The implementation of `mlcc` and `mld`: A Machine-Independent Compiler and A Retargetable, Optimizing Linker

Mary Fernandez

May 8, 1995

Contents

1 Introduction **4**

2 `mlcc`: A C to `mill` compiler **5**
- 2.1 Program initialization and termination . . . 5
- 2.2 Generating `mill` object code . . . 9
- 2.3 Building flow graphs . . . 14
- 2.4 Building call graphs . . . 16
- 2.5 Data representation . . . 17
- 2.6 Symbol representation . . . 19
- 2.7 Type representation . . . 23
- 2.8 Target-dependent code . . . 29
- 2.9 Miscellaneous support . . . 32
- 2.10 Common data structures . . . 34

3 Emitting `mill` object files **43**
- 3.1 Modules . . . 43
- 3.2 Call Graphs . . . 46
- 3.3 Data . . . 47
- 3.4 Flow Graphs . . . 48
- 3.5 Procedures . . . 54
- 3.6 Symbols . . . 57
- 3.7 Types . . . 62
- 3.8 Miscellaneous . . . 64
- 3.9 Pickling and unpickling `mill` instructions . . . 67

4 Reading `mill` object files **79**
- 4.1 Modules . . . 79
- 4.2 Call Graphs . . . 80
- 4.3 Data . . . 81
- 4.4 Flow Graphs . . . 81
- 4.5 Procedures . . . 82
- 4.6 Symbols . . . 83
- 4.7 Types . . . 86
- 4.8 Miscellaneous . . . 89

5 Simplifying expressions in `mlcc` and `mld` 92
5.1 Simplifying `Tree` and `Node` expressions . . . 92
5.2 Simplification rules . . . 97
5.3 Constant folding . . . 109

6 `mld`: A `mill` linker and optimizer 118
6.1 Module and procedure reachability . . . 120
6.2 Identifying external symbols . . . 122
6.3 Data Layout . . . 123
6.4 Optimizations . . . 128
6.5 Code Generation . . . 129
6.6 Command-line options . . . 132
6.7 Tracing, timing, and instrumentation . . . 137
6.8 `mldpp`: A link-time preprocessor . . . 139
6.9 `mranlib`: An indexer for `mill` archives . . . 141
6.10 Boilerplates for `mld`, `mldpp`, and `mranlib` . . . 144
6.11 Resolving external symbols . . . 147
6.12 Converting packed `mill` to unpacked `mill` code . . . 152
6.13 `mld`'s interface to code generator and emitter . . . 159

7 Link-time binding files 171
7.1 Parsing a binding file . . . 173
7.2 Using binding values . . . 180

8 Inlining 183
8.1 Identifying reachable procedures . . . 183
8.2 Inlining heuristics . . . 186
8.3 Inlining . . . 187
8.4 Transforming inlined code . . . 200
8.5 Miscellaneous procedures . . . 203

9 Specialization 206

10 Data-flow analysis 211
10.1 Definitions and uses . . . 211
10.1.1 Defs . . . 217
10.1.2 Uses . . . 220
10.2 Managing `def` and `use` sets . . . 223
10.3 Computing `gen` and `kill` sets . . . 225
10.4 Computing the fixed point . . . 229
10.5 Ud-chains and du-chains . . . 231
10.6 Reaching definitions . . . 235
10.7 Dominators . . . 235

11 Constant propagation 237
11.1 Functions on `Node`s . . . 248
11.2 Miscellaneous . . . 251

12 Analysis and optimization of Modula–3 programs 254
12.1 Generating binding files for Modula–3 programs . . . 254
12.2 Optimization analysis for Modula-3 programs . . . 268
12.2.1 Type analysis . . . 268
12.2.2 Identifying and converting method calls . . . 273

13 Machine-independent binary emission **278**
13.1 Segments . . . 281
13.2 Assembly Code . . . 282
13.3 Support for link-time initialization . . . 284
13.4 Support for New Jersey Machine-Code Toolkit . . . 284
13.5 Segments for SPARC and Intel486 . . . 286

14 MIPS binary emission procedures **288**
14.1 Code generation specification . . . 288
14.2 Symbols and registers . . . 291
14.3 Segments . . . 293
14.4 Synthesized instructions . . . 295
14.5 Assembly and startup code . . . 297
14.6 Executable format . . . 301

15 SPARC binary emission procedures **304**
15.1 Code generation specification . . . 304
15.2 Segments . . . 308
15.3 Assembly and startup code . . . 309
15.4 Synthesized instructions . . . 316
15.5 Executable format . . . 316

16 Intel486 binary emission procedures **319**
16.1 Code generation specification . . . 319
16.2 Assembly and startup code . . . 329
16.3 Executable format . . . 331

17 File and archive management **333**
17.1 Buffer management . . . 344

18 Internal data structures and support procedures **348**
18.1 Memory management . . . 348
18.2 `mill` Tables (`MTables`) . . . 351
18.3 Sets . . . 355
18.3.1 Caching results of set operations . . . 359
18.3.2 Temporary sets . . . 361
18.4 Boilerplate . . . 362
18.5 String management . . . 363
18.5.1 Emitting strings in `mill` code . . . 368
18.6 List management . . . 370

List of chunks **372**

References **381**

1 Introduction

This document describes and includes the implementation of `mlcc`, a machine-independent compiler, and `mld`, a retargetable, optimizing linker. `mld` is designed to reduce the run-time overhead of Modula–3's features that support module encapsulation and objects by delaying optimization until link time. `mld` links and transforms `mill`, a machine-independent intermediate code that is generated by `mlcc`. The motivation for, application of, and effectiveness of link-time optimization are described in a technical paper (Fernández 1995b) and thesis (Fernández 1995a). This document details the design and implementation of `mlcc` and `mld`, and of `mldpp` and `mranlib`, two tools that support use of `mld`. `mldpp` is a link-time pre-processor that produces a binding file for a Modula–3 program and `mranlib` produces a table of contents for an archive of `mill` object files.

The programs' source appears in named code "chunks". Code chunks contain the programs' source code and references to other code chunks. Chunk names appear italicized within angle brackets, e.g., ⟨*propagate constant to use 42*⟩ and contain the page number of the chunk's first definition. When two or more chunks appear on the same page, a letter is appended to the page number, e.g. 42a and 42b. Documentation is interleaved between code chunks. The source and documentation are extracted and written into files for compilation or for formatting using the `noweb` literate programming system (Ramsey 1994).

2 mlcc: A C to mill compiler

`mlcc` includes `lcc`'s front end and a new code generator for an abstract machine that emits `mill` code instead of target-dependent object code.

This module contains the `mill` code generator for `mlcc`. It implements `lcc`'s code-generation interface (Fraser and Hanson 1995).

2.1 Program initialization and termination

`progbeg` is called by `mlcc`'s front end to initialize the code generator. Tables for the global symbols, data segment, procedure index, and types are created and initialized. Flags recognized by the backend are processed. The appropriate `mill` files (binary and possibly ASCII) are opened and the module header is emitted.

`mlcc` recognizes the following options:

`-ma`*file*	Emit an ASCII `mill` code in *file*.
`-mb`*file*	Emit binary `mill` code in *file*, overriding default file.
`-mx`*proc*	Exclude procedure *proc* from reachability analysis at link time.
`-msum`	Emit summary of compilation statistics.
`-m3`	Recognize Modula–3 file names.

5a ⟨*local declarations* 5a⟩≡
```
dclproto(static void progbeg,(int, char **));
dclproto(static void progend,(void));
static BaseTable baseTbl;
static Symbol    indirproc;        /* generic symbol for indirect procedure calls */
static int       amillfd, sumflag = 0;
static char      *m3file = 0;      /* M3 file name */
```
This definition is continued in chunks 5b, 9a, 14–19, 22e, 23b, 26a, 32c, and 33b.
This code is used in chunk 33d.

`excludedsym` is true if the current global symbol is excluded from the reachability search.

5b ⟨*local declarations* 5a⟩+≡
```
static int       excludedsym = 0;
static MTable    excludedsyms;
```

6 ⟨*definitions* 6⟩≡

```
static void progbeg(argc, argv) char *argv[]; {
    DataItem d;
    int i, j;
    char *bname;
    char *amillfile = 0, *bmillfile = 0, *tmp, m3tmp[64];
    static Type proto[2];

    bzero((char *)&baseTbl, BASETBLSZ);

    proto[0] = voidtype; proto[1] = (Type)0;
    /* Init data segment area */
    newtable(&excludedsyms, 8, sizeof(char *), &permanent);
    newtable(&baseTbl.datatab, 1024, sizeof(DataItem), &permanent);
    newtable(&baseTbl.dataGlobs, 0, SHORTSZ, &permanent);
    newtable(&baseTbl.procindex, PROCIDXTABSZ, PROCIDXSZ, &permanent);
    newtable(&baseTbl.glbsymtab, GLBSYMTABSZ, sizeof(Symbol), &permanent);

    ⟨ install basic types, symbols and constants 7a⟩

    indirproc = install("_indirproc", &externals, GLOBAL, PERM);
    indirproc->sclass = EXTERN;
    indirproc->type = func(inttype, proto, 0);
    newglobalsym(indirproc);
    amillfd = 0;

    if (firstfile) bname = basename(firstfile);
    else fatal("mlcc", "Empty source file\n", 0);
    ⟨process command-line arguments 7b⟩

    if (amillfile) {
        if (strcmp(amillfile, "-") == 0) amillfd = dup(2);
        else amillfd = open(amillfile, O_WRONLY|O_CREAT);
        if (amillfd == -1)
            error("mlcc: Cannot open ASCII MILL file %s\n", amillfile);
        else chmod(amillfile, 00644);
    }
    if (!bmillfile)    {
        baseTbl.fd = outfd;
        lseek(1, 0, SEEK_SET);
    }
    else if ((baseTbl.fd = open(bmillfile, O_WRONLY|O_CREAT)) == -1)
        fatal("mlcc:", "Cannot open binary MILL file\n", 0);
    if (bmillfile) chmod(bmillfile, 00644);

    /* Emit Module Header; Return base address */
    bemitModuleHdr(&baseTbl, firstfile);
    baseTbl.pbt.procsReg.base = tell(baseTbl.fd);

    /* Procedures are emitted by bemitProcedure() */
}
```

This definition is continued in chunks 7–10, 15–28, and 33.
This code is used in chunk 33d.

7a ⟨ *install basic types, symbols and constants* 7a⟩≡

```
newtable(&baseTbl.typetab, TYPETABSZ, sizeof(Type), &permanent);
for (i = 1; i <= NBASICTYPES; i++)
    typesym(*basictypes[i]);
newglobalsym(intconst(0));
for (i = 1, j = 2; j <= NCONSTANTS; i <<= 1, j++)
    newglobalsym(intconst(i));
```

This code is used in chunk 6.

7b ⟨*process command-line arguments* 7b⟩≡

```
for (i = 1; i < argc; i++) {
    if (strncmp(argv[i], "-ma", 3) == 0) {
        if (strlen(argv[i]) > 3)
            amillfile = string(&(argv[i][3]));
        else amillfile = concat(bname, ".amill");
    }
    else if (strncmp(argv[i], "-mb", 3) == 0) {
        if (strlen(argv[i]) > 3)
            bmillfile = string(&(argv[i][3]));
    }
    else if (strncmp(argv[i], "-msum", 4) == 0) sumflag = 1;
    else if (strncmp(argv[i], "-mx", 3) == 0) {
        addtotable(char **, &excludedsyms, string(&(argv[i][3])));
    }
    else if (strncmp(argv[i], "-m3", 3) == 0) {
        tmp = strstr(firstfile, "_m.c");
        if (tmp) {
            strcpy(m3tmp, firstfile);
            m3tmp[tmp - firstfile] = '\0';
            m3file = string(m3tmp);
        }
    }
    else error("mlcc: Argument %s ignored\n", argv[i]);
}
```

This code is used in chunk 6.

7c ⟨*definitions* 6⟩+≡

```
void memory_usage() {
    fprint(2,"memory_usage not implemented\n");
}
```

progend emits **mill** code for all of the procedures, emits summary information, and closes any open **mill** files.

8a ⟨*definitions* 6⟩+≡

```
static void progend() {
    Symbol s;
    int i,j;
    ProcIndex *pi;

    typedef_uses();
    /* Emit all the procedures and clean up */
    gentable(baseTbl.procindex, i, 0) {
        pi = &(tableentry(ProcIndex *, baseTbl.procindex, i));
        assert(s = idxtable(Symbol *, baseTbl.glbsymtab, symid(pi->id)));
        emitProcedure(amillfd, s);
        bemitProcedure(&baseTbl, s, pi);
        mlcc_deallocate(&(mfunc(s)->arena));
    }

    baseTbl.pbt.procsReg.size = tell(baseTbl.fd) - baseTbl.pbt.procsReg.base;
    gentable(baseTbl.glbsymtab, i, 1) {
        s = idxtable(Symbol *, baseTbl.glbsymtab, i);
        fixupTypes(s->type);        /* <== this is a kludge */

        /* If a function is not defined locally, it must be an EXTERN */
        if (s->type && isfunc(s->type) &&
            isemptytable(mfn(s).flowgraph) && s->sclass != STATIC)
            s->sclass = EXTERN;
    }
    baseTbl.pbt.minlabel = (unsigned short)genlabel(0);
    for (i = 0; i < 6; i++)
        baseTbl.pbt.segmentsizes[i] = roundup(baseTbl.pbt.segmentsizes[i], 16);
    bemitProgram(amillfd, &baseTbl);
    if (sumflag) {
        emitSummary(2, &baseTbl);
        resusage(2);
    }
    if (baseTbl.fd)
        close(baseTbl.fd);
    if (amillfd)
        close(amillfd);
}
```

8b ⟨*definitions* 6⟩+≡

```
static void resusage(fd) {
  struct rusage r;
  getrusage(RUSAGE_SELF, &r);
  fprint(2,"User %d,%d\n", r.ru_utime.tv_sec, r.ru_utime.tv_usec);
  fprint(2,"Sys %d,%d\n", r.ru_stime.tv_sec, r.ru_stime.tv_usec);
  fprint(2,"PF %d\n", r.ru_majflt);          /* page faults */
  fprint(2,"IN %d\n", r.ru_inblock);         /* block input operations */
  fprint(2,"OUT %d\n", r.ru_oublock);         /* block output operations */
  fprint(2,"RESSET %d\n", r.ru_maxrss); /*  resident set */

}
```

2.2 Generating mill object code

`function` is called by `mlcc`'s front end to generate code for the function represented by the symbol `f`. Each function has several `mill` tables (`MTables`) associated with it; these tables are created and initialized in `function`. `function` then calls `gencode` to generate `mill` code from the function's forest of dags. After code generation, the function is assigned a unique index in the module's procedure table and the table is updated.

Unfortunately, we need some globals to maintain context across calls from/to `mlcc`'s frontend. `cfun` is the current function, `cfgn` is the current flow graph node in `cfun`. `centryn` is the last visited entry node in `cfun`. `cfgnid` and `cdagid` are the unique ids of the `cfgn` and the root of the current dag being processed.

9a ⟨*local declarations* 5a⟩+≡

```
dclproto(static void function,(Symbol, Symbol [], Symbol [], int));
Symbol cfun = (Symbol)0;    /* current function */
static FGNode  cfgn;        /* current flow graph node */
static FGNode  centryn;     /* current entry node */
static int     cfgnid = 1;  /* current flow graph node id */
static int     cdagid = 1;  /* current root dag id */
```

9b ⟨*definitions* 6⟩+≡

```
static void function(f, caller, callee, ncalls) Symbol f, caller[], callee[]; {
    int i, arg;
    Env env;
    ProcIndex pi;

    cfun = f;
    newglobalsym(f);
    mfn(cfun).sym = f;
    mfn(cfun).bt = &baseTbl;

    cfgnid = 1;
    cdagid = 1;
    if (f->name == string("main"))
        mfunc(f)->ncalls = 1;
    mfunc(f)->leaf = 1;        /* assume its a leaf function */
    ⟨ initialize f's tables  10a⟩
    cfgn = newfgnode(UnknownN, (FGNode)0);
    blockbeg(&env);
    for (i = 0; caller[i] && callee[i]; i++) {
        arg = newsym(caller[i], &(mfn(cfun).localsymtab), centryn, cfgn);
        updatetable(unsigned short *, (&mfn(f).caller), i, arg);
        arg = newsym(callee[i], &(mfn(cfun).localsymtab), centryn, cfgn);
        updatetable(unsigned short *, (&mfn(f).callee), i, arg);
    }
    gencode(caller, callee);
    blockend(&env);

    ⟨ add f to procedure index  10b⟩
    cfgn = centryn = 0;
}
```

10a ⟨ *initialize* `f` *'s tables* 10a⟩≡
```
newtable(&(mfn(f).flowgraph), FGTABSZ, sizeof(FGNode), &(mfunc(cfun)->arena));
newtable(&(mfn(f).localsymtab), 8, sizeof(Symbol), &(mfunc(cfun)->arena));
newtable(&(mfunc(f)->callgraph), 0, CALLGRNODESZ, &permanent);
newtable(&(mfn(f).dagtable), DAGTABSZ, sizeof(unsigned char), &(mfunc(cfun)->arena));
newtable(&(mfn(f).caller), CALLTABSZ, SHORTSZ, &(mfunc(cfun)->arena));
newtable(&(mfn(f).callee), CALLTABSZ, SHORTSZ, &(mfunc(cfun)->arena));
```
This code is used in chunk 9b.

10b ⟨ *add* `f` *to procedure index* 10b⟩≡
```
pi.id = msym(f).id;
pi.stridx = msym(f).stridx;
addtotable(ProcIndex *, &(baseTbl.procindex), pi);
```
This code is used in chunk 9b.

`gen` is called by `gencode` to generate `mill` code for a forest of trees represented by `p`. Concurrent with `mill` code generation, the procedure's flow graph and the module's call graph are constructed.

10c ⟨*definitions* 6⟩+≡
```
Node gen(p) Node p; {
    int last;
    CallGraphNode cgn;
    Symbol gsym, argsym, temp;
    Node target, lastp, q, savp, newn;
    int low, argnid = 0, asgncall = 0, retsymid = 0;

    ⟨preprocess codelist 11a⟩
    for (; p; p = p->link) {
        low = cdagid;
        last = (p->link == 0);
        ⟨generate mill code for p 11c⟩
        ⟨update flow graph 12⟩
    }
    ⟨postprocess codelist 11b⟩
    return p;
}
```

The codelist is preprocessed so that `mlcc` can associate `CALL` nodes with their related `ARG` nodes. `ARG` nodes can be isolated from their `CALL` nodes by intervening assignments to their children. Any such assignment is reassigned to a temporary and the temporary replaces the current argument.

11a ⟨*preprocess codelist* 11a⟩≡

```
savp = p;
for (lastp = 0; p; lastp = p, p = p->link) {
  switch(p->op){
  case ARGI: case ARGD: case ARGF:
    if (generic(p->kids[0]->op) == INDIR && isaddrop(p->kids[0]->kids[0]->op)) {
      argsym = p->kids[0]->kids[0]->syms[0];
      for(q = p->link; q && generic(q->op) != CALL &&
        !(generic(q->op) == ASGN && generic(q->kids[1]->op) == CALL);
        q = q->link) {
        if (generic(q->op) == ASGN && isaddrop(q->kids[0]->op) &&
          q->kids[0]->syms[0] == argsym) {
          /* assign ARG value to a temp; update ARG's kid node */
          temp = newtemp(AUTO, optype(p->op));
          temp->temporary = 0;
          local(temp);
          newn = newnode(ASGN+optype(p->op), newnode(ADDRL+P, 0, 0, temp), p->kids[0], 0);
          newn->syms[0] = intconst(temp->type->size);
          newn->syms[1] = intconst(temp->type->align);
          if (lastp) lastp->link = newn;
          else savp = newn;
          newn->link = p;
          p->kids[0] = newnode(INDIR+optype(p->kids[0]->op),
                               newnode(ADDRL+P, 0, 0, temp), 0, 0);
          break;
        }
      }
    }
  default: break;
  }
}
p = savp;
```

This code is used in chunk 10c.

After the codelist has been traversed, it is postprocessed. A new basic block is created so that dags are announced to the backend on the same codelist as they were emitted by the front-end.

11b ⟨*postprocess codelist* 11b⟩≡

```
assert(cfgn);
if (cfgn->type != UnknownN)
    cfgn = newfgnode(UnknownN, cfgn);
```

This code is used in chunk 10c.

11c ⟨*generate* `mill` *code for* `p` 11c⟩≡

```
cdagid = genmillcode(p, &(mfn(cfun)), cdagid, centryn, cfgn);
```

This code is used in chunk 10c.

The type of the current node determines if the flow graph must be updated.

