)
                 }
```
Uses Cbracket 321b, Cprec 321b, Enot 79, and prettyC 311.

316c ⟨*cases for* prettyC 312a⟩+≡ (311) ◁316b 316d▷
```
  "Enosimp"   : prettyC(e.x, precedence, associativity)
```
Uses Enosimp 79 and prettyC 311.

316d ⟨*cases for* prettyC 312a⟩+≡ (311) ◁316c 316e▷
```
  "Sstmts"    : "$t${{ $c" || commaseparate(maplist2(prettyC, e.x), " $c") || " $b$c}$}"
```
Uses prettyC 311 and Sstmts 79.

316e ⟨*cases for* prettyC 312a⟩+≡ (311) ◁316d 316f▷
```
  "Gcall" : { l := []; every put(l, prettyC(!e.args))
              Cnoreserve(e.name) || "(" || commaseparate(l) || ")"
          }
```
Uses Cnoreserve 322b, Gcall 281a, and prettyC 311.

316f ⟨*cases for* prettyC 312a⟩+≡ (311) ◁316e 317a▷
```
  "Einstance_input" : {
                  s := prettyC(e.x, Cprec["."]) || ".u." || e.cons.name || "." || e.name
                  Cbracket(s, ".", precedence)
                }
  "Einstance_tagged" : {
                  s := prettyC(e.x, Cprec["="]) || ".tag == " || e.cons.name || "_TAG"
                  Cbracket(s, "==", precedence)
                }
```
Uses Cbracket 321b, Cprec 321b, Einstance_input 79, Einstance_tagged 79, and prettyC 311.

I assume the if statement has been simplified, so there are no arms beyond any `else` arm

317a ⟨*cases for* `prettyC` 312a⟩+≡ (311) ◃316f 317b▹
```
"Sif"      : { preif := ""
               s := ""
               every a := !e.arms do {
                 s ||:= preif || Ctest(a.guard) || Carm(a.x)
                 preif := "$celse "
               }
               "${" || s || "$}"
             }
```
Uses `Carm` 317d, `Ctest` 317c, and `Sif` 79.

317b ⟨*cases for* `prettyC` 312a⟩+≡ (311) ◃317a 317e▹
```
"Efail" : error("Generate-time operation failed: ", e.msg)
```
Uses `Efail` 79.

317c ⟨* 311⟩+≡ ◃315f 317d▹
```
procedure Ctest(guard)
  return if guard_always_satisfied(guard) then ""
         else "if $t${(" || prettyC(guard) || ")$}$b "
end
```
Defines:
`Ctest`, used in chunk 317a.
Uses `guard_always_satisfied` 93c and `prettyC` 311.

317d ⟨* 311⟩+≡ ◃317c 318e▹
```
procedure Carm(e)
  return if type(e) == ("Gblock"|"Sstmts") then prettyC(e) || " "
         else "$t$c${" || prettyC(e) || "$}$b "
end
```
Defines:
`Carm`, used in chunk 317a.
Uses `Gblock` 281a, `prettyC` 311, and `Sstmts` 79.

317e ⟨*cases for* `prettyC` 312a⟩+≡ (311) ◃317b 318a▹
```
"Stagcase" : { s := "switch (" || prettyC(e.x, Cprec["."]) || ".tag) {$t${"
              every c := kept_constructors(e.type) do
                s ||:= "$ccase " || c.name || "_TAG: $t$c${" ||
                            prettyC(e.arms[c]) || " $cbreak;$}$b"
              s || "$cdefault: assert(0);$}$b$c} /* " || prettyC(e.x) || "*/"
            }
```
Uses `Cprec` 321b, `kept_constructors` 31d, `prettyC` 311, and `Stagcase` 178a.

318a ⟨*cases for* prettyC 312a⟩+≡ (311) ◁317e 318b▷
```
  "Sfail"    : { s := "fail($t${" || image(escape_dollars(e.fmt))
                 every a := e.a1 | e.a2 | e.a3 do
                   s ||:= ", $o" || prettyC(\a, Cprec[","])
                 s ||:= "$}$b);"
                 Cbracket(s, "app", precedence)
               }
  "Sepsilon" : "{/* skip */}"
```
Uses Cbracket 321b, Cprec 321b, escape_dollars 109b, fail 396, prettyC 311, Sepsilon 79, and Sfail 182c.

318b ⟨*cases for* prettyC 312a⟩+≡ (311) ◁318a 318c▷
```
  "Sclosure" : prettyC(\e.creation) | impossible("creating closure")
```
Uses prettyC 311 and Sclosure 185f.

318c ⟨*cases for* prettyC 312a⟩+≡ (311) ◁318b 318d▷
```
  "Glines"    : { s := ""; every s ||:= prettyC(!e.x) || "\n"; s }
  "Gresynch"  : "$n#line " || e.line  || " " || image(\e.file | "generated-code")
```
Uses file 345a, Glines 281a, Gresynch 281a, line 345a, and prettyC 311.

318d ⟨*cases for* prettyC 312a⟩+≡ (311) ◁318c 319a▷
```
  "Gblock"    : { s := "{ ${$t$c"
                  every s ||:= prettyC(!e.decls) || ";$n"
                  s ||:= commaseparate(maplist(prettyC, e.stmts), "$c")
                  s || "$b $c}$}"
                }
  "Gdecl"     : { s := "${" || prettyC(\e.type | infer_C_type(e.init))
                  s ||:= " " || e.name
                  s ||:= " = $t$c" || prettyC(\e.init) || "$b"
                  s || "$}"
                }
```
Uses Gblock 281a, Gdecl 281a, infer_C_type 318e, and prettyC 311.

318e ⟨* 311⟩+≡ ◁317d 320c▷
```
  procedure infer_C_type(value)
    if /value then impossible("variable with no type and no value")
    return case type(value) of {
      "Ewiden" : Tsigned(value.n)
      default  : Tunsigned(wordsize)
    }
  end
```
Defines:
  infer_C_type, used in chunk 318d.

Uses Ewiden 79, Tsigned 282a, Tunsigned 282a, value 132a, and wordsize 36c.

319a ⟨*cases for* `prettyC` 312a⟩+≡ (311) ◁318d 319c▷
```
  "Gdeclnamearray" : {
    /na_count := 0
    /e.na.codename := "MATCH_name_" || e.na.field.field.name || "_" || (na_count +:= 1)
    /e.na.storageclass := "static"
    s := e.na.storageclass || " char *" || e.na.codename || "[] = {"
    s ||:= "${$t"
    every i := 0 to e.na.hi - 1 do
      s ||:= " $c" || image(\e.na.tbl[i]|0) || ","
    s || "$b $c$}}"
  }
```
Uses `field` 53 and `Gdeclnamearray` 281b.

319b ⟨*local declarations for* `prettyC` 313b⟩+≡ (311) ◁315c
```
  static na_count
```

319c ⟨*cases for* `prettyC` 312a⟩+≡ (311) ◁319a 319d▷
```
  "Gsetname"  : "char *" || e.lhs || " = ${$t$o" || case type(e.name) of {
                        "string"    : image(e.name)
                        "namearray" : e.name.codename || "[" || prettyC(e.name.field) || "]"
                        default     : impossible("type of node name field")
                  } || "$b$}"
```
Uses `field` 53, `Gsetname` 282a, `namearray` 279a, `node` 260a, and `prettyC` 311.

319d ⟨*cases for* `prettyC` 312a⟩+≡ (311) ◁319c 320a▷
```
  "Gcase"     : { s := "switch (" || prettyC(e.x) || ") {$t${ "
                   every s ||:= prettyC(!e.arms)
                   s || "$cdefault: assert(0);$}$b$c} /* " || prettyC(e.x) || "*/"
                 }
  "Gcasearm"  : { s := "$n"
                   every i := 1 to *e.tags by 2 do {
                     if e.tags[i] + 1 = e.tags[i+1] then
                       s ||:= "$ocase " || e.tags[i] || ": "
                     else
                       every s ||:= "$ocase " || (e.tags[i] to (e.tags[i+1]-1)) || ": "
                   }
                   s ||:= "$t$c${" || prettyC(e.x) || "$cbreak;$}$b"
                 }
```
Uses `Gcase` 282a, `Gcasearm` 282a, and `prettyC` 311.

320a ⟨*cases for* prettyC 312a⟩+≡ (311) ◁319d 320b▷
```
  "Ginrange"  : { x := prettyC(e.x, Cprec["<"])
                  if e.lo + 1 = e.hi then {
                    Cbracket(x || " == " || e.lo, "==", Cprec["&&"])
                  } else {
                    c1 := Cbracket(e.lo || " <= " || x, "<=", Cprec["&&"])
                    c2 := Cbracket(x || " < " || e.hi, "<", Cprec["&&"])
                    Cbracket(c1 || " && " || c2, "&&", precedence)
                  }
                }
  "Gsetname"  : "${" || e.lhs || " = $t$o" || case type(e.name) of {
                     "string"    : image(e.name)
                     "namearray" : e.name.codename || "[" || e.name.field.name || "]"
                     default     : impossible("type of node name field")
                } || ";$b$}"
  "Gasgn"  : "${" || prettyC(e.lhs) || " = $t$o" || prettyC(e.x, Cprec["="]) || ";$b$}"
```
Uses `Cbracket` 321b, `Cprec` 321b, `field` 53, `Gasgn` 282a, `Ginrange` 282a, `Gsetname` 282a, `namearray` 279a, `node` 260a, and `prettyC` 311.

320b ⟨*cases for* prettyC 312a⟩+≡ (311) ◁320a 321a▷
```
  "Gnomatch"  : "assert(0); /* no match */$n"
  "Tunsigned" : unsigned_type(e.width)
  "Tsigned"   : signed_type(e.width)
```
Uses `Gnomatch` 282a, `signed_type` 320c, `Tsigned` 282a, `Tunsigned` 282a, and `unsigned_type` 320c.

320c ⟨* 311⟩+≡ ◁318e 321b▷
```
  procedure unsigned_type(width)
    return "unsigned" || unsigned_width_comment(width)
  end
  procedure signed_type(width)
    return "int" || signed_width_comment(width)
  end
  procedure unsigned_width_comment(width)
    return if \width < wordsize then " /* [0.." || (2^width-1) || "] */" else ""
  end
  procedure signed_width_comment(width)
    return if \width < wordsize then
      " /* [" || -(2^(width-1)) || ".." || (2^(width-1)-1) || "] */"
    else
      ""
  end
```
Defines:
`signed_type`, used in chunks 235 and 320b.
`unsigned_type`, used in chunks 195a, 235, 288a, and 320b.
Uses `wordsize` 36c.

321a ⟨*cases for* `prettyC` 312a⟩+≡ (311) ◁320b

```
"absolute_field" :
   prettyC(super_simplify(Eslice(literal(wordname(e)), e.field.lo, fwidth(e.field))),
           precedence, associativity, nohex)
```

Uses `absolute_field` 256, `Eslice` 79, `field` 53, `fwidth` 112d, `literal` 43e, `prettyC` 311, `super_simplify` 90a, and `wordname` 286b.

321b ⟨* 311⟩+≡ ◁320c 322a▷

```
global Cprec, Cassoc
procedure Cbracket(s, op, p, a)
  /a := "L"
  return "${" || (if Cprec[op] > p | (Cprec[op] = p & Cassoc[p] == a) then s
                  else "(" || s || ")") || "$}"
end
```

Defines:
`Cassoc`, used in chunk 321c.
`Cbracket`, used in chunks 312–16, 318a, and 320a.
`Cprec`, used in chunks 312–14, 316–18, 320a, and 321c.

321c ⟨*initialize* `Cprec` *and* `Cassoc` 321c⟩≡ (311)

```
ops := ["N", ["low"],
        "L", [";"],
        "L", [","],
        "R", ["=", "+=", "-=", "*=", "/=", "%=", "&=", "^=", "|=", ">>=", "<<="],
        "R", [":?"],
        "L", ["||"],
        "L", ["&&"],
        "L", ["|"],
        "L", ["^"],
        "L", ["&"],
        "L", ["==", "!="],
        "L", ["<=", "<", ">=", ">"],
        "L", [">>", "<<"],
        "L", ["+", "-"],
        "L", ["%", "/", "*"],
        "R", ["!", "~", "++", "--", "cast", "sizeof"],
        "L", [".", "->", "app"],
        "N", ["high"]
       ]
Cprec := table([])  # missed lookups break arithmetic comparisons
Cassoc := table()
every i := 1 to *ops by 2 do {
  every Cprec[!ops[i+1]] := i
  Cassoc[i] := ops[i]
}
```

Uses `Cassoc` 321b and `Cprec` 321b.

322a ⟨* 311⟩+≡ ◁321b 322b▷
```
procedure Cop(op)
  return if op == "=" then "==" else op
end
```
Defines:
`Cop`, used in chunk 312a.

322b ⟨* 311⟩+≡ ◁322a 322c▷
```
procedure Cnoreserve(word)
  static reserved
  initial reserved := set([
        "asm", "auto", "break", "case", "char", "const", "continue",
        "default", "double", "do", "else", "enum", "extern", "float", "for",
        "goto", "if", "int", "long", "register", "return", "short",
        "signed", "sizeof", "static", "struct", "switch", "typedef", "union",
        "unsigned", "void", "volatile", "while"])
  return if member(reserved, word) then Cnoreserve(word||"_") else word
end
```
Defines:
`Cnoreserve`, used in chunks 203, 204a, 219a, 234, 242, and 316e.

322c ⟨* 311⟩+≡ ◁322b 323a▷
```
procedure hexstring(i,n)
   local s
   if i = 0 then s := "0"
   else {
      s := ""
      while i ~= 0 do {
         s := "0123456789abcdef"[iand(i,15) + 1] || s
         i := ishift(i,-4)
         }
      }
   if \n > *s then s := right(s,n,"0")
   return s
end
```
Defines:
`hexstring`, used in chunk 312a.

17.1 Support for decoding code in C

323a ⟨* 311⟩+≡ ◁322c 323b▷
```
procedure Generate_C()
  pretty := prettyC
  gen_outer_decls := []
  gen_file_header := ["#include <assert.h>\n",
     "#define sign_extend(N,SIZE) ", # no newline after #define
         "(((int)((N) << (sizeof(unsigned)*8-(SIZE)))) >> (sizeof(unsigned)*8-(SIZE)))"
     ]
  fetchtab["type"] := "unsigned char *";
  fetchtab["add"] := "((%a) + %o)"
  fetchtab["integer"] := "(unsigned)(%a)"
#  fetchtab[8]  := "(unsigned)*(unsigned char *) (%a+%o/8)"
#  fetchtab[16] := "(unsigned)*(unsigned short *)(%a+%o/16)"
#  fetchtab[32] := "(unsigned)*(unsigned *)      (%a+%o/32)"
end
```
Defines:
Generate_C, used in chunks 10i and 18.
Uses fetchtab 283a, gen_file_header 282d, gen_outer_decls 282d, pretty 16a, and prettyC 311.

Used to declare functions according to the proper style of indirection.

323b ⟨* 311⟩+≡ ◁323a
```
procedure c_function_declaration(pp, returntype, name, args, indirect)
  emit_template(pp, if \indirectname then "indirect-proto.t" else "direct-proto.t",
                  "return", returntype, "name", name, "args", args)
  return
end
```
Defines:
c_function_declaration, used in chunks 203a, 204a, and 222a.
Uses indirectname 16a.

323c ⟨*direct-proto.t* 323c⟩≡
```
extern %return %name(%args);
```

323d ⟨*indirect-proto.t* 323d⟩≡
```
%return (*%name)(%args);
```

Chapter 18

Converting Expressions to Modula-3

`prettyM3` returns Modula-3 code for an expression (or statement) with embedded prettyprinting escapes. `nohex` is a purely internal flag. If it is non-null, constants are in hex; otherwise they're in decimal.

325a ⟨* 325a⟩≡ 329e ▷

```
procedure prettyM3(e, precedence, associativity, nohex)
  ⟨local declarations for prettyM3 327b⟩
  initial {⟨initialize M3prec and M3assoc 333a⟩}
  /precedence := 0
  /associativity := "L"
  return case type(e) of {
    ⟨cases for prettyM3 325b⟩
    default    : impossible("Bad code to prettyM3")
  }
end
```

Defines:
`prettyM3`, used in chunks 326–32 and 334.

325b ⟨*cases for* `prettyM3` 325b⟩≡ (325a) 326a ▷

326a ⟨*cases for* `prettyM3` 325b⟩+≡ (325a) ◁325b 327c▷

```
  "list"     : commaseparate(maplist2(prettyM3, e, M3prec[","]), ", $o")
  "set"      : commaseparate(maplist2(prettyM3, sort(e), M3prec["AND"]), " AND $o")
  "eqn"      : { o := M3op(e.op)
                 M3bracket(prettyM3(e.left,  M3prec[o]) || " " || o || " $o" ||
                           prettyM3(e.right, M3prec[o]), "=", precedence)
               }
  "table"    : {                       # standard normal form
      ⟨make s represent table e, but elide additive and multiplicative units 326b⟩
      "$t${" || s || "$}$b"
    }
  "string"   : if e == "$pc" then "$$pc" else e
  "literal"  : list_to_string(e.s)
  "integer"  : string(e)
```

Uses `eqn` 81c, `literal` 43e, `M3bracket` 332d, `M3op` 333b, `M3prec` 332d, and `prettyM3` 325a.

The old version printed in any old order, but the new version prints positive terms first, then negative terms, with the constant term last in its group. ge #23

326b ⟨*make* `s` *represent table* `e`*, but elide additive and multiplicative units* 326b⟩≡ (326a)

