New Jersey Machine-Code Toolkit
Source Code

Norman Ramsey and Mary F. Fernandez

April 4, 2001

Contents

Part I

Icon code

Chapter 1

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.

7a ⟨*gdb-binoutput* 7a⟩≡

```
disas main _asmoutput
```

Uses `main`.

7b ⟨*gdb-asmoutput* 7b⟩≡

```
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.

8 ⟨*chkrfilter* 8⟩≡

```
#!/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` and `next`.

1.1 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.

9 ⟨*rules to build check address* 9⟩≡

```
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`.

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.

1.2 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.

10 ⟨* 10⟩≡ 11▷

```
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 11 and 13a.

TESTLIMIT, used in chunk 19.
zerotab, used in chunk 17a.
Uses cdebug, crhs, expimage, generate_checker 11, generate_input_values 15, implementation_extension, indirectname, interface_extension, kept_constructors, openfile, parmtab, pattern_to_case, subst_tab, and verbose.

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.

11 ⟨* 10⟩+≡ ◁10 12▷

```
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 10.
Uses chkr_input_tbl 10, Gcall, generate_tests 12, indirectname, instructionctype, kept_constructors, line, openfile, and pretty.

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.

12 ⟨* 10⟩+≡ ◁11 13a▷

```
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 16e⟩
  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 11 and 13a.
Uses cdebug, expimage, literal, and tests_of 13a.

13a ⟨* 10⟩+≡ ◁12 14▷

```
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 13b⟩
        suspend tbl[input.name]
      }
      "string" : {
        ⟨check that test values exist for input 13b⟩
        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 13b⟩
        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 12.
Uses chkr_input_tbl 10, constype, field, Gcall, generate_tests 12, input, kept_constructors, literal, lookuptype, typeerror, and unsigned_to_raddr.

13b ⟨*check that test values exist for input* 13b⟩≡ (13a)

```
inputs := \tbl[input.name] | \
error("No test values specified for `", input.name, "'")
```

Uses input.

merge_chkr_tables merges two input tables and complains if test values have already been supplied for an input.

14 ⟨* 10⟩+≡ ◃13a 15▹

```
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`.

1.3 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.

15 ⟨*10⟩+≡ ◁14 16c▷

```
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 16a⟩
        ⟨create initial set of test_values 20c⟩
        ⟨generate input values 19⟩
        put(test_sets, test_values)
      }
    }
  }
  return test_sets
end
```

Defines:
`generate_input_values`, used in chunk 10.

Uses `Epc`, `kept_constructors`, `pattern`, `remove_duplicate_conditions`, and `Stagcase`.

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`.

16a ⟨*extract* `d`*'s conditions* 16a⟩≡ (15)
```
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 16b⟩
⟨add constraints on unconstrained inputs 17a⟩
⟨add constraints on field inputs 17b⟩
⟨omit constraints and conditions on constructor-typed operands 17c⟩
⟨print fits and conds 16d⟩
```
Uses `Epatlabel_to_Epc`, `extract_conditions`, `gsubst`, `sanitize_sequents`, and `set_patlabel_offsets`.

Relocatable addresses are always "forceable", because the checker determines their values.

16b ⟨*eliminate* `Eforce` *on relocatable inputs* 16b⟩≡ (16a)
```
every f := !fits do f.x := unforce(f.x)
```
Uses `unforce` 16c.

16c ⟨* 10⟩+≡ ◁15 20a▷
```
procedure unforce(x)
  return if (type(x) == "Eforce") then x.x else x
end
```
Defines:
`unforce`, used in chunks 16b and 20a.
Uses `Eforce`.

16d ⟨*print* `fits` *and* `conds` 16d⟩≡ (16a)
```
if (*fits > 0) then write(\cdebug, "Width constraints for " || cons.name)
write(\cdebug, "   " || expimage(fits))
write(\cdebug, print_conditions(conds))
```
Uses `cdebug`, `constraints`, `expimage`, and `print_conditions` 21c.

16e ⟨*print* `test_values` 16e⟩≡ (12)
```
every k := key(test_values) do
  write(\cdebug, "test_values[", expimage(k), "] = ", expimage(test_values[k]))
```
Uses `cdebug` and `expimage`.