12 ⟨*update flow graph* 12⟩≡

```
switch(generic(p->op)){
case ARG:
    ⟨create an ArgN flow graph node 13a⟩
    if (last) assert(0);   /* ARG nodes must be on list w/ CALL node */
    break;
case ASGN:
    if (generic(p->kids[1]->op) != CALL) {
        if (cfgn->type == UnknownN) cfgn->type = StmtsN;
        adddagrange((&cfgn->dagnodes), low, cdagid - 1);
        cfgn->codelist = last;
        break;
    }
    /* fall through */
    retsymid = msym(p->kids[0]->syms[0]).id;
    asgncall = 1;
case CALL:
    if (asgncall) q = p->kids[1];
    else q = p;
    mfunc(cfun)->leaf = 0;  /* Not a leaf function */

    ⟨create a CallN (procedure call) flow graph node 13b⟩
    ⟨insert call in function's call graph 17b⟩

    adddagrange((&cfgn->dagnodes), low, cdagid - 1);
    cfgn->codelist = last;
    callN(cfgn).retsymid = retsymid;

    ⟨add new label following CALL 14c⟩
    cfgn = newfgnode(UnknownN, cfgn);
    asgncall = 0;
    break;
case RET:
    ⟨create a RetN (procedure return) flow graph node 14a⟩
    break;
case LABEL:
    ⟨create a StmtsN (straight-line) flow graph node 14b⟩
    break;
case EQ: case GE: case GT: case LE: case LT: case NE:
    ⟨create a TestN (conditional branch) flow graph node 13c⟩
    break;
case JUMP:
    ⟨create a JumpN (unconditional branch) flow graph node 13d⟩
    break;
default:
    if (cfgn->type == UnknownN) cfgn->type = StmtsN;
    adddagrange((&cfgn->dagnodes), low, cdagid - 1);
    cfgn->codelist = last;
    break;
}
```

This code is used in chunk 10c.

When a new flow graph node is created, it is labelled with the appropriate type and its `dagnodes` table is updated with the range of dags contained in the node.

Arguments are strung together by `ArgN` nodes. `argnid` is the id of the first node in the string.

13a ⟨*create an ArgN flow graph node* 13a⟩≡
```
if (p->op == ARGP && isaddrop(p->kids[0]->op))
    msym(p->kids[0]->syms[0]).downarg = 1;
if (cfgn->type == UnknownN || cfgn->type == ArgN)
    cfgn->type = ArgN;
else cfgn = newfgnode(ArgN, cfgn);
if (!argnid) argnid = cfgn->id;
adddagrange((&cfgn->dagnodes), low, cdagid - 1);
```

This code is used in chunk 12.

The `argnid` field in a `CallN` node is the id of the first flow graph node that computes an argument for the call.

13b ⟨*create a CallN (procedure call) flow graph node* 13b⟩≡
```
if (cfgn->type == UnknownN || cfgn->type == ArgN)
    cfgn->type = CallN;
else cfgn = newfgnode(CallN, cfgn);
callN(cfgn).argnid = argnid ? argnid : cfgn->id;
argnid = 0;
callN(cfgn).dagn = p->x.dagid;
```

This code is used in chunk 12.

Conditional and unconditional branches create new flow graph nodes. The target label is tagged as the target of the given node.

13c ⟨*create a TestN (conditional branch) flow graph node* 13c⟩≡
```
if (cfgn->type == UnknownN || cfgn->type == StmtsN)
    cfgn->type = TestN;
else cfgn = newfgnode(TestN, cfgn);
marklabel(p->syms[0], cfgn);
adddagrange((&cfgn->dagnodes), low, cdagid - 1);
cfgn->codelist = last;
cfgn = newfgnode(UnknownN, cfgn);
```

This code is used in chunk 12.

13d ⟨*create a JumpN (unconditional branch) flow graph node* 13d⟩≡
```
if (cfgn->type == UnknownN || cfgn->type == StmtsN)
    cfgn->type = JumpN;
else if (cfgn->type != MultitestN)
    cfgn = newfgnode(JumpN, cfgn);
if (generic(p->kids[0]->op) == ADDRG)
    marklabel(p->kids[0]->syms[0], cfgn);
adddagrange((&cfgn->dagnodes), low, cdagid - 1);
cfgn->codelist = last;
cfgn = newfgnode(UnknownN, (FGNode)0);
```

This code is used in chunk 12.

14a ⟨*create a RetN (procedure return) flow graph node* 14a⟩≡
```
if (cfgn->type != UnknownN && cfgn->type != StmtsN)
    cfgn = newfgnode(UnknownN, cfgn);
cfgn->type = RetN;
adddagrange((&cfgn->dagnodes), low, cdagid - 1);
cfgn->codelist = last;
cfgn = newfgnode(UnknownN, cfgn);
```

This code is used in chunk 12.

14b ⟨*create a StmtsN (straight-line) flow graph node* 14b⟩≡
```
if (cfgn->type == UnknownN) cfgn->type = StmtsN;
else cfgn = newfgnode(StmtsN, cfgn);
cfgn->labelsym = msym(p->syms[0]).id;
backpatch(p->syms[0], cfgn);
cfgn->codelist = last;
```

This code is used in chunk 12.

A label is inserted after each CALL node for use by `mld`; the label can be used as a return target for a procedure inlined at the call site.

14c ⟨*add new label following CALL* 14c⟩≡
```
target = p->link;
if (last || generic(target->op) != LABEL) { /* add target label for CALL */
    p->link = newnode(LABEL+V, 0, 0, findlabel(genlabel(1)));
    p->link->link = target;
    target = p->link;
}
```
This code is used in chunk 12.

2.3 Building flow graphs

A procedure's flow graph is computed concurrently with the generation of `mill` code. `newfgnode` creates a new flowgraph node and labels the current flow graph node (`cfgn`) with its id and type. If `pred` is non-null, it is a predecessor of the new node. A change in scope (indicated by calls to `blockbeg` and `blockend`) is considered a change in flow control and therefore adds new nodes to the flow graph. `linknodes` links a successor to a predecessor in the flowgraph. `multitest` creates a flow graph node of type `MultitestN`, which represents a switch statement.

14d ⟨*local declarations* 5a⟩+≡
```
dclproto(static FGNode newfgnode, (unsigned char, FGNode));
dclproto(static void linknodes, (FGNode, FGNode));
```

15a ⟨*definitions 6*⟩+≡

```
static FGNode newfgnode(unsigned char type, FGNode pred) {
    FGNode nd;

    nd = (FGNode)yyalloc(sizeof(struct fgnode), (mfunc(cfun)->arena));
    bzero((char *)nd, sizeof *nd);
    nd->id = cfgnid++;
    nd->type = type;
    newtable(&(nd->succs), 2, SHORTSZ, &(mfunc(cfun)->arena));
    newtable(&(nd->preds), 2, SHORTSZ, &(mfunc(cfun)->arena));
    linknodes(nd, pred);
    newtable(&(nd->dagnodes), LOCALDAGTABSZ, RANGESZ, &(mfunc(cfun)->arena));
    updatetable(FGNode *, (&mfn(cfun).flowgraph), nd->id, nd);
    newtable(&(nd->tempsymids), 0, SHORTSZ, &(mfunc(cfun)->arena));
    newtable(&(nd->localsymids), 0, SHORTSZ, &(mfunc(cfun)->arena));
    return nd;
}
static void linknodes(succ, pred) FGNode succ, pred; {
    if (pred && succ) {
        addtotable(short *, (&pred->succs), srelIndex(pred->id, succ->id));
        addtotable(short *, (&succ->preds), srelIndex(succ->id, pred->id));
    }
}
```

blockbeg/**blockend** begins/ends a compound statement and creates a new flow graph node.

15b ⟨*definitions 6*⟩+≡

```
void blockbeg(e) Env *e; {
    e->entry = centryn;
    if (cfgn->type == UnknownN)
        cfgn->type = EntryN;
    else
        cfgn = newfgnode(EntryN, cfgn);
    centryn = cfgn;
}
void blockend(e) Env *e; {
    if (cfgn->type == UnknownN)
        cfgn->type = ExitN;
    else
        cfgn = newfgnode(ExitN, cfgn);
    entryN(centryn).exitNid = cfgn->id;
    exitN(cfgn).entryNid = centryn->id;
    centryn = e->entry;
}
```

multitest creates a flow graph node that represents a switch statement.

15c ⟨*local declarations 5a*⟩+≡

```
dclproto(extern Symbol equated,(Symbol));
```

16a ⟨*definitions* 6⟩+≡
```
void multitest(tab, s, lab, size, values, lbls) Symbol tab, s, lab, *lbls; int *values; {
    int i;
    if (cfgn->type == UnknownN || cfgn->type == StmtsN)
        cfgn->type = MultitestN;
    else
        cfgn = newfgnode(MultitestN, cfgn);
    cfgn->codelist = 1;
    cfgn->labelsym = newsym(tab, &(mfn(cfun).localsymtab), centryn, cfgn);
    multitestN(cfgn).sym = newsym(s, &(mfn(cfun).localsymtab), centryn, cfgn);
    multitestN(cfgn).deflab =
        newsym(equated(lab), &(mfn(cfun).localsymtab), centryn, cfgn);
    multitestN(cfgn).size = size;
    newtable(&multitestN(cfgn).labels, size+1, SHORTSZ, &permanent);
    newtable(&multitestN(cfgn).values, size+1, LONGSZ, &permanent);
    for (i = 0; i < size; i++) {
        marklabel(lbls[i], cfgn);
        updatetable(unsigned short *, (&multitestN(cfgn).labels), i,
            newsym(equated(lbls[i]), &(mfn(cfun).localsymtab), centryn, cfgn));
        updatetable(int *, (&multitestN(cfgn).values), i, values[i]);
    }
}
```

`backpatch` associates the current node with the given label and backpatches all those nodes for which the label is the node's target. `marklabel` links the node with the given label as the target of its predecessor; if the label is not associated with a flow graph node, the label is tagged with the predecessor for later update by `backpatch`.

16b ⟨*local declarations* 5a⟩+≡
```
dclproto(static void backpatch, (Symbol, FGNode));
dclproto(static void marklabel, (Symbol, FGNode));
```

16c ⟨*definitions* 6⟩+≡
```
static void backpatch(s, nd) Symbol s; FGNode nd; {
    List l;

    if (!mlabel(s).fgnode) {
        mlabel(s).fgnode = nd;
        for (l = mlabel(s).targetof; l; l = l->link)
            linknodes(nd, (FGNode)l->x);
    }  else
        error("mlcc: Flow graph node already associated with label %s", s->name);
}

static void marklabel(s, pred) Symbol s; FGNode pred; {
    linknodes(mlabel(s).fgnode, pred);
    mlabel(s).targetof = push(pred, mlabel(s).targetof);
}
```

2.4 Building call graphs

`insertingraph` inserts a call graph node in the given call graph. The calls are inserted in order sorted by the function's unique identifier.

16d ⟨*local declarations* 5a⟩+≡
```
dclproto(static void insertingraph, (MTable *, CallGraphNode));
```

17a ⟨*definitions* 6⟩+≡
```
static void insertingraph(callgraph, cgn) MTable *callgraph; CallGraphNode cgn; {
    int i, j;
    CallGraphNode tmp;

    gentable((*callgraph), i, 0) {
        tmp = tableentry(CallGraphNode *, (*callgraph), i);
        if (cgn.fnid < tmp.fnid)
            break;
    }    /* insert at position i */
    for (j = tablesize((*callgraph)); j > i; j--) {
        updatetable(CallGraphNode *, callgraph, j,
                    tableentry(CallGraphNode *, (*callgraph), j-1));
    }
    updatetable(CallGraphNode *, callgraph, i, cgn);
}
```

If a procedure call is direct, the callee's symbol id is inserted in the call graph. Otherwise, the default symbol `indirproc` is inserted.

17b ⟨*insert call in function's call graph* 17b⟩≡
```
if (generic(q->kids[0]->op) == ADDRG) {
    gsym = tableentry(Symbol *, baseTbl.glbsymtab, symid(msym(q->kids[0]->syms[0]).id));
    mfunc(gsym)->ncalls += 1;
    cgn.ref = q->kids[0]->syms[0]->ref;
} else {
    gsym = indirproc;
    cgn.ref = 0;
}
cgn.fnid = msym(gsym).id;
cgn.fgnid = cfgn->id;
insertingraph((&mfunc(cfun)->callgraph), cgn);
```

This code is used in chunk 12.

2.5 Data representation

The values of initialized variables are emitted in abstract data segments. Each initialized variable is tagged with its abstract segment. The data is emitted into the `mill` file's data region, which contains *data type*, *value* pairs. The data types are those supported by `lcc`: `C, I, U, S, D, F`. Two new types, `X` and `Y`, represent symbol addresses and strings, respectively. If a variable is initialized, it is tagged with the index of its value in the `mill` data segment.

`segment` sets the current abstract segment to `seg`. `space` allocates empty space in the current segment.

17c ⟨*local declarations* 5a⟩+≡
```
dclproto(static void segment,(int));
dclproto(static void space,(int));
static int     cseg;         /* current segment */
```

18a ⟨*definitions* 6⟩+≡
```
static void segment(seg) {
    cseg = seg;
}
static void space(n) {
    DataItem d;

    d.ty = Z;
    d.x.u = n;
    baseTbl.pbt.segmentsizes[cseg] += n;
    addtotable(DataItem *, (&baseTbl.datatab), d);
}
```

`defaddress` emits an address, `defconst` emits a constant, and `defstring` emits a string in the current data segment. The total size of the current segment is incremented by the size of the object emitted.

18b ⟨*local declarations* 5a⟩+≡
```
dclproto(static void defaddress,(Symbol));
dclproto(static void defconst,(int, Value));
dclproto(static void defstring,(int, char *));
```

18c ⟨*definitions* 6⟩+≡
```
static void defaddress(p) Symbol p; {
    DataItem d;

    assert(p);
    newsym(p, 0, centryn, cfgn);

    d.ty = X;
    d.x.us = msym(p).id;
    if (!excludedsym || (p->type && !isfunc(p->type)))
        msym(p).dataseg = 1;
    baseTbl.pbt.segmentsizes[cseg] += sizeof(d.x.p);
    addtotable(DataItem *, (&baseTbl.datatab), d);
}
```

19a ⟨*definitions* 6⟩+≡
```
static void defconst(ty, v) Value v; {
    DataItem d;
    int    sz;

    d.ty = ty;
    switch(ty) {
        case C: d.x.uc = v.uc ; sz = sizeof(v.uc); break;
        case I:
        case U: d.x.u = v.u;    sz = sizeof(v.u);  break;
        case S: d.x.us = v.us;  sz = sizeof(v.us); break;
        case F: d.x.f = v.f;    sz = sizeof(v.f);  break;
        case D: d.x.u = ((unsigned *)(&v.d))[0];
            addtotable(DataItem *, (&baseTbl.datatab), d);
            d.x.u = ((unsigned *)(&v.d))[1];
            sz = sizeof(v.d);
            break;
        case P: d.x.p = v.p;    sz = sizeof(v.p);  break;
        default: fatal("mlcc", "defconst: unknown type\n", 0);
            break;
    }
    baseTbl.pbt.segmentsizes[cseg] += sz;
    addtotable(DataItem *, (&baseTbl.datatab), d);
}
```

19b ⟨*definitions* 6⟩+≡
```
static void defstring(len, s) char *s; {
    unsigned short    idx;
    DataItem d;

    d.ty = Y;
    d.x.us = stridx(s, len);
    baseTbl.pbt.segmentsizes[cseg] += len;
    addtotable(DataItem *, (&baseTbl.datatab), d);
}
```

2.6 Symbol representation

When a symbol is seen by `mlcc`'s backend for the first time, it is assigned a unique identifier and added to either the global or the current local symbol table, depending upon the symbol's scope and storage class.

`local` adds a local symbol to the current local symbol table. `global` adds a global symbol to the global symbol table and allocates space for the symbol in the appropriate data segment.

19c ⟨*local declarations* 5a⟩+≡
```
dclproto(static void global,(Symbol));
dclproto(static int newglobalsym,(Symbol));
dclproto(static void local,(Symbol));
dclproto(extern int newsym,(Symbol, MTable *, FGNode, FGNode));
static int glbsymid = 1;
```

20a ⟨*definitions* 6⟩+≡
```
static void global(p) Symbol p; {
    int i;

    ⟨increment current data segment size 20b⟩
    newglobalsym(p);
    mglob(p).dataloc = tablesize(baseTbl.datatab);
    addtotable(unsigned short *, (&baseTbl.dataGlobs), msym(p).id);
    ⟨check if p is excluded from reachability search 20c⟩
}
```

20b ⟨*increment current data segment size* 20b⟩≡
```
if (p->type->align > 4)
    baseTbl.pbt.segmentsizes[cseg] = roundup(baseTbl.pbt.segmentsizes[cseg],4);
else
    baseTbl.pbt.segmentsizes[cseg] =
        roundup(baseTbl.pbt.segmentsizes[cseg],p->type->align);
```
This code is used in chunk 20a.

20c ⟨*check if* `p` *is excluded from reachability search* 20c⟩≡
```
excludedsym = 0;
gentable(excludedsyms, i, 0) {
    if (p->name == tableentry(char **, excludedsyms, i)) {
        excludedsym = 1;
        break;
    }
}
```
This code is used in chunk 20a.

newglobalsym inserts a global symbol in the global symbol table and selects a single symbol if the front end refers to more than one global symbol with the same name. If the symbol is a new function, it initializes a new arena for the function.

21 ⟨*definitions* 6⟩+≡

```
static int newglobalsym(p) Symbol p; {
    Symbol gsym;
    int i;
    Arena ar;

    if (msym(p).id != 0) return msym(p).id;
            /* Resolve references to global symbols */
    if (!msym(p).typesym && (p->sclass == EXTERN || p->scope < LOCAL)
        && p->name && !num(p->name)) {
        if (p->type && isfunc(p->type)) {
            ar = newarena();
            msym(p).u.func = (MFuncSym)yyalloc(MFNSYMSZ, permanent);
            BZERO(msym(p).u.func, *(msym(p).u.func));
            mfunc(p)->arena = ar;
        }
        gentable(baseTbl.glbsymtab, i, 1) {
            gsym = tableentry(Symbol *, baseTbl.glbsymtab, i);
            if (!msym(gsym).typesym && p->name == gsym->name) {
                msym(p).id = msym(gsym).id;
                if (p->sclass != EXTERN) {
                    mergesymbols(p, tableentry(Symbol *, baseTbl.glbsymtab, i));
                    tableentry(Symbol *, baseTbl.glbsymtab, i) = p;
                }
            }
            if (msym(p).id) break;    /* Found a match */
        }
    }
    if (!msym(p).id) {
        msym(p).id = GLOBSYM(glbsymid);
        updatetable(Symbol *, (&baseTbl.glbsymtab), glbsymid, p);
        glbsymid++;
    }
    ⟨add p's name to string table 22c⟩
    ⟨add p's type to type table 22d⟩
    return msym(p).id;
}
```

`newsym` assigns and returns a symbol's unique id and updates the local or global symbol table. Local symbols must be associated with a flow graph node so `mld` knows when to announce the symbol to the target-dependent backends. `newsym` associates local temporaries to the current flow graph node; bonafide locals are associated with the current entry node.

22a ⟨*definitions* 6⟩+≡
```
static void local(p) Symbol p; {
    newsym(p, &(mfn(cfun).localsymtab), centryn, cfgn);
}
int newsym(p, localsymtab, entryn, fgn)
    Symbol p; MTable *localsymtab; FGNode entryn, fgn; {

    assert(p);
    if (msym(p).id != 0) return msym(p).id;
    if (⟨p belongs in global symbol table 22b⟩)
        newglobalsym(p);
    else {
        if (p->scope == PARAM && !p->computed)
            msym(p).id = newlocalsym(p, localsymtab, 0);
        else if (p->temporary == 1 && p->sclass == REGISTER) {
            msym(p).id = newlocalsym(p, localsymtab, &fgn->tempsymids);
            if (p->u.t.cse != NULL) mtemp(p).cse = 1;
        }
        else {
            assert(entryn);
            msym(p).id = newlocalsym(p, localsymtab, &entryn->localsymids);
        }
        ⟨add p's name to string table 22c⟩
        ⟨add p's type to type table 22d⟩
    }
    return msym(p).id;
}
```

22b ⟨`p` *belongs in global symbol table* 22b⟩≡
```
(msym(p).typesym || p->sclass == TYPEDEF ||
p->scope == CONSTANTS || p->scope == GLOBAL ||
(p->scope != LABELS && (p->sclass == STATIC || p->sclass == EXTERN)))
```

This code is used in chunk 22a.

22c ⟨*add* `p`*'s name to string table* 22c⟩≡
```
if (p->name) msym(p).stridx = stridx(p->name, strlen(p->name));
```
This code is used in chunks 21 and 22a.

22d ⟨*add* `p`*'s type to type table* 22d⟩≡
```
typesym(p->type);
```

This code is used in chunks 21 and 22a.

`address` initializes `q` to the addressing expression `p+n`.

22e ⟨*local declarations* 5a⟩+≡
```
dclproto(static void address,(Symbol, Symbol, int));
```

23a ⟨*definitions* 6⟩+≡
```
static void address(q, p, n) Symbol q, p; {
    newsym(p, cfun ? &(mfn(cfun).localsymtab) : 0, centryn, cfgn);
    q->name = stringf("%s%s%d", p->name, n > 0 ? "+" : "", n);
    newsym(q, cfun ? &(mfn(cfun).localsymtab) : 0, centryn, cfgn);
    maddr(q).offset = n;
    maddr(q).symid = msym(p).id;
}
```

2.7 Type representation

Types are emitted into a `mill` file's type region. `lcc`'s default types are not emitted because they can be reconstructed by `mld` at link time. User-defined types are represented by walking the abstract syntax tree of the type's constructor and emitting the tree in a bottom-up traversal.