```
  {
    s := ""; leadingsign := ""
    ⟨add positive terms to s 326c⟩
    leadingsign := " - $o"
    ⟨add negative terms to s 326d⟩
    s := if s == "" then "0" else M3bracket(s, "+", precedence)
  }
```

Uses `M3bracket` 332d.

326c ⟨*add positive terms to* `s` 326c⟩≡ (326b)

```
  every e[k := 1 ~=== key(e)] > 0 do {
      s ||:= leadingsign
      s ||:= (1 ~= abs(e[k])) || "*"    # print coefficient if not 1
      s ||:= prettyM3(k, M3prec["*"])
      leadingsign := " + $o"
  }
  if e[1] > 0 then s ||:= leadingsign || string(e[1])
```

Uses `M3prec` 332d and `prettyM3` 325a.

326d ⟨*add negative terms to* `s` 326d⟩≡ (326b)

```
  every e[k := 1 ~=== key(e)] < 0 do {
      s ||:= leadingsign
      s ||:= (1 ~= abs(e[k])) || "*"    # print coefficient if not 1
      s ||:= prettyM3(k, M3prec["*"])
  }
  if e[1] < 0 then s ||:= leadingsign || string(-e[1])
```

Uses `M3prec` 332d and `prettyM3` 325a.

327a ⟨*old make* s *represent table* e*, but elide additive and multiplicative units* 327a⟩≡

```
{
  # print constant term if nonzero
  if e[1] ~= 0 then { s := string(e[1]) ; leadingsign := " + $o" }
  else s := leadingsign := ""
  # print every nonconstant term k
  every k := 1 ~=== key(e) do {
      s ||:= if e[k] < 0 then " - " else leadingsign
      leadingsign := " + $o"
      s ||:= (1 ~= abs(e[k])) || "*"    # print coefficient if not 1
      s ||:= prettyM3(k, M3prec["*"])
  }
  s := if s == "" then "0" else M3bracket(s, "+", precedence)
}
```

Uses constant 111b, M3bracket 332d, M3prec 332d, and prettyM3 325a.

327b ⟨*local declarations for* prettyM3 327b⟩≡ (325a) 331b ▹

```
local leadingsign
```

327c ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◃326a 327d ▹

```
"Eorb"     : "Word.Or(${$t$c" || prettyM3(e.x) || ", $c" || prettyM3(e.y) || "$b$})"
"Eand"     : { s := prettyM3(e.x, M3prec["AND"], "L")
               t := prettyM3(e.y, M3prec["AND"], "R")
               M3bracket(s || " AND $o" || t, "AND", precedence)
             }
```

Uses Eand 79, Eorb 79, M3bracket 332d, M3prec 332d, and prettyM3 325a.

327d ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◃327c 327e ▹

```
"Eslice"   : "Word.Extract(${$t$c" || prettyM3(e.x) || ", $c" || e.lo || ", " ||
                                                                     e.n || "$b$})"
"Eshift"   : "Word.Shift(${$t$c"   || prettyM3(e.x) || ", $c" || e.n || "$b$})"
```

Uses Eshift 79, Eslice 79, and prettyM3 325a.

327e ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◃327d 328a ▹

```
"Enarrowu" : prettyM3(e.x, precedence, associativity) # checked, so must be identity
"Enarrows" : prettyM3(super_simplify(Eslice(e.x, 0, e.n)), precedence, associativity)
"Ewiden"   : "Word.Insert(${$t$c" || prettyM3(e.x) ||
                         ", $cMATCH_bits[" || prettyM3(Eslice(e.x, e.n-1, 1)) ||
                         "], $c" || e.n ||
                         ", $cWord.Size - " || e.n || "$b$})"
```

Uses Enarrows 79, Enarrowu 79, Eslice 79, Ewiden 79, prettyM3 325a, and super_simplify 90a.

328a ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁327e 328b▷
```
  "Ediv"     : { s := prettyM3(e.x, M3prec["DIV"])
                 s ||:= " DIV " || e.n
                 M3bracket(s, "DIV", precedence)
               }
  "Emod"     : { s := prettyM3(e.x, M3prec["MOD"])
                 s ||:= " MOD " || e.n
                 M3bracket(s, "MOD", precedence)
               }
```
Uses Ediv 79, Emod 79, M3bracket 332d, M3prec 332d, and prettyM3 325a.

328b ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁328a 328c▷
```
  "Semit"    : { s := emittername || "$t(" || prettyM3(e.x, M3prec[","]) || ", $o" ||
                      e.n || ");$b"
                 M3bracket("$c" || s, "app", precedence)
               }
```
Uses emittername 16a, M3bracket 332d, M3prec 332d, prettyM3 325a, and Semit 79.

328c ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁328b 328d▷
```
  "Efitsu"   : "Word.LT(${$t$c" || prettyM3(e.x) || ", $c" || (2^e.n) || "$b$})"
  "Efitss"   : "Word.LT(${$t$c" || prettyM3(addconst(e.x, 2^(e.n-1))) || ", $c" ||
                                                                   (2^e.n) || "$b$})"
```
Uses addconst 111a, Efitss 79, Efitsu 79, and prettyM3 325a.

328d ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁328c 328e▷
```
  "Eforce" | "Eforceable"  : {
                 z := if type(e) == "Eforceable" then "_known" else ""
                 case s := prettyM3(e.x) of {
                   "$$pc"     : "RBlocks.cur_pc" || z || "()"
                   "_c->loc" : "RBlocks.pc_location" || z || "(_c.loc)"
                   default    : "RBlocks.location" || z || "$t${(" || s || ")$}$b"
                 }
               }
```
Uses cur_pc 389c, Eforce 79, Eforceable 79, location, pc_location 389c, and prettyM3 325a.

328e ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁328d 328f▷
```
  "Enot"      : { s := "NOT " || prettyM3(e.x, M3prec["NOT"])
                  M3bracket(s, "NOT", precedence)
                }
```
Uses Enot 79, M3bracket 332d, M3prec 332d, and prettyM3 325a.

328f ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁328e 329a▷
```
  "Enosimp"   : prettyM3(e.x, precedence, associativity)
```
Uses Enosimp 79 and prettyM3 325a.

329a ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁328f 329b▷
```
  "Sstmts"    : "${" || commaseparate(maplist2(prettyM3, e.x), " $c") || "$}"
```
Uses prettyM3 325a and Sstmts 79.

329b ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁329a 329c▷
```
  "Gcall" : { l := []; every put(l, prettyM3(!e.args))
              M3noreserve(e.name) || "(" || commaseparate(l) || ")"
           }
  "Einstance_input" : {
                 s := "NARROW(" || prettyM3(e.x) ||
                      ", Encode." || e.cons.name || "Instance)." || e.name
                 M3bracket(s, ".", precedence)
                }
  "Einstance_tagged" : {
                 s := "ISTYPE(" || prettyM3(e.x) ||
                      ", Encode." || e.cons.name || "Instance)"
                 M3bracket(s, "app", precedence)
                }
```
Uses Einstance_input 79, Einstance_tagged 79, Gcall 281a, Instance 391 391, M3bracket 332d, M3noreserve 333c, and prettyM3 325a.

I assume the if statement has been simplified, so there are no arms beyond any else arm

329c ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁329b 329d▷
```
  "Sif"      : { ifword := "IF "
                 s := ""
                 every a := !e.arms do {
                   s ||:= ifword || M3test(a.guard) || prettyM3(a.x)
                   ifword := "$cELSIF "
                 }
                 "${" || s || " $cEND;$}"
                }
```
Uses M3test 329e, prettyM3 325a, and Sif 79.

329d ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁329c 330a▷
```
  "Efail" : error("Generate-time operation failed: ", e.msg)
```
Uses Efail 79.

329e ⟨* 325a⟩+≡ ◁325a 332d▷
```
  procedure M3test(guard)
    return if guard_always_satisfied(guard) then "TRUE THEN $o"
           else "$t${" || prettyM3(guard) || "$}$b THEN $o"
  end
```
Defines:
M3test, used in chunk 329c.
Uses guard_always_satisfied 93c and prettyM3 325a.

330a ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁329d 330b▷
```
"Stagcase" : { s := "TYPECASE " || prettyM3(e.x) || " OF$t${"
               every c := kept_constructors(e.type) do
                 s ||:= "$n| Encode." || c.name || "Instance => $o$t${" ||
                           prettyM3(e.arms[c]) || "$}$b"
               s || "$nELSE <* ASSERT FALSE *>$}$b$nEND; (* " || prettyM3(e.x) || "*)"
             }
```
Uses Instance 391 391, kept_constructors 31d, prettyM3 325a, and Stagcase 178a.

330b ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁330a 330c▷
```
"Sfail"    : { s := "fail($t${" || image(escape_dollars(e.fmt))
               every a := e.a1 | e.a2 | e.a3 do
                 s ||:= ", $o" || prettyM3(\a, M3prec[","])
               s ||:= "$}$b);"
               M3bracket(s, "app", precedence)
             }
"Sepsilon" : "(* skip *)"
```
Uses escape_dollars 109b, fail 396, M3bracket 332d, M3prec 332d, prettyM3 325a, Sepsilon 79, and Sfail 182c.

330c ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁330b 330d▷
```
"Sclosure" : prettyM3(\e.creation) | impossible("creating closure")
```
Uses prettyM3 325a and Sclosure 185f.

330d ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁330c 330e▷
```
"Glines"    : { s := ""; every s ||:= prettyM3(!e.x) || "\n"; s }
"Gresynch"  : "<* LINE " || e.line  || " " || image(\e.file | "generated-code") || "*>"
```
Uses file 345a, Glines 281a, Gresynch 281a, line 345a, and prettyM3 325a.

330e ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁330d 331a▷
```
"Gblock"    : { s := "${"
                every s ||:= prettyM3(!e.decls) || ";$n"
                if *e.decls > 0 then
                  s ||:= "BEGIN ${$t$c"
                s ||:= commaseparate(maplist(prettyM3, e.stmts), "$c")
                if *e.decls > 0 then
                  s ||:= "$b$cEND$}"
                s || "$}"
              }
"Gdecl"     : { s := "VAR ${" || e.name
                s ||:= " : " || prettyM3(\e.type)
                s ||:= " := $t$c" || prettyM3(\e.init) || "$b"
                s || "$}"
              }
```
Uses Gblock 281a, Gdecl 281a, and prettyM3 325a.

331a ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁330e 331c▷

```
  "Gdeclnamearray" : {
    /na_count := 0
    /e.na.codename := "MATCH_name_" || e.na.field.field.name || "_" || (na_count +:= 1)
    s := "CONST ${" || e.na.codename || " = ARRAY [0.." || (e.na.hi - 1) || "] OF TEXT {"
    s ||:= "${$t"
    every i := 0 to e.na.hi - 1 do {
      s ||:= " $c" || (image(\e.na.tbl[i])|"NIL")
      if i < e.na.hi - 1 then s ||:= ","
    }
    s || "$b$c$}}"
  }
```

Uses field 53 and Gdeclnamearray 281b.

331b ⟨*local declarations for* prettyM3 327b⟩+≡ (325a) ◁327b

```
  static na_count
```

331c ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁331a 332a▷

```
  "Gcase"     : { s := "CASE " || prettyM3(e.x) || " OF "
                  every s ||:= prettyM3(!e.arms)
                  "${" || s || "$nEND; (* CASE " || prettyM3(e.x) ||
                    " *)$}"
                 }
  "Gcasearm"  : { s := "$n| $t"
                  every i := 1 to *e.tags by 2 do {
                    if i > 1 then s ||:= ", $o"
                    s ||:= if e.tags[i] + 1 = e.tags[i+1] then e.tags[i]
                           else e.tags[i] || ".." || (e.tags[i+1]-1)
                  }
                  s ||:= " => "  || "$t$c" || prettyM3(e.x)
                  "${" || s || "$b$b$}"
                }
```

Uses Gcase 282a, Gcasearm 282a, and prettyM3 325a.

332a ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁331c 332b▷

```
"Ginrange"  : { x := prettyM3(e.x, M3prec["<"])
                 c1 := M3bracket(e.lo || " <= " || x, "<=", M3prec["AND"])
                 c2 := M3bracket(x || " < " || e.hi, "<", M3prec["AND"])
                 M3bracket(c1 || " AND " || c2, "AND", precedence)
               }
"Gsetname"  : "${VAR " || e.lhs || " := $t$o" || case type(e.name) of {
                    "string"    : image(e.name)
                    "namearray" : e.name.codename || "[" || prettyM3(e.name.field) || "]"
                    default     : impossible("type of node name field")
               } || "$b$}"
"Gasgn"  : "${" || e.lhs || " := $t$o" || prettyM3(e.x, M3prec["="]) || ";$b$}"
```

Uses `field` 53, `Gasgn` 282a, `Ginrange` 282a, `Gsetname` 282a, `M3bracket` 332d, `M3prec` 332d, `namearray` 279a, `node` 260a, and `prettyM3` 325a.

332b ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁332a 332c▷

```
"Gnomatch"  : "<* ASSERT FALSE *> (* no match *)"
"Tunsigned" : if \e.width < wordsize then
                  "[0.." || (2^e.width-1) || "]"
               else
                  "Word.T"
```

Uses `Gnomatch` 282a, `Tunsigned` 282a, and `wordsize` 36c.

332c ⟨*cases for* prettyM3 325b⟩+≡ (325a) ◁332b

```
"absolute_field" : prettyM3(Eslice(literal(wordname(e)), e.field.lo, fwidth(e.field)))
```

Uses `absolute_field` 256, `Eslice` 79, `field` 53, `fwidth` 112d, `literal` 43e, `prettyM3` 325a, and `wordname` 286b.

332d ⟨* 325a⟩+≡ ◁329e 333b▷

```
global M3prec, M3assoc
procedure M3bracket(s, op, p, a)
  /a := "L"
  return "${" || (if M3prec[op] > p | (M3prec[op] = p & M3assoc[p] == a) then s
                   else "(" || s || ")") || "$}"
end
```

Defines:
`M3assoc`, used in chunk 333a.
`M3bracket`, used in chunks 326–30 and 332a.
`M3prec`, used in chunks 326–28, 330b, 332a, and 333a.

333a ⟨*initialize* M3prec *and* M3assoc 333a⟩≡ (325a)

```
ops := ["N", ["low"],
        "L", [";"],
        "L", [","],
        "L", ["OR"],
        "L", ["AND"],
        "L", ["NOT"],
        "L", ["<=", "<", ">=", ">", "=", "#", "IN"],
        "L", ["+", "-", "&"],
        "L", ["MOD", "DIV", "*", "/"],
        "L", ["^"],
        "L", ["app"],
        "L", ["."],
        "N", ["high"]
       ]
M3prec := table([])  # missed lookups break arithmetic comparisons
M3assoc := table()
every i := 1 to *ops by 2 do {
  every M3prec[!ops[i+1]] := i
  M3assoc[i] := ops[i]
}
```

Uses M3assoc 332d and M3prec 332d.

333b ⟨* 325a⟩+≡ ◃332d 333c▹

```
procedure M3op(op)
  return if op == "!=" then "#" else op
end
```

Defines:
M3op, used in chunk 326a.

333c ⟨* 325a⟩+≡ ◃333b 334▹

```
procedure M3noreserve(word)
  static reserved
  initial reserved := set([
       "AND", "ANY", "ARRAY", "AS", "BEGIN", "BITS", "BRANDED", "BY", "CASE",
       "CONST", "DIV", "DO", "ELSE", "ELSIF", "END", "EVAL", "EXCEPT",
       "EXCEPTION", "EXIT", "EXPORTS", "FINALLY", "FOR", "FROM", "GENERIC",
       "IF", "IMPORT", "IN", "INTERFACE", "LOCK", "LOOP", "METHODS", "MOD",
       "MODULE", "NOT", "OBJECT", "OF", "OR", "OVERRIDES", "PROCEDURE",
       "RAISE", "RAISES", "READONLY", "RECORD", "REF", "REPEAT", "RETURN",
       "REVEAL", "ROOT", "SET", "THEN", "TO", "TRY", "TYPE", "TYPECASE",
       "UNSAFE", "UNTIL", "UNTRACED", "VALUE", "VAR", "WHILE", "WITH"])
  return if member(reserved, word) then M3noreserve(word||"_") else word
end
```

Defines:
M3noreserve, used in chunk 329b.

18.1 Support for decoding code in Modula-3

334 ⟨* 325a⟩+≡ ◁333c

```
procedure Generate_M3()
  eqntoC := eqntoM3
  pretty := prettyM3
  gen_outer_decls := [literal(
       "CONST MATCH_bits = ARRAY [0..1] OF Word.T {0, Word.Not(0)}")]
  gen_file_header := ""
  fetchtab["type"] := "ADDRESS"
  fetchtab["add"] := "LOOPHOLE(%a+%o, Word.T)"
  fetchtab["pc"] :=  "%a"
#  fetchtab[8]  := "LOOPHOLE(%a+(%o/ADRSIZE(CHAR)), UNTRACED REF [0..255])^"
#  fetchtab[16] := "LOOPHOLE(%a+(%o/ADRSIZE(CHAR)), UNTRACED REF [0..65535])^"
#  fetchtab[32] := "LOOPHOLE(%a+(%o/ADRSIZE(CHAR)), UNTRACED REF INTEGER)^"
end
```

Defines:
`Generate_M3`, used in chunk 15a.
Uses `fetchtab` 283a, `gen_file_header` 282d, `gen_outer_decls` 282d, `literal` 43e, `pretty` 16a, and `prettyM3` 325a.

Chapter 19

Miscellaneous code-generation procedures

Value computation and shifting is straightforward; the only detail is that signed values have to be masked to remove the high-order (sign) bits.

335a ⟨* 335a⟩≡ 335b ▷