17a ⟨*add constraints on unconstrained inputs* 17a⟩≡ (16a)
```
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`, `Efitsu`, `Emod`, `eqn`, `inputs_of`, `wordsize`, and `zerotab` 10.

We insert constraints on field operands into `fits` so we can generate test values for them.

17b ⟨*add constraints on field inputs* 17b⟩≡ (16a)
```
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`, `Efitsu`, `field`, `fwidth`, and `inputs_of`.

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.

17c ⟨*omit constraints and conditions on constructor-typed operands* 17c⟩≡ (16a)
```
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` and `subterms_matching`.

18a ⟨*insert equality conditions in* `test_values` 18a⟩≡ (20c)
```
every c := !conds  do {
  if ((⟨equality condition 18b⟩) then {
    verbose("inserting test_values[" || expimage(x) || "] = " || v)
    test_values[x] := v
  }
}
```
Uses `expimage` and `verbose`.

18b ⟨*equality condition* 18b⟩≡ (18a)
```
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` and `untableexp`.

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.

18c ⟨*sparsely populated* 18c⟩≡ (19)
```
names := \fieldname_table(i.meaning)
```
Uses `fieldname_table`.

19 ⟨*generate input values* 19⟩≡ (15)

```
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 18c⟩) 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 21b⟩
      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 21b⟩
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`, `constype`, `disjunct`, `expimage`, `field`, `inputs_of`, `interleave_slices` 20b, print_conditions 21c, random_nbit_value 22a, slices_of 20a, TESTLIMIT 10, and verbose.

slices_of projects conditions dependent on slices of v.

20a ⟨* 10⟩+≡ ◁16c 20b▷

```
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 19.
Uses eqn, Eslice, unforce 16c, and untableexp.

Now try to interleave the slices, generating random bits for the unspecified parts. No checking for overlapping slices! The solver should do that.

20b ⟨* 10⟩+≡ ◁20a 21c▷

```
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 19.
Uses mask.

20c ⟨*create initial set of* test_values 20c⟩≡ (15)

```
every i := inputs_of(cons, "string") &
      i.meaning == "reloc" & test_values[i.name] := "reloc"
n_tests := 0
⟨insert equality conditions in test_values 18a⟩
while (*test_values > n_tests) do {
  n_tests := *test_values
  ⟨simplify conditions 21a⟩
  ⟨insert equality conditions in test_values 18a⟩
}
```

Uses inputs_of.

21a ⟨*simplify conditions* 21a⟩≡ (20c)
```
new_conds := set()
every insert(new_conds, (simplify_conditions(conds, test_values))[2])
conds := new_conds
```
Uses `simplify_conditions` 22b.

Simplifying conditions can produce satisfied conditions, unsatisfied conditions, or conditions with unknown satisfiability.

21b ⟨*check conditions* 21b⟩≡ (19)
```
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`, `reduceable` 23, `simplify_conditions` 22b, and `tautology` 23.

21c ⟨* 10⟩+≡ ◁20b 22a▷
```
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 16d and 19.
Uses `expimage`.

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.

22a ⟨* 10⟩+≡ ◃21c 22b▹

```
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 19.
Uses constraint, Efitss, Efitsu, expimage, and field.

simplify_conditions simplifies each condition in a set of conditions and then suspends a list pair which contains the original and new condition.

22b ⟨* 10⟩+≡ ◃22a 23▹

```
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 21.
Uses expimage, simplify, and subst_tab.

An equation is reduceable if it contains a variable expression.

23 ⟨* 10⟩+≡ ◁22b

```
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 21b.
  tautology, used in chunk 21b.
Uses untableexp.

24 ⟨*checker.t* 24⟩≡

```
#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;
asmprintfd = stdout;
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(".globl _main\n");
printf("reloc:\n");
printf("main:\n");
printf("_main:\n");
```

Uses addr_new, asmprintf, asmprintfd, asmprintreloc, block_label, block_new, fail, label, location, location_known, main, mc_alloc_closure, Mc_alloc_pool, RAddr, RBlock, RClosure, set_address, and set_block.

25a ⟨*inchecker.t* 25a⟩≡

```
printf(".text\n");
printf(".globl _asmoutput\n");
printf("_asmoutput:\n");
```

25b ⟨*afterchecker.t* 25b⟩≡

```
printf(".text\n");
printf(".globl _endoutput\n");
printf("_endoutput:\n");
exit(0);
```

Part II

Library code

Part III

Back matter

1.4 List of chunks

1.5 Index