`typesym` assigns a unique id to a type and updates the type table. For non-simple types, `flattentypesym` flattens the type recursively by traversing the type's constructor. After the entire source file is processed, `fixuptypes` checks that all types used in the file have been identified and converted to `mill` code.

23b ⟨*local declarations* 5a⟩+≡
```
dclproto(static void typesym, (Type));
dclproto(static void flattentypesym,(Symbol));
dclproto(static void fixupTypes, (Type));
static int     typeid = 1;
```

24a ⟨*definitions* 6⟩+≡

```
static void typesym(ty) Type ty; {
    int    i;
    Type    *protos;
    Symbol    *ids;
    Field    f;

    if (!ty || ty->x.typeno != 0) return;
    if (isfunc(ty)) {
        ty->x.nprotos = 0;
        if (ty->u.f.proto)
            for (protos = ty->u.f.proto; *protos; protos++, ty->x.nprotos++)
                typesym(*protos);
    }
    else if (ty->u.sym) {
        if ((isstruct(ty) || isunion(ty)) && ty->u.sym->u.s.flist != 0){
           ⟨ flatten fields of structs and unions 24b⟩
        }  else if (isenum(ty) && ty->u.sym->u.idlist != 0) {
           ⟨ flatten enum 24c⟩
        }
        if ((isstruct(ty) || isunion(ty) || isenum(ty)) &&
            (*ty->u.sym->name >= '1' && *ty->u.sym->name <= '9')) {
            Symbol p = findtype(ty);
            flattentypesym(p);
        }
        flattentypesym(ty->u.sym);
    }
    typesym(ty->type);
    ty->x.typeno = typeid++;
    updatetable(Type *, (&baseTbl.typetab), ty->x.typeno, ty);
}
```

24b ⟨ *flatten fields of* `struct`*s and* `union`*s* 24b⟩≡

```
msym(ty->u.sym).field = 1;
for (f = ty->u.sym->u.s.flist; f; f = f->link) {
    Symbol p = findtype(ty);
    flattentypesym(p);
}
```

This code is used in chunk 24a.

24c ⟨ *flatten* `enum` 24c⟩≡

```
msym(ty->u.sym).enumsym = 1;
i = 0;
for (ids = ty->u.sym->u.idlist; *ids; ids++, i++) {
    /* count number of enum identifiers */
}
menum(ty->u.sym) = i;
```

This code is used in chunk 24a.

25a ⟨*definitions* 6⟩+≡
```
static void flattentypesym(p) Symbol p; {
    Type ty;
    if (p) {
        msym(p).typesym = 1;     /* Type symbols are STATIC */
        ty = p->type;
        p->type = 0;
        newglobalsym(p);
        p->type = ty;
    }
}
```

25b ⟨*definitions* 6⟩+≡
```
static void fixupTypes(t) Type t;{
    Type *tt;
    Field f;
    Type ty;

    if (t && !t->x.fixed) {
        t->x.fixed = 1;
        switch(t->op) {
        case CHAR: case CONST: case DOUBLE: case FLOAT:
        case INT: case LONG: case SHORT: case UNSIGNED: case VOID:
        case VOLATILE:
            typesym(t);
            break;
        case POINTER: case ARRAY: case ENUM:
            typesym(t);
            fixupTypes(t->type);
            break;
        case FUNCTION:
            typesym(t);
            if (isfunc(t) && t->u.f.proto)
                for (tt = t->u.f.proto; *tt; tt++)
                    fixupTypes(*tt);
            break;
        case STRUCT: case UNION:
            typesym(t);
            assert(t->u.sym);
            for (f = t->u.sym->u.s.flist; f; f = f->link)
                fixupTypes(f->type);
            break;
        default: printtype(t, 2); error("mlcc: Unknown type ");
            break;
        }
    }
}
```

For each declaration of a global or local symbol, `typedef_uses` determines if the variable's declaration refers to a typedef. If it does, the symbol's auxilliary type is assigned the typedef. `findtypdef` finds the use of a typedef closest to the `x` coordinate of the symbol's declaration in the source file.

For Modula–3 programs, a variable's auxilliary type encodes the type descriptor of the variable's type. `mld` needs this type information so it can associate the type of a C variable with its original type in the Modula–3 source.

26a ⟨*local declarations* 5a⟩+≡

```
dclproto(static void adduses,(Symbol, Generic));
dclproto(static void typedef_uses,(void));
dclproto(static Symbol findtypedef,(MTable *, Coordinate));
typedef struct {
    Symbol sym;
    char *file;
    unsigned x;
} TypeDef;
#define TYPEDEFSZ sizeof(TypeDef)
```

`adduses` adds a use of a typedef to the `usetable`.

26b ⟨*definitions* 6⟩+≡

```
static void adduses(p, usetable) Symbol p; Generic usetable; {
    MTable *uses = (MTable *)usetable, *useentry;
    List uselist, head;
    Coordinate *use;
    TypeDef t;
    MTable *newtab;

    if (p->sclass == TYPEDEF) {
        for (head = uselist = p->uses; uselist; uselist = uselist->link) {
            use = (Coordinate *)uselist->x;
            if (use->y >= tablesize(*uses) ||
                tableentry(MTable **, *uses, use->y) == 0) {
                newtab = (MTable *)talloc(MTABLESZ);
                updatetable(MTable **, uses, ((int)use->y), newtab);
                newtable(newtab, 4, TYPEDEFSZ, &transient);
            }
            useentry = idxtable(MTable **, *uses, use->y);
            t.sym = p;
            t.file = use->file;
            t.x = use->x;
            addtotable(TypeDef *, useentry, t);
            if (uselist->link == head) break; /* Lists are circular; */
        }
    }
}
```

For each variable, `typedef_uses` finds the closest use of a typedef that is on the line of the variable's declaration.

27a ⟨*definitions* 6⟩+≡
```
static void typedef_uses() {
    MTable uses;
    TypeDef t;
    int i,j;
    ProcIndex *pi;
    Symbol s, f;
    extern int lineno;

    ⟨initialize uses, table of all typedef uses 27b⟩

    if (!emptytable(uses)) {
        ⟨for each global, find global's auxilliary type 27c⟩
        ⟨for each local, find local's auxilliary type 27d⟩
    }
}
```

27b ⟨*initialize* `uses`*, table of all typedef uses* 27b⟩≡
```
newtable(&uses, lineno+1, sizeof(MTable *), &transient);
foreach(identifiers, GLOBAL, adduses, (Generic)&uses);
```

This code is used in chunk 27a.

27c ⟨*for each global, find global's auxilliary type* 27c⟩≡
```
gentable(baseTbl.glbsymtab, i, 1) {
    s = tableentry(Symbol *, baseTbl.glbsymtab, i);
    ⟨find s's auxilliary type 27e⟩
}
```
This code is used in chunk 27a.

27d ⟨*for each local, find local's auxilliary type* 27d⟩≡
```
gentable(baseTbl.procindex, i, 0) {
    pi = &(tableentry(ProcIndex *, baseTbl.procindex, i));
    assert(f = idxtable(Symbol *, baseTbl.glbsymtab, symid(pi->id)));
    gentable(mfn(f).localsymtab, j, 1) {
        s = tableentry(Symbol *, mfn(f).localsymtab, j);
        ⟨find s's auxilliary type 27e⟩
    }
}
```
This code is used in chunk 27a.

27e ⟨*find* `s`*'s auxilliary type* 27e⟩≡
```
if (!s->computed && !s->generated && s->src.y < tablesize(uses)) {
    msym(s).auxtype = findtypedef(idxtable(MTable **, uses, s->src.y), s->src);
}
```

This code is used in chunk 27.

findtypedef finds the closest typedef that does not overlap with symbol. This is a hack because of the following code:

```
typedef X int;
typedef char *Y;
#line 1
X   x;
#line 1
  Y   b;
```

28 ⟨*definitions* 6⟩+≡

```
static Symbol findtypedef(uses, src) MTable *uses; Coordinate src; {
    int i;
    TypeDef t, closest;

    if (!uses) {
        return 0;
    }
    closest.x = 0;
    closest.sym = 0;
    gentable(*uses, i, 0) {
        t = tableentry(TypeDef *, *uses, i);
        if (t.file == src.file &&
            t.x < src.x && t.x >= closest.x &&
            strlen(t.sym->name) + t.x < src.x) {
            closest = t;
        }
    }
    if (closest.sym) {
        newglobalsym(closest.sym);
        msym(closest.sym).typesym = 1;
    }
    return closest.sym;
}
```

2.8 Target-dependent code

Although the form of `mill` code is machine-independent, its content is not. The sizes and alignments of basic types and the endianness of the target are embedded in `mill` code. All of this data is encapsulated in the targets' interfaces.

29 ⟨*interface definitions* 29⟩≡

```
Interface mipsebIR = {
        1, 1, 0,    /* char */
        2, 2, 0,    /* short */
        4, 4, 0,    /* int */
        4, 4, 1,    /* float */
        8, 8, 1,    /* double */
        4, 4, 0,    /* T * */
        0, 1, 0,    /* struct */
    0,    /* little_endian */
    0,    /* mulops_are_calls */
    0,    /* wants_callb */
    1,    /* wants_argb */
    1,    /* left_to_right */
    0,    /* wants_dag */
    ⟨standard code-generator interfaces 32a⟩
    0, 0, 0, 0, 0, 0, 0,
}, mipselIR= {
        1, 1, 0,    /* char */
        2, 2, 0,    /* short */
        4, 4, 0,    /* int */
        4, 4, 1,    /* float */
        8, 8, 1,    /* double */
        4, 4, 0,    /* T * */
        0, 1, 0,    /* struct */
    1,    /* little_endian */
    0,    /* mulops_are_calls */
    0,    /* wants_callb */
    1,    /* wants_argb */
    1,    /* left_to_right */
    0,    /* wants_dag */
    ⟨standard code-generator interfaces 32a⟩
    0, 0, 0, 0, 0, 0, 0,
};
```

This definition is continued in chunks 30 and 31.
This code is used in chunk 33d.

30a ⟨*interface definitions* 29⟩+≡
```
Interface sparcIR = {
        1, 1, 0,    /* char */
        2, 2, 0,    /* short */
        4, 4, 0,    /* int */
        4, 4, 1,    /* float */
        8, 8, 1,    /* double */
        4, 4, 0,    /* T * */
        0, 1, 0,    /* struct */
    0,        /* little_endian */
    1,        /* mulops_are_calls */
    1,        /* wants_callb */
    0,        /* wants_argb */
    1,        /* left_to_right */
    0,        /* wants_dag */
    ⟨standard code-generator interfaces 32a⟩
    0, 0, 0, 0, 0, 0, 0,
};
```

30b ⟨*interface definitions* 29⟩+≡
```
Interface solarisIR = {
        1, 1, 0,    /* char */
        2, 2, 0,    /* short */
        4, 4, 0,    /* int */
        4, 4, 1,    /* float */
        8, 8, 1,    /* double */
        4, 4, 0,    /* T * */
        0, 1, 0,    /* struct */
    0,    /* little_endian */
    1,    /* mulops_calls */
    1,    /* wants_callb */
    0,    /* wants_argb */
    1,    /* left_to_right */
    0,    /* wants_dag */
    ⟨standard code-generator interfaces 32a⟩
    0, 0, 0, 0, 0, 0, 0,
};
```

31a ⟨*interface definitions* 29⟩+≡
```
Interface x86IR= {
    1, 1, 0,    /* char */
    2, 2, 0,    /* short */
    4, 4, 0,    /* int */
    4, 4, 1,    /* float */
    8, 4, 1,    /* double */
    4, 4, 0,    /* T * */
    0, 4, 0,    /* struct */
    1,        /* little_endian */
    0,        /* mulops_are_calls */
    0,        /* wants_callb */
    1,        /* wants_argb */
    0,        /* left_to_right */
    0,        /* wants_dag */
    ⟨standard code-generator interfaces 32a⟩
    0, 0, 0, 0, 0, 0, 0,    /* no stabs */
};
```

31b ⟨*interface definitions* 29⟩+≡
```
Interface x86linuxIR= {
    1, 1, 0,    /* char */
    2, 2, 0,    /* short */
    4, 4, 0,    /* int */
    4, 4, 1,    /* float */
    8, 4, 1,    /* double */
    4, 4, 0,    /* T * */
    0, 1, 0,    /* struct */
    1,        /* little_endian */
    0,        /* mulops_are_calls */
    0,        /* wants_callb */
    1,        /* wants_argb */
    0,        /* left_to_right */
    0,        /* wants_dag */
    ⟨standard code-generator interfaces 32a⟩
    0, 0, 0, 0, 0, 0, 0,    /* no stabs */
};
```

32a ⟨*standard code-generator interfaces* 32a⟩≡
```
address,
blockbeg,
blockend,
defaddress,
defconst,
defstring,
defsymbol,
emit,
export,
function,
gen,
global,
import,
local,
progbeg,
progend,
segment,
space,
```
This code is used in chunks 29–31.

2.9 Miscellaneous support

basename returns the base name for a file name, e.g. /usr/drh/foo.c =¿ foo. **concat** returns the concatenation of strings s1 and s2.

32b ⟨*external definitions* 32b⟩≡
```
extern int outfd;
int offset, maxoffset;
```
This code is used in chunk 33d.

32c ⟨*local declarations* 5a⟩+≡
```
dclproto(static char *basename,(char *name));
dclproto(static char *concat,(char *,char *));
```

33a ⟨*definitions* 6⟩+≡
```
static char *basename(name) char *name; {
    char *s, *b, *t = 0;

    if (!name) return name;
    for (b = s = name; *s; s++)
        if (*s == '/') {
            b = s + 1;
            t = 0;
        } else if (*s == '.')
            t = s;
    s = stringn(b, t ? t-b : strlen(b));
    return s;
}
static char *concat(s1, s2) char *s1, *s2; {
    int n = strlen(s1);
    char *s = malloc(n + strlen(s2) + 1);

    strcpy(s, s1);
    strcat(s, s2);
    return s;
}
```
The remaining procedures are stubs because `mlcc`'s code generator does not emit any code when they are called.

33b ⟨*local declarations* 5a⟩+≡
```
dclproto(static void defsymbol,(Symbol));
dclproto(static void export,(Symbol));
dclproto(static void import,(Symbol));
```

33c ⟨*definitions* 6⟩+≡
```
static void defsymbol(p) Symbol p; {}
void emit(p) Node p; {}
static void export(p) Symbol p; {}
static void import(p) Symbol p; {}
```

33d ⟨*gen.c* 33d⟩≡
```
#include <stdio.h>
#include <c.h>
#include <sys/fcntl.h>
#include <string.h>
#include <aemit.h>
#include <bemit.h>
#include <sys/time.h>
#include <sys/resource.h>

dclproto(extern char *strcat,(char *, char *));
dclproto(extern char *strcpy,(char *, char *));
dclproto(extern char *strstr,(const char *, const char *));
dclproto(extern int  dup,(int));
⟨external definitions 32b⟩
⟨local declarations 5a⟩
⟨definitions 6⟩
⟨interface definitions 29⟩
```
This code is written to file `gen.c`.

2.10 Common data structures

All of these data structures should be integrated with the code for emitting and reading `mill` code.

Procedure indices, regions, and callgraphs are simple.

34a ⟨*data structures* 34a⟩≡
```
typedef struct procidx {
    unsigned short id;
    unsigned long  offset;
    unsigned long  size;
    unsigned short stridx;
}ProcIndex;
#define    PROCIDXSZ    (sizeof(struct procidx))
#define PROCIDXTABSZ    20        /* assume 20 procs/module */
```

This definition is continued in chunks 34–40.
This code is used in chunk 42b.

34b ⟨*data structures* 34a⟩+≡
```
typedef struct  {
    unsigned long    base;
    unsigned long    size;
}Region;
#define    REGSZ    (sizeof(Region))
```

34c ⟨*data structures* 34a⟩+≡
```
typedef struct {
    unsigned short  fnid;           /* Function id */
    unsigned short  fgnid;          /* Flow graph node id */
    unsigned char inline;   /* Inlineable ? */
    float   ref;            /* Weighted reference count */
}CallGraphNode;
#define CALLGRNODESZ    (sizeof(CallGraphNode))
```

An individual call site includes a reference count.

34d ⟨*data structures* 34a⟩+≡
```
typedef struct {
    Symbol callee;
    int  id;
    int  count;
}CallSiteNode;
#define CALLSITENODESZ    (sizeof(CallSiteNode))
```

The data region contains type tag, value pairs. A value may be a character, unsigned short, unsigned, integer, floating point, pointer, string, or uninitialized bytes. `X` is the type tag for a pointer, `Y` for a string, `Z` for uninitialized bytes; the other tags use `mlcc`'s type ops `C`,`I`, etc.

35a ⟨*data structures* 34a⟩+≡
```
typedef struct {
    unsigned char    ty;
    union {
        unsigned char uc;
        unsigned short us;
        unsigned int u;
        float f;
        char *p;
    }x;
}DataItem;
```

35b ⟨*data structures* 34a⟩+≡
```
#define X    B+1
#define Y    B+2
#define Z    B+3
#define DATAITEMSZ   (sizeof(DataItem))
#define MAXDATAITEM  200
#define DATAREGSZ    (MAXDATAITEM*DATAITEMSZ)
```

A packed module header contains offsets to each region in the `mill` object file.

35c ⟨*data structures* 34a⟩+≡
```
typedef struct {
    Region     procsReg;        /* Procedure region */
    Region     procindexReg;    /* Index of procedure region */
    Region     inlineindexReg;  /* Index of inlining info */
    Region     callgraphReg;    /* Call graph */
    Region     glbsymsReg;      /* Global symbol table */
    Region     typesReg;        /* Type table */
    Region     dataReg;         /* Data region */
    Region     stringReg;       /* String region */
    long       maxregsz;        /* Maximum region or procedure size */
    unsigned short     minlabel;
    int        segmentsizes[6]; /* Sizes in bytes of data segments */
}PackedBaseTable;
#define PCKDBASETBLSZ sizeof(PackedBaseTable)
```

35d ⟨*data structures* 34a⟩+≡
```
typedef struct summary {
    int    procs;
    int    flowgraphs;
    int    code;
    int    localsyms;
    int    nlocalsyms;
    int    auxsyms;
}Summary;
```

An unpacked module header contains all of the tables that represent the module.

36a ⟨*data structures* 34a⟩+≡
```
typedef struct btable {
    PackedBaseTable    pbt; /* Packed data stored in BMILL file */
    /* Space for unpacked data after processing BMILL file */
    char    *pathname;    /* path name of file */
    char    *filename;
    long    baseTableLoc;
    MTable  glbsymtab;    /* Unpacked global symbol table */
    MTable  typetab;      /* Unpacked type table */
    MTable  procindex;    /* Unpacked proc index */
    MTable  dataGlobs;    /* Globals that refer to data region */
    MTable  stringmap;    /* Table of string region */
    MTable  datatab;      /* Unpacked data region */
    int     fd;           /* File descriptor of BMILL file */
    int     optvarct;     /* Count of variables used in optimization */
    long    offset;       /* Non-zero if file is in an archive */
    int     maxbufsz;
    Filebuf buf;          /* Buffer for file's contents */
    struct arena *arena;  /* Arena for module's buffer & tables */
    unsigned visited:1;   /* 1 if module required during linking */
    Summary sumsizes;     /* Summary info */
    struct  btable *next;
}BaseTable;
#define BASETBLSZ (sizeof(BaseTable))
```

36b ⟨*data structures* 34a⟩+≡
```
typedef struct mfunction {
    Symbol    sym;
    BaseTable *bt;          /* containing module */
    MTable    flowgraph;    /* list or table of flow graph nodes */
    MTable    caller;       /* caller/callee symbols */
    MTable    callee;
    MTable    localsymtab;
    MTable    dagtable;     /* table of dag roots */
    MTable    nodetable;    /* table of nodes after unmilling */
    int       dagroots;     /* # of dag roots */
#ifdef MLD
    struct optinfo *optinfo;     /* information for optimization */
#endif
} MFunction;
#define MFNSZ    (sizeof(MFunction))
```

Flow graph nodes are too complex. Their fields are self explanatory.