```
procedure unsignedval(f, val)
  constant(val) = 0 | return Cshift(val, "left", f.lo)
end
procedure signedval(f, val)
  constant(val) = 0 |
  return Cshift(Cmask(val, fwidth(f)), "left", f.lo)
end
```

Defines:
`signedval`, used in chunk 335c.
`unsignedval`, used in chunk 335c.
Uses `constant` 111b and `fwidth` 112d.

Signed and unsigned values are stored in `fieldval`

335b ⟨* 335a⟩+≡ ◁ 335a 336 ▷

```
procedure init_tests()
  ⟨init 335c⟩
  return
end
global fieldval
```

Defines:
`init_tests`, used in chunk 18.

335c ⟨*init* 335c⟩≡ (335b) 337c ▷

```
fieldvals := table()
fieldvals["field"] := unsignedval; fieldvals["extended"] := signedval
```

Uses `field` 53, `signedval` 335a, and `unsignedval` 335a.

Size testing is straightforward but tedious. Constants, which usually come from constraints in disjuncts, are tested at generation time. Unsigned values require a single test; signed values require two (i.e. a range test).

336 ⟨* 335a⟩+≡ ◁335b 337a▷

```
procedure unsignedtest(outfile, f, value, inputs)
  bits := fwidth(f)
  bits >= f.class.size |
  if v := constant(value) then
    v < 2^bits |
    error("spec forces ", f.name, " = ", v, ", which won't fit in ", bits, " bits")
  else {
    value := eqntoC(value, inputs, 1)
    write(outfile, indent(), "if ((unsigned)(", value, ") > ", mask(bits), ")\n",
      indent(2), "(*fail)(",
      image(f.name || " = %d won't fit in " || bits || " bits"),
      ", ", value, ");")
  }
  return
end
procedure unsignedcond(f, value, inputs)
  bits := fwidth(f)
  value := eqntoC(value, inputs, 1)
  return "((unsigned)(" || value || ") <= " || mask(bits) || ")"
end
```

Defines:
`unsignedcond`, used in chunk 337c.
`unsignedtest`, used in chunk 337c.

Uses `constant` 111b, `fail` 396, `fwidth` 112d, `indent` 338b, `mask` 338a, and `value` 132a.

337a ⟨* 335a⟩+≡ ◃336 337b▹
```
procedure signedtest(outfile, f, value, inputs)
  bits := fwidth(f)
  bits >= f.class.size |
  if v := constant(value) then
    -(2^(bits-1)) <= v < 2^(bits-1) |
    error("spec forces ", f.name, " = ", v, ", which won't fit in ", bits, " signed bits")
  else {
    value := eqntoC(value, inputs, 1)
    write(outfile, indent(), "if ((int)(", value, ") < -", tworaised(bits-1), " ||\n",
      indent(2), "(int)(", value, ") >= ", tworaised(bits-1), ")\n",
      indent(2), "(*fail)(",
      image(f.name || " = %d won't fit in "|| bits || " signed bits"),
      ", ", value, ");")
  }
  return
end
procedure signedcond(f, value, inputs)
  bits := fwidth(f)
  value := eqntoC(value, inputs, 1)
  return "((int)(" || value || ") >= -" || tworaised(bits-1) || " && \n" ||
         "(int)(" || value || ") < " || tworaised(bits-1) || ")"
end
```
Defines:
`signedcond`, used in chunk 337c.
`signedtest`, used in chunk 337c.
Uses `constant` 111b, `fail` 396, `fwidth` 112d, `indent` 338b, `tworaised` 338a, and `value` 132a.

337b ⟨* 335a⟩+≡ ◃337a 337d▹
```
global fieldconds, fieldtests
```
Defines:
`fieldconds`, used in chunk 337c.
`fieldtests`, used in chunk 337c.

337c ⟨*init* 335c⟩+≡ (335b) ◃335c
```
every fieldconds | fieldtests := table()
fieldconds["field"] := unsignedcond; fieldconds["extended"] := signedcond
fieldtests["field"] := unsignedtest; fieldtests["extended"] := signedtest
```
Uses `field` 53, `fieldconds` 337b, `fieldtests` 337b, `signedcond` 337a, `signedtest` 337a, `unsignedcond` 336, and `unsignedtest` 336.

`testcondition` generates code to test an arbitrary condition generated by the solver.

337d ⟨* 335a⟩+≡ ◃337b 338a▹
```
procedure testcondition(outfile, cond, inputs)
    write(outfile, indent(), "if (!(", eqntoC(cond, inputs, 1), "))\n",
          indent(2), "(*fail)(\"Condition ", expimage(cond), " not satisified.\");")
end
```
Uses `expimage` 98a, `fail` 396, and `indent` 338b.

These functions compute the constants against which values are tested.

338a ⟨* 335a⟩+≡ ◁337d 338b▷

```
procedure mask(bits)
  return "0x" || ((0 ~= (2^(bits % 4) - 1)) | "") || repl("f", bits / 4)
end
procedure tworaised(bits)
  return "0x" || (2^(bits % 4)) || repl("0", bits / 4)
end
```

Defines:
mask, used in chunks 304b, 313d, and 336.
tworaised, used in chunks 315g and 337a.

338b ⟨* 335a⟩+≡ ◁338a

```
procedure indent(n)
    initial /sindent := 2
    return repl(" ", sindent + (\n | 0))
end
```

Defines:
indent, used in chunks 336 and 337.

Chapter 20

Field-dependent information

20.1 Field categories

There are a number of things one wants to know about each field, depending on whether one is to assemble, disassemble, or generate C code for assembly. Here's a tentative classification:

- How to disassemble
  - array of names
  - enumeration (for sparsely populated fields)
  - `printf` format string (e.g. for MIPS `spec03`)?
- Type of argument to pass to C or Modula-3 procedure (could be enumeration or subrange type, for example). Different types may require different value checking within the procedure.
- Literals
  - C `enum`s or constants
  - use in assembly code (syntactic category?)

If field names are specified, they are kept in a name-to-value table. That table is associated with the field in the global table `fnt` (standing for "field-name table").

339 ⟨* 339⟩≡ 340a ▷

```
global fnt                # field -> name -> value
```

Defines:
`fnt`, used in chunks 340a and 342a.
Uses `field` 53 and `value` 132a.

340a ⟨* 339⟩+≡ ◁339 340f▷

```
procedure fieldinfo(f, info)
    initial { /fnt := table(); /fieldasm := table() }
    case type(info) of {
        "string" :  case info of { "checked"    : { ⟨make f checked 340b⟩ }
                                   "unchecked"  : { ⟨make f unchecked 340c⟩ }
                                   "guaranteed" : { ⟨make f guaranteed 340d⟩ }
                                   default : { ⟨complain about fieldspec info 340e⟩ }
                                 }
        "namespec"  : (/fnt[f] := check_namespec(info, f)) |
                          error("Field names for field ", f.name, " already specified")
        default     : fail
    }
    return
end
```

Defines:
`fieldinfo`, used in chunk 35c.

Uses `check_namespec` 341d, `fail` 396, `field` 53, `fnt` 339, and `namespec` 341a.

340b ⟨*make* `f` *checked* 340b⟩≡ (340a)

```
every delete(guaranteed_fields | unchecked_fields, f)
```

Uses `guaranteed_fields` 340f and `unchecked_fields` 340f.

340c ⟨*make* `f` *unchecked* 340c⟩≡ (340a)

```
insert(unchecked_fields, f)
delete(guaranteed_fields, f)
```

Uses `guaranteed_fields` 340f and `unchecked_fields` 340f.

340d ⟨*make* `f` *guaranteed* 340d⟩≡ (340a)

```
every insert(guaranteed_fields | unchecked_fields, f)
```

Uses `guaranteed_fields` 340f and `unchecked_fields` 340f.

340e ⟨*complain about fieldspec* `info` 340e⟩≡ (340a)

```
error("Unknown field info specifier `", info, "'")
```

Uses `field` 53.

Fields can be checked or unchecked at assembly time.

340f ⟨* 339⟩+≡ ◁340a 341a▷

```
global unchecked_fields, guaranteed_fields
```

Defines:
`guaranteed_fields`, used in chunks 30b, 187, and 340.
`unchecked_fields`, used in chunks 30b, 182b, 187, 191c, 202d, and 340.

Field values may be assigned pneumonic names that are used during disassembly of instructions. The names bound to field values within an enumeration are used to synthesize the sames of opcodes.

341a ⟨* 339⟩+≡ ◁340f 341b▷

```
record namespec(nametable, full)
procedure sparse_name_table(bindlist)
  t := table()
  every b := !bindlist do
    (/t[b.name] := b.val) | error("duplicate field names")
  return namespec(t)
end
```

Defines:
namespec, used in chunks 230a, 340a, and 341c.
sparse_name_table, used in chunks 35c and 36a.
Uses field 53.

341b ⟨* 339⟩+≡ ◁341a 341c▷

```
record enumbinding(name, val)
```

Defines:
enumbinding, used in chunks 35c and 36a.

341c ⟨* 339⟩+≡ ◁341b 341d▷

```
procedure full_name_table(namelist, f)
  t := table()
  every i := 1 to *namelist do
    (/t[namelist[i]] := i - 1) | error("duplicate field names")
  return namespec(t, 1)
end
```

Defines:
full_name_table, used in chunks 35c and 36a.
Uses field 53 and namespec 341a.

341d ⟨* 339⟩+≡ ◁341c 342a▷

```
procedure check_namespec(spec, f)
  if \spec.full then
    *spec.nametable = 2^fwidth(f) |
       error("Field ", f.name, " has ", 2^fwidth(f), " values, not ", *spec.nametable)
  if x := spec.nametable[n := key(spec.nametable)] & (x < 0 | x >= 2^fwidth(f)) then
    error("name ", image(n), " = ", x,
          " falls outside the value range of field ", f.name)
  return spec.nametable
end
```

Defines:
check_namespec, used in chunks 230a and 340a.
Uses field 53, fwidth 112d, and value 132a.

342a ⟨* 339⟩+≡ ◃341d 342b▹
```
procedure fieldname_table(f)
  initial /fnt := table()
  return fnt[f]
end
```
Defines:
  fieldname_table, used in chunks 158b, 159b, 230, 245, 302c, 342b, and 343b.
Uses fnt 339.

342b ⟨* 339⟩+≡ ◃342a 342c▹
```
procedure fieldname_env_for(f)
  return [\fieldname_table(f)] | []
end

procedure fieldname_env_for_ipt(ipt)
  return (type(ipt.meaning) == "field", fieldname_env_for(ipt.meaning)) | []
end
```
Defines:
  fieldname_env_for, used in chunk 75a.
  fieldname_env_for_ipt, used in chunk 171a.
Uses field 53 and fieldname_table 342a.

20.2 Emitting field names

We concoct a `namearray` and pass it to `pretty`.

342c ⟨* 339⟩+≡ ◃342b 343b▹
```
procedure emit_fieldnames(base)
  local header, data, fields
  data := open(base || implementation_extension, "w")
  header := open(base || interface_extension, "w");
  if interface_extension == ".h" then
    write(data, "#include \"", base, interface_extension, "\"")
  pp := PPnew(data);
  ⟨make fields an array of all fields sorted by name 343a⟩
  every PPxwrite(pp, pretty(Gdeclnamearray(fieldnamearray(!fields))), ";")
  every write(header, "extern char *", (!fields).name, "_names[];")
  every close(data | header)
  return
end
```
Defines:
  emit_fieldnames, used in chunk 18.
Uses fieldnamearray 343b, Gdeclnamearray 281b, implementation_extension 16a,
  interface_extension 16a, and pretty 16a.

343a ⟨*make* `fields` *an array of all fields sorted by name* 343a⟩≡ (342c)

```
t := table()
fields := []
every f := !symtab & type(f) == "field" do t[f.name] := f
every put(fields, (!sort(t))[2])
```

Uses `field` 53 and `symtab` 30a 353a.

To build a name array, we first create a table of all invalid field values. The valid ones are then overwritten using names from `fieldname_table(f)`.

343b ⟨* 339⟩+≡ ◁342c 344a▷

```
procedure fieldnamearray(f)
  return name_array_from_table(\fieldname_table(f), 2^fwidth(f), f.name)
end
procedure name_array_from_table(t, limit, fieldname)
  local name
  limit <= 1024 |
    error("Tried to enumerate ", limit, " names for field or operand ", fieldname)
  na := namearray(field, table(), limit, fieldname || "_names", "")
  every i := 0 to na.hi - 1 do
    na.tbl[i] := bad_operand_name(fieldname, i)
  every name := key(t) do
    na.tbl[t[name]] := name
  return na
end
procedure bad_operand_name(name, value)
  return "??" || name || "=" || value || "?!"
end
```

Defines:
- `bad_operand_name`, used in chunk 236.
- `fieldnamearray`, used in chunk 342c.
- `name_array_from_table`, used in chunk 236.

Uses `field` 53, `fieldname_table` 342a, `fwidth` 112d, `namearray` 279a, and `value` 132a.

For use in identifying fields with identical naming patterns, we have to turn a name table into a suitable string. This means covering the keys in order.

344a ⟨* 339⟩+≡ ◁343b

```
procedure nametablekey(nametab)
  local min, max
  static cache
  initial cache := table()
  if \cache[nametab] then return cache[nametab]
  if /nametab then return cache[f] := "(no name table)"
  u := table()
  every k := key(nametab) do u[nametab[k]] := k
  ⟨make min and max smallest and largest values in nametab 344b⟩
  s := ""
  every i := min to max do
    s ||:= \u[i] || "@" || i || "\0"
  return cache[f] := s
end
```

Defines:
`nametablekey`, used in chunk 237d.
Uses `max` 396 and `min` 396.

344b ⟨*make* `min` *and* `max` *smallest and largest values in* `nametab` 344b⟩≡ (344a)

```
min := max := !nametab | 0
every min >:= !nametab
every max <:= !nametab
```

Uses `max` 396 and `min` 396.

Chapter 21

Lexical analysis

`token` describes the category of the token, and `tval` is its value. `token == tval` except for integers and identifiers. `file` and `line` describe the source file and line number from which the token came.

345a ⟨* 345a⟩≡ 345b ▷

```
global token, tval, file, line
```

Defines:
- `file`, used in chunks 9–14, 21, 22, 99a, 106d, 218a, 251a, 259b, 263c, 264a, 272a, 273a, 281a, 282d, 289a, 292, 318c, 330d, 346a, 353, 354, and 358–60.
- `line`, used in chunks 15j, 21a, 99a, 106d, 161b, 196c, 251a, 259b, 263c, 264a, 272a, 273a, 281a, 282d, 289a, 295, 318c, 330d, 346a, 358a, and 360.
- `token`, used in chunks 19d, 79, 188b, and 347–49.
- `tval`, used in chunks 19d, 347, and 349.

There are three special tokens.

345b ⟨* 345a⟩+≡ ◁345a 346b ▷

```
global EOF, IDENT, INT, CODELINE, CASELINE, NEWLINE, WHITESPACE, lexstate
```

Defines:
- `CASELINE`, used in chunks 49a, 346a, and 347a.
- `CODELINE`, used in chunks 49a, 346, and 347.
- `EOF`, used in chunks 19d and 346a.
- `IDENT`, used in chunks 47c, 52, 346a, 348a, and 349a.
- `INT`, used in chunks 32–34, 36b, 39c, 47c, 49b, 52, 346a, and 349a.
- `lexstate`, used in chunks 346, 347, 349b, and 351a.
- `NEWLINE`, used in chunks 43a, 50c, 346a, and 348a.
- `WHITESPACE`, used in chunks 52, 346a, and 348b.

346a ⟨*initialize* `tokens` 346a⟩≡ (346b)

```
CASELINE   := " case ... of "
CODELINE   := " code line "
EOF        := " end of file "
IDENT      := " identifier "
INT        := " integer "
NEWLINE    := " newline "
WHITESPACE := " white space "
```

Uses `CASELINE` 51d 345b, `CODELINE` 51d 345b, `EOF` 345b, `file` 345a, `IDENT` 51d 345b 346b, `INT` 51d 345b, `line` 345a, `NEWLINE` 51d 345b, `white` 350a, and `WHITESPACE` 51d 345b.

346b ⟨* 345a⟩+≡ ◁345b 348a▷

```
global SEMANTICS, IDENT, STARTSEM, ENDSEM, succptr
procedure scantokens(eol)
  static alpha, alphanum, multichar
  initial {
    alpha      := &letters ++ '_.'
    alphanum   := alpha ++ &digits
    multichar  := set(["<=", ">=", "!=", ">>", "<<", "=>", "..."])
    /lexstate  := "INITIAL"
    ⟨initialize tokens 346a⟩
  }
  if \eol then return possible_newline()
  case lexstate of {
    "INITIAL" | "CASEPAT" : {⟨lexer for INITIAL and CASEPAT 349a⟩}
    ⟨cases for other lexical states 347a⟩
  }
end
```

Defines:
`IDENT`, used in chunks 47c, 52, 346a, 348a, and 349a.
`succptr`, used in chunks 49a, 247, 248, 251a, 259b, 288a, and 347a.

Uses `lexstate` 345b and `possible_newline` 348a.

347a ⟨*cases for other lexical states* 347a⟩≡ (346b) 347b ▷