36c ⟨*data structures* 34a⟩+≡
```
typedef enum fgtypes { ArgN, CallN, EntryN, ExitN, JumpN,
MultitestN, RetN, StmtsN, TestN, UnknownN } FGType;
```

37a ⟨*data structures* 34a⟩+≡

```
typedef struct fgnode{
    unsigned short id;
    unsigned char type;    /* FGType */
    unsigned codelist:1;   /* 1 if end of a codelist */
    MTable   succs;        /* List of signed chars */
    MTable   preds;        /* List of signed chars */
    MTable   dagnodes;     /* Table of dag leaf,root pairs  */
    MTable   dagroots;     /* Table of root ids */
    Node     node;         /* Head of list of dag nodes for gen() */
    MTable   tempsymids;   /* Ids of temps associated with dags in this node */
    unsigned short   labelsym;     /* Id of target label */
    Env      env;          /* Environment to save for backend */
    MTable   localsymids;  /* ids of syms local to this block */
#ifdef MLD
    struct set dominators;   /* Set of dominators */
#endif
    union {
        struct {           /* 1 succ; 1+ preds */
            ⟨callN info 37c⟩
        }callN;
         struct {          /* 1 succ; 0/1 pred */
            unsigned short     exitNid;
        }entryN;
        struct {           /* 1 succ; 1+ preds */
            unsigned short    entryNid;
        }exitN;
        struct {              /* 1+ succs; 1+ preds */
            ⟨multitestN info 38a⟩
        }multitestN;
        struct {              /* 0/1 succ; 1+ preds */
            ⟨returnN info 38b⟩
        }returnN;
    }u;
}*FGNode;
```

37b ⟨*data structures* 34a⟩+≡

```
#define callN(N)     ((N)->u.callN)
#define catchN(N)    ((N)->u.catchN)
#define entryN(N)    ((N)->u.entryN)
#define exitN(N)     ((N)->u.exitN)
#define multitestN(N)    ((N)->u.multitestN)
#define returnN(N)   ((N)->u.returnN)
```

37c ⟨callN *info* 37c⟩≡

```
unsigned short argnid; /* first flow graph node that contains
                          arguments of this call node */
unsigned short retsymid;
unsigned short dagn;   /* id of CALL dagnode */
MFunction *mfn;    /* Inlined function */
```

This code is used in chunk 37a.

38a ⟨**multitestN** *info* 38a⟩≡
```
unsigned short sym;       /* Switch symbol */
unsigned short deflab;    /* Symbol id of default label */
unsigned short size;      /* # integers in switch */
MTable labels;    /* table of label values */
MTable values;    /* table of test values */
Symbol deflabsym;
MTable labelsyms;
```
This code is used in chunk 37a.

38b ⟨**returnN** *info* 38b⟩≡
```
unsigned short    retval; /* Symbol id of return value */
unsigned short    entryNid;
#ifdef MLD
unsigned typeval;      /* type identifier for M3 */
#endif
```
This code is used in chunk 37a.

An **MFuncSym** contains extra information about a procedure symbol.

38c ⟨*data structures* 34a⟩+≡
```
typedef struct mfuncsym { /* MILL function symbol */
    unsigned leaf:1;      /* 1 if leaf fn */
    unsigned indircall:1; /* 1 if fn can be called indirectly */
    unsigned asmcode:1;   /* 1 if fn's text is assembly code */
    unsigned onstack:1;   /* 1 if fn on DFS stack */
    unsigned cached:1;    /* 1 if fn cached in memory */
    unsigned emitted:1;   /* 1 if fn emitted */
    unsigned noopt:1;     /* 1 if fn is excluded from optimization */
    unsigned int ninlines;    /* # inline call sites */
    unsigned short procid;    /* index in module */
    struct arena *arena;      /* fn's arena */
    unsigned int cgnid;       /* Call graph node id */
    unsigned short ncalls;    /* # direct procedure calls */
    unsigned short dagsize;   /* # of real dag nodes */
    MTable    callgraph;      /* table of called fn's */
    MTable  inlinesites;      /* table of individual sites to inline */
    MFunction mfn;
}*MFuncSym;
#define MFNSYMSZ sizeof(struct mfuncsym)
```

MSymbol is mlcc's backend representation of a symbol and mld's frontend representation.

39 ⟨*data structures* 34a⟩+≡
```
typedef struct msym {    /* MILL symbol extension */
    unsigned short  id;    /* unique identifier */
    unsigned short  oid;   /* unique identifier for optimization */
    unsigned short  stridx;    /* index of string in string table */
    unsigned announced:1;  /* 1 if announced to backend */
    unsigned typesym:1;    /* 1 if symbol is a type symbol */
    unsigned field:1;      /* 1 if symbol rep's a struct/union && # fields > 0 */
    unsigned enumsym:1;    /* 1 if symbol represents an enum */
    unsigned dataseg:1;    /* 1 if symbol appears in data segment */
    unsigned downarg:1;    /* 1 if passed as a VAR parameter */
    unsigned written:1;    /* 1 if symbol is ever written */
    unsigned method:1;     /* 1 if symbol is a method */
    unsigned replacement:1; /* 1 if symbol replaces another symbol */
    Symbol auxtype;        /* Symbol of auxilliary type */
    MTable *symtab;        /* Symbol table in which symbol resides */
    int    stackaddr;      /* stack address when copying between arenas */
    Symbol fwdsym;         /* forwarded symbol */
    Symbol replacesym;     /* replacement symbol */
    union {
        struct {    /* computed address: offset & symbol */
            int offset;
            unsigned short symid;
        }addr;
        struct {
            unsigned short dataloc; /* globals & statics: location in data seg */
            Symbol constsym;        /* constant symbol if global represents a constant */
            DataValue datavalue;    /* initialized data value */
        }glob;
        struct {
            unsigned cse: 1;       /* 1 if temporary represents a CSE */
            DataValue datavalue;    /* initialized data value */
        }temp;
        unsigned short     enumsz;    /* # enum ids if sym rep's an enum */
        struct {    /* label: flow graph node & predecessors */
            FGNode fgnode;
            struct list *targetof;
        }label;
        MFuncSym func;
    }u;
#ifdef MLD
    RAddr raddr;      /* relocatable address of symbol */
    unsigned typeval; /* type identifier for M3 */
#endif
}MSymbol;
```

40a ⟨*data structures* 34a⟩+≡
```
#define msym(_s)     ((_s)->x.m)
#define maddr(_s)    (msym(_s).u.addr)
#define menum(_s)    (msym(_s).u.enumsz)
#define mfunc(_s)    (msym(_s).u.func)
#define mfn(_s)      (mfunc(_s)->mfn)
#define mglob(_s)    (msym(_s).u.glob)
#define mlabel(_s)   (msym(_s).u.label)
#define mtemp(_s)    (msym(_s).u.temp)
```

`MDag` represents an unpickled `Node`.

40b ⟨*data structures* 34a⟩+≡
```
#define OPTYPES    B+1
#define NOP 0
typedef struct mdag {
    unsigned int opcode;
    int k0, k1;
    unsigned short sym0, sym1;
}MDag;
```

40c ⟨*external declarations* 40c⟩≡
```
#define LOCALDAGTABSZ  4
#define DAGTABSZ      200
#define GLBSYMTABSZ   128

#define CALLTABSZ    8
#define FGTABSZ      128
#define LOCALTABSZ   8
#define TYPETABSZ    128

#define isglobal(_s) ((_s) & ~(SHRT_MAX))
#define symid(_s)    ((_s) & SHRT_MAX)
#define GLOBSYM(_s)  ((_s) | ~(SHRT_MAX))
#define basesymbol(p) ((p)->computed ? \
                      idxtable(Symbol *, (*msym(p).symtab), symid(maddr(p).symid)) : (p))
#define relIndex(x,y)     (((y-x > SCHAR_MAX) || (y-x < SCHAR_MIN)) ? \
    error("Relative index %d (%d-%d) out of range\n",y-x,y,x),y : (char)(y-x))
#define srelIndex(x,y)    (((y-x > SHRT_MAX) || (y-x < SHRT_MIN)) ? \
    error("Relative index %d (%d-%d) out of range\n",y-x,y,x),y : (short)(y-x))
#define globalsym(_p) ((_p)->scope == GLOBAL || ((_p)->scope != LABELS && (_p)->sclass == STATIC))
#define num(_s)       (_s[0] >= '0' && _s[0] <= '9')
#define max(_x,_y)    (_x >= _y ? _x : _y)

#define CHARSZ    (sizeof(char))
#define SHORTSZ   (sizeof(unsigned short))
#define LONGSZ    (sizeof(long))

#define NBASICTYPES   14
#define CONSTANTIDX   NBASICTYPES+1
#define NCONSTANTS    8
```

This definition is continued in chunks 41 and 42a.
This code is used in chunk 42b.

C Library.

41a ⟨*external declarations* 40c⟩+≡
```
dclproto(extern void abort,(void));
dclproto(extern int atoi,(char *));
dclproto(extern void bcopy,(char *b1, char *b2, int len));
dclproto(extern void bzero,(char *,int));
dclproto(extern int  lseek,(int,int,int));
dclproto(extern void free,(void *));
dclproto(extern void *malloc,(unsigned));
dclproto(extern char *realloc,(char *ptr, int sz));
dclproto(extern char *strcpy,(char *, char *));
dclproto(extern char *strcat,(char *, char *));
dclproto(extern int tell,(int));
dclproto(extern int write,(int, char *, int));
```

`mill.c`.

41b ⟨*external declarations* 40c⟩+≡
```
dclproto(Node caller2callee,(Symbol, Symbol));
dclproto(unsigned short constantsym,(int));
dclproto(void genfgnode,(FGNode, MFunction *));
dclproto(int genmillcode, (Node, MFunction *, int, FGNode, FGNode));
dclproto(void insertintable,(MTable *, unsigned int));
dclproto(int intable,(MTable, unsigned int));
dclproto(int instringtable,(MTable, char *));
dclproto(unsigned char mapopcode,(int));
dclproto(void mergesymtab,(MTable *, MTable));
dclproto(void mergesymbols,(Symbol p,Symbol q));
dclproto(struct arena *newarena,(void));
dclproto(unsigned short newlocalsym,(Symbol, MTable *, MTable *));
dclproto(Symbol newtempsym,(Type, MTable *, MTable *));
dclproto(void nopdag,(int, MTable *));
dclproto(int readdag,(int, MDag *, MTable));
dclproto(int writedag,(int, MDag , MTable *));
dclproto(int unmapopcode,(unsigned char));
dclproto(int unmilldag,(int, MDag *, MTable, MTable *));
```

41c ⟨*external declarations* 40c⟩+≡
```
extern Type *basictypes[];
#define basictype(t)     (t && t->x.typeno <= NBASICTYPES)
#define adddagrange(dagnodes, l, h) \
        { Range r; r.low = l; r.hi = h; addtotable(Range *, dagnodes, r); }
#define setdagrange(dagnodes, i, l , h) \
        { tableentry(Range *, dagnodes, i).low = l; \
          tableentry(Range *, dagnodes, i).hi = h; }
#define dagrange(dagnodes, i, l, h) \
        { l = tableentry(Range *, dagnodes, i).low; \
          h = tableentry(Range *, dagnodes, i).hi; }
#define indircalledproc(p) (mfunc(p)->indircall || msym(p).dataseg || msym(p).downarg)
```

`bread.c`.

42a ⟨*external declarations* 40c⟩+≡

```
dclproto(void breadData,(BaseTable *));
dclproto(void breadModuleHdr,(BaseTable *));
dclproto(void breadProcedure,(Symbol));
dclproto(void breadProgram,(BaseTable *));
dclproto(int  breadProgramSize,(BaseTable *));
dclproto(void breadStrings,(BaseTable *));
dclproto(void breadRegion,(BaseTable *, Region));
```

42b ⟨*mill.h* 42b⟩≡

```
#ifndef INCLUDE_MILLH
#define INCLUDE_MILLH

#include <mtable.h>
#include <dataimage.h>
#include <mlcc_ops.h>
#ifdef MLD
#include <set.h>
#include <mclib.h>
#endif
#include <buf.h>

⟨data structures 34a⟩
⟨external declarations 40c⟩
#endif
```

This code is written to file `mill.h`.

3 Emitting `mill` object files

A module is emitted into a `mill` object file in a binary format. To aid in debugging, an ASCII representation of the `mill` code also can be emitted. This module includes the procedures for emitting `mill` code in both binary and ASCII formats. The names of binary emitting procedures begin with `bemit`; ASCII emitters begin with `emit`.

3.1 Modules

A module contains the following components, contained in "regions":

- Header
- Procedures
- Procedure index
- Inlining index
- Call graph index
- Global symbol table
- Type table
- Strings

A module's header contains the absolute path name of the source file and a `BaseTable` that contains the offsets to each region in the file. `bemitModuleHdr` emits the module's name and reserves space for its base table. Each individual procedure is emitted by `bemitProcedure`. After all procedures are emitted, `bemitProgram` emits all the global module data and then updates the module's base table to point to the absolute offsets of each region.

43a ⟨*binary external declarations* 43a⟩≡

```
dclproto(void bemitModuleHdr,(BaseTable *, char *));
dclproto(void bemitProgram,(int, BaseTable *));
```

This definition is continued in chunks 54b and 55b.
This code is used in chunk 66c.

43b ⟨*binary definitions* 43b⟩≡

```
void bemitModuleHdr(bt, filename) BaseTable *bt; char *filename; {

    if (!bt->fd) return;
    bt->filename = string(filename);
    bemitString(bt->fd, filename);
    bt->baseTableLoc = tell(bt->fd);
    lseek(bt->fd, PCKDBASETBLSZ, SEEK_CUR);    /* Seek past base table */
    bt->maxbufsz = 0;
    bufalloc(bt);
}
```

This definition is continued in chunks 44a, 46b, 47c, 49–51, 54–56, 58–60, and 62–65.
This code is used in chunk 66d.

bemitProgram emits the program represented by bt in the file referred to by afd.

44a ⟨*binary definitions* 43b⟩+≡
```
void bemitProgram(afd, bt) BaseTable *bt; {
    List procs;
    int maxregsz;
    int i;
    ProcIndex pi;

    maxregsz = 0;
    if (!bt->fd) return;

    ⟨ emit procedure index 44b⟩
    /* bt->inlineIndex = bemitInlineIndex(inlineIndex) */
    bt->pbt.inlineindexReg.base = 0;
    bt->pbt.inlineindexReg.size = 0;

    ⟨ emit global symbol table 44c⟩
    ⟨ emit type table 44d⟩
    ⟨ emit module's call graph 44e⟩
    ⟨ emit abstract data segments 45a⟩
    ⟨ emit strings 45b⟩
    ⟨ update region offsets 45c⟩

}
```

44b ⟨ *emit procedure index* 44b⟩≡
```
emitProcIndex(afd, bt->procindex);
bt->pbt.procindexReg = bemitProcIndex(bt);
gentable(bt->procindex, i, 0) {
    pi = tableentry(ProcIndex *, bt->procindex, i);
    maxregsz = max(maxregsz, pi.size);
}
```

This code is used in chunk 44a.

44c ⟨ *emit global symbol table* 44c⟩≡
```
emitSymbols(afd, bt->glbsymtab, "");
bt->pbt.glbsymsReg = bemitSymbols(bt, bt->glbsymtab, NBASICTYPES+NCONSTANTS+1);
maxregsz = max(maxregsz, bt->pbt.glbsymsReg.size);
```

This code is used in chunk 44a.

44d ⟨ *emit type table* 44d⟩≡
```
emitTypes(afd, bt->typetab);
bemitTypes(bt);
maxregsz = max(maxregsz, bt->pbt.typesReg.size);
```

This code is used in chunk 44a.

44e ⟨ *emit module's call graph* 44e⟩≡
```
emitCallgraph(afd, bt);
bt->pbt.callgraphReg = bemitCallgraph(bt);
maxregsz = max(maxregsz, bt->pbt.callgraphReg.size);
```

This code is used in chunk 44a.

45a ⟨*emit abstract data segments* 45a⟩≡
```
emitData(afd, bt->dataGlobs, bt->datatab);
bemitData(bt);
maxregsz = max(maxregsz, bt->pbt.dataReg.size);
```
This code is used in chunk 44a.

45b ⟨*emit strings* 45b⟩≡
```
bemitStrings(bt);
maxregsz = max(maxregsz, bt->pbt.stringReg.size);
```
This code is used in chunk 44a.

45c ⟨*update region offsets* 45c⟩≡
```
bt->pbt.maxregsz = maxregsz;
lseek(bt->fd, bt->baseTableLoc, SEEK_SET);
write(bt->fd, (char *)&(bt->pbt), PCKDBASETBLSZ);
emitBaseTable(afd, *bt);
```
This code is used in chunk 44a.

45d ⟨*local declarations* 45d⟩≡
```
static char *segnames[] = { "", "TEXT", "BSS", "DATA", "LIT", "SYM" };
```
This definition is continued in chunks 46e, 51, 53c, 56d, and 60d.
This code is used in chunk 66b.

45e ⟨*external declarations* 45e⟩≡
```
dclproto(void emitBaseTable,(int, BaseTable));
```
This definition is continued in chunks 46c, 47d, 51f, 56b, 60c, 63c, and 65c.
This code is used in chunk 66a.

45f ⟨*definitions* 45f⟩≡
```
void emitBaseTable(fd, b) BaseTable b; {
    int i;

    if (!fd) return;
    fprint(fd, "Filename\t%s\n", (b.filename ? b.filename : ""));
    fprint(fd, "ProcsReg\t%d\t%d\n", b.pbt.procsReg.base, b.pbt.procsReg.size);
    fprint(fd, "ProcIndex\t%d\t%d\n", b.pbt.procindexReg.base, b.pbt.procindexReg.size);
    fprint(fd, "CallGraph\t%d\t%d\n", b.pbt.callgraphReg.base, b.pbt.callgraphReg.size);
    fprint(fd, "GlbSyms\t%d\t%d\n", b.pbt.glbsymsReg.base, b.pbt.glbsymsReg.size);
    fprint(fd, "Types\t%d\t%d\n", b.pbt.typesReg.base, b.pbt.typesReg.size);
    fprint(fd, "Data\t%d\t%d\n", b.pbt.dataReg.base, b.pbt.dataReg.size);
    fprint(fd, "Strings\t%d\t%d\n", b.pbt.stringReg.base, b.pbt.stringReg.size);
    fprint(fd, "Max Region Size\t%d\n", b.pbt.maxregsz);
    fprint(fd, "SIZES:\n");
    fprint(fd, "PSym\t%d\tPType\t%d\tPField\t%d\n", PSYMSZ, PTYPESZ, PFIELDSZ);
    fprint(fd, "ProcIdx\t%d\tRegion\t%d\tDataItem\t%d\n", PROCIDXSZ, REGSZ, DATAITEMSZ);
    fprint(fd, "BaseTab\t%d\n", PCKDBASETBLSZ);
    for (i = 1; i < 6; i++)
        fprint(fd, "%s %d\n", segnames[i], b.pbt.segmentsizes[i]);

}
```
This definition is continued in chunks 46–48, 51d, 52, 54a, 56c, 57a, 61, 62a, 64a, and 65d.
This code is used in chunk 66b.

3.2 Call Graphs

Call graphs are represented by `<Caller ID><List of Callee IDs>`. `bemitCallgraph` emits the callgraph for the module in `bt`. `emitDotCallgraph` emits a `dot` file (?) that represents the call graph.

46a ⟨*binary local declarations* 46a⟩≡

```
dclproto(static Region  bemitCallgraph,(BaseTable *bt));
```

This definition is continued in chunks 47b, 48c, 51a, 54d, 55d, 57b, 62b, 64b, and 65a.
This code is used in chunk 66d.

46b ⟨*binary definitions* 43b⟩+≡

```
static Region bemitCallgraph(bt) BaseTable *bt;  {
    Region r;
    int i;
    Symbol fn;
    ProcIndex pi;

    bufflush(bt);
    r.base = tell(bt->fd);
    gentable(bt->procindex, i, 0) {
        pi = tableentry(ProcIndex *, bt->procindex, i);
        fn = (Symbol) idxtable(Symbol *, bt->glbsymtab, symid(pi.id));
        bemitMTable(bt, mfunc(fn)->callgraph, CALLGRNODESZ);
    }
    bufflush(bt);
    r.size = tell(bt->fd) - r.base;
    return r;
}
```

46c ⟨*external declarations* 45e⟩+≡

```
dclproto(void emitCallgraph,(int, BaseTable *));
```

46d ⟨*definitions* 45f⟩+≡

```
void emitCallgraph(fd, bt) BaseTable *bt; {
    int i, j;
    Symbol fn;
    CallGraphNode callee;
    ProcIndex pi;

    if (!fd) return;
    fprint(fd, "CALLGRAPH\n");
    gentable(bt->procindex, i, 0) {
        pi = tableentry(ProcIndex *, bt->procindex, i);
        fn = (Symbol) tableentry(Symbol *, bt->glbsymtab, symid(pi.id));
        fprint(fd, "%s:", fn->name, mfunc(fn)->callgraph.maxelem);
        gentable(mfunc(fn)->callgraph, j, 0) {
            callee = tableentry(CallGraphNode *, mfunc(fn)->callgraph, j);
            fprint(fd, "\n\t%s %d %d", idxtable(Symbol *, bt->glbsymtab,
                symid(callee.fnid))->name, (int)callee.ref, callee.fgnid);
        }
        fprint(fd, "\n");
    }
}
```

46e ⟨*local declarations* 45d⟩+≡

```
dclproto(void emitDotCallgraph,(Symbol));
```

47a ⟨*definitions* 45f⟩+≡
```
void emitDotCallgraph(fn) Symbol fn; {
    int i;
    Symbol s;
    CallGraphNode cgn;

    fprint(2,"%s [label=\"%s,%d\"];\n",fn->name, fn->name, mfunc(fn)->cgnid);
    gentable(mfunc(fn)->callgraph, i, 0) {
        cgn = tableentry(CallGraphNode *, mfunc(fn)->callgraph, i);
        s = idxtable(Symbol *, (*msym(fn).symtab), symid(cgn.fnid));
        fprint(2, "%s -> %s [label=\"%d\"];\n", fn->name, s->name, (int)cgn.ref);
        if (cgn.inline) fprint(2,"%s [shape=diamond];\n",s->name);
    }
}
```

3.3 Data

`bemitData` emits the table of symbols that refer to initialized data in the data region and then emits the data itself.