```
"CODE": {
  if (optwhite(), ="match ", optwhite(),
           (="[", optwhite(), any(&letters),
               succptr := tab(many(alphanum)), optwhite(), ="]", optwhite()) |
        (succptr := &null),
     tval := tab(find(" to")), =" to", optwhite(), pos(0)) then {
    lexstate := "CASEPAT"
    return CASELINE
  } else if optwhite() & ="match " & tval := tab(find(" to")) & =" to" &
                                          optwhite() & pos(0) then {
    impossible("bogus code in lex")
    lexstate := "CASEPAT"
    return CASELINE
  } else {    # warning message for ill-formed case?
    tval := tab(0)
    return CODELINE
  }
}
```

Uses bogus 113b, CASELINE 51d 345b, CODELINE 51d 345b, lexstate 345b, optwhite 350a, succptr 346b, and tval 345a.

347b ⟨*cases for other lexical states* 347a⟩+≡ (346b) ◁ 347a

```
"CASEARM": {
  if optwhite() & tval := (="|" | 1(=("else" | "endmatch"), white() | pos(0))) then {
    lexstate :=
        case tval of { "|" : "CASEPAT" ; "else" : "CASEARM" ; "endmatch" : "CODE" }
    return token := tval
  } else {
    tval := tab(0)
    return CODELINE
  }
}
```

Uses CODELINE 51d 345b, lexstate 345b, optwhite 350a, token 345a, tval 345a, and white 350a.

There's a ghastly hack that makes exactly one newline visible after every opcode of a constructor spec. What we do is, before parsing a constructor spec, we turn "newline-vision" on if the next token is an identifier. Encountering a newline or reserved word or { or other stuff turns it off again.

348a ⟨* 345a⟩+≡ ◁346b 348b▷

```
global NEWLINEVISION
procedure see_newline()
  if token == IDENT then NEWLINEVISION := 1
  return NEWLINEVISION
end
procedure ignore_newlines()
  return NEWLINEVISION := &null
end
procedure possible_newline()
  if \NEWLINEVISION then {
    NEWLINEVISION := &null
    return NEWLINE
  }
  return possible_whitespace()
end
```

Defines:
ignore_newlines, used in chunk 349a.
possible_newline, used in chunks 346b and 349a.
see_newline, used in chunks 42a and 50a.
Uses IDENT 51d 345b 346b, NEWLINE 51d 345b, possible_whitespace 348b, and token 345a.

There's an equally ghastly hack that makes one unit of white space visible. Note that we don't require an identifier.

348b ⟨* 345a⟩+≡ ◁348a 350a▷

```
global WHITESPACEVISION
procedure see_whitespace()
  WHITESPACEVISION := 1
  return WHITESPACEVISION
end
procedure ignore_whitespace()
  return WHITESPACEVISION := &null
end
procedure possible_whitespace()
  if \WHITESPACEVISION then {
    return WHITESPACE
  }
end
```

Defines:
ignore_whitespace, used in chunk 52.
possible_whitespace, used in chunks 348a and 349a.
see_whitespace, used in chunk 52.
Uses WHITESPACE 51d 345b.

349a ⟨*lexer for INITIAL and CASEPAT* 349a⟩≡ (346b)
```
if tval := white() then  # don't backtrack past white()
  if token := possible_whitespace() then
    return token
if {not pos(0)} then {
  thispos := &pos
  if ="#" then {
    tab(0) & (token := possible_newline() | fail)
  } else if tval := (hexint() | octalint() | binaryint() | decimalint()) then {
    token := INT
  } else if tval := (any(alpha), tab(many(alphanum))) then {
    token := 1(reserved(tval), ignore_newlines()) | IDENT
  } else if tval := =!multichar then {
    token := tval
    ⟨change lexer state from CASEPAT to CASEARM if tval is "=>" 349b⟩
  } else if token := ="\"" then {
    delim := move(-1) # recover quote
    tval := ""
    while =delim do {  # turn 'don''t into "don't"
      tval ||:= tab(upto(delim) | 0)
      =delim | error("unclosed ", delim)
    }
  } else if token := ="'" then { # character constant
      tval := charconst()
      ="'" | error("character constant too large (or unclosed single quote)")
  } else {
    token := tval := move(1)
    if token == ("{"|".") then ignore_newlines()
  }
  return token
}
return possible_newline()
```
Uses `binaryint` 350d, `charconst` 351b, `constant` 111b, `decimalint` 350e, `fail` 396, `hexint` 350b, `IDENT` 51d 345b 346b, `ignore_newlines` 348a, `INT` 51d 345b, `octalint` 350c, `possible_newline` 348a, `possible_whitespace` 348b, `token` 345a, `tval` 345a, and `white` 350a.

349b ⟨*change lexer state from* `CASEPAT` *to* `CASEARM` *if* `tval` *is* `"=>"` 349b⟩≡ (349a)
```
case tval of {
  "=>" : if lexstate == "CASEPAT" then lexstate := "CASEARM"
}
```
Uses `lexstate` 345b and `tval` 345a.

350a ⟨* 345a⟩+≡ ◁348b 350b▷
```
procedure optwhite()
  suspend white() | ""
end

procedure white()
  suspend tab(many(' \t'))
end
```
Defines:
optwhite, used in chunks 243 and 347.
white, used in chunks 42b, 346a, 347b, and 349a.

ge #26

350b ⟨* 345a⟩+≡ ◁350a 350c▷
```
procedure hexint()
  static hexdigits
  initial hexdigits  := &digits ++ 'abcdefABCDEF'
  suspend ="0x" & integer("16r" || tab(many(hexdigits)))
end
```
Defines:
hexint, used in chunk 349a.

350c ⟨* 345a⟩+≡ ◁350b 350d▷
```
procedure octalint()
  static octaldigits
  initial octaldigits  := '01234567'
  suspend ="0" & integer("8r" || tab(many(octaldigits)))
end
```
Defines:
octalint, used in chunk 349a.

350d ⟨* 345a⟩+≡ ◁350c 350e▷
```
procedure binaryint()
  static binarydigits
  initial binarydigits  := '01'
  suspend ="0b" & integer("2r" || tab(many(binarydigits)))
end
```
Defines:
binaryint, used in chunk 349a.

350e ⟨* 345a⟩+≡ ◁350d 351a▷
```
procedure decimalint()
  static decimaldigits
  initial decimaldigits  := &digits
  suspend integer(tab(many(&digits)))
end
```
Defines:
decimalint, used in chunk 349a.

351a ⟨* 345a⟩+≡ ◁350e 351b▷
```
procedure codelex(in)
  lexstate := "CODE"
  return lex(in)
end
```
Defines:
codelex, used in chunk 19a.
Uses lexstate 345b.

21.1 Support for character constants

351b ⟨* 345a⟩+≡ ◁351a 352a▷
```
procedure charconst()
   local ns, c

   return if ="\\" then {⟨backslash escape 351c⟩} else move(1)
end
```
Defines:
charconst, used in chunk 349a.

351c ⟨*backslash escape* 351c⟩≡ (351b)
```
c := map(move(1)) | fail # backslash newline fails
case c of {
   "b":  "\b"
   "d":  "\d"
   "e":  "\e"
   "f":  "\f"
   "l":  "\n"
   "n":  "\n"
   "r":  "\r"
   "t":  "\t"
   "v":  "\v"
   "'":  "'"
   "\"":  "\""
   "\\" : "\\"
   "x":  hexchar()
   !"01234567":  octcode()
   default:  error("bad backslash escape \\", c, tab(0))
}
```
Uses fail 396 and octcode 352b.

352a ⟨* 345a⟩+≡ ◁351b 352b▷

```
procedure hexcode()
   local i, s
   static hdigits
   initial hdigits := ~'0123456789ABCDEFabcdef'

   move(i := 2 | 1) ? s := tab(upto(hdigits) | 0)
   move(*s - i)
   return char("16r" || s)
end
```

352b ⟨* 345a⟩+≡ ◁352a

```
procedure octcode()
   local i, s
   static odigits
   initial odigits := ~'01234567'

   move(-1)
   move(i := 3 | 2 | 1) ? s := tab(upto(odigits) | 0)
   move(*s - i)
   if s > 377 then {    # back off if too large
      s := s[1:3]
      move(-1)
      }
   return char("8r" || s)
end
```

Defines:
`octcode`, used in chunk 351c.

Chapter 22

Output

Routines for producing ASCII and "dag" representations of patterns and decision trees.

353a ⟨* 353a⟩≡ 353b ▹

```
global equivclasses, symtab

procedure outspec(file)
    every c := !equivclasses do {
        writes(file, "fields(", c.size, ") ")
        every f := !c.fields do writes(file, fieldimage(f))
        write(file)
    }
    outpatterndefs(file, set())
    write(file, "%%")
end
```

Defines:
`equivclasses`, used in chunks 30b and 33a.
`outspec`, used in chunk 18.
`symtab`, used in chunks 30b, 33, 37b, 43–45, 48b, 112, 166b, 168b, 182b, 230, 235, 250a, 343a, 354a, and 355a.

Uses `fieldimage` 353b 353b, `file` 345a, and `outpatterndefs` 354a 354a.

353b ⟨* 353a⟩+≡ ◃353a 354a ▹

```
procedure fieldimage(f)
    return f.name || " " || f.lo || ":" || f.hi || " "
end
```

Defines:
`fieldimage`, used in chunks 25c and 353a.

354a ⟨* 353a⟩+≡ ◁353b 354b▷

```
procedure outpatterndefs(file, written)
    /written := patnames()
    pats := sort(symtab)
    # first do patterns that don't depend on other patterns, then catch as catch can
    every pair := !pats & name := pair[1] & p := pair[2] & type(p) == "pattern" do {
        if not ((!p.disjuncts).name ~== name) then # doesn't depend on others
            outpatterndef(file, name, p, written)
    }
    every pair := !pats & type(pair[2]) == "pattern" do
        outpatterndef(file, pair[1], pair[2], written)
    return
end
```

Defines:
`outpatterndefs`, used in chunk 353a.
Uses `file` 345a, `outpatterndef` 354b 354b, `patnames` 355a, `pattern` 53, and `symtab` 30a 353a.

354b ⟨* 353a⟩+≡ ◁354a 354c▷

```
procedure outpatterndef(file, name, p, written)
    if not member(written, name) then {
        every d := !p.disjuncts & name ~== \d.name do
            outpatterndef(file, d.name, lookup(d.name), written)
        writes(file, "pattern ", name, " : ")
        write(file, patternimage(p, written))
        insert(written, name)
    }
    return
end
```

Defines:
`outpatterndef`, used in chunk 354a.
Uses `file` 345a, `lookup` 23a, `pattern` 53, and `patternimage` 355b.

354c ⟨* 353a⟩+≡ ◁354b 355a▷

```
procedure outpattern(file, p, written)
    /written := patnames()
    return writes(file, patternimage(p, written))
end
```

Defines:
`outpattern`, used in chunks 265b, 267b, and 360.
Uses `file` 345a, `patnames` 355a, and `patternimage` 355b.

355a ⟨* 353a⟩+≡ ◁354c 355b▷
```
procedure patnames()
  static s
  initial { s := set(); every insert(s, key(symtab)) } # cheat
  return s
end
```
Defines:
patnames, used in chunk 354.
Uses symtab 30a 353a.

355b ⟨* 353a⟩+≡ ◁355a 355c▷
```
procedure patternimage(p, written)
    /written := emptyset
    return if member(written, p.name) then # temporary hack
        p.name
    else {
        l := []; every put(l, disjunctimage(!p.disjuncts, p.name))
        if *l > 0 then commaseparate(l, " | ") else "<NOMATCH>"
    }
end
```
Defines:
patternimage, used in chunks 25c, 354, and 363a.
Uses disjunctimage 355d and emptyset 16a.

355c ⟨* 353a⟩+≡ ◁355b 355d▷
```
link commafy
```

355d ⟨* 353a⟩+≡ ◁355c 356a▷
```
procedure disjunctimage(d, name)
    return if \name == \d.name | /d.name then
        case type(d) of {
            "disjunct" : sequenceimage(d.sequents)
            "adisjunct" :
               if adalwaysmatches(d) then "<ELSE>"
               else commaseparate(maplist(constraintimage, d.aconstraints), " & ")
            default : impossible("disjunct type")
        }
    else
        d.name
end
```
Defines:
disjunctimage, used in chunk 355b.
Uses adalwaysmatches 261, adisjunct 256, constraintimage 357a 357a, disjunct 53,
and sequenceimage 356a.

356a ⟨* 353a⟩+≡ ◁355d 356b▷

```
procedure sequenceimage(l)
  return if *l = 0 then "<EPSILON>"
         else if l[1] === dots then "... " || sequenceimage(l[2:0])
         else if l[-1] === dots then sequenceimage(l[1:-1]) || " ..."
         else commaseparate(maplist(sequentimage, l), "; ")
end
```

Defines:
sequenceimage, used in chunks 66 and 355d.
Uses dots 60c and sequentimage 356b.

356b ⟨* 353a⟩+≡ ◁356a 357a▷

```
procedure sequentimage(s)
  return if s === dots then impossible("image of dots")
         else if type(s) == ("patlabel" | "latent_patlabel") then expimage(s)
         else if *s.constraints = 0 then "<Impossible ELSE?>"
         else commaseparate(maplist(constraintimage, s.constraints), " & ")
end
```

Defines:
sequentimage, used in chunk 356a.
Uses constraintimage 357a 357a, constraints 132a, dots 60c, expimage 98a,
latent_patlabel 53, and patlabel 53.

357a ⟨* 353a⟩+≡ ◁356b 357b▷

```
procedure constraintimage(c)
    local bitcount
    s := ""
    case type(c) of {
    "constraint": {
        bitcount := fieldbitcount(c.field)
        if c.lo + 1 = c.hi then
            s ||:= patimage(c.field) || " == " || c.lo
        else if c.lo = 0 & c.hi = 2^bitcount then
            s ||:= "<ANY " || bitcount || "-bit " || patimage(c.field)|| ">"
        else if c.lo >= c.hi then
            s ||:= "<IMPOSSIBLE " || patimage(c.field) || ">"
        else {
            if c.lo > 0 then s ||:= patimage(c.field) || " >= " || c.lo
            if c.lo > 0 & c.hi < 2^bitcount then s ||:= " & "
            if           c.hi < 2^bitcount then
                        s ||:=  patimage(c.field) || " < " || c.hi
        }
    }
    "fieldbinding" : s ||:= patimage(c.field) || " = " || image(c.code)
    default : impossible("constraint type")
    }
    return s
end
```

Defines:
`constraintimage`, used in chunks 67a, 187, 355d, and 356b.
Uses `constraint` 53, `field` 53, `fieldbinding` 53, `fieldbitcount` 357b, and `patimage` 358a.

357b ⟨* 353a⟩+≡ ◁357a 358a▷

```
procedure fieldbitcount(f)
   return case type(f) of {
     "absolute_field" : fieldbitcount(f.field)
     "field" : fwidth(f)
     default : impossible("field type")
   }
end
```

Defines:
`fieldbitcount`, used in chunk 357a.
Uses `absolute_field` 256, `field` 53, and `fwidth` 112d.

358a ⟨* 353a⟩+≡ ◁357b 358b▷

```
procedure patimage(v)
    return case type(v) of {
        "list" | "set" :  "[" || commaseparate(maplist(patimage, v)) || "]"
        "matching_stmt" : "MATCH " || patimage(v.arms) || "ENDMATCH"
        "arm" : image(v.file) || ", line " || string(v.line) || ": " ||
                       patimage(v.pattern) || " => " || v.code
        "pattern" : commaseparate(maplist(patimage, v.disjuncts), " | ")
        "disjunct" : if *v.sequents = 0 then "epsilon"
                     else commaseparate(maplist(patimage, v.sequents), "; ")
        "adisjunct": if *v.aconstraints = 0 then "<MATCH>"
                     else commaseparate(maplist(patimage, v.aconstraints), " & ")
        "sequent" : if *v.constraints = 0 then "some " || v.class.name
                    else commaseparate(maplist(patimage, v.constraints), " & ")
        "constraint" : stringininterval(patimage(v.field), v.lo, v.hi)
        "fieldbinding" : patimage(v.field) || " = " || image(v.code)
        "field" : v.name
        "absolute_field" : "{" || patimage(v.field) || " at " || v.offset || "}"
        default : image(v)
    }
end
```

Defines:
`patimage`, used in chunks 35b, 70c, 71b, 264a, 266, 268a, 278a, 357a, and 359b.
Uses `absolute_field` 256, `adisjunct` 256, `arm` 259b, `constraint` 53, `constraints` 132a, `disjunct` 53, `epsilon` 60b, `field` 53, `fieldbinding` 53, `file` 345a, `line` 345a, `matching_stmt` 259b, `pattern` 53, `sequent` 53, and `stringininterval` 358b.

358b ⟨* 353a⟩+≡ ◁358a 359a▷

```
procedure stringininterval(name, lo, hi)
    local r, result
    if type(lo) == "set" then {
        r := sort(lo ++ hi)
        if *r = 0 then return "<EMPTY RANGE??>"
        result := "( " ; result ||:= stringininterval(name, get(r), get(r))
        while result ||:= " | " || stringininterval(name, get(r), get(r))
        return result || " )"
    } else if lo + 1 = hi then return name || " == " || fnum(lo)
    else return fnum(lo) || " <= " || name || "  < " || fnum(hi)
end
```

Defines:
`stringininterval`, used in chunks 101a, 266, 358a, and 359b.
Uses `fnum` 359a.

359a ⟨* 353a⟩+≡ ◁358b 359b▷

```
procedure fnum(n)
    n := string(n) | impossible("format")
    if *n < 2 then n := right(n, 2)
    return n
end
```

Defines:
`fnum`, used in chunk 358b.

22.1 Writing trees in ascii format

359b ⟨* 353a⟩+≡ ◁359a 360▷

```
procedure outtree(file, n, prefix, shownode)
  if /prefix then write(file, "TREE ", image(n.name))
  else write(image(n.name))
  if *n.children > 0 then {
    span := *patimage(n.field) + 14
    every e := !n.children do
      outtree(file, e.node, (\prefix || " & " | "  ") \ 1 ||
        right(stringininterval(patimage(n.field), e.lo, e.hi),span), shownode)
  } else {
    if *n.cs.arms = 0 then write(file, \prefix | "  ", " : NO MATCH")
    else
      every write(file, (\prefix | "  ANY?  ")\1, " : ", expimage(!n.cs.arms))
  }
  if /prefix then write(file, "END TREE\n")
  return
end
```

Defines:
`outtree`, used in chunks 20a, 278b, and 283b.
Uses `expimage` 98a, `field` 53, `file` 345a, `node` 260a, `patimage` 358a, and `stringininterval` 358b.

22.2 Writing trees and dags in dot(1) format

360 ⟨* 353a⟩+≡ ◁359b 361b▷

```
global pagesize
procedure dotouttree(file, root)
    write(file, "digraph decisions {")
    write(file, "  page=", image(pagesize), ";")
    write(file, "  ratio=compress;")
    dotoutnode(file, root, table(), create(seq()))
    write(file, "}")
end

procedure dotoutnode(file, n, known, number)
    if /known[n] then {
        if *n.children > 0 then {
            if alt := alternates(n) then {
                writes(file, "N", known[n] := @number, " [label=\"")
                every i := 1 to *alt.constraints do {
                   if i > 1 then writes(file, ",")
                   writes(file, alt.constraints[i].field.field.name)
                   writes(file, "@", 0 < alt.constraints[i].field.offset)
                }
                write(file, "\"];")
                every dotoutnode(file, alt.thennode | alt.elsenode, known, number)
                ⟨make s a label string for alt.constraints 361a⟩
                write(file, "N", known[n], " -> N", known[alt.thennode], " [label=",
                             image(s), "];")
                write(file, "N", known[n], " -> N", known[alt.elsenode],
                            " [label=\"else\"];")
            } else {
                write(file, "N", known[n] := @number,
                            " [label=", image(n.field.field.name),
                            "@" || (0 < n.field.offset) | "", "];")
                every e := !n.children do {
                    dotoutnode(file, e.node, known, number)
                    r := sort(e.lo ++ e.hi)
                    lo := get(r) & hi := get(r)
                    s := ""
                    s ||:= if lo+1 = hi then lo else lo || "-" || (hi - 1)
                    while lo := get(r) & hi := get(r) do
                      s ||:= ", " || if lo+1 = hi then lo else lo || "-" || (hi - 1)
                    write(file, "N", known[n], " -> N", known[e.node], " [label=",
                          image(commaseparate(split10(s), "\n")), "];")
                }
            }
        } else if *n.cs.arms = 0 then
```

```
            write(file, "N", known[n] := @number, " [label=", image("NO MATCH"), "];")
        else {
            writes(file, "N", known[n] := @number, " [label=\"")
#           outpattern(file, n.cs.arms[1].original.pattern)
            writes(file, \n.cs.arms[1].original.pattern.name |
                         "[" || n.cs.arms[1].original.line || "]")
            write(file, "\"];")
        }
    }
    return
end
```

Defines:
dotouttree, used in chunk 20a.
pagesize, used in chunks 14e and 17a.
Uses alternates 20b, constraints 132a, field 53, file 345a, label 369a, line 345a, node 260a, outpattern 354c, pattern 53, and split10 361b.

361a ⟨*make* s *a label string for* alt.constraints 361a⟩≡ (360)

```
s := ""
every i := 1 to *alt.constraints do {
   if i > 1 then s ||:= ","
   c := alt.constraints[i]
   if *c.lo > 1 then s ||:= "..."
   else if ?c.lo + 1 = ?c.hi then s ||:= ?c.lo
   else s ||:= ?c.lo || "-" || (?c.hi - 1)
}
s := split10(s)
```

Uses constraints 132a and split10 361b.

361b ⟨* 353a⟩+≡ ◁360

```
procedure split10(s)
  static N
  initial N := 18
  if *s > N & find(", ", s) <= N then {
     every k := find(", ", s) <= n
     return s[1:k+1] || "\n" || split10(s[k+2:0]) # keep comma, drop space
  } else {
    return s
  }
end
```

Defines:
split10, used in chunks 360 and 361a.

362a ⟨* 362a⟩≡ 363a▷

```
global debug, debugs, debugstack
procedure wrerr(L[])
    write ! ([&errout] ||| L)
end
procedure wrerrs(L[])
    writes ! ([&errout] ||| L)
end
procedure nop()
end

procedure debug_on()
  /debugstack := []
  every push(debugstack, debug | debugs)
  debug := wrerr
  debugs := wrerrs
  return
end

procedure debug_off()
  /debugstack := []
  every push(debugstack, debug | debugs)
  debug := debugs := nop
  return
end

procedure debug_pop()
  every debugs | debug := pop(debugstack)
  return
end
```

Defines:
`debug`, used in chunks 11, 12a, 17a, 132–36, 138–41, 144b, 146a, 148, 150, and 153.
`debug_on`, used in chunk 11k.
`debugs`, used in chunks 17a, 132a, and 141e.
`nop`, used in chunks 17a and 262.

362b ⟨*integer-input error* 362b⟩≡

```
# error("unknown integer input `", image(i),"' of type `", type(i), "'")
```

Uses `input` 44a.

Note this procedure doesn't parenthesize properly.

363a ⟨* 362a⟩+≡ ◁362a 363c▷

```
procedure astimage(n)
    return case type(n) of {
        "pattern"  : "<pattern> " || patternimage(n)
        "Pident"   : n.name
        "Pcon"     : n.name || " " || n.relop || " " || astimage(n.value)
        "Pand"     : commaseparate(astimage(n.patterns), " & ")
        "Por"      : commaseparate(astimage(n.patterns), " | ")
        "Pseq"     : ⟨image of Pseq (possibly with dots) 363b⟩
        "Papp"     : astimage(n.cons) || "(" || commaseparate(astimage(n.args)) || ")"
        "integer"  : n
        "string"   : n
        "Glist"    : astimage(n.values)
        "list"     : maplist(astimage, n)
        default    : image(n)
    }
end
```

Defines:
`astimage`, used in chunk 363b.
Uses `Glist` 40b, `Pand` 39a, `Papp` 39a, `pattern` 53, `patternimage` 355b, `Pcon` 39a, `Pident` 39a, `Por` 39a, `Pseq` 39a, and `value` 132a.

363b ⟨*image of* `Pseq` *(possibly with dots)* 363b⟩≡ (363a)

```
if *n.patterns = 2 & n.patterns[1] === dots_pattern() then
  "... " || astimage(n.patterns[2])
else if *n.patterns = 2 & n.patterns[2] === dots_pattern() then
  astimage(n.patterns[1]) || " ..."
else
  commaseparate(astimage(n.patterns), "; ")
```

Uses `astimage` 363a and `dots_pattern` 60c.

363c ⟨* 362a⟩+≡ ◁363a 364a▷

```
procedure solnimage(soln)
  s := envimage(soln.answers, "answers") || "\n"
  s ||:= "Uses " || commafy(maplist(expimage, sort(\soln.used))) || "\n"
  s ||:= "Defines " || commafy(maplist(expimage, sort(\soln.defined))) || "\n"
  s ||:= "(constraints: " || commafy(maplist(expimage, \soln.constraints)) || " )"
  return s
end
```

Defines:
`solnimage`, used in chunk 251.
Uses `constraints` 132a, `defined` 132a, `envimage` 25a, `expimage` 98a, and `used` 132a.

364a ⟨* 362a⟩+≡ ◁363c 364b▷

```
procedure injectimage(i)
  return "\tas pattern: " || expimage(i.pattern) || "\n" ||
         "\tas integer: " || expimage(i.integer) || "\n" ||
         "\tas consop: " || expimage(i.consop) || "\n"
end
```

Uses expimage 98a and pattern 53.

Show with positive coefficients.

364b ⟨* 362a⟩+≡ ◁364a

```
procedure zeroimage(z)
  e := eqn(table(0), "=", table(0))
  every k := key(z) do
    if z[k] > 0 then e.left[k] := z[k] else e.right[k] := -z[k]
  return expimage(e)
end
```

Uses eqn 81c and expimage 98a.

Part II

Library code

Chapter 23

The library

This file implements the toolkit library. The declarations of exported procedures and types appear in `library.nw`. Some procedures and types are exported here to help support exported macros, but users aren't supposed to know about them.

Our goal in this implementation is to make sequential emission of instructions very fast.

23.1 Relocatable Blocks

Figure 23.1 shows the parts of a relocatable block that handle its `contents`. The block uses a linked list of buffers. Each buffer is shown as a dashed box. Within the buffer, `data` points to the first byte of the data area (solid box), and `limit` points just beyond the last byte.

367 ⟨*exported data declarations* 367⟩≡ (395) 368a ▷

```
typedef struct rblock_buffer {
  unsigned char *data, *limit;
  struct rblock_buffer *next;
  unsigned datalc;
```

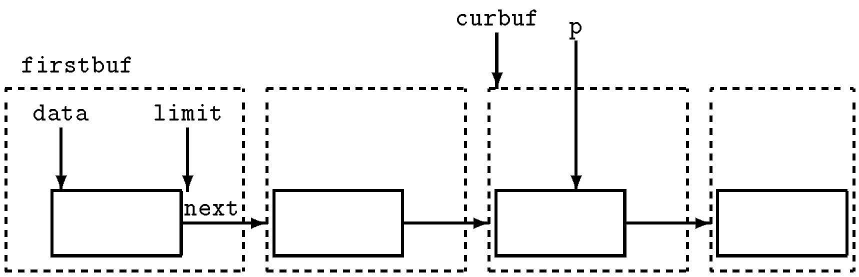

Figure 23.1: Relocatable block with 4 buffers

```
} *Buffer;
```

Defines:
Buffer, used in chunks 368b, 377a, 386, and 387.
Uses next 385c.

`datalc` caches the value of the location counter `lc` that is associated with the beginning of the buffer, so that it doesn't always have to be recomputed by walking along the list.

Although a relocatable block has a location counter, we don't actually store it. Instead, we store `p`, which is a pointer into the current buffer, called `curbuf`; from that we can compute an integer location counter if we have to.[1] But what happens initially, before there's any data? We want to be able to move the location counter around without allocating any buffer space, so we give every relocatable block an initial, empty buffer called `firstbuf`. We can then move the location counter around by changing `firstbuf.datalc`. Moreover, we can tell when the low water mark has been set by seeing when `firstbuf.next` is non-null, in which case `firstbuf.datalc` is the low water mark:

368a ⟨*exported data declarations* 367⟩+≡ (395) ◁367 370a▷

```
struct reloc_block {
  struct rblock_buffer firstbuf, *curbuf;
  unsigned char *p;      /* points to current lc -- dirty */
  unsigned size;         /* highest lc ever -- dirty */
  ⟨other fields of a relocatable block 369a⟩
};
```

Uses lc 382b.

A *dirty* field is one that is invalid in the presence of certain optimizations; we give more dirt below.

368b ⟨*initialize a new relocatable block* rb 368b⟩≡ (384d) 369b▷

```
rb->curbuf = &rb->firstbuf;
⟨set data and limit in a new rb->firstbuf 371d⟩
rb->p = rb->firstbuf.data;
rb->firstbuf.next = (Buffer)0;
rb->firstbuf.datalc = 0;         /* lc is initially 0 */
```

Uses Buffer 367, lc 382b, and next 385c.

Again because of sneaky optimizations, we won't tell you just how we're initializing `data` and `limit` in a new block, except to say they're equal (thus the first block is empty).

[1] Actually, as we show below, p needn't point into the current buffer, but its meaning is still relative to the start of the current buffer.

A relocatable block has three other properties: an address, a label, and associated register information. Some of this properties can be null, so I include some flags that indicate when values are null. These flags are put in a structure called **known**, so that they have the correct values (not known) if a relocatable block is initialized to all zeroes. This property makes an all-zeroes undefined relocatable block valid—the undefined block mustn't have a known `address` because that makes it possible to see whether a label is defined with a simple check to see if the address of the label's block is known. Otherwise we would need an extra check to make sure the block wasn't the undefined block.

369a ⟨*other fields of a relocatable block* 369a⟩≡ (368a) 376d ▹

```
struct { unsigned address:1, reg:1; } known;
unsigned address;       /* absolute address of block in memory */
unsigned reg;           /* number of register pointing into this block */
unsigned reglc;         /* when reg non-null, location to which it points */
struct label *label;    /* (anonymous) label associated with this block */
```

Defines:
`address`, used in chunks 45a, 49b, 79, 184, 185d, 289, 290, 369b, 385, and 389c.
`label`, used in chunks 39a, 71b, 77a, 158a, 168a, 175, 283b, 308, 360, 369b, 385a, 389b, 393, 394, and 398b.
`reg`, used in chunks 369b and 384a.
`reglc`, used in chunk 384a.
Uses `location`.

369b ⟨*initialize a new relocatable block* `rb` 368b⟩+≡ (384d) ◃368b 376e ▹

```
rb->known.address = rb->known.reg = 0;
rb->label = label_new("unnamed");
rb->label->block = rb;
rb->label->offset = 0;
```

Uses `address` 369a, `label` 369a, `label_new` 390b, and `reg` 369a.

23.1.1 Optimizing sequential emission

Given the representation above, emitting an instruction becomes an elaborate business. We have to fetch the pointer `rb->p`, add a size, and compare the result against `rb->curbuf->limit` before we can store the bytes and increment `p`, and afterward we have to compute the location counter and see if `rb->size` needs to be updated. This is all far too expensive, but we can make it cheaper for the common case of sequential emission to the current relocatable block `currb`. In this special case, we store `p` and `limit` pointers in global variables, and we don't bother to update `rb->size`. `rb->p` and `rb->size` are *dirty* in that they don't reflect the truth; the truth is in global variables.[2]

370a ⟨*exported data declarations* 367⟩+≡ (395) ◁368a 385d▷

```
#define crb() currb
extern RBlock currb;             /* the current relocatable block */
extern unsigned char *currb_p;   /* the true version of currb->p */
extern unsigned char *currb_safe;
             /* the true version of currb->curbuf->limit - MARGIN */
```

Defines:
`crb`, used in chunk 394a.
Uses `currb` 371e, `currb_p` 371e, `currb_safe` 371e, `MARGIN` 370b, `RBlock`, `relocatable` 45a, and `version` 16a.

These variables must satisfy the two invariants:

$$\begin{aligned} \texttt{lc} &= \texttt{currb\_p} - \texttt{currb->curbuf->data} + \texttt{currb->curbuf->datalc} \\ \texttt{currb\_safe} &= \texttt{currb->curbuf->limit} - \texttt{MARGIN} \end{aligned}$$

They must be updated whenever `currb` or `currb->curbuf` changes.

Using a difference of `MARGIN` is a trick that, in most cases, means we needn't add anything to `p` before comparing it. `MARGIN` must be big enough to hold any integral value, and it must match the largest `n` accepted in the `emitl` and `emitb` procedures defined below.

370b ⟨*exported macro definitions* 370b⟩≡ (395) 377b▷

```
#define MARGIN 8
```

Defines:
`MARGIN`, used in chunks 370, 371, 374c, and 382c.

We've made `MARGIN` 8 instead of 4 so we won't have to make changes for 64-bit machines.

[2] We have to export these variables because they're used in exported macros.

When we want the values in `currb` to tell the truth, such as when we are about to change blocks, we `make_clean`.

371a ⟨*definitions* 371a⟩≡ (396) 371b ▷

```
void make_clean() {
  if (currb) {
    unsigned curlc = lc();
    currb->p = currb_p;
    if (currb->size < curlc) currb->size = curlc;
  }
}
```

Defines:
`make_clean`, used in chunks 371b, 372a, 383a, and 386a.
Uses `currb` 371e, `currb_p` 371e, and `lc` 382b.

The other operation that has to make sure the global variables are consistent is changing the current block.

371b ⟨*definitions* 371a⟩+≡ (396) ◁ 371a 371e ▷

```
void set_block(rb) RBlock rb; {
  make_clean();
  currb = rb;
  currb_p = currb->p;
  currb_safe = currb->curbuf->limit - MARGIN;
}
```

Defines:
`set_block`, used in chunk 308.
Uses `currb` 371e, `currb_p` 371e, `currb_safe` 371e, `make_clean` 371a, `MARGIN` 370b, and `RBlock`.

This tricky subtraction of `MARGIN`, which will save us a comparison later on, makes initializing a `firstbuf` a nuisance—we can't set `data` and `limit` to zero, because then the subtraction would make the pointer wrap around, and we would try to fill our whole address space with garbage. Instead we go mega-compulsive, making sure it's OK by ANSI to subtract MARGIN:

371c ⟨*local declarations* 371c⟩≡ (396) 375a ▷

```
static unsigned char bogus_buffer[MARGIN];
```

Uses `MARGIN` 370b.

371d ⟨*set* `data` *and* `limit` *in a new* `rb->firstbuf` 371d⟩≡ (368b)

```
rb->firstbuf.data = rb->firstbuf.limit = bogus_buffer + MARGIN;
```

Uses `MARGIN` 370b.

The rest of this section is boilerplate.

371e ⟨*definitions* 371a⟩+≡ (396) ◁ 371b 372b ▷