47b ⟨*binary local declarations* 46a⟩+≡
```
dclproto(static void bemitData,(BaseTable *));
```

47c ⟨*binary definitions* 43b⟩+≡
```
static void bemitData(bt) BaseTable *bt; {
    List l;
    DataItem *d;
    int i, regsz;

    bufflush(bt);
    bt->pbt.dataReg.base = tell(bt->fd);
    bemitMTable(bt, bt->dataGlobs, SHORTSZ);
    bemitMTable(bt, bt->datatab, DATAITEMSZ);
    bufflush(bt);
    bt->pbt.dataReg.size = tell(bt->fd) - bt->pbt.dataReg.base;
}
```

47d ⟨*external declarations* 45e⟩+≡
```
dclproto(void emitData,(int, MTable, MTable));
dclproto(void emitDataItem,(int, DataItem *));
#define emitSymid(_fd,_s) (isglobal(_s) ? \
                           fprint(_fd,"G%d ",symid(_s)) : fprint(_fd,"%d ",_s))
```

48a ⟨*definitions* 45f⟩+≡
```
void emitDataItem(fd, d) DataItem *d; {
    char *names = "*XYZ";

    if (!fd) return;
    fprint(fd, "%c\t", d->ty < B ? TYPENAMES[d->ty] : names[d->ty - B]);
    switch(d->ty) {
        case C: fprint(fd, "%d\n", (int)d->x.uc); break;
        case I:
        case U: fprint(fd, "%d\n", (int)d->x.u);    break;
        case S: fprint(fd, "%d\n", (int)d->x.us); break;
        case F: fprint(fd, "%f\n", (float)d->x.f); break;
        case D: fprint(fd, "%d\n", d->x.u); break;
        case P: fprint(fd, "%d\n", (int)d->x.p);    break;
        case X: emitSymid(fd, d->x.u); fprint(fd, "\n"); break;
        case Y: fprint(fd, "%d\n", (int)d->x.us); break;
        case Z: fprint(fd, "%d\n", d->x.u); break;
        default: error("emitData: unknown type\n"); break;
    }
}
```

48b ⟨*definitions* 45f⟩+≡
```
void emitData(int fd, MTable dataGlobs, MTable datatab) {
    List l;
    DataItem *d;
    int i, max, *j;

    if (!fd) return;
    fprint(fd, "DATA REGION %d\n", tablesize(datatab));
    gentable(dataGlobs, i, 0) {
        emitSymid(fd, tableentry(unsigned short *, dataGlobs, i));
    }
    gentable(datatab, i, 0) {
        d = &(tableentry(DataItem *, datatab, i));
        emitDataItem(fd, d);
    }
}
```

3.4 Flow Graphs

`bemitFlowgraph` emits a table of flowgraph nodes, in the form `<Number of nodes><Node list>`.

48c ⟨*binary local declarations* 46a⟩+≡
```
dclproto(static int     bemitFlowgraph,(BaseTable *, MTable));
dclproto(static int     bemitFlowgraphnode,(BaseTable *, FGNode));
```

49 ⟨*binary definitions* 43b⟩+≡

```
static int bemitFlowgraph(BaseTable *bt, MTable flowgraph) {
    int sz, i;
    unsigned short    len;

    len = flowgraph.maxelem+1;
    assert(len >= 0);
    bufwrite(bt, (char *)&len, SHORTSZ);
    gentable(flowgraph, i, 1) {
        sz += bemitFlowgraphnode(bt, tableentry(FGNode *, flowgraph, i));
    }
    return sz;
}
```

bemitFlowgraphnode emits one flow graph node by flattening the dynamic representation of the node into a packed node (**PFGNode**). Data is emitted only for those components of the node that have non-zero size, i.e., space is not allocated for each component by default.

50 ⟨*binary definitions* 43b⟩+≡

```
static int bemitFlowgraphnode(bt, fgnode) BaseTable *bt; FGNode fgnode; {
    char gennode;
    int sz = 0;
    PFGNode pfg;

    pfg.id = fgnode->id;
    pfg.type = fgnode->type;
    pfg.gennode = fgnode->codelist;
    pfg.temps = !emptytable(fgnode->tempsymids);
    pfg.succs = !emptytable(fgnode->succs);
    pfg.preds = !emptytable(fgnode->preds);

    bufwrite(bt, (char *)&pfg, PFGSZ);    /* Emit type */
    sz = PFGSZ;
    if (pfg.temps) sz += bemitSMTable(bt, fgnode->tempsymids);
    if (pfg.succs) sz += bemitSMTable(bt, fgnode->succs);    /* Emit successors */
    if (pfg.preds) sz += bemitSMTable(bt, fgnode->preds);    /* Emit predecessors */
    sz += bemitMTable(bt, fgnode->dagnodes, RANGESZ);    /* Emit dagnode table */
    bufwrite(bt, (char *)&(fgnode->labelsym), SHORTSZ);
    sz += SHORTSZ;
    switch(fgnode->type) {
    case CallN:
        bufwrite(bt, (char *)&(callN(fgnode).argnid), SHORTSZ);
        bufwrite(bt, (char *)&(callN(fgnode).dagn), SHORTSZ);
        bufwrite(bt, (char *)&(callN(fgnode).retsymid), SHORTSZ);
        sz += SHORTSZ;
        break;
    case EntryN:
        sz += bemitMTable(bt, fgnode->localsymids, SHORTSZ);
        break;
    case ExitN:
        bufwrite(bt, (char *)&(exitN(fgnode).entryNid), SHORTSZ);
        sz += SHORTSZ;
        break;
    case MultitestN:
        bufwrite(bt, (char *)&(multitestN(fgnode).sym), SHORTSZ);
        bufwrite(bt, (char *)&(multitestN(fgnode).deflab), SHORTSZ);
        bufwrite(bt, (char *)&(multitestN(fgnode).size), SHORTSZ);
        sz += SHORTSZ*4;
        sz += bemitMTable(bt, multitestN(fgnode).values, LONGSZ);
        sz += bemitMTable(bt, multitestN(fgnode).labels, SHORTSZ);
        break;
    default: break;
    }
    return sz;
}
```

`bcopyFlowgraphnode` copies a flow graph node into a new arena. This is used by `mld`'s inliner to copy a callee's flow graph.

51a ⟨*binary local declarations* 46a⟩+≡
```
dclproto(static FGNode bcopyFlowgraphnode,(FGNode, Arena *, MTable *));
```

51b ⟨*binary definitions* 43b⟩+≡
```
static FGNode bcopyFlowgraphnode(oldfgnode, arena, stack)
        FGNode oldfgnode; Arena *arena; MTable *stack; {
    FGNode newfgnode = (FGNode)yyalloc(sizeof(*newfgnode), (*arena));

    BZERO(newfgnode, *newfgnode);
    bcopy((char *)oldfgnode, (char *)newfgnode, sizeof(*newfgnode));
    copytable(&(oldfgnode->succs), &(newfgnode->succs), arena);
    copytable(&(oldfgnode->preds), &(newfgnode->preds), arena);
    copytable(&(oldfgnode->dagnodes), &(newfgnode->dagnodes), arena);
    copytable(&(oldfgnode->tempsymids), &(newfgnode->tempsymids), arena);
    if (newfgnode->type == EntryN || newfgnode->type == CallN)
        copytable(&(oldfgnode->localsymids), &(newfgnode->localsymids), arena);
    if (newfgnode->type == MultitestN) {
        copytable(&(multitestN(oldfgnode).labels), &(multitestN(newfgnode).labels), arena);
        copytable(&(multitestN(oldfgnode).values), &(multitestN(newfgnode).values), arena);
    }
    else if (newfgnode->type == CallN && callN(oldfgnode).mfn != 0)
        callN(newfgnode).mfn = bcopyAuxMFunction(callN(oldfgnode).mfn, arena, stack);
    return newfgnode;
}
```

51c ⟨*local declarations* 45d⟩+≡
```
dclproto(static void emitFlowgraph,(int, MTable, MTable, char *));
```

51d ⟨*definitions* 45f⟩+≡
```
static void emitFlowgraph(fd, flowgraph, dagtable, tab)
    MTable flowgraph; MTable dagtable; char *tab;{
    int i;
    MTable    stack;

    newtable(&stack, 10, sizeof(int), &permanent);
    gentable(flowgraph, i, 1) {
        emitFlowgraphnode(fd, tableentry(FGNode *, flowgraph, i), dagtable, &stack, tab);
    }
}
```

51e ⟨*local declarations* 45d⟩+≡
```
char *fgtypenames[] = { "Arg", "Call", "Entry", "Exit", "Jump",
                        "Multitest", "Return", "Stmts", "Test", "Unknown", };
```

51f ⟨*external declarations* 45e⟩+≡
```
dclproto(void emitFlowgraphnode,(int, FGNode, MTable, MTable *, char *));
```

52 ⟨*definitions* 45f⟩+≡
```
void emitFlowgraphnode(fd, fg, dagtable, stack, tab)
        FGNode fg; MTable dagtable, *stack; char *tab;{
    List    l1, l2;
    unsigned short    l;
    int i, j, k, v, low, hi;
    MDag mdag;
    char *tabs = talloc(strlen(tab)+2);

    if (!fd) return;
    fprint(fd, "\n%s%s", tab, fg->codelist ? "*" : "");
    fprint(fd, "FGNODE%d:%s", fg->id, fgtypenames[fg->type]);
    fprint(fd, "\n%sSuccs: ", tab);
    gentable(fg->succs, i, 0) {
        fprint(fd, "%d ", (int)tableentry(short *, fg->succs, i));
    }
    fprint(fd, "\n%sPreds: ", tab);
    gentable(fg->preds, i, 0) {
        fprint(fd, "%d ", (int)tableentry(short *, fg->preds, i));
    }
    fprint(fd, "\n%sDAGs: ", tab);
    gentable(fg->dagnodes, i, 0) {
        dagrange(fg->dagnodes, i, low, hi);
        fprint(fd, "%d..%d ", low, hi);
    }
    if (!emptytable(fg->tempsymids)) {
        fprint(fd, "\n%sTEMPS: ", tab);
        gentable(fg->tempsymids, i, 0) {
            fprint(fd, "%d ", tableentry(unsigned short *, fg->tempsymids, i));
        }
    }
    ⟨ print type of node 53a⟩
    ⟨ print contents of node 53b⟩
}
```

53a ⟨*print type of node* 53a⟩≡
```
switch(fg->type) {
case CallN:
    fprint(fd, "\n%sCALL DAG: %d", tab, callN(fg).dagn);
    fprint(fd, "\n%sARG NODE: %d", tab, callN(fg).argnid);
        /* fall through */
case EntryN:
    if (!emptytable(fg->localsymids)) fprint(fd, "\n%sLOCALSYMS: ", tab);
    gentable(fg->localsymids, i, 0) {
        fprint(fd, "%d ", tableentry(unsigned short *, fg->localsymids, i));
    }
    if (fg->type == CallN && callN(fg).mfn != 0) {
        strcpy(tabs, tab);
        strcat(tabs,"    ");
        emitAuxMFunction(2, callN(fg).mfn, tabs);
    }
    break;
case ExitN:
    fprint(fd, "\n%sENTRYID: %d\n", tab, exitN(fg).entryNid);
    break;
case MultitestN:
    fprint(fd, "\n%sValues: ", tab);
    gentable(multitestN(fg).labels, i, 0) {
        l = tableentry(unsigned short *, multitestN(fg).labels, i);
        v = idxtable(int *, multitestN(fg).values, i);
        fprint(fd, "%d %d,", l, v);
    }
    break;
default: break;
}
fprint(fd, "\n");
```
This code is used in chunk 52.

53b ⟨*print contents of node* 53b⟩≡
```
if (fg->labelsym) fprint(fd, "%sLABEL %d: \n", tab, fg->labelsym);
gentable(fg->dagnodes, i, 0) {
    dagrange(fg->dagnodes, i, low, hi);
    for (j = low; j <= hi; j = k) {
        k = unmilldag(j, &mdag, dagtable, stack);
        emitDag(fd, j, mdag, tab);
    }
}
```
This code is used in chunk 52.

53c ⟨*local declarations* 45d⟩+≡
```
dclproto(static void emitDag,(int, int, MDag, char *));
```

54a ⟨*definitions* 45f⟩+≡
```
static void emitDag(fd, i, mdag, tab) MDag mdag; char *tab;{
    if (!fd) return;
    fprint(fd, "%s%d:%s #%d #%d ", tab, i, opname(mdag.opcode), mdag.k0, mdag.k1);
    if (mdag.sym0) emitSymid(fd, mdag.sym0);
    else fprint(fd, "0 ");
    if (mdag.sym1) emitSymid(fd, mdag.sym1);
    else fprint(fd, "0 ");
    fprint(fd, "\n");
}
```

3.5 Procedures

`bemitProcedure` emits a single procedure in the form: `<id><local symbol table><caller symbols><callee symbols> <flowgraph><dag table>`.

54b ⟨*binary external declarations* 43a⟩+≡
```
dclproto(void bemitProcedure,(BaseTable *, Symbol, ProcIndex *));
```

54c ⟨*binary definitions* 43b⟩+≡
```
void bemitProcedure(bt, f, pi) BaseTable *bt;  Symbol f; ProcIndex *pi; {
    short id;
    Region localsyms;

    bufflush(bt);
    pi->offset = tell(bt->fd) - bt->pbt.procsReg.base;

    id = msym(f).id;
    bufwrite(bt, (char *)&id, SHORTSZ);
    bt->sumsizes.procs += SHORTSZ;
    localsyms = bemitSymbols(bt, mfn(f).localsymtab, 1);
    bt->sumsizes.localsyms += localsyms.size;
    bt->sumsizes.nlocalsyms += tablesize(mfn(f).localsymtab);
    bt->sumsizes.procs += bemitSMTable(bt, mfn(f).caller);
    bt->sumsizes.procs += bemitSMTable(bt, mfn(f).callee);
    bt->sumsizes.flowgraphs += bemitFlowgraph(bt, mfn(f).flowgraph);
    bt->sumsizes.code += bemitMTable(bt, mfn(f).dagtable, CHARSZ);
    bufflush(bt);
    pi->size = tell(bt->fd) - pi->offset;
}
```

`bemitProcIndex` emits a module's procedure index.

54d ⟨*binary local declarations* 46a⟩+≡
```
dclproto(static Region     bemitProcIndex,(BaseTable *));
```

55a ⟨*binary definitions* 43b⟩+≡
```
static Region bemitProcIndex(bt) BaseTable *bt; {
    Region r;

    bufflush(bt);
    r.base = tell(bt->fd);
    bemitMTable(bt, bt->procindex, PROCIDXSZ);
    bufflush(bt);
    r.size = tell(bt->fd) - r.base;
    return r;
}
```

`bcopyMFunction` copies a `mill` function into a new arena.

55b ⟨*binary external declarations* 43a⟩+≡
```
dclproto(MFunction *bcopyMFunction,(MFunction *, struct arena **));
```

55c ⟨*binary definitions* 43b⟩+≡
```
MFunction *bcopyMFunction(oldmfn, arena) MFunction *oldmfn; Arena *arena; {
    MTable stack;     /* for use when copying symbols between arenas */

    newtable(&stack, 16, sizeof(Symbol), &transient);
    return bcopyAuxMFunction(oldmfn, arena, &stack);
}
```

55d ⟨*binary local declarations* 46a⟩+≡
```
dclproto(static MFunction *bcopyAuxMFunction,(MFunction *, Arena *, MTable *));
```

56a ⟨*binary definitions* 43b⟩+≡
```
static MFunction *bcopyAuxMFunction(oldmfn, arena, stack)
        MFunction *oldmfn; Arena *arena; MTable *stack; {
    MFunction *mfn = (MFunction *)yyalloc(MFNSZ, (*arena));
    int i;
    Symbol s;

    BZERO(mfn, *mfn);
    mfn->bt = oldmfn->bt;
    mfn->sym = oldmfn->sym;
    copytable(&(oldmfn->caller), &(mfn->caller), arena);
    copytable(&(oldmfn->callee), &(mfn->callee), arena);
    copytable(&(oldmfn->localsymtab), &(mfn->localsymtab), arena);
    gentable(oldmfn->localsymtab, i, 1) {        /* Copy symbol if not already in arena */
        s = tableentry(Symbol *, oldmfn->localsymtab, i);
        if (!isglobal(msym(s).id)) {
            if (emptytable((*stack)) || msym(s).stackaddr >= tablesize((*stack)) ||
                tableentry(Symbol *, (*stack), msym(s).stackaddr) != s) {
                msym(s).fwdsym = (Symbol)yyalloc(sizeof(struct symbol), (*arena));
                bcopy((char *)s, (char *)msym(s).fwdsym, sizeof(*s));
                msym(s).stackaddr = tablesize((*stack));
                addtotable(Symbol *, stack, s);
            } else
                assert(msym(s).fwdsym);
            tableentry(Symbol *, mfn->localsymtab, i) = msym(s).fwdsym;
        }
    }
    copytable(&(oldmfn->dagtable), &(mfn->dagtable), arena);
    copytable(&(oldmfn->flowgraph), &(mfn->flowgraph), arena);
    gentable(oldmfn->flowgraph, i, 1) {
        tableentry(FGNode *, mfn->flowgraph, i) =
            bcopyFlowgraphnode(tableentry(FGNode *, oldmfn->flowgraph, i), arena, stack);
    }
    return mfn;
}
```

56b ⟨*external declarations* 45e⟩+≡
```
dclproto(void emitProcedure,(int, Symbol));
dclproto(void emitMFunction,(int, MFunction *));
dclproto(void emitProcIndex,(int, MTable));
```

56c ⟨*definitions* 45f⟩+≡
```
void emitProcedure(fd, f) Symbol f; {
    if (!fd) return;
    fprint(fd, "PROC %d %s\n", symid(msym(f).id), f->name);
    emitMFunction(fd, &mfn(f));
}
```

56d ⟨*local declarations* 45d⟩+≡
```
dclproto(static void emitAuxMFunction,(int, MFunction *, char *));
```

57a ⟨*definitions* 45f⟩+≡
```
static void emitAuxMFunction(fd, mfn, tab) MFunction *mfn; char *tab; {
    int     i;
    unsigned short    c1, c2;

    emitSymbols(fd, mfn->localsymtab, tab);
    fprint(fd, "%s > >%s\n", tab,  (mfn->sym) ? mfn->sym->name : "???");
    fprint(fd, "%sCALLER/CALLEE \n", tab);
    gentable(mfn->caller, i, 0) {
        c1 = tableentry(unsigned short *, mfn->caller, i);
        c2 = idxtable(unsigned short *, mfn->callee, i);
        fprint(fd, "%sSYM %d/%d\n", tab, c1, c2);
    }
    emitFlowgraph(fd, mfn->flowgraph, mfn->dagtable, tab);
    fprint(fd, "%s < <%s\n", tab, mfn->sym->name);
}

void emitMFunction(fd, mfn) MFunction *mfn; {
    emitAuxMFunction(fd, mfn, "");
}

void emitProcIndex(fd, procs) MTable procs; {
    ProcIndex p;
    int i;

    if (!fd) return;
    gentable(procs, i, 0) {
        p = tableentry(ProcIndex *, procs, i);
        fprint(fd, "PROCIDX: ");
        emitSymid(fd, p.id);
        fprint(fd, ":offset %d stridx %d\n", p.offset, p.stridx);
    }
}
```

3.6 Symbols

`bemitSymbols` emits a symbol table and returns the table's region data.

57b ⟨*binary local declarations* 46a⟩+≡
```
dclproto(static Region bemitSymbols,(BaseTable *, MTable, int));
dclproto(static void bemitPackedSyms,(BaseTable *, MTable, int));
dclproto(static void packSym,(Symbol, Psymbol *));
dclproto(static int packAuxSym,(Symbol, PAuxsymbol *));
dclproto(static void bemitFields,(BaseTable *, int, int, Field));
```

58a ⟨*binary definitions* 43b⟩+≡

```
static Region bemitSymbols(bt, syms, startsym) BaseTable *bt; MTable syms; {
    Region r;
    Symbol s;

    bufflush(bt);
    r.base = tell(bt->fd);
    bemitPackedSyms(bt, syms, startsym);
    bufflush(bt);
    r.size = tell(bt->fd) - r.base;
    return r;
}
```

`bemitPackedSyms` emits a table of packed symbols. `mill` preserves the complete representation of a symbol provided by `mlcc`. This is probably overkill, but much of the info is necessary for optmizations and code generation. Because `mlcc`'s representation of symbols is complex and large, symbols are packed into a denser, variable-sized representation on disk. Every on-disk symbol shares some common information, which is picked in `packSym`. `packAuxSym` produces an auxilliary symbol that pickles extra data required by more complex symbols.