```
RBlock currb = (RBlock) 0;
unsigned char *currb_p, *currb_safe;
```

Defines:
`currb`, used in chunks 370–74, 377a, 382, 383, 386a, 389c, and 390a.
`currb_p`, used in chunks 370–74 and 377–83.
`currb_safe`, used in chunks 370–74, 378–80, and 382c.
Uses `RBlock`.

372a ⟨*exported procedure declarations* 372a⟩≡ (395) 393d▹
```
void make_clean(void);
```
Uses `make_clean` 371a.

23.1.2 Emitting data into relocatable blocks

Having shown you our dirt, it seems only fair to dive right into the *raison d'être* for that dirt—the emission optimization. We have emission procedures that work under any circumstances, plus some emission macros that work only when the alignments are just right. We use different strategies for the two—the procedures use a particular byte order, while the macros give you whatever byte order you happen to have on your machine. We begin with the procedures because they're slightly saner.

Emission procedures for known endianness

We have an emission procedure for each endianness. The little-endian procedure is the sanest because the positions of the bytes don't depend on how many there are. This fact lets us play awful games with a switch statement and macros. We number bytes, with byte 0 being the least significant, and we define a macro `BYTE(N)` that emits byte `N` and then falls through to emit bytes $N-1$ down to 0.

372b ⟨*definitions* 371a⟩+≡ (396) ◃371e 373a▹
```
void emitl(val, n) unsigned long val; unsigned n; {
  if (currb_p <= currb_safe || currb_p + n <= currb->curbuf->limit) {
    register unsigned char *p = currb_p;
    ⟨emit n bytes of val at p (little-endian) 372c⟩
    currb_p = p + n;
  } else {
    ⟨slow path for little-endian emission 374a⟩
  }
}
```
Defines:
`emitl`, used in chunks 374a and 378a.
Uses `currb` 371e, `currb_p` 371e, and `currb_safe` 371e.

372c ⟨*emit* `n` *bytes of* `val` *at* `p` *(little-endian)* 372c⟩≡ (372b 375c)
```
switch (n) {
  #define BYTE(N) case N+1: p[N] = val >> 8*N; /* fall through */
  BYTE(7) BYTE(6) BYTE(5) BYTE(4) BYTE(3) BYTE(2) BYTE(1) BYTE(0)
  #undef BYTE
  case 0: /* do nothing */
    break;
  default: assert(("unsigned long bigger than 8 bytes", 0));
}
```

Big-endian machines suck. If we were willing to pay the run-time cost of computing `n-N-1`, which is the position of byte `N` on a big-endian machine, we could use the exact code we've just shown you, with just a slightly different definition of `BYTE`. Unfortunately, we're trying to squeeze every cycle we can out of these procedures, so we insist on doing the arithmetic at compile time (once for each possible value of `n`). We use a macro `S(POS, N)` that stores byte `N` in position `POS`. This makes `emitb` ugly.

373a ⟨*definitions* 371a⟩+≡ (396) ◁372b 375b▷

```
void emitb(val, n) unsigned long val; unsigned n; {
  if (currb_p <= currb_safe || currb_p + n <= currb->curbuf->limit) {
    register unsigned char *p = currb_p;
    ⟨emit n bytes of val at p (big-endian) 373b⟩
    currb_p = p + n;
  } else {
    ⟨slow path for big-endian emission 374b⟩
  }
}
```

Defines:
`emitb`, used in chunks 374b and 378a.
Uses `currb` 371e, `currb_p` 371e, and `currb_safe` 371e.

373b ⟨*emit* `n` *bytes of* `val` *at* `p` *(big-endian)* 373b⟩≡ (373a 376a)

```
switch (n) {
  #define S(POS, N) (p[POS] = val >> 8*N)
  case 0: break;
  case 1: S(0,0); break;
  case 2: S(0,1); S(1,0); break;
  case 3: S(0,2); S(1,1); S(2,0); break;
  case 4: S(0,3); S(1,2); S(2,1); S(3,0); break;
  case 5: S(0,4); S(1,3); S(2,2); S(3,1); S(4,0); break;
  case 6: S(0,5); S(1,4); S(2,3); S(3,2); S(4,1); S(5,0); break;
  case 7: S(0,6); S(1,5); S(2,4); S(3,3); S(4,2); S(5,1); S(6,0); break;
  case 8: S(0,7); S(1,6); S(2,5); S(3,4); S(4,3); S(5,2); S(6,1); S(7,0); break;
  default: assert(("unsigned long bigger than 8 bytes", 0));
}
```

OK, what about the slow paths? We split the value into two pieces: one that will fit in the current buffer, and one that will have to go into the next buffer. Sometimes the next buffer has to be allocated. When computing `roomhere`, the size of the code that will fit in the current buffer, we have to be careful not to subtract pointers where it doesn't make sense. In fact, we're playing fast and loose with the ANSI rules for pointer comparison and arithmetic.

374a ⟨*slow path for little-endian emission* 374a⟩≡ (372b)

```
⟨make sure currb->curbuf->next is non-null, allocating if necessary 377a⟩
{ unsigned roomhere =
        currb_p < currb->curbuf->limit ? currb->curbuf->limit - currb_p : 0;
  if (roomhere > 0) emitl(val, roomhere);
  ⟨move currb->curbuf to the next buffer, updating globals 374c⟩
  emitl(val >> 8*roomhere, n - roomhere);
}
```

Uses `currb` 371e, `currb_p` 371e, and `emitl` 372b.

We may follow this path to great depth if we are walking along a chain of buffers,[3] so it may one day be worth it to eliminate the tail recursion.

The only difference for big-endians is that is is the high bits which go in the current buffer.

374b ⟨*slow path for big-endian emission* 374b⟩≡ (373a)

```
⟨make sure currb->curbuf->next is non-null, allocating if necessary 377a⟩
{ unsigned roomhere =
        currb_p < currb->curbuf->limit ? currb->curbuf->limit - currb_p : 0;
  if (roomhere > 0) emitb(val >> 8*(n-roomhere), roomhere);
  ⟨move currb->curbuf to the next buffer, updating globals 374c⟩
  emitb(val, n - roomhere);
}
```

Uses `currb` 371e, `currb_p` 371e, and `emitb` 373a.

When changing `curbuf`, we must maintain the location-counter invariant on page 370.

374c ⟨*move* `currb->curbuf` *to the next buffer, updating globals* 374c⟩≡ (374)

```
{ unsigned templc = lc();
  currb->curbuf = currb->curbuf->next;
  currb_safe = currb->curbuf->limit - MARGIN;
  currb_p = templc - currb->curbuf->datalc + currb->curbuf->data;
}
```

Uses `currb` 371e, `currb_p` 371e, `currb_safe` 371e, `lc` 382b, `MARGIN` 370b, and `next` 385c.

[3] The durn code ought to be tested to see if that works!

There's no need to make the block *clean* just because we're changing the buffer.

Emission procedures for closures

Applications have to provide emission procedures for closures, but most application writers just want to do the standard thing, which is to emit into our ordinary relocatable blocks, so we had better give them procedures that do so.

The major difference in implementation is that we're not using the "current" block or location counter; instead, we're emitting directly into a known block at a known location. Also, since we're guaranteed that placeholder tokens have already been emitted into these slots, we don't have to worry about maintaining low or high water marks or about allocating buffers. The code that actually spits out the bits (but not the slow path) can be re-used as the chunk ⟨*emit* `n` *bytes of* `val` *at* `p`⟩.

375a ⟨*local declarations* 371c⟩+≡ (396) ◃371c 377c▹

```
static void initial_cl_emitm(RBlock rb, unsigned lc, unsigned long bits, unsigned size);
```

Uses `initial_cl_emitm` 375b 375b, `lc` 382b, and `RBlock`.

375b ⟨*definitions* 371a⟩+≡ (396) ◃373a 375c▹

```
static void initial_cl_emitm(rb, lc, bits, size)
         RBlock rb; unsigned lc; unsigned long bits; unsigned size;
{
  static union { unsigned u; unsigned char c[sizeof(unsigned)]; } u = { 0xaabbccdd };
  switch (u.c[0]) {
    case 0xaa: cl_emitm = &cl_emitb; break;
    case 0xdd: cl_emitm = &cl_emitl; break;
    default:   assert(("byte-order botch", 0));
  }
  (*cl_emitm)(rb, lc, bits, size);
}
void (*cl_emitm)(RBlock rb, unsigned lc, unsigned long, unsigned) = &initial_cl_emitm;
```

Defines:
`initial_cl_emitm`, used in chunk 375a.
Uses `cl_emitb` 376a, `cl_emitl` 375c, `lc` 382b, and `RBlock`.

375c ⟨*definitions* 371a⟩+≡ (396) ◃375b 376a▹

```
void cl_emitl(rb, lc, val, n) RBlock rb; unsigned lc; unsigned long val; unsigned n; {
  struct rblock_buffer *cb;
  register unsigned char *p;
  for (cb = &rb->firstbuf; cb->next && lc >= cb->next->datalc; cb = cb->next);
  p = cb->data + (lc - cb->datalc);
  if (p + n <= cb->limit) {⟨emit n bytes of val at p (little-endian) 372c⟩}
  else {⟨slow path for little-endian closure emission 376b⟩}
}
```

Defines:
`cl_emitl`, used in chunks 375 and 376.
Uses `lc` 382b, `next` 385c, and `RBlock`.

376a ⟨*definitions* 371a⟩+≡ (396) ◁375c 378a▷
```
void cl_emitb(rb, lc, val, n) RBlock rb; unsigned lc; unsigned long val; unsigned n; {
  struct rblock_buffer *cb;
  register unsigned char *p;
  for (cb = &rb->firstbuf; cb->next && lc >= cb->next->datalc; cb = cb->next);
  p = cb->data + (lc - cb->datalc);
  if (p + n <= cb->limit) {⟨emit n bytes of val at p (big-endian) 373b⟩}
  else {⟨slow path for big-endian closure emission 376c⟩}
}
```
Defines:
cl_emitb, used in chunk 375b.
Uses lc 382b, next 385c, and RBlock.

376b ⟨*slow path for little-endian closure emission* 376b⟩≡ (375c)
```
{ unsigned roomhere = cb->limit - p;
  assert(roomhere > 0 && roomhere < n);
  cl_emitl(rb, lc, val, roomhere);
  cl_emitl(rb, lc+roomhere, val >> 8*roomhere, n - roomhere);
}
```
Uses cl_emitl 375c and lc 382b.

376c ⟨*slow path for big-endian closure emission* 376c⟩≡ (376a)
```
{ unsigned roomhere = cb->limit - p;
  assert(roomhere > 0 && roomhere < n);
  cl_emitl(rb, lc, val >> 8*roomhere, roomhere);
  cl_emitl(rb, lc+roomhere, val, n - roomhere);
}
```
Uses cl_emitl 375c and lc 382b.

Allocating new buffers

The only real question here is how big a buffer to allocate. We may use a size hint from the application, but we insist always on being able to get to lc plus 32.[4] The size hint, once used (or if never set), becomes BUFSZ.

376d ⟨*other fields of a relocatable block* 369a⟩+≡ (368a) ◁369a 385c▷
```
int size_hint;
```
Defines:
size_hint, used in chunks 376e, 377a, and 384d.

376e ⟨*initialize a new relocatable block* rb 368b⟩+≡ (384d) ◁369b
```
rb->size_hint = BUFSZ;
```
Uses BUFSZ 396 and size_hint 376d.

[4] Why 32? I have no idea. I'm just guessing. It only matters when this allocation results from a big set_lc.

377a ⟨*make sure* `currb->curbuf->next` *is non-null, allocating if necessary* 377a⟩≡ (374)
```
if (!currb->curbuf->next) {
  int curbufsize = (currb->curbuf->limit - currb->curbuf->data);
  int minsize = 32 + (currb_p - currb->curbuf->data) - curbufsize;
  int size = roundup(max(minsize, currb->size_hint), 8);
  Buffer b = (Buffer) mc_alloc(roundup(sizeof(*b), 8) + size, RBlock_data_pool);