58b ⟨*binary definitions* 43b⟩+≡

```
static void bemitPackedSyms(bt, syms, startsym) BaseTable *bt; MTable syms; {
    Psymbol p;
    PAuxsymbol aux;
    Symbol s, id;
    int i, enums;
    unsigned short    sidx;
    unsigned short    len;

    len = syms.maxelem+1-startsym;
    assert(len >= 0);
    bufwrite(bt, (char *)&len, SHORTSZ);
    gentable(syms, i, startsym) {
        s = tableentry(Symbol *, syms, i);
        packSym(s, &p);
        bufwrite(bt, (char *)(&p), PSYMSZ);
        if (packAuxSym(s, &aux)) {
            bufwrite(bt, (char *)(&aux), PAUXSYMSZ);
            bt->sumsizes.auxsyms++;
        }
        if (p.field)     /* emit fields */
            bemitFields(bt, s->u.s.cfields, s->u.s.vfields, s->u.s.flist);
        else if (!p.temporary && p.enumsym_or_cse) { /* emit enum identifiers */
            bufwrite(bt, (char *)&(menum(s)), SHORTSZ);
            for (enums = 0; enums < menum(s) ; enums++) {
                id = s->u.idlist[enums];
                sidx = stridx(id->name, strlen(id->name));
                bufwrite(bt, (char *)&sidx, SHORTSZ);
                bufwrite(bt, (char *)&id->u.c.v.i, sizeof(int));
            }
        }
    }
}
```

59 ⟨*binary definitions* 43b⟩+≡

```
static int packAuxSym(s, p) Symbol s; PAuxsymbol *p; {
    int auxsym = 1;
    if (s->scope == CONSTANTS)
        p->v = s->u.c.v;
    else if (s->computed) {            /* check if computed before global */
        p->addr.offset = maddr(s).offset;
        p->addr.sym = maddr(s).symid;
    }
    else if (s->type && isfunc(s->type)) {
        p->func.auxtype = msym(s).auxtype ? msym(msym(s).auxtype).id : 0;
        if (mfunc(s)) {
            p->func.ncalls = mfunc(s)->ncalls;
            p->func.dagsize = mfunc(s)->dagsize;
            p->func.leaf = mfunc(s)->leaf;
            p->func.indircall = mfunc(s)->indircall;
        }
    }
    else if (globalsym(s)) {
        p->glbstat.auxtype = msym(s).auxtype ? msym(msym(s).auxtype).id : 0;
        p->glbstat.seg = (unsigned char)s->u.seg;
        p->glbstat.dataloc = mglob(s).dataloc;
    }
    else if (msym(s).auxtype)
        p->auxtype = msym(s).auxtype ? msym(msym(s).auxtype).id : 0;
    else auxsym = 0;
    return auxsym;
}
```

`packSym` returns a compacted representation of symbol `s`.

60a ⟨*binary definitions* 43b⟩+≡

```
static void packSym(s, p) Symbol s; Psymbol *p; {
    bzero((char *)p, PSYMSZ);
    if (s->type) {
        assert(s->type->x.typeno);
        p->type = s->type->x.typeno;
    }
    else p->type = 0;
    p->stridx = msym(s).stridx;
    p->ref = s->ref;
    p->scope = s->scope;
    p->class = s->sclass;
    p->temporary = s->temporary;
    p->generated = s->generated;
    p->computed = s->computed;
    p->addressed = s->addressed;
    p->structarg = s->structarg;
    p->dataseg = msym(s).dataseg;
    p->downarg = msym(s).downarg;
    p->typesym = msym(s).typesym;
    p->field = msym(s).field;
    if (p->temporary) p->enumsym_or_cse = mtemp(s).cse; /* HACK !!! */
    else p->enumsym_or_cse = msym(s).enumsym;
    p->written = msym(s).written;
    p->auxtype = msym(s).auxtype ? 1 : 0;
}
```

`bemitFields` emits a compact representation of a list of struct/union fields.

60b ⟨*binary definitions* 43b⟩+≡

```
static void bemitFields(bt, cfields, vfields, flist) BaseTable *bt; Field flist; {
    Pfield p;

    assert(flist);
    if (flist->name) p.nameidx = stridx(flist->name, strlen(flist->name));
    else p.nameidx = 0;
    p.type = (assert(flist->type && flist->type->x.typeno), flist->type->x.typeno);
    p.offset = flist->offset;
    p.bitsize = flist->bitsize;
    p.lsb = flist->lsb;
    p.cfields = cfields;
    p.vfields = vfields;
    p.link = (flist->link ?  1 : 0);
    bufwrite(bt, (char *)&p, PFIELDSZ);
    if (flist->link) bemitFields(bt, cfields, vfields, flist->link);
}
```

60c ⟨*external declarations* 45e⟩+≡

```
dclproto(void emitSymbol,(int, Symbol, char *));
dclproto(void emitSymbols,(int, MTable, char *));
```

60d ⟨*local declarations* 45d⟩+≡

```
dclproto(static void emitSymHead,(int, Symbol));
```

61 ⟨*definitions* 45f⟩+≡

```
static void emitSymHead(fd, s) Symbol s; {

    if (!fd) return;
    fprint(fd, "SYM ", s);
    emitSymid(fd, msym(s).id);
    fprint(fd, " *%s**%s* ", s->name, s->x.name ? s->x.name : "");

    fprint(fd, "SEG*%d* LOC*%d* ", s->u.seg, mglob(s).dataloc);
    fprint(fd, "%d ", s->scope);
    switch(s->sclass) {
    case AUTO:      fprint(fd, "%s ", "auto"); break;
    case EXTERN:    fprint(fd, "%s ", "extern"); break;
    case REGISTER:  fprint(fd, "%s ", "register"); break;
    case STATIC:    fprint(fd, "%s ", "static"); break;
    case TYPEDEF:   fprint(fd, "%s ", "typedef"); break;
    default: break;
    }
    if (s->temporary)  fprint(fd, "temp ");
    if (mtemp(s).cse)  fprint(fd, "cse ");
    if (s->addressed)  fprint(fd, "addr'd ");
    if (s->generated)  fprint(fd, "gen'd ");
    if (s->computed)   fprint(fd, "comp'd ");
    if (msym(s).announced)  fprint(fd, "annc'd ");
    if (msym(s).downarg) fprint(fd, "downarg ");
    if (msym(s).dataseg) fprint(fd, "dataseg ");
    if (msym(s).typesym) fprint(fd, "typesym ");
    if (msym(s).field)   fprint(fd, "fieldsym ");
    if (msym(s).enumsym) fprint(fd, "enumsym ");
    if (msym(s).written) fprint(fd, "written ");
    if (msym(s).auxtype) {
        fprint(fd, "auxtype:");
        emitSymid(fd, msym(msym(s).auxtype).id);
    }
    if (s->type && isfunc(s->type) && mfunc(s)) {
        fprint(fd, "called%d ", mfunc(s)->ncalls);
        if (mfunc(s)->leaf) fprint(fd, "leaf ");
        if (mfunc(s)->indircall) fprint(fd, "indircall ");
        fprint(fd,"size%d ", mfunc(s)->dagsize);
    }
    if (s->scope == CONSTANTS) fprint(fd, "VAL %d ", s->u.c.v.i);
    if (s->type) fprint(fd, "type %d ", s->type->x.typeno);
}
```

62a ⟨*definitions* 45f⟩+≡

```
void emitSymbol(fd, s, tab) Symbol s; char *tab; {
    if (!fd) return;
    fprint(fd, tab);
    emitSymHead(fd, s);
    fprint(fd, "\n");
}

void emitSymbols(fd, syms, tab) MTable syms; char *tab; {
    Symbol s;
    int    i;

    if (!fd) return;
    fprint(fd, "\n");
    gentable(syms, i, 1) {
        s = tableentry(Symbol *, syms, i);
        emitSymbol(fd, s, tab);
    }
}
```

3.7 Types

bemitTypes emits a type table and returns its region info.

62b ⟨*binary local declarations* 46a⟩+≡

```
dclproto(static void     bemitTypes,(BaseTable *));
dclproto(static void     bemitPackedTypes,(BaseTable *));
dclproto(static void     packType,(Type, Ptype *));
```

62c ⟨*binary definitions* 43b⟩+≡

```
static void bemitTypes(bt) BaseTable *bt; {

    bufflush(bt);
    bt->pbt.typesReg.base = tell(bt->fd);
    bemitPackedTypes(bt);
    bufflush(bt);
    bt->pbt.typesReg.size = tell(bt->fd) - bt->pbt.typesReg.base;
}
```

`bemitPackedTypes` emits a table of packed types.

63a ⟨*binary definitions* 43b⟩+≡
```
static void bemitPackedTypes(bt) BaseTable *bt; {
    Ptype p;
    Type t, *pt;
    Symbol id;
    int i;
    unsigned short len;

    /* Emit size of table */
    len = bt->typetab.maxelem-NBASICTYPES;
    assert(len >= 0);
    bufwrite(bt, (char *)&len, SHORTSZ);
    gentable(bt->typetab, i, NBASICTYPES+1) {
        t = tableentry(Type *, bt->typetab, i);
        packType(t, &p);
        bufwrite(bt, (char *)(&p), PTYPESZ);
        if (isfunc(t) && t->x.nprotos > 0) {
            for (pt = t->u.f.proto; *pt; pt++)
                bufwrite(bt, (char *)&((*pt)->x.typeno), SHORTSZ);
        }
    }
}
```

`packType` returns a packed representation of type `t`.

63b ⟨*binary definitions* 43b⟩+≡
```
static void packType(t, p) Type t; Ptype *p; {
    bzero((char *)p, PTYPESZ);

    assert(p->op < UCHAR_MAX && p->align < UCHAR_MAX && p->size < USHRT_MAX);
    p->op = (unsigned char)t->op;
    p->align = (unsigned char)t->align;
    p->size = t->size;
    if (t->type) p->type = t->type->x.typeno;
    else p->type = 0;
    if (isfunc(t))
        p->u.nprotos = t->x.nprotos;
    else if (t->u.sym) {
        assert(msym(t->u.sym).id);
        p->u.sym = msym(t->u.sym).id;
    }
    else p->u.sym = 0;
}
```

63c ⟨*external declarations* 45e⟩+≡
```
dclproto(void emitTypes,(int, MTable));
```

64a ⟨*definitions* 45f⟩+≡

```
void emitTypes(fd, types) MTable types; {
    Type t, *tt;
    int i;

    if (!fd) return;
    gentable(types, i, 1) {
        t = tableentry(Type *, types, i);
        fprint(fd, "TYPE%d:", t->x.typeno);
        printtype(t, fd);
    }
}
```

3.8 Miscellaneous

`bemitMTable` emits an `MTable` in the binary form: `<unsigned short len> T[0] T[1] ... T[len]`.

64b ⟨*binary local declarations* 46a⟩+≡

```
dclproto(static int     bemitMTable,(BaseTable *, MTable, int));
dclproto(static int     bemitSMTable,(BaseTable *, MTable));
```

64c ⟨*binary definitions* 43b⟩+≡

```
static int bemitMTable(bt, t, esz) BaseTable *bt; MTable t; {
    unsigned short len;
    int sz;

    assert(t.maxelem+1 < USHRT_MAX);
    len = (unsigned short)t.maxelem+1;
    bufwrite(bt, (char *)&len, SHORTSZ);
    sz = SHORTSZ;
    if (len != 0) {
        assert(t.table && t.elemsz);
        assert(esz == t.elemsz);
        bufwrite(bt, (char *)t.table, t.elemsz*len);
        sz += t.elemsz*len;
    }
    return sz;
}
```

64d ⟨*binary definitions* 43b⟩+≡

```
static int bemitSMTable(bt, t) BaseTable *bt; MTable t; {
    unsigned short len;
    int sz;

    assert(t.maxelem+1 < USHRT_MAX);
    len = (unsigned short)t.maxelem+1;
    bufwrite(bt, (char *)&len, SHORTSZ);
    sz = SHORTSZ;
    if (len != 0) {
        assert(t.table && t.elemsz);
        assert(t.elemsz == SHORTSZ);
        bufwrite(bt, (char *)t.table, SHORTSZ*len);
        sz += SHORTSZ*len;
    }
    return sz;
}
```

`bemitString` emits the binary string `<LENGTH><STRING>` for a string not in the string region.

65a ⟨*binary local declarations* 46a⟩+≡
```
dclproto(static void     bemitString,(int, char *));
```

65b ⟨*binary definitions* 43b⟩+≡
```
static void bemitString(fd, s) char *s; {
    unsigned short len;
    long offset;

    if (!fd) return;
    len = strlen(s);
    write(fd, (char *)(&len), SHORTSZ);
    write(fd, (char *)s, len);
}
```

`emitSummary` summarizes the sizes of all the regions in the `mill` object file.

65c ⟨*external declarations* 45e⟩+≡
```
dclproto(void emitSummary,(int, BaseTable *));
```

65d ⟨*definitions* 45f⟩+≡
```
void emitSummary(fd, bt) BaseTable *bt; {
    int totalsz = 0;

    if (!fd) return;
    fprint(fd, "%s\n", bt->filename);
    fprint(fd, "File Header\t%d\n", strlen(bt->filename) + PCKDBASETBLSZ);
    totalsz += strlen(bt->filename) + PCKDBASETBLSZ;
    fprint(fd, "Proc Descs\t%d\t%d\t%d\n", bt->pbt.procindexReg.size,
            bt->pbt.procindexReg.size/PROCIDXSZ, PROCIDXSZ);
    totalsz += bt->pbt.procindexReg.size;
    fprint(fd, "Procedures\t%d\n", bt->sumsizes.procs);
    totalsz += bt->sumsizes.procs;
    fprint(fd, "Flow Graphs\t%d\n", bt->sumsizes.flowgraphs);
    totalsz += bt->sumsizes.flowgraphs;
    fprint(fd, "Call Graphs\t%d\n", bt->pbt.callgraphReg.size);
    totalsz += bt->pbt.callgraphReg.size;
    fprint(fd, "Local Symbol\t%d\t%d\t%d\n", bt->sumsizes.localsyms,
                bt->sumsizes.nlocalsyms, PSYMSZ);
    totalsz += bt->sumsizes.localsyms;
    fprint(fd, "External Symbol\t%d\t%d\t%d\n", bt->pbt.glbsymsReg.size,
               tablesize(bt->glbsymtab), PSYMSZ);
    totalsz += bt->pbt.glbsymsReg.size;
    fprint(fd, "Aux Symbols\t%d\n", bt->sumsizes.auxsyms);
    fprint(fd, "Types\t\t%d\t%d\t%d\n", bt->pbt.typesReg.size,
               tablesize(bt->typetab), PTYPESZ);
    totalsz += bt->pbt.typesReg.size;
    fprint(fd, "Data\t\t%d\n", bt->pbt.dataReg.size);
    totalsz += bt->pbt.dataReg.size;
    fprint(fd, "Code\t\t%d\n", bt->sumsizes.code);
    totalsz += bt->sumsizes.code;
    fprint(fd, "Strings\t\t%d\n", bt->pbt.stringReg.size);
    totalsz += bt->pbt.stringReg.size;
    fprint(fd, "Total\t\t%d\n", totalsz);
    fprint(fd, "Total writes %d\n", totwrites);
}
```

66a ⟨*aemit.h* 66a⟩≡
```
#ifndef INCLUDE_AEMITH
#define INCLUDE_AEMITH

⟨external declarations 45e⟩
#endif
```

This code is written to file aemit.h.

66b ⟨*aemit.c* 66b⟩≡
```
#include <c.h>
#include <bmill.h>
#include <aemit.h>

⟨local declarations 45d⟩
⟨definitions 45f⟩
```

This code is written to file aemit.c.

66c ⟨*bemit.h* 66c⟩≡
```
#ifndef INCLUDE_BEMITH
#define INCLUDE_BEMITH
#include <string.h>
⟨binary external declarations 43a⟩
dclproto(void bemitStrings,(BaseTable *));
dclproto(void bemitLStrings,(BaseTable *, Strings *));
#endif
```

This code is written to file bemit.h.

66d ⟨*bemit.c* 66d⟩≡
```
#include <stdio.h>
#include <c.h>
#include <bmill.h>
#include <buf.h>
#include <string.h>
#include <aemit.h>
#include <bemit.h>

⟨binary local declarations 46a⟩
⟨binary definitions 43b⟩
```

This code is written to file bemit.c.

3.9 Pickling and unpickling mill instructions

Pickling converts the dynamic representation of `mill` instructions (`Node`s) into a flattened, on-disk representation.

67a ⟨*local declarations* 67a⟩≡

```
dclproto(static int gendagcode, (Node, MFunction *, int, FGNode, FGNode));
dclproto(static int readescape,(int, unsigned short *, MTable));
dclproto(static int readsyms,(int, unsigned int, unsigned short *, unsigned short *, MTable));
dclproto(static int writeescape,(int, unsigned short, MTable *));
dclproto(static int writesyms,(int, unsigned int, unsigned short, unsigned short, MTable *));
dclproto(extern int newsym,(Symbol, MTable *, FGNode, FGNode));
```

This definition is continued in chunks 72b, 73b, and 75b.
This code is used in chunk 78.

`genfgnode` generates `mill` code for the instructions in `fgn`.

67b ⟨*definitions* 67b⟩≡

```
void genfgnode(fgn, mfn) FGNode fgn; MFunction *mfn;{
    Node p;
    int low, cdagid;
    MTable *mt;

    /* Add nodes to flow graph as necessary */
    cdagid = low = tablesize(mfn->dagtable);
    for (p = fgn->node; p; p = p->link) {
        cdagid = genmillcode(p, mfn, cdagid, fgn, fgn);
        mt = &fgn->dagnodes;
        adddagrange(mt, low, cdagid - 1);
        low = cdagid;
    }
    fgn->node = (Node)0;
    fgn->codelist = 1;
}
```

This definition is continued in chunks 68–77.
This code is used in chunk 78.

gendagcode pickles a single **Node** by pickling each of its children (bottom-up, post-order) and then by mapping **p**'s opcode into a small integer. The pickled **Node** is emitted into **dagtable**. Along the way, **p** is inspected to acquire useful information about symbols.

68 ⟨*definitions* 67b⟩+≡

```
static int gendagcode(p, mfn, id, entry, fgn) Node p; MFunction *mfn; FGNode entry, fgn; {
    Symbol gsym;
    MDag mdag;
    Node q;

    p->x.dagid = id;
    mdag.opcode = p->op;
    mdag.sym0 = p->syms[0] ? newsym(p->syms[0], &(mfn->localsymtab), entry, fgn) : 0;
    mdag.sym1 = p->syms[1] ? newsym(p->syms[1], &(mfn->localsymtab), entry, fgn) : 0;

    switch (generic(p->op)) {
    case EQ: case GE: case GT: case LE: case LT: case NE:
    case ADD:  case MUL: case DIV:  case MOD: case SUB:
    case BAND: case BOR: case BXOR: case LSH: case RSH:
    case INDIR: case BCOM: case NEG:
    case CVC:   case CVD:  case CVF: case CVI:   case CVP:  case CVS: case CVU:
    case JUMP:
        id = writedag(id, mdag, &(mfn->dagtable));
        break;
    case RET:
        ⟨determine if p returns a function 69a⟩
        id = writedag(id, mdag, &(mfn->dagtable));
        break;
    case ASGN:
        if (isaddrop(p->kids[0]->op))
            msym(p->kids[0]->syms[0]).written = 1;
        ⟨determine if p takes address of a function 69b⟩
        /* fall through */
    case ARG:
        id = writedag(id, mdag, &(mfn->dagtable));
        break;
    case ADDRG:
        /*if (p->op == ADDRGP)
            p->syms[0]->addressed = 1;
         fall through */
    case ADDRF: case ADDRL: case CNST: case CALL:
        id = writedag(id, mdag, &(mfn->dagtable));
        break;
    case LABEL: newsym(p->syms[0], &(mfn->localsymtab), entry, fgn);
        break;     /* label in flowgraph node */
    default: assert(0);
    }
    return id;
}
```

69a ⟨*determine if* p *returns a function* 69a⟩≡
```
if (p->op != RETV) {
    q = p->kids[0];
    while (isconvertop(generic(q->op))) q = q->kids[0];
    if (isaddrop(q->op) && isfunc(q->syms[0]->type)) {
        gsym = tableentry(Symbol *, mfn->bt->glbsymtab, symid(msym(q->syms[0]).id));
        mfunc(gsym)->indircall = 1;
    }
}
```
This code is used in chunk 68.

69b ⟨*determine if* p *takes address of a function* 69b⟩≡
```
if (p->op == ASGNP && isaddrop(p->kids[1]->op)) {
    if (isfunc(p->kids[1]->syms[0]->type)) {
        gsym = tableentry(Symbol *, mfn->bt->glbsymtab, symid(msym(p->kids[1]->syms[0]).id));
        mfunc(gsym)->indircall = 1;
    } else p->kids[1]->syms[0]->addressed = 1;
}
```
This code is used in chunk 68.

`genmillcode` linearizes the code list `p` in bottom-up, post-fix order.

69c ⟨*definitions* 67b⟩+≡
```
int genmillcode(p, mfn, dagid, entry, fgn) Node p; MFunction *mfn; FGNode entry, fgn; {
    if (p && !p->x.visited) {
        dagid = genmillcode(p->kids[0], mfn, dagid, entry, fgn);
        dagid = genmillcode(p->kids[1], mfn, dagid, entry, fgn);
        dagid = gendagcode(p, mfn, dagid, entry, fgn);
        p->x.visited = 1;
#ifndef MLD
        {   extern Symbol cfun;
            mfunc(cfun)->dagsize++;
        }
#endif
    }
    return dagid;
}
```

`mapopcode` maps `op` from set of `Node` opcodes into a small integer.

70 ⟨*definitions* 67b⟩+≡

```
unsigned char mapopcode(op) {
    switch (op) {
    case NOP: return 0;
    case CNSTC: return 1;    case CNSTD: return 2;
    case CNSTF: return 3;    case CNSTI: return 4;
    case CNSTP: return 5;    case CNSTS: return 6;
    case CNSTU: return 7;
    case ARGB: return 8;     case ARGD: return 9;
    case ARGF: return 10;
    case ARGI: return 11;    case ARGP: return 12;
    case ASGNB: return 13;   case ASGNC: return 14;
    case ASGND: return 15;   case ASGNF: return 16;
    case ASGNI: return 17;   case ASGNS: return 18;
    case ASGNP: return 19;
    case INDIRB: return 20;  case INDIRC: return 21;
    case INDIRD: return 22;  case INDIRF: return 23;
    case INDIRI: return 24;  case INDIRS: return 25;
    case INDIRP: return 26;
    case CVCI: return 27;    case CVCU: return 28;
    case CVDF: return 29;    case CVDI: return 30;
    case CVFD: return 31;    case CVIC: return 32;
    case CVID: return 33;    case CVIS: return 34;
    case CVIU: return 35;    case CVPU: return 36;
    case CVSI: return 37;    case CVSU: return 38;
    case CVUC: return 39;    case CVUI: return 40;
    case CVUP: return 41;    case CVUS: return 42;
    case NEGD: return 43;    case NEGF: return 44;
    case NEGI: return 45;
    case CALLB: return 46;   case CALLD: return 47;
    case CALLF: return 48;   case CALLI: return 49;
    case CALLV: return 50;
    case LOADB: return 51;   case LOADC: return 52;
    case LOADD: return 53;   case LOADF: return 54;
    case LOADI: return 55;   case LOADP: return 56;
    case LOADS: return 57;   case LOADU: return 58;
    case RETD: return 59;    case RETF: return 60;
    case RETI: return 61;    case RETV: return 62;
    case ADDRGP: return 63;  case ADDRFP: return 64;
    case ADDRLP: return 65;
    case ADDD: return 66;    case ADDF: return 67;
    case ADDI: return 68;    case ADDP: return 69;
    case ADDU: return 70;
    case SUBD: return 71;    case SUBF: return 72;
    case SUBI: return 73;    case SUBP: return 74;
    case SUBU: return 75;
    case LSHI: return 76;    case LSHU: return 77;
    case MODI: return 78;    case MODU: return 79;
    case RSHI: return 80;    case RSHU: return 81;
    case BANDU: return 82;   case BCOMU: return 83;
    case BORU: return 84;    case BXORU: return 85;
    case DIVD: return 86;    case DIVF: return 87;
    case DIVI: return 88;    case DIVU: return 89;
```

```
    case MULD: return 90;    case MULF: return 91;
    case MULI: return 92;    case MULU: return 93;
    case EQD: return 94;     case EQF: return 95;
    case EQI: return 96;    /* case EQU: return 97; */
    case GED: return 98;     case GEF: return 99;
    case GEI: return 100;    case GEU: return 101;
    case GTD: return 102;    case GTF: return 103;
    case GTI: return 104;    case GTU: return 105;
    case LED: return 106;    case LEF: return 107;
    case LEI: return 108;    case LEU: return 109;
    case LTD: return 110;    case LTF: return 111;
    case LTI: return 112;    case LTU: return 113;
    case NED: return 114;    case NEF: return 115;
    case NEI: return 116;    /* case NEU: return 117; */
    case JUMPV: return 118;  LABELV: return 119;
    default: error("Invalid opcode %d\n", op); assert(0);
    }
    return 0;
}
```

unmapopcode is the inverse of mapopcode.

71 ⟨*definitions* 67b⟩+≡

```
int unmapopcode(unsigned char op) {
static int opcodemap[] = { 0, CNSTC, CNSTD, CNSTF, CNSTI,
    CNSTP, CNSTS, CNSTU, ARGB, ARGD, ARGF, ARGI,
    ARGP, ASGNB, ASGNC, ASGND, ASGNF, ASGNI, ASGNS, ASGNP,
    INDIRB, INDIRC, INDIRD, INDIRF,
    INDIRI, INDIRS, INDIRP, CVCI, CVCU, CVDF, CVDI,
    CVFD, CVIC, CVID, CVIS, CVIU, CVPU,
    CVSI, CVSU, CVUC, CVUI, CVUP, CVUS, NEGD,
    NEGF, NEGI, CALLB, CALLD, CALLF, CALLI,
    CALLV, LOADB, LOADC, LOADD, LOADF, LOADI,
    LOADP, LOADS, LOADU, RETD, RETF, RETI,
    RETV, ADDRGP, ADDRFP, ADDRLP, ADDD, ADDF,
    ADDI, ADDP, ADDU, SUBD, SUBF, SUBI, SUBP,
    SUBU, LSHI, LSHU, MODI, MODU, RSHI, RSHU,
    BANDU, BCOMU, BORU, BXORU, DIVD, DIVF, DIVI,
    DIVU, MULD, MULF, MULI, MULU, EQD, EQF, EQI, 0,
    GED, GEF, GEI, GEU, GTD, GTF, GTI,
    GTU, LED, LEF, LEI, LEU, LTD, LTF, LTI, LTU,
    NED, NEF, NEI, 0, JUMPV, LABELV };

    if (op < 0 || op >= sizeof(opcodemap)/sizeof(int)) return 0;
    return opcodemap[(int)op];
}
```

unmilldag unpickles a **mill** instruction.

72a ⟨*definitions* 67b⟩+≡

```
int unmilldag(id, mdag, dagtab, stack) MDag *mdag; MTable dagtab, *stack; {
unsigned char op;
    int j;

    j =  readdag(id, mdag, dagtab);
    switch (generic(mdag->opcode)) {
    case LABEL: case NOP: break;
    case EQ: case GE: case GT: case LE: case LT: case NE:
        mdag->k1 = popfromtable(int *, stack);
        mdag->k0 = popfromtable(int *, stack);
        break;
    case ASGN:
    case ADD:  case MUL: case DIV:  case MOD: case SUB:
    case BAND: case BOR: case BXOR: case LSH: case RSH:
        mdag->k1 = popfromtable(int *, stack);
        /* fall through */
    case ARG:
    case INDIR: case BCOM: case NEG:
    case CVC:   case CVD:  case CVF:
    case CVI:   case CVP:  case CVS: case CVU:
    case JUMP:
        mdag->k0 = popfromtable(int *, stack);
        if (generic(mdag->opcode) != JUMP)
            addtotable(int *, stack, id);
        break;
    case RET:
        if (mdag->opcode != RETV)
            mdag->k0 = popfromtable(int *, stack);
        break;
    case CALL:
        if (mdag->opcode == CALLB)
            mdag->k1 = popfromtable(int *, stack);
        mdag->k0 =  popfromtable(int *, stack);
        if (generic(mdag->opcode) != CALLB)
            addtotable(int *, stack, id);
        break;
    case ADDRF: case ADDRG: case ADDRL: case CNST:
        addtotable(int *, stack, id);
        break;
    default:
        assert(0);
        break;
    }
    return j;
}
```

72b ⟨*local declarations* 67a⟩+≡

```
#define lowerbyte(_s)    ((unsigned char)((_s) & 0x00ff))
#define upperbyte(_s)    lowerbyte(_s >> 8)
```

`writeescape` writes a 2-byte escape sequence into `dagtab`. `readescape` reads an escape opcode.

73a ⟨*definitions* 67b⟩+≡
```
static int writeescape(int dagid, unsigned short escape, MTable *dagtab) {
    updatetable(unsigned char *, dagtab, dagid, upperbyte(escape));
    updatetable(unsigned char *, dagtab, dagid+1, lowerbyte(escape));
    return dagid += 2;
}
```

73b ⟨*local declarations* 67a⟩+≡
```
#define hibyte(_s) ((_s << 8) & 0xff00)
```

73c ⟨*definitions* 67b⟩+≡
```
static int readescape(i, escape, dagtab) unsigned short *escape; int i; MTable dagtab; {
    unsigned short ubyte, lbyte;

    ubyte = (unsigned short)dagfor(i);
    lbyte = (unsigned short)dagfor(i+1);
    ubyte = hibyte(ubyte) | lbyte;
    *escape = ubyte;
    return i += 2;
}
```

`writesyms` writes a `Node`'s pickled symbols into `dagtab`. `readsyms` unpickles reference to a `Node`'s symbol.

73d ⟨*definitions* 67b⟩+≡
```
static int writesyms(int dagid, unsigned int opcode,
                     unsigned short sym0, unsigned short sym1, MTable *dagtab) {

    switch (generic(opcode)) {
    case ADDRF: case ADDRG: case ADDRL: case CNST:
    case LABEL: case CALL:
    case EQ: case GE: case GT: case LE: case LT: case NE:
        dagid = writeescape(dagid, sym0, dagtab);
        break;
    case ASGN: case ARG:
        if (opcode == ASGNB || opcode == ARGB) {
            dagid = writeescape(dagid, sym0, dagtab);
            dagid = writeescape(dagid, sym1, dagtab);
        }
        break;
    case ADD:  case MUL: case DIV:  case MOD: case SUB:
    case BAND: case BOR: case BXOR: case LSH: case RSH:
    case INDIR: case BCOM: case NEG:
    case CVC:   case CVD:  case CVF:
    case CVI:   case CVP:  case CVS: case CVU:
    case JUMP: case RET: case NOP:
        break;
    default:
        assert(0);
        break;
    }
    return dagid;
}
```

74a ⟨*definitions* 67b⟩+≡
```
static int readsyms(int id, unsigned int opcode,
                    unsigned short *sym0, unsigned short *sym1, MTable dagtab) {
    *sym0 = *sym1 = 0;
    switch (generic(opcode)) {
    case ADDRF: case ADDRG: case ADDRL: case CNST:
    case LABEL: case CALL:
    case EQ: case GE: case GT: case LE: case LT: case NE:
        id = readescape(id, sym0, dagtab);
        break;
    case ASGN: case ARG:
        if (opcode == ASGNB || opcode == ARGB) {
            id = readescape(id, sym0, dagtab);
            id = readescape(id, sym1, dagtab);
        }
        else {
            *sym0 = constantsym(btot(opcode)->size);
            *sym1 = constantsym(btot(opcode)->align);
        }
        break;
    case ADD:  case MUL: case DIV:  case MOD: case SUB:
    case BAND: case BOR: case BXOR: case LSH: case RSH:
    case INDIR: case BCOM: case NEG:
    case CVC:   case CVD:  case CVF:
    case CVI:   case CVP:  case CVS: case CVU:
    case JUMP: case RET: case NOP:
        break;
    default:
        assert(0);
        break;

    }
    return id;
}
```

`writedag` writes a pickled `Node` into `dagtab`. `readdag` reads a pickled `Node`.

74b ⟨*definitions* 67b⟩+≡
```
int writedag(int id, MDag mdag, MTable *dagtab) {
    int j = id;

    updatetable(unsigned char *, dagtab, j, mapopcode(mdag.opcode));
    j = writesyms(j+1, mdag.opcode, mdag.sym0, mdag.sym1, dagtab);
    return j;
}
int readdag(id, mdag, dagtab) MDag *mdag; MTable dagtab; {
    int j = id;

    mdag->opcode = unmapopcode(dagfor(j));
    j = readsyms(j+1, mdag->opcode, &(mdag->sym0), &(mdag->sym1), dagtab);
    mdag->k0 = mdag->k1 = 0;
    return j;
}
```

`nopdag` emits a pickled `NOP` into `dagtab`.

75a ⟨*definitions* 67b⟩+≡
```
void nopdag(id, dagtab) MTable *dagtab; {
    MDag mdag;

    mdag.opcode = unmapopcode(idxtable(unsigned char *, *dagtab, id));
    mdag.sym0 = mdag.sym1 = 0;
    writedag(id, mdag, dagtab);
    tableentry(unsigned char *, *dagtab, id) = NOP;
}
```

75b ⟨*local declarations* 67a⟩+≡
```
Type *basictypes[NBASICTYPES+1] = { 0, &chartype, &doubletype,
    &floattype, &inttype, &longdouble,
    &longtype, &shorttype, &signedchar, &unsignedchar, &unsignedlong,
    &unsignedshort, &unsignedtype, &voidtype, &voidptype};

#define dagfor(_i) (assert((_i) > 0), (_i) <= dagtab.maxelem ? \
                    tableentry(unsigned char *, dagtab, (_i)) : 0)
```

75c ⟨*definitions* 67b⟩+≡
```
Node caller2callee(p, q) Symbol p, q; {
    extern Node forest;

    forest = (Node)0;
    if (p->sclass != q->sclass || p->type != q->type) {
        listnodes(asgn(p, idtree(q)), 0, 0);
    }
    return forest;
}
```

75d ⟨*definitions* 67b⟩+≡
```
unsigned short constantsym(i) {
    int j;
    if (i == 0 || i == 1)
        return GLOBSYM(CONSTANTIDX+i);
    else if (j = ispow2(i))
        return GLOBSYM(CONSTANTIDX+j+1);
    else assert(0);
    return 0;
}
```

`insertintable` inserts value `v` by increasing sorted order in table `t` if it is not already in `t`. `intable` returns the index of value `v` in sorted table `tab` or -1 if not in `tab`. `instringtable` returns 1 if given `str` is in the table of strings.

76a ⟨*definitions* 67b⟩+≡

```
void insertintable(tab, v) MTable *tab; unsigned int v; {
    int i, j;
    unsigned short tmp;

    gentable((*tab), i, 0) {
        tmp = tableentry(unsigned short *, (*tab), i);
        if ((unsigned short)v == tmp) return;
        else if ((unsigned short)v < tmp)
            break;
    }    /* insert at position i */
    for (j = tablesize((*tab)); j > i; j--) {
        updatetable(unsigned short *, tab, j, tableentry(unsigned short *, (*tab), j-1));
    }
    updatetable(unsigned short *, tab, i, (unsigned short)v);
}
```

76b ⟨*definitions* 67b⟩+≡

```
int intable(tab, v) MTable tab; unsigned int v; {
    int i;
    unsigned short tmp;
    gentable(tab, i, 0) {
        tmp = tableentry(unsigned short *, tab, i);
        if ((unsigned short)v == tmp)  return i;
        else if ((unsigned short)v < tmp) return -1;
    }
    return -1;
}
```

76c ⟨*definitions* 67b⟩+≡

```
int instringtable(tab, str) MTable tab; char *str; {
    int i;
    char *tmp;
    gentable(tab, i, 0) {
        tmp = tableentry(char **, tab, i);
        if (str == tmp)  return 1;
    }
    return 0;
}
```

76d ⟨*definitions* 67b⟩+≡

```
void mergesymtab(dest, source) MTable *dest, source; {
    int i;
    gentable(source, i, 0) {
        addtotable(unsigned short *, dest, tableentry(unsigned short *, source, i));
    }
}
```

mergesymbols merges symbol information by updating first symbol.

77a ⟨*definitions* 67b⟩+≡
```
void mergesymbols(p, q) Symbol p, q; {
    int i;
    Symbol comp;
    CallGraphNode cgn;

    p->addressed |= q->addressed;
    if (p->type && isfunc(p->type)) {
        if (mfunc(q)->cgnid) mfunc(p)->cgnid = mfunc(q)->cgnid;
        mfunc(p)->ncalls += mfunc(q)->ncalls;
        mfunc(p)->leaf |= mfunc(q)->leaf;
        mfunc(p)->indircall |= mfunc(q)->indircall;
        gentable(mfunc(q)->callgraph, i, 0) {
            cgn = tableentry(CallGraphNode *, mfunc(q)->callgraph, i);
            addtotable(CallGraphNode *, (&mfunc(p)->callgraph), cgn);
        }
    }
    msym(p).dataseg |= msym(q).dataseg;
    msym(p).downarg |= msym(q).downarg;
    msym(p).written |= msym(q).written;
}
```

77b ⟨*definitions* 67b⟩+≡
```
unsigned short newlocalsym(Symbol sym, MTable *localsymtab, MTable *symids) {
    unsigned short id;
    id = max(1,tablesize(*localsymtab));
    updatetable(Symbol *, localsymtab, id, sym);
    if (symids)
        addtotable(unsigned short *, symids, id);
    return id;
}
```

77c ⟨*definitions* 67b⟩+≡
```
Symbol newtempsym(type, localsymtab, symids) Type type; MTable *localsymtab, *symids; {
        Symbol p;

    assert(type);
    p = (Symbol)yyalloc(sizeof *p, (*localsymtab->arena));
    BZERO(p, struct symbol);
    p->name = stringd(genlabel(1));
    p->scope = LOCAL;
    p->sclass = AUTO;
    p->type = type;
    p->generated = 1;
    msym(p).id = newlocalsym(p, localsymtab, symids);
    return p;
}
```

78 ⟨*mill.c* 78⟩≡
```
#include <c.h>
#include <simp.h>
#include <bemit.h>

⟨local declarations 67a⟩
⟨definitions 67b⟩
```

This code is written to file `mill.c`.

4 Reading mill object files

This section contains all of the procedures for reading **mill** object code. The sections appear in the same order as for emitting **mill** code. The two sections should be merged.

4.1 Modules

79a ⟨*definitions* 79a⟩≡
```
void breadModuleHdr(bt) BaseTable *bt; {
    char *filename, *machtype;
    long baseTblLoc;

    fileOpen(bt);
    lseek(bt->fd, bt->offset, SEEK_SET);
    filename = breadString(bt->fd);
/*    machtype = breadString(bt->fd); */
    read(bt->fd, (unsigned char *)&(bt->pbt), PCKDBASETBLSZ);
    bt->filename = filename;
}
```
This definition is continued in chunks 79–90.
This code is used in chunk 91.

breadProgram reads the procedure index, string region, type region, global symbol table and data region for the program in **BaseTable bt**.

79b ⟨*definitions* 79a⟩+≡
```
void breadProgram(bt) BaseTable *bt; {
    assert(bt->fd);

    bt->arena = newarena();
    bt->maxbufsz = bt->pbt.maxregsz;
    breadProcIndex(bt);
    breadStrings(bt);
    breadTypes(bt);
    breadRegion(bt, bt->pbt.glbsymsReg);
    breadSymbols(bt, bt->typetab, &(bt->glbsymtab), &(bt->arena), 1);
    backpatchTypes(bt->typetab, bt->glbsymtab);
    backpatchAuxTypes(&(bt->glbsymtab), &(bt->glbsymtab));
    breadCallgraph(bt);
}
```

`breadProgramSize` emits the runtime size of program.

80a ⟨*definitions* 79a⟩+≡

```
int breadProgramSize(bt) BaseTable *bt; {
    int arenasz, tmpsz, totsz = 0;

    arenasz = arenasize(bt->arena);
    fprint(2,"%s:\n", bt->filename);
    fprint(2,"maxbuf  %d\n", bt->maxbufsz);
    totsz += bt->maxbufsz;
    tmpsz = (tablesize(bt->glbsymtab) * sizeof(struct symbol)) + (tablesize(bt->procindex) * MFNSYMSZ);
    fprint(2,"globalsyms  %d\n", tmpsz);
    totsz += tmpsz;
    tmpsz = tablesize(bt->typetab) * sizeof(struct type);
    fprint(2,"types  %d\n", tmpsz);
    totsz += tmpsz;
    tmpsz = tablesize(bt->procindex) * PROCIDXSZ;
    fprint(2,"procindex  %d\n", tmpsz);
    totsz += tmpsz;
    tmpsz = tablesize(bt->dataGlobs) * SHORTSZ;
    fprint(2,"datasyms  %d\n", tmpsz);
    totsz += tmpsz;
    tmpsz = tablesize(bt->datatab) * DATAITEMSZ;
    fprint(2,"datasegment  %d\n", tmpsz);
    totsz += tmpsz;
    fprint(2,"arena  %d\n", arenasz);
    fprint(2,"other  %d\n", arenasz > totsz ? arenasz - totsz : 0);
    return arenasz;

}
```

4.2 Call Graphs

80b ⟨*local declarations* 80b⟩≡

```
dclproto(static void   breadCallgraph,(BaseTable *bt));
```

This definition is continued in chunks 81–83, 86b, and 88–90.
This code is used in chunk 91.

81a ⟨*definitions* 79a⟩+≡
```
static void breadCallgraph(bt) BaseTable *bt;  {
    int i,j;
    Symbol fn;
    CallGraphNode *cgn;
    ProcIndex pi;

    breadRegion(bt, bt->pbt.callgraphReg);
    gentable(bt->procindex, i, 0) {
        pi = tableentry(ProcIndex *, bt->procindex, i);
        fn = (Symbol) idxtable(Symbol *, bt->glbsymtab, symid(pi.id));
        breadMTable(bt, &(mfunc(fn)->callgraph), CALLGRNODESZ, &(bt->arena));
        gentable(mfunc(fn)->callgraph, j, 0) {
            cgn = &(tableentry(CallGraphNode *, mfunc(fn)->callgraph, j));
            cgn->inline = 0;
        }
    }
}
```

4.3 Data

`breadData` reads the data region.