  b->data = (unsigned char *)(b) + roundup(sizeof(*b), 8);
  b->limit = b->data + size;
  b->next = (Buffer)0;
  b->datalc = currb->curbuf->datalc + curbufsize;
  currb->curbuf->next = b;
  currb->size_hint = BUFSZ;
}
```
Uses `Buffer` 367, `BUFSZ` 396, `currb` 371e, `currb_p` 371e, `max` 396, `next` 385c, `roundup` 396, and `size_hint` 376d.

Calculating `curbufsize` separately is conventient, and it makes sure we don't subtract `limit` from `p` when `p` is too small. For efficiency, we allocate once, and we use the resulting memory to hold both the buffer structure and its data area. Also, we round everything in sight to a multiple of 8 just to keep good alignment.

The low water mark is "automagically" set by this code; if we're allocating the first buffer, `currb->firstbuf.datalc` becomes the low water mark by virtue of `currb->firstbuf.next` becoming non-null.

Emission using the byte order of the machine

To emit data using the byte order of the machine, clients should call through the function pointer `emitm`. Clients must exercise care when copying or passing the value of `emitm`, because `emitm` works by a hack: it's initialized to a procedure that dynamically determines the byte order, then resets `emitm` to point either to `emitl` or `emitb`. I believe in avoiding `#ifdef`; it's too easy to get wrong.

This section also defines some shortcuts for emitting data of known size.

377b ⟨*exported macro definitions* 370b⟩+≡ (395) ◁370b 378b▷
```
#define emitul(BITS) emitm((BITS), sizeof(unsigned long))
#define emitu(BITS)  emitm((BITS), sizeof(unsigned))
#define emitus(BITS) emitm((BITS), sizeof(unsigned short))
#define emituc(BITS) emitm((BITS), sizeof(unsigned char))
```
Uses `emitm` 378a.

377c ⟨*local declarations* 371c⟩+≡ (396) ◁375a 384c▷
```
static void initial_emitm(unsigned long bits, unsigned size);
```

378a ⟨*definitions* 371a⟩+≡ (396) ◁376a 382c▷
```
static void initial_emitm(bits, size) unsigned long bits; unsigned size; {
  static union { unsigned u; unsigned char c[sizeof(unsigned)]; } u = { 0xaabbccdd };
  switch (u.c[0]) {
    case 0xaa: emitm = &emitb; break;
    case 0xdd: emitm = &emitl; break;
    default:   assert(("byte-order botch", 0));
  }
  (*emitm)(bits, size);
}
void (*emitm)(unsigned long, unsigned) = &initial_emitm;
```
Defines:
  `emitm`, used in chunks 17a and 377–81.
Uses `emitb` 373a and `emitl` 372b.

Macros for super-fast emission

`emitfastul`, `emitfastu`, `emitfastus`, and `emitfastuc` are all macros for emitting unsigned longs, words, shorts, and characters, respectively, into the current buffer. `emitfast` calls one of these macros or `emitm`, as appropriate. The macros themselves call `emitm` if they can't guarantee sufficient space. Applications calling these macros *must* guarantee that the current location counter has the proper alignment.

The macros differ only in the type of the datum they emit quickly, so we make that a parameter:

378b ⟨*exported macro definitions* 370b⟩+≡ (395) ◁377b 379a▷
```
#define emitfastT0(BITS, TYPE) do { \
  if (currb_p < currb_safe) { \
     *((TYPE*)currb_p) = (TYPE) (BITS); \
      currb_p += sizeof(TYPE); \
  } else (*emitm)((BITS), sizeof(TYPE)); \
} while (0)
#define emitfastul0(BITS) emitfastT0((BITS), unsigned long)
#define emitfastu0(BITS)  emitfastT0((BITS), unsigned)
#define emitfastus0(BITS) emitfastT0((BITS), unsigned short)
#define emitfastuc0(BITS) emitfastT0((BITS), unsigned char)
```
Defines:
  `emitfastu0`, used in chunk 379a.
  `emitfastuc0`, used in chunk 379.
  `emitfastul0`, used in chunk 379a.
  `emitfastus0`, used in chunk 379a.
Uses `currb_p` 371e, `currb_safe` 371e, and `emitm` 378a.

We have several versions of these macros; the trailing `0` indicates this is version 0.

`emitfast` chooses the right macro. We use nested `if-then-else` instead of a `switch` because we think more compilers will eliminate the unreachable code at compile time.

379a ⟨*exported macro definitions* 370b⟩+≡ (395) ◁378b 379b▷
```
#define emitfast0(BITS, SIZE) do { \
  if      ((SIZE) == sizeof(unsigned long))  emitfastul0((BITS)); \
  else if ((SIZE) == sizeof(unsigned))       emitfastu0((BITS)); \
  else if ((SIZE) == sizeof(unsigned short)) emitfastus0((BITS)); \
  else if ((SIZE) == sizeof(unsigned char))  emitfastuc0((BITS)); \
  else emitm((BITS), (SIZE)); \
} while(0)
```
Uses `emitfastu0` 378b, `emitfastuc0` 378b, `emitfastul0` 378b, `emitfastus0` 378b, and `emitm` 378a.

We can't provide much help to the poor CISC people, because they can't guarantee proper alignment, but we can at least give them a fast path for a single byte:

379b ⟨*exported macro definitions* 370b⟩+≡ (395) ◁379a 379c▷
```
#define emitfastbyte0(BITS, SIZE) do { \
  if ((SIZE) == sizeof(unsigned char)) \
    emitfastuc0((BITS)); \
  else \
    emitm((BITS), (SIZE)); \
} while(0)
```
Uses `emitfastuc0` 378b and `emitm` 378a.

Version 1 To speed things up a bit, we use a temporary to hold `currb_p` while we're using it.

379c ⟨*exported macro definitions* 370b⟩+≡ (395) ◁379b 380a▷
```
#define emitfastT1(BITS, TYPE) do { \
  register unsigned char *p = currb_p; \
  if (p < currb_safe) { \
     *((TYPE*)p) = (TYPE) (BITS); \
      currb_p = p + sizeof(TYPE); \
  } else (*emitm)((BITS), sizeof(TYPE)); \
} while (0)
#define emitfastul1(BITS) emitfastT1((BITS), unsigned long)
#define emitfastu1(BITS)  emitfastT1((BITS), unsigned)
#define emitfastus1(BITS) emitfastT1((BITS), unsigned short)
#define emitfastuc1(BITS) emitfastT1((BITS), unsigned char)
```
Defines:
- `emitfastu1`, used in chunk 380a.
- `emitfastuc1`, used in chunk 380.
- `emitfastul1`, used in chunk 380a.
- `emitfastus1`, used in chunk 380a.

Uses `currb_p` 371e, `currb_safe` 371e, and `emitm` 378a.

380a ⟨*exported macro definitions* 370b⟩+≡ (395) ◁379c 380b▷
```
#define emitfast1(BITS, SIZE) do { \
  if      ((SIZE) == sizeof(unsigned long))  emitfastul1((BITS)); \
  else if ((SIZE) == sizeof(unsigned))       emitfastu1((BITS)); \
  else if ((SIZE) == sizeof(unsigned short)) emitfastus1((BITS)); \
  else if ((SIZE) == sizeof(unsigned char))  emitfastuc1((BITS)); \
  else emitm((BITS), (SIZE)); \
} while(0)
```
Uses `emitfastu1` 379c, `emitfastuc1` 379c, `emitfastul1` 379c, `emitfastus1` 379c, and `emitm` 378a.

380b ⟨*exported macro definitions* 370b⟩+≡ (395) ◁380a 380c▷
```
#define emitfastbyte1(BITS, SIZE) do { \
  if ((SIZE) == sizeof(unsigned char)) \
    emitfastuc1((BITS)); \
  else \
    emitm((BITS), (SIZE)); \
} while(0)
```
Uses `emitfastuc1` 379c and `emitm` 378a.

Version 2 Version 2 works around a problem that crops up with `lcc` and other compilers; the many `if` statements generate lots of unreachable code, killing VM performances. `gcc -O` works around this, but both `gcc` and `lcc` generate much better code if we use conditional expressions instead of conditional statements.

We need one trick; we declare the pointer in the outermost macro. That's because the inner macros have to be expressions, because C compilers like `lcc` and `gcc` will optimize away unnecessary branches and labels for conditional expressions but not for conditional statements.

380c ⟨*exported macro definitions* 370b⟩+≡ (395) ◁380b 381a▷
```
#define emitfastT2(BITS, TYPE) ( \
  (EMITFAST_p < currb_safe) \
    ? (*((TYPE*)EMITFAST_p) = (TYPE) (BITS), \
       currb_p = EMITFAST_p + sizeof(TYPE), \
       (void)0) \
    : (*emitm)((BITS), sizeof(TYPE)) \
  )
#define Xemitfastul2(BITS) emitfastT2((BITS), unsigned long)
#define Xemitfastu2(BITS)  emitfastT2((BITS), unsigned)
#define Xemitfastus2(BITS) emitfastT2((BITS), unsigned short)
#define Xemitfastuc2(BITS) emitfastT2((BITS), unsigned char)
```
Defines:
`Xemitfastu2`, used in chunk 381.
`Xemitfastuc2`, used in chunk 381.
`Xemitfastul2`, used in chunk 381.
`Xemitfastus2`, used in chunk 381.

Uses `currb_p` 371e, `currb_safe` 371e, and `emitm` 378a.

381a ⟨*exported macro definitions* 370b⟩+≡ (395) ◁380c 381b▷
```
#define emitfast2(BITS, SIZE) do { \
  register unsigned char *EMITFAST_p = currb_p; \
  ((SIZE) == sizeof(unsigned long))  ? Xemitfastul2((BITS)) : \
  ((SIZE) == sizeof(unsigned))       ? Xemitfastu2((BITS))  : \
  ((SIZE) == sizeof(unsigned short)) ? Xemitfastus2((BITS)) : \
  ((SIZE) == sizeof(unsigned char))  ? Xemitfastuc2((BITS)) : \
  emitm((BITS), (SIZE)); \
} while(0)
```
Defines:
  `emitfast2`, used in chunk 382a.
Uses `currb_p` 371e, `emitm` 378a, `Xemitfastu2` 380c, `Xemitfastuc2` 380c, `Xemitfastul2` 380c, and `Xemitfastus2` 380c.

381b ⟨*exported macro definitions* 370b⟩+≡ (395) ◁381a 381c▷
```
#define emitfastbyte2(BITS, SIZE) do { \
  register unsigned char *EMITFAST_p = currb_p; \
  ((SIZE) == sizeof(unsigned char)) ? emitfastuc2((BITS)) : \
  emitm((BITS), (SIZE)); \
} while(0)
```
Defines:
  `emitfastbyte2`, used in chunk 382a.
Uses `currb_p` 371e, `emitfastuc2` 381c, and `emitm` 378a.

381c ⟨*exported macro definitions* 370b⟩+≡ (395) ◁381b 382a▷
```
#define emitfastul2(BITS) do { \
  register unsigned char *EMITFAST_p = currb_p; Xemitfastul2((BITS)); \
} while(0)
#define emitfastu2(BITS) do { \
  register unsigned char *EMITFAST_p = currb_p; Xemitfastu2((BITS)); \
} while(0)
#define emitfastus2(BITS) do { \
  register unsigned char *EMITFAST_p = currb_p; Xemitfastus2((BITS)); \
} while(0)
#define emitfastuc2(BITS) do { \
  register unsigned char *EMITFAST_p = currb_p; Xemitfastuc2((BITS)); \
} while(0)
```
Defines:
  `emitfastu2`, used in chunk 382a.
  `emitfastuc2`, used in chunks 381b and 382a.
  `emitfastul2`, used in chunk 382a.
  `emitfastus2`, used in chunk 382a.
Uses `currb_p` 371e, `Xemitfastu2` 380c, `Xemitfastuc2` 380c, `Xemitfastul2` 380c, and `Xemitfastus2` 380c.

We use version 2.

382a ⟨*exported macro definitions* 370b⟩+≡ (395) ◁381c 382b▷
```
#define emitfast        emitfast2
#define emitfastbyte    emitfastbyte2
#define emitfastul      emitfastul2
#define emitfastu       emitfastu2
#define emitfastus      emitfastus2
#define emitfastuc      emitfastuc2
```
Uses `emitfast2` 381a, `emitfastbyte2` 381b, `emitfastu2` 381c, `emitfastuc2` 381c, `emitfastul2` 381c, and `emitfastus2` 381c.

23.1.3 Location-counter manipulations

We export the following macros and procedures. Note that we use the global variables, not the dirty fields in `currb`.

382b ⟨*exported macro definitions* 370b⟩+≡ (395) ◁382a 383a▷
```
#define lc() (currb_p - currb->curbuf->data + currb->curbuf->datalc)
#define org(LOC)        set_lc(currb,(LOC))
#define addlc(N) ((void)(currb_p += (N)))
```
Defines:
`addlc`, used in chunk 383.
`lc`, used in chunks 13, 14, 368, 371a, 374–76, 383c, 387–90, 394a, and 398c.
Uses `currb` 371e, `currb_p` 371e, and `set_lc` 382c.

When the location counter changes, we always maintain the invariant on page 370. The invariant doesn't require that `currb_p` point into the current buffer, so we have to update `currb->curbuf` only if the location counter moves back to precede the beginning of the buffer. If it moves forwards, we needn't touch anything, because we will lazily walk to the proper buffer when we try to emit.

382c ⟨*definitions* 371a⟩+≡ (396) ◁378a 383c▷
```
void set_lc(rb, newlc) RBlock rb; unsigned newlc; {
    ⟨if low water mark not set, update first datalc and return 383b⟩
    if (newlc < currb->curbuf->datalc) {
      currb->curbuf = &currb->firstbuf;
      currb_safe = currb->curbuf->limit - MARGIN;
    }
    assert(("above low water", newlc >= currb->curbuf->datalc));
    currb_p = newlc - currb->curbuf->datalc + currb->curbuf->data;
}
```
Defines:
`set_lc`, used in chunk 382b.
Uses `currb` 371e, `currb_p` 371e, `currb_safe` 371e, `MARGIN` 370b, and `RBlock`.

We have to provide access to *all* the fields of any relocatable block, even if it isn't the `crb`. `block_lc` and `block_low` return the location counter and low-water mark of `rb`. If `rb` is `crb`, `make_clean` is called so we don't access any dirty fields.

383a ⟨*exported macro definitions* 370b⟩+≡ (395) ◁382b 384b▷

```
#define block_lc(rb) \
  ((rb) == currb \
      ? (make_clean(),((rb)->p - (rb)->curbuf->data + (rb)->curbuf->datalc)) \
      :               ((rb)->p - (rb)->curbuf->data + (rb)->curbuf->datalc))
#define block_low(rb) \
  ((rb) == currb \
      ? (make_clean(), (rb)->firstbuf.datalc) \
      :                (rb)->firstbuf.datalc)
```

Uses `currb` 371e and `make_clean` 371a.

If the low water mark isn't set, then we can change the location counter and pin `currb_p` to the beginning of the sentinel buffer.

383b ⟨*if low water mark not set, update first* `datalc` *and return* 383b⟩≡ (382c)

```
if (!currb->firstbuf.next) {
  assert(currb->curbuf == &currb->firstbuf);
  currb->firstbuf.datalc = newlc;
  currb_p = currb->curbuf->data; /* might have been advanced by addlc */
  return;
}
```

Uses `addlc` 382b, `currb` 371e, `currb_p` 371e, and `next` 385c.

Aligning always increases the location counter, so we use `addlc`, which is fast.

383c ⟨*definitions* 371a⟩+≡ (396) ◁382c 384a▷

```
void align(n) unsigned n; {
    unsigned oldlc = lc();
    unsigned newlc;
    newlc = (oldlc + n - 1);
    newlc -= newlc % n;
    addlc(newlc-oldlc);
}
```

Uses `addlc` 382b and `lc` 382b.

23.1.4 Pointing registers into relocatable blocks

Wherein we define the following exported functions:

`bind_register(rb, reg, rlc)` binds register `reg` to the relocatable block `rb` at location counter `rlc`. Registers can't be rebound.

384a ⟨*definitions* 371a⟩+≡ (396) ◁383c 384d▷
```
void set_register(rb, reg, rlc) RBlock rb; unsigned reg, rlc; {
    assert(!rb->known.reg);
    rb->reg = reg;
    rb->reglc = rlc;
    rb->known.reg = 1;
}
```
Uses `RBlock`, `reg` 369a, and `reglc` 369a.

23.1.5 Other operations on relocatable blocks

Here's the rest of 'em:

A block is defined if it's non-null and it's not the undefined block.

384b ⟨*exported macro definitions* 370b⟩+≡ (395) ◁383a 385a▷
```
extern struct reloc_block undefrblock;
#define block_defined(rb) ((rb) && (rb) != &undefrblock)
```
Defines:
`block_defined`, used in chunk 390a.
Uses `undefrblock` 384c.

384c ⟨*local declarations* 371c⟩+≡ (396) ◁377c 387b▷
```
struct reloc_block undefrblock;
```
Defines:
`undefrblock`, used in chunks 384b and 390b.

Note that in initializing the undefined block we rely on the fact that the `known` flags default to 0, since they're used in the `location_known` tests.

`block_new` creates a new relocatable block. `size` is a hint to the library as to the maximum size of the block's contents. Rather than allocate a buffer immediately, however, we just save away the size hint. At the cost of a word per block, this makes it cheap to create blocks that are never used. We're doing it because someday somebody might want to create *lots* of them.

384d ⟨*definitions* 371a⟩+≡ (396) ◁384a 385e▷
```
RBlock block_new(size) unsigned size; {
    RBlock rb, tmprb;

    rb = (RBlock) mc_alloc(sizeof(*rb), RBlock_pool);
    ⟨initialize a new relocatable block rb 368b⟩
    rb->size_hint = size > 0 ? size : BUFSZ;
    ⟨count the new block rb and string it on a list 385b⟩
    return rb;
}
```
Defines:
`block_new`, used in chunk 308.
Uses `BUFSZ` 396, `RBlock`, and `size_hint` 376d.

385a ⟨*exported macro definitions* 370b⟩+≡ (395) ◁384b 385f▷
```
#define block_label(rb) ((rb)->label)
```
Defines:
`block_label`, used in chunk 308.
Uses `label` 369a.

For some reason, we have a bunch of instrumentation attached to relocatable blocks. For example, we keep them on a list `defined_rbs` linked by `next` fields. The next few chunks can go if we ditch the instrumentation.

385b ⟨*count the new block* `rb` *and string it on a list* 385b⟩≡ (384d)
```
nblocks++; /* do we really need this instrumentation? */
rb->next = defined_rbs;          /* instrumentation */
defined_rbs = rb;                /* instrumentation */
```
Uses `defined_rbs` 385e, `nblocks` 385e, and `next` 385c.

385c ⟨*other fields of a relocatable block* 369a⟩+≡ (368a) ◁376d
```
struct reloc_block *next;        /* instrumentation */
```
Defines:
`next`, used in chunks 292, 367, 368b, 374–77, 383b, and 385–87.

385d ⟨*exported data declarations* 367⟩+≡ (395) ◁370a 389b▷
```
extern RBlock defined_rbs;       /* instrumentation */
extern int nblocks;              /* instrumentation */
```
Uses `defined_rbs` 385e, `nblocks` 385e, and `RBlock`.

385e ⟨*definitions* 371a⟩+≡ (396) ◁384d 385g▷
```
RBlock defined_rbs = (RBlock) 0;/* instrumentation */
int nblocks = 0;                 /* instrumentation */
```
Defines:
`defined_rbs`, used in chunk 385.
`nblocks`, used in chunk 385.
Uses `RBlock`.

Setting and reading an address is straightforward.

385f ⟨*exported macro definitions* 370b⟩+≡ (395) ◁385a 386a▷
```
#define block_address_known(rb) ((rb)->known.address)
#define block_address(rb)       ((rb)->address)
```
Uses `address` 369a.

385g ⟨*definitions* 371a⟩+≡ (396) ◁385e 386b▷
```
void set_address(rb, address) RBlock rb; unsigned address; {
    assert(rb);
    rb->address = address;
    rb->known.address = 1;
}
```
Defines:
`set_address`, used in chunk 308.
Uses `address` 369a and `RBlock`.

Computing the block size is a pain in the ass, because the block we're looking at just might be dirty.

386a ⟨*exported macro definitions* 370b⟩+≡ (395) ◁385f 389c▷
```
#define block_size(rb) ((rb) == currb ? (make_clean(), (rb)->size) : (rb)->size)
```
Uses `currb` 371e and `make_clean` 371a.

23.1.6 Accessing contents of relocatable blocks

`block_copy` copies `n` bytes of relocatable block `rb` into `dest` beginning at location `start`. `block_write` writes `n` bytes of relocatable block `rb` into file defined by file descriptor `fd` beginning at location `start`. We use a macro `TRANSFER(DEST, SRC, N)` to transfer the data, so we can re-use the body.

386b ⟨*definitions* 371a⟩+≡ (396) ◁385g 386d▷
```
void block_copy(dest, rb, start, n) unsigned char *dest; RBlock rb; unsigned start, n; {
  void *memcpy(void *dest, const void *src, size_t n);
  #define TRANSFER(SRC, N) (memcpy(dest, (SRC), (N)), dest += (N))
  ⟨transfer n bytes from start 386c⟩
  #undef TRANSFER
}
```
Defines:
`block_copy`, used in chunks 387c and 388a.
Uses `RBlock`.

386c ⟨*transfer* `n` *bytes from* `start` 386c⟩≡ (386)
```
{ Buffer buf;
  unsigned char *p;
  for (buf = buffer_at_lc(rb, start), p = buf->data + (start - buf->datalc);
       buf && (n > 0);
       buf = buf->next, p = buf ? buf->data : p
      ) {
    unsigned blk_size = min(buf->limit - p, n);
    TRANSFER(p, blk_size);
    n -= blk_size;
  }
  assert(n == 0);
}
```
Uses `Buffer` 367, `min` 396, and `next` 385c.

386d ⟨*definitions* 371a⟩+≡ (396) ◁386b 387a▷
```
void block_write(fd, rb, start, n) RBlock rb; unsigned start, n; {
  #define TRANSFER(SRC, N) (write(fd, SRC, N))
  ⟨transfer n bytes from start 386c⟩
  #undef TRANSFER
}
```
Uses `RBlock`.

Searching for a buffer is easy. We look either from the current buffer or from the first buffer, depending on whether we've already overshot. If the point we're looking for is on the boundary between two buffers, we return the first buffer, because if this point is at the very end of the contents, there might not be a second buffer. ßChange #6

387a ⟨*definitions* 371a⟩+≡ (396) ◁386d 387c▷

```
static Buffer buffer_at_lc(rb, start) RBlock rb; unsigned start; {
  Buffer b = rb->curbuf;
  if (start < b->datalc)
    b = &rb->firstbuf;
  assert(start >= b->datalc);
  while (b && start > b->datalc + (b->limit - b->data))
    b = b->next;
  assert(b);
  return b;
}
```

Uses `Buffer` 367, `next` 385c, and `RBlock`.

387b ⟨*local declarations* 371c⟩+≡ (396) ◁384c 388d▷

```
static Buffer buffer_at_lc(RBlock rb, unsigned start);
```

Uses `Buffer` 367 and `RBlock`.

I assume an efficient fetch isn't needed, since the only thing I'm using it for so far is to simulate relocation in the style of GNU BFD. For that reason, I implement fetch using `block_copy`.

387c ⟨*definitions* 371a⟩+≡ (396) ◁387a 388a▷

```
unsigned long block_fetchl(rb, lc, n)
    RBlock rb; unsigned lc; unsigned n;
{
  unsigned char bytes[8];
  unsigned long val = 0;
  assert(n <= sizeof(bytes));
  block_copy(bytes, rb, lc, n);
  ⟨convert n bytes of bytes into val (little-endian) 388b⟩
  return val;
}
```

Defines:
  `block_fetchl`, used in chunk 389a.
Uses `block_copy` 386b, `lc` 382b, and `RBlock`.

388a ⟨*definitions* 371a⟩+≡ (396) ◁387c 389a▷
```
unsigned long block_fetchb(rb, lc, n)
    RBlock rb; unsigned lc; unsigned n;
{
  unsigned char bytes[8];
  unsigned long val;
  assert(n <= sizeof(bytes));
  block_copy(bytes, rb, lc, n);
  ⟨convert n bytes of bytes into val (big-endian) 388c⟩
  return val;
}
```
Defines:
`block_fetchb`, used in chunk 389a.
Uses `block_copy` 386b, `lc` 382b, and `RBlock`.

388b ⟨*convert* `n` *bytes of* `bytes` *into* `val` *(little-endian)* 388b⟩≡ (387c)
```
val = 0;
switch (n) {
  #define BYTE(N) case N+1: val |= bytes[N] << 8*N; /* fall through */
  BYTE(7) BYTE(6) BYTE(5) BYTE(4) BYTE(3) BYTE(2) BYTE(1) BYTE(0)
  #undef BYTE
  case 0: /* do nothing */
    break;
  default: assert(("unsigned long bigger than 8 bytes", 0));
}
```

388c ⟨*convert* `n` *bytes of* `bytes` *into* `val` *(big-endian)* 388c⟩≡ (388a)
```
switch (n) {
  #define B(POS, N) (bytes[POS] << 8*N)
  case 0: break;
  case 1: val = B(0,0); break;
  case 2: val = B(0,1) | B(1,0); break;
  case 3: val = B(0,2) | B(1,1) | B(2,0); break;
  case 4: val = B(0,3) | B(1,2) | B(2,1) | B(3,0); break;
  case 5: val = B(0,4) | B(1,3) | B(2,2) | B(3,1) | B(4,0); break;
  case 6: val = B(0,5) | B(1,4) | B(2,3) | B(3,2) | B(4,1) | B(5,0); break;
  case 7: val = B(0,6) | B(1,5) | B(2,4) | B(3,3) | B(4,2) | B(5,1) | B(6,0); break;
  case 8: val = B(0,7) | B(1,6) | B(2,5) | B(3,4) | B(4,3) | B(5,2) | B(6,1) | B(7,0);
          break;
  default: assert(("unsigned long bigger than 8 bytes", 0));
  #undef B
}
```

388d ⟨*local declarations* 371c⟩+≡ (396) ◁387b 392c▷
```
static unsigned long initial_block_fetchm(RBlock rb, unsigned lc, unsigned size);
```
Uses `initial_block_fetchm` 389a 389a, `lc` 382b, and `RBlock`.

389a ⟨*definitions* 371a⟩+≡ (396) ◁388a 390a▷
```
static unsigned long initial_block_fetchm(rb, lc, size)
         RBlock rb; unsigned lc; unsigned size;
{
  static union { unsigned u; unsigned char c[sizeof(unsigned)]; } u = { 0xaabbccdd };
  switch (u.c[0]) {
    case 0xaa: block_fetchm = &block_fetchb; break;
    case 0xdd: block_fetchm = &block_fetchl; break;
    default:   assert(("byte-order botch", 0));
  }
  return (*block_fetchm)(rb, lc, size);
}
unsigned long (*block_fetchm)(RBlock rb, unsigned lc, unsigned) = &initial_block_fetchm;
```
Defines:
`initial_block_fetchm`, used in chunk 388d.
Uses `block_fetchb` 388a, `block_fetchl` 387c, `lc` 382b, and `RBlock`.

23.2 Labels

389b ⟨*exported data declarations* 367⟩+≡ (395) ◁385d 392a▷
```
struct label {
    RBlock block;
    unsigned offset;
    char *name;
};
```
Uses `label` 369a and `RBlock`.

`label_location_known` is true if the given label is located in a relocatable block with a known address. `label_location` returns the absolute address of the label, if it is known. If the address is unknown, `label_location` is undefined. `cur_pc_known` and `cur_pc` are analogous macros for the current program counter `pc`; `pc_location_known` and `pc_location` are simliar but are applied to the location of a closure.

389c ⟨*exported macro definitions* 370b⟩+≡ (395) ◁386a
```
#define label_location_known(lbl)  (((lbl) && (lbl)->block->known.address))
#define label_location(lbl)        ((lbl)->block->address + (lbl)->offset)
#define pc_location_known(loc)     ((loc).dest_block->known.address)
#define pc_location(loc)           ((loc).dest_block->address + (loc).dest_lc)
#define cur_pc_known()             (currb->known.address)
#define cur_pc()                   (currb->address + lc())
```
Defines:
`cur_pc`, used in chunks 315, 328d, and 398a.
`cur_pc_known`, used in chunks 315d and 398a.
`pc_location`, used in chunk 328d.
Uses `address` 369a, `currb` 371e, and `lc` 382b.

`label_define` defines label `lbl` at the current location counter plus `offset` in the current relocatable block. `label_define_at` is a more general operation.

390a ⟨*definitions* 371a⟩+≡ (396) ◁389a 390b▷

```
void label_define(lbl, offset) RLabel lbl; int offset; {
    label_define_at(lbl, currb, lc() + offset);
}
void label_define_at(lbl, block, lc) RLabel lbl; RBlock block; unsigned lc; {
    assert(lbl);
    assert(!block_defined(lbl->block));  /* fail on multiple definition */
    lbl->block = block;
    lbl->offset = lc;
}
```

Uses `block_defined` 384b, `currb` 371e, `fail` 396, `lc` 382b, `RBlock`, and `RLabel`.

`label_new` returns a new label.

390b ⟨*definitions* 371a⟩+≡ (396) ◁390a 393a▷

```
RLabel label_new(name) char *name; {
    RLabel lbl;

    lbl = (RLabel) mc_alloc(sizeof(*lbl), RLabel_pool);
    lbl->block = &undefrblock;
    lbl->name = name;
    return lbl;
}
```

Defines:
  `label_new`, used in chunk 369b.
Uses `RLabel` and `undefrblock` 384c.

23.3 Relocation and closures

Closures are implemented in a two-level structure. `ClosureHeader` *The specification needs to be revised. Meanwhile, there's a two-level structure.* `CHeader` *shows the header of a constructor, which retains a tag and inputs until they are to be used in an instruction.* `RClosure` *is the relocation closure, which delays evaluation until labels (including those nested inside constructor inputs) are known.*

With nested constructors, the simple array scheme won't work any more, and we will need to emit label-locating functions for each constructor type (with callbacks). Similarly for PCs.

391 ⟨*C emitter spec* 391⟩≡ (392a)

```
typedef void (*RelocCallback)(void *closure, RAddr reloc);
typedef void (*RelocEnumerator)(struct rclosure *cp, RelocCallback f, void *closure);

typedef struct instance {
    int tag;         /* need at least 11 bits for Intel */
    union { struct { void *inputs[1]; } consname; } u;
} Instance;
```

Defines:
`Instance`, used in chunks 201d, 329b, and 330a.
`RelocCallback`, used in chunk 201.
`RelocEnumerator`, used in chunk 392a.
Uses `RAddr` 397a.

392a ⟨*exported data declarations* 367⟩+≡ (395) ◁389b 392b▷
```
⟨C emitter spec 391⟩
typedef void (*ApplyMethod)(RClosure cl, Emitter emitter, FailCont fail);

typedef struct closure_header {
    ApplyMethod apply;
    RelocEnumerator relocfn;
    int uses_pc;
    int size_in_bytes;
} *ClosureHeader;       /* always allocated statically */

typedef struct closure_location {
    RBlock dest_block;        /* block written */
    unsigned dest_lc;         /* location within block to write */
} ClosureLocation;

struct rclosure {
    ClosureHeader h;
    ClosureLocation loc;
    /* fields may follow */
};
```
Defines:
`ApplyMethod`, used in chunks 215a and 392b.
`ClosureHeader`, used in chunks 198c, 393d, and 394a.
`ClosureLocation`, used in chunk 198c.
Uses `apply` 266, `Emitter`, `fail` 396, `FailCont`, `location`, `RBlock`, `RClosure`, `RelocEnumerator` 391 391, and `uses_pc` 200c.

In order to be able to export closures to disk, we need to be able to map the closure functions into some other representation. For the moment, we use a bastard form of PostScript.

392b ⟨*exported data declarations* 367⟩+≡ (395) ◁392a
```
typedef struct closurepostfix {
    ApplyMethod apply;
    char *postfix;
} ClosurePostfix;
```
Defines:
`ClosurePostfix`, used in chunk 215a.
Uses `apply` 266 and `ApplyMethod` 392a.

The library provides the closure-independent procedure `apply_closure`. `apply_closure` checks that all relocatable blocks on which the closure is dependent are known before applying the closure-dependent `apply` procedure which performs the relocation.

392c ⟨*local declarations* 371c⟩+≡ (396) ◁388d 394c▷
```
static void fail_if_undef(void *, RAddr);
```
Uses `RAddr` 397a.

393a ⟨*definitions* 371a⟩+≡ (396) ◁390b 393b▷
```
static void fail_if_undef(void *closure, RAddr a) {
    if (!location_known(a))
       ((FailCont)closure)("Label %s undefined or unbound", a->label->name);
}
```
Uses `FailCont`, `label` 369a, `location_known`, and `RAddr` 397a.

393b ⟨*definitions* 371a⟩+≡ (396) ◁393a 394a▷
```
void apply_closure(RClosure cl, Emitter emitter, FailCont fail) {
    (*cl->h->relocfn)(cl, &fail_if_undef, (void*)fail);
    (*cl->h->apply)(cl, emitter, fail);
}
```
Uses `apply` 266, `Emitter`, `fail` 396, `FailCont`, and `RClosure`.

393c ⟨*old definitions* 393c⟩≡
```
void apply_closure(RClosure cl, Emitter emitter, FailCont fail) {
    int n = cl->t.depends_count;
    RLabel *p = &cl->s.depends[0];

    if (cl->t.depends_on_dest && !location_known(cl->t.dest_block->label))
        (*fail)("Destination %s undefined or unbound", cl->t.dest_block->label->name);
    while (n-- > 0)
        if (location_known(*p))
            p++;
        else
            (*fail)("Label %s undefined or unbound",(*p)->name);
    (*cl->t.apply)(cl, emitter, fail);
}
```
Uses `apply` 266, `Emitter`, `fail` 396, `FailCont`, `label` 369a, `location_known`, `RClosure`, and `RLabel`.

To help reduce code bloat in generated encoding procedures, here is a function that allocates and initializes a closure:

393d ⟨*exported procedure declarations* 372a⟩+≡ (395) ◁372a
```
extern RClosure mc_create_closure_here(unsigned size, ClosureHeader h);
```
Uses `ClosureHeader` 392a, `mc_create_closure_here` 394a, and `RClosure`.

394a ⟨*definitions* 371a⟩+≡ (396) ◁393b 394b▷
```
RClosure mc_create_closure_here(unsigned size, ClosureHeader h) {
  RBlock   this_block = crb();
  unsigned this_lc    = lc();
  RClosure c = (RClosure) mc_alloc_closure(size, this_block, this_lc);
  c->h = h;
  c->loc.dest_block = this_block;
  c->loc.dest_lc    = this_lc;
  return c;
}
```
Defines:
`mc_create_closure_here`, used in chunks 196c and 393d.
Uses `ClosureHeader` 392a, `crb` 370a, `lc` 382b, `mc_alloc_closure`, `RBlock`, and `RClosure`.

23.4 Support for printing relocatable addresses

394b ⟨*definitions* 371a⟩+≡ (396) ◁394a 394d▷
```
static void label_print(struct label *label) {
  asmprintf(asmprintfd, "%s", label->name ? label->name : "???");
}
```
Uses `asmprintf` 394d, `asmprintfd` 394d, and `label` 369a.

394c ⟨*local declarations* 371c⟩+≡ (396) ◁392c
```
static void label_print(struct label *label);
```
Uses `label` 369a.

394d ⟨*definitions* 371a⟩+≡ (396) ◁394b
```
typedef void (*asmprinter)(void *closure, const char *fmt, ...);
extern int fprintf(FILE *stream, const char *fmt, ...); /* needed at Purdue -- ugh */
void (*asmprintf)(void *closure, const char *fmt, ...) = (asmprinter) fprintf;
void *asmprintfd = (void *)stdout;
void (*asmprintreloc)(RAddr reloc) = reloc_print; /* calls asmprintf */
```
Defines:
`asmprintf`, used in chunks 236, 308, and 394b.
`asmprintfd`, used in chunks 236, 239b, 308, and 394b.
`asmprintreloc`, used in chunks 235 and 308.
Uses `RAddr` 397a.

23.5 Boilerplate

395 ⟨*mclib.h* 395⟩≡

```
#ifndef INCLUDE_MCLIBH
#define INCLUDE_MCLIBH

#ifndef assert
#include <assert.h>
#endif

/* this group is available for users */

⟨library types (never defined)⟩
⟨library procedures (never defined)⟩
⟨declarations of procedures provided by the application (never defined)⟩

/* this group either supports macros or is needed by emission procedures */

⟨exported data declarations 367⟩
⟨exported macro definitions 370b⟩
⟨exported procedure declarations 372a⟩

#endif
```

Note the **roundup** macro is good only for powers of two.

396 ⟨*mclib.c* 396⟩≡

```
#include <mclib.h>
#include <stdio.h>

#define max(X,Y) ((X) > (Y) ? (X) : (Y))
#define min(X,Y) ((X) < (Y) ? (X) : (Y))
#define roundup(X,N) (((X)+((N)-1))&(~((N)-1)))


static void mcfail(char *fmt, ...) {
  assert(("application didn't set fail", 0));  /* apps should provide their own */
}
void (*fail)(char *fmt, ...) = &mcfail;

#define BUFSZ 1024

⟨local declarations 371c⟩
⟨definitions 371a⟩
```

Defines:
BUFSZ, used in chunks 376e, 377a, and 384d.
fail, used in chunks 27, 69a, 75–77, 83, 84, 91, 97c, 114–17, 124, 125a, 127–30, 135c, 138b, 151a, 168a, 172, 173, 176, 180a, 187, 189c, 194b, 199, 202, 216, 232a, 250b, 251c, 254b, 260b, 261, 273b, 308, 318a, 330b, 336, 337, 340a, 349a, 351c, 390a, 392, and 393.
max, used in chunks 14, 59, 229b, 268a, 344, and 377a.
min, used in chunks 344 and 386c.
roundup, used in chunk 377a.

Chapter 24

Relocatable Addresses

Encoding applications often want to work with addresses whose values aren't known. Common examples include calls to unknown procedures, or forward branches within a procedures. This kind of work can be done with labels alone, but it is more convenient to use a more general abstraction: the *relocatable address*. In its most general form, a relocatable address is an expression whose value depends on one or more labels.[1] Most of the toolkit treats relocatable addresses simply as an abstract data type:

397a ⟨*library types* 397a⟩≡

```
typedef struct relocatable_address *RAddr;
```

Defines:
`RAddr`, used in chunks 195a, 308, 391–94, 397, and 398.

The operations available on a relocatable address are to see whether it has a known value, and to extract that value:

397b ⟨*faux declarations of library macros* 397b⟩≡ 398a ▷

```
extern int location_known(RAddr addr);
extern unsigned location (RAddr addr);
```

Uses `location`, `location_known`, and `RAddr` 397a.

`location_known(addr)` will be true whenever all the labels on which `addr` depends have known values. If an application writer wants to change our representation of relocatable addresses, his only obligation is to provide `location_known` and `location`.

[1]Note that our definition is different from the more traditional definition, in which, for example, the difference of two labels is taken to be an absolute address, not a reloctable address. This traditional definition works only if the two labels are in the same compilation unit.

It is often convenient to refer to the "program counter" (pc), i.e., the address of the current location in the current relocatable block, as if it were a relocatable address. In our current scheme, it is not representable as a relocatable address, but we provide a similar interface through macros:

398a ⟨*faux declarations of library macros* 397b⟩+≡ ◁397b

```
extern int cur_pc_known(void);
extern unsigned cur_pc(void);
```

Uses cur_pc 389c and cur_pc_known 389c.

cur_pc returns pc's value if cur_pc_known is true.

The representation we have chosen for relocatable addresses does not support the fully general form defined above, because we would need a little language for representing and interpreting addressing expressions, and we despair of devising a language whose definition and implementation would be acceptable to all application writers. Instead, the toolkit provides relocatable addresses of the form $L + n$, where L is a label and n is an integer offset. A relocatable address ra therefore has two fields:

```
a.label
ra.offset
```

and one can be created by calling the following procedure:

398b ⟨*library procedures* 398b⟩≡ 398c▷

```
extern RAddr addr_new(RLabel label, int offset);
```

Uses addr_new, label 369a, RAddr 397a, and RLabel.

You can also convert an unsigned integer to a relocatable address:

398c ⟨*library procedures* 398b⟩+≡ ◁398b

```
extern RAddr unsigned_to_raddr(unsigned lc);
```

Uses lc 382b, RAddr 397a, and unsigned_to_raddr.

Part III

Back matter

24.1 List of chunks

24.2 Index