81b ⟨*definitions* 79a⟩+≡
```
void breadData(bt) BaseTable *bt; {
    breadRegion(bt, bt->pbt.dataReg);
    breadMTable(bt, &bt->dataGlobs, SHORTSZ, &(bt->arena));
    breadMTable(bt, &bt->datatab, DATAITEMSZ, &(bt->arena));
}
```

4.4 Flow Graphs

`breadFlowgraph` reads a list of flowgraph nodes and return its length.

81c ⟨*local declarations* 80b⟩+≡
```
dclproto(static void   breadFlowgraph,(BaseTable *, MTable *, Arena *));
dclproto(static FGNode breadFlowgraphnode,(BaseTable *, int, Arena *));
```

81d ⟨*definitions* 79a⟩+≡
```
static void breadFlowgraph(bt, flowgraph, arena) BaseTable *bt; MTable *flowgraph; Arena *arena; {
    int i;
    FGNode fgnode;
    unsigned short len;

    bufread(bt, (unsigned char *)&len, SHORTSZ);
    newtable(flowgraph, len, sizeof(FGNode), arena);
    for (i = 1; i < len; i++) {
        fgnode = breadFlowgraphnode(bt, i, arena);
        updatetable(FGNode *, flowgraph, i, fgnode);
    }
}
```

`breadFlowgraphnode` reads a single flow graph node.

82a ⟨*definitions* 79a⟩+≡
```
static FGNode breadFlowgraphnode(bt, id, arena) BaseTable *bt; Arena *arena; {
    FGNode     fgnode;
    unsigned short    len;
    PFGNode pfg;

    fgnode = (FGNode)yyalloc(sizeof(*fgnode), (*arena));
    BZERO(fgnode, *fgnode);
    cleartable(fgnode->tempsymids, arena);
    cleartable(fgnode->succs, arena);
    cleartable(fgnode->preds, arena);
    cleartable(fgnode->localsymids, arena);
    fgnode->tempsymids.elemsz = fgnode->localsymids.elemsz = SHORTSZ;
    bufread(bt, (char *)&pfg, PFGSZ);
    fgnode->id = pfg.id;
    if (fgnode->id != id)
        error("Invalid flow graph node id %d != %d\n", fgnode->id, id);
    fgnode->type = pfg.type;    /* Read type */
    fgnode->codelist = pfg.gennode;
    if (pfg.temps) breadSMTable(bt, &(fgnode->tempsymids), arena);
    if (pfg.succs) breadSMTable(bt, &fgnode->succs, arena);    /* Read successors */
    if (pfg.preds) breadSMTable(bt, &fgnode->preds, arena);    /* Read predecessors */
    breadMTable(bt, &(fgnode->dagnodes), RANGESZ, arena);    /* Read dagnode table */
    bufread(bt, (char *)&(fgnode->labelsym), SHORTSZ);
    switch(fgnode->type) {
    case CallN:    bufread(bt, (char *)&(callN(fgnode).argnid), SHORTSZ);
            bufread(bt, (char *)&(callN(fgnode).dagn), SHORTSZ);
            bufread(bt, (char *)&(callN(fgnode).retsymid), SHORTSZ);
            break;
    case EntryN:    breadMTable(bt, &(fgnode->localsymids), SHORTSZ, arena);
            break;
    case ExitN:    bufread(bt, (char *)&(exitN(fgnode).entryNid), SHORTSZ);
            break;
    case MultitestN:
            bufread(bt, (char *)&(multitestN(fgnode).sym), SHORTSZ);
            bufread(bt, (char *)&(multitestN(fgnode).deflab), SHORTSZ);
            bufread(bt, (char *)&(multitestN(fgnode).size), SHORTSZ);
            breadMTable(bt, &(multitestN(fgnode).values), LONGSZ, arena);
            breadMTable(bt, &(multitestN(fgnode).labels), SHORTSZ, arena);
            break;
    default: break;
    }
    return fgnode;
}
```

4.5 Procedures

`breadProcIndex` reads the index of procedures in `bt`.

82b ⟨*local declarations* 80b⟩+≡
```
dclproto(static void   breadProcIndex,(BaseTable *));
```

83a ⟨*definitions* 79a⟩+≡
```
static void breadProcIndex(bt) BaseTable *bt; {
    breadRegion(bt, bt->pbt.procindexReg);
    breadMTable(bt, &bt->procindex, PROCIDXSZ, &(bt->arena));
}
```

`breadProcedure` reads a single procedure.

83b ⟨*definitions* 79a⟩+≡
```
void breadProcedure(fn) Symbol fn; {
    unsigned short    id, len = 0;
    Region    procreg;
    char    *buf;
    BaseTable *bt = mfn(fn).bt;
    ProcIndex pi;

    if (mfunc(fn)->arena == 0)
        mfunc(fn)->arena = newarena();
    pi = idxtable(ProcIndex *, bt->procindex, mfunc(fn)->procid);
    procreg.base = bt->pbt.procsReg.base + pi.offset;
    procreg.size = pi.size;
    breadRegion(bt, procreg);
    bufread(bt, (char *)&id, SHORTSZ);
    if (id != pi.id) {
        error("Invalid procedure id %d != %d\n", symid(id), symid(pi.id));
        exit(1);
    }
    breadSymbols(bt, bt->typetab, &(mfn(fn).localsymtab), &(mfunc(fn)->arena), 0);
if (dbgflag) fprint(2,"%d\t", tablesize(mfn(fn).localsymtab));
    backpatchAuxTypes(msym(fn).symtab, &(mfn(fn).localsymtab));
    mfn(fn).sym = fn;
    breadSMTable(bt, &(mfn(fn).caller), &(mfunc(fn)->arena));
    breadSMTable(bt, &(mfn(fn).callee), &(mfunc(fn)->arena));
    breadFlowgraph(bt, &(mfn(fn).flowgraph), &(mfunc(fn)->arena));
if (dbgflag) fprint(2,"%d\t", tablesize(mfn(fn).flowgraph));
    breadMTable(bt, &(mfn(fn).dagtable), CHARSZ, &(mfunc(fn)->arena));
    cleartable(mfn(fn).nodetable, &(mfunc(fn)->arena));
    mfunc(fn)->cached = 1;
}
```

4.6 Symbols

`breadSymbols` reads in a table of symbols and returns its size.

83c ⟨*local declarations* 80b⟩+≡
```
dclproto(static Symbol unpackSym,(BaseTable *, Psymbol *, MTable, MTable, int, Arena *));
dclproto(static void   unpackAuxSym,(BaseTable *, Symbol, PAuxsymbol *, MTable, Arena *));
dclproto(static void   breadSymbols,(BaseTable *, MTable, MTable *, Arena *, int));
```

84 ⟨*definitions* 79a⟩+≡

```
static void breadSymbols(bt, typetab, symtab, arena, global)
        BaseTable *bt; MTable typetab, *symtab; Arena *arena; {
    Symbol    s, s1;
    Psymbol p, p1;
    PAuxsymbol aux;
    int     i, j, startsym, cfields, vfields;
    unsigned short     id, sidx;
    unsigned short    len;

    bufread(bt, (char *)&len, SHORTSZ);
    if (global) startsym = NBASICTYPES+NCONSTANTS+1;
    else startsym = 1;
    len += startsym;
    newtable(symtab, len, sizeof(Symbol), arena);
    if (global) {
        for (i = 1; i <= NBASICTYPES; i++) {
            updatetable(Symbol *, symtab, i, (*basictypes[i])->u.sym);
            msym((*basictypes[i])->u.sym).id = i;
            msym((*basictypes[i])->u.sym).typesym = 1;
        }
        updatetable(Symbol *, symtab, CONSTANTIDX, intconst(0));
        for (i = 1, j = CONSTANTIDX+1; j < CONSTANTIDX+NCONSTANTS; i <<= 1, j++)
            updatetable(Symbol *, symtab, j, intconst(i));
    }
    for (i = startsym; i < len; i++) {
        bufread(bt, (char *)&p, PSYMSZ);
        s = unpackSym(bt, &p, *symtab, typetab, global ? GLOBSYM(i) : i, arena);
        if (auxsym(s)) {
            bufread(bt, (char *)&aux, PAUXSYMSZ);
            unpackAuxSym(bt, s, &aux, *symtab, arena);
        }
        updatetable(Symbol *, symtab, i, s);
        msym(s).symtab = symtab;
        if (p.field) {
            s->u.s.flist = breadFields(bt, &cfields, &vfields, typetab);
            s->u.s.cfields = cfields;
            s->u.s.vfields = vfields;
        }
        else if (!p.temporary && p.enumsym_or_cse) {             /* emit enum identifiers */
            bufread(bt, (char *)&(menum(s)), SHORTSZ);
            s->u.idlist = (Symbol *)yyalloc((menum(s) + 1)*sizeof(Symbol), (bt->arena));
            for (j = 0; j < menum(s); j++) {
                s1 = (Symbol)yyalloc(sizeof(struct symbol), (bt->arena));
                BZERO(s1, *s1);
                bufread(bt, (char *)&sidx, SHORTSZ);
                s1->name = mapstridx(bt, sidx);
                bufread(bt, (char *)&(s1->u.c.v.i), sizeof(int));
                msym(s).id = 0;
                s1->type = s->type;
                s->u.idlist[j] = s;
            }
            s->u.idlist[j] = 0;
        }
```

```
    }
}
```

unpackSym unpacks a representation of symbol **p** and stores it in **symtab**.

85 ⟨*definitions* 79a⟩+≡

```
static Symbol unpackSym(bt, p, symtab, typetab, i, arena)
    BaseTable *bt; Psymbol *p; MTable symtab, typetab; Arena *arena; {
    Symbol s;

    s = (Symbol)yyalloc(sizeof(struct symbol), (*arena));
    BZERO(s, *s);
    msym(s).id = i;
    if (p->type != 0)
        s->type = idxtable(Type *, typetab, p->type);
    s->name = mapstridx(bt, p->stridx);
    s->ref = p->ref;
    s->scope = p->scope;
    s->sclass = p->class;
    s->temporary = p->temporary;
    s->generated = p->generated;
    s->computed = p->computed;
    s->addressed = p->addressed;
    s->structarg = p->structarg;
    msym(s).dataseg = p->dataseg;
    msym(s).downarg = p->downarg;
    msym(s).typesym = p->typesym;
    msym(s).field = p->field;
    if (s->temporary) {
        mtemp(s).cse = p->enumsym_or_cse; /* HACK!!!!! */
    } else
        msym(s).enumsym = p->enumsym_or_cse;
    msym(s).written = p->written;
    msym(s).auxtype = (Symbol)p->auxtype;
    if (s->scope == LABELS && !s->computed)
        msym(s).raddr = addr_new(label_new(s->name), 0);
    return s;
}
```

86a ⟨*definitions* 79a⟩+≡

```
static void unpackAuxSym(bt, s, p, symtab, arena)
    BaseTable *bt; Symbol s; PAuxsymbol *p; MTable symtab; Arena *arena; {
    if (s->scope == CONSTANTS)
        s->u.c.v = p->v;
    else if (s->computed) {
        maddr(s).offset = p->addr.offset;
        maddr(s).symid = p->addr.sym;
    }
    else if (s->type && isfunc(s->type)) {
        msym(s).u.func = (MFuncSym)yyalloc(MFNSYMSZ, (*arena));
        BZERO(msym(s).u.func, *(msym(s).u.func));
        mfn(s).bt = bt;
        msym(s).auxtype = (Symbol)p->auxtype;
        mfunc(s)->ncalls = p->func.ncalls;
        mfunc(s)->dagsize = p->func.dagsize;
        mfunc(s)->leaf = p->func.leaf;
        mfunc(s)->indircall = p->func.indircall;
    }
    else if (globalsym(s)) {
        msym(s).auxtype = (Symbol)p->auxtype;
        s->u.seg = (int)p->glbstat.seg;
        mglob(s).dataloc = p->glbstat.dataloc;
    }
    else if (msym(s).auxtype)
        msym(s).auxtype = (Symbol)p->auxtype;
}
```

4.7 Types

`breadTypes` reads in table of types and returns its size. `unpackType` unpacks the representation of type `p` and updates `typetab`.

86b ⟨*local declarations* 80b⟩+≡

```
dclproto(static Type   unpackType,(Ptype *, BaseTable *, int));
dclproto(static void   breadTypes,(BaseTable *));
dclproto(static void   backpatchTypes,(MTable, MTable));
dclproto(static void   backpatchAuxTypes,(MTable *, MTable *));
```

87a ⟨*definitions* 79a⟩+≡
```
static void breadTypes(bt) BaseTable *bt; {
    Type    *typetab, t;
    Ptype     p;
    int     i, j;
    unsigned short     ptid;
    unsigned short  len;
    MTable    *mt;

    breadRegion(bt, bt->pbt.typesReg);
    bufread(bt, (char *)&len, SHORTSZ);
    len += NBASICTYPES;
    newtable(&bt->typetab, len+1, sizeof(Type), &(bt->arena));
    mt = &bt->typetab;
    updatetable(Type *, mt, 0, 0);
    for (i = 1; i <= NBASICTYPES; i++) {
        updatetable(Type *, mt, i, *basictypes[i]);
        idxtable(Type *, bt->typetab, i)->x.typeno = i;
    }
    for (i = NBASICTYPES+1; i <= len; i++) {
        bufread(bt, (char *)&p, PTYPESZ);
        t = unpackType(&p, bt, i);
        updatetable(Type *, mt, i, t);
        if (isfunc(t) && t->x.nprotos > 0) {
            t->u.f.proto = (Type *)yyalloc((t->x.nprotos+1) * sizeof(Type), (bt->arena));
            t->u.f.proto[t->x.nprotos] = 0;
            for (j = 0; j < t->x.nprotos; j++) {
                bufread(bt, (char *)&ptid, SHORTSZ);
                t->u.f.proto[j] = idxtable(Type *, bt->typetab, ptid);
            }
        }
    }
}
```

87b ⟨*definitions* 79a⟩+≡
```
static Type unpackType(p, bt, id) Ptype *p; BaseTable *bt; {
    Type t;

    t = (Type)yyalloc(sizeof *t, (bt->arena));
    BZERO(t, *t);
    t->x.typeno = id;
    t->op = p->op;
    t->align = p->align;
    t->size = p->size;
    if (p->type) {
        t->type = idxtable(Type *, bt->typetab, (int)p->type);
    }
    else t->type = (Type)0;
    if (isfunc(t)) t->x.nprotos = p->u.nprotos;
    else t->x.symno = p->u.sym;
    return t;
}
```

`backpatchTypes` replaces the symbol table place holders in type table from symbol table symtab.

88a ⟨*definitions* 79a⟩+≡
```
static void backpatchTypes(typetab, symtab) MTable typetab, symtab; {
    int i, symno;
    Symbol p;
    Type t;

    gentable(typetab, i, 1) {
        t = tableentry(Type *, typetab, i);
        if (t->x.symno && !t->u.sym) {
            symno = (int)symid(t->x.symno);
            t->u.sym = idxtable(Symbol *, symtab, symno);
        }
    }
}
```

`backpatchAuxTypes` backpatches any references to auxilliary type symbols in `symtab`.

88b ⟨*definitions* 79a⟩+≡
```
static void backpatchAuxTypes(gsymtab, symtab) MTable *gsymtab, *symtab; {
    int i;
    unsigned short id;
    Symbol auxtype, s;

    gentable(*symtab, i, 1) {
        s = tableentry(Symbol *, *symtab, i);
        if (msym(s).auxtype) {
            id = (unsigned short) msym(s).auxtype;
            assert(auxtype = idxtable(Symbol *, *gsymtab, symid(id)));
            msym(s).auxtype = auxtype;
        }
    }
}
```

`breadFields` reads in a list of struct/union fields.

88c ⟨*local declarations* 80b⟩+≡
```
dclproto(static Field  breadFields,(BaseTable *, int *, int *, MTable));
```

89a ⟨*definitions* 79a⟩+≡
```
static Field breadFields(bt, cfields, vfields, typetab) BaseTable *bt; int *cfields, *vfields; MTable typetab; {
    Pfield p;
    Field flist;

    flist = (Field)yyalloc(sizeof(*flist), (bt->arena));
    BZERO(flist, *flist);
    bufread(bt, (char *)&p, PFIELDSZ);
    flist->name = mapstridx(bt, p.nameidx);
    flist->type = idxtable(Type *, typetab, p.type);
    flist->offset = p.offset;
    flist->bitsize = p.bitsize;
    flist->lsb = p.lsb;

    if (p.link)
        flist->link = breadFields(bt, cfields, vfields, typetab);
    else     flist->link = (Field)0;
    *cfields = p.cfields;
    *vfields = p.vfields;
    return flist;
}
```

4.8 Miscellaneous

`breadMTable` reads a table in the form: `<len>T[0],T[1]...T[len]`.

89b ⟨*local declarations* 80b⟩+≡
```
dclproto(static void   breadMTable,(BaseTable *, MTable *, int, Arena *));
dclproto(static void   breadSMTable,(BaseTable *, MTable *, Arena *));
```

89c ⟨*definitions* 79a⟩+≡
```
static void breadMTable(bt, t, esz, arena) BaseTable *bt; MTable *t; Arena *arena; {
    unsigned short len;

    assert(t);
    bufread(bt, (char *)&len, SHORTSZ);
    t->tablesz = len;
    t->maxelem = (int)len-1;
    t->arena = arena;
     t->elemsz = esz;
    if (len != 0) {
        t->table = (Generic *)yyalloc(t->elemsz * (len+1), (*arena));
        assert(t->table && t->elemsz);
        bufread(bt, (char *)t->table, t->elemsz*len);
    }
}
```

90a ⟨*definitions* 79a⟩+≡
```
static void breadSMTable(bt, t, arena) BaseTable *bt; MTable *t; Arena *arena; {
    unsigned short len;

    assert(t);
    bufread(bt, (char *)&len, SHORTSZ);
    t->tablesz = len;
    t->maxelem = (int)len-1;
    t->arena = arena;
    t->elemsz = SHORTSZ;
    if (len != 0) {
        t->table = (Generic *)yyalloc(SHORTSZ * (len+1), (*arena));
        assert(t->table);
        bufread(bt, (char *)t->table, SHORTSZ*len);
    }
}
```

`breadString` reads the string `<len><string>` for a string not in string region.

90b ⟨*local declarations* 80b⟩+≡
```
dclproto(static char   *breadString,(int));
```

90c ⟨*definitions* 79a⟩+≡
```
static char *breadString(fd) {
    char tmpspace[64], *s;
    unsigned short len;

    if (!fd) return 0;
    read(fd, (unsigned char *)(&len), SHORTSZ);
    if (len > 63) s = (char *)alloc(memalign(len+1));
    else s = tmpspace;
    read(fd, (unsigned char *)s, len);
    s[len] = '\0';
    return string(s);
}
```

`breadRegion` reads in region `reg` into the common read buffer.

90d ⟨*definitions* 79a⟩+≡
```
void breadRegion(bt, reg) BaseTable *bt; Region reg; {
    fileOpen(bt);
    assert(bt->fd);
    lseek(bt->fd, reg.base + bt->offset, SEEK_SET);
    bufalloc(bt);
    assert(bufsize(bt) >= reg.size);
    read(bt->fd, (unsigned char *)bt->buf->ptr, reg.size);
}
```

91 ⟨*bread.c* 91⟩≡
```
#include <stdio.h>
#include <c.h>
#include <mld.h>
#include <bmill.h>
#include <buf.h>
#include <string.h>
dclproto(int fileOpen,(BaseTable *));
⟨local declarations 80b⟩
⟨definitions 79a⟩
```
This code is written to file `bread.c`.

5 Simplifying expressions in mlcc and mld

This section contains code simplifying expressions in **mill** code. Most of the rules for simplifying expressions are specified using **BURG** rewrite rules; the rules are translated into a simplifier using **eburs**. The rules simplify addressing expressions, fold constants, and recognize and implify idioms that appear in Modula–3 code. Additional procedures support the **BURG** simplifier.

The specification can be used to simplify **Tree**s, as it used in **mlcc**, or to simplify **Node**s, as it is used in **mld**. Macros (sort of) blur the distinction between the two representations.

5.1 Simplifying Tree and Node expressions

The **BURG** expression simplifier for **Tree**s and **Node**s share the same reducer and simplification rules. The labellers are distinct, because **mld** uses the labelling pass to identify variables that refer to link-time constants.

92 ⟨*local declarations* 92⟩≡
```
dclproto(static TN label,(TN, int));
dclproto(static TN reduce,(TN, Type, int, jmp_buf));
dclproto(static void relabel,(TN));
```

This definition is continued in chunks 95e, 105a, 107d, 109a, and 111a.
This code is used in chunks 113 and 115a.

`mld`'s `label` recursively labels Node `p`. `p` is relabelled if `relabel` is set. Re-labelling is required if link-time constants are identified, because they may trigger simplification of other expressions.

93 ⟨*node definitions* 93⟩≡

```
static TN label(p, relabel) TN p; {
    int nindir = 0, offset = 0;
    TN q;
    Symbol s;
    DataValue datavalue;

    if (p == 0) return 0;
    p->x.rereduce = 0;
    if (STATE(p) && !relabel)  return p;
    ⟨ label kids first  94a⟩
    if (OP(p) != OVFU && OP(p) != OVFB)  /* special overflow operator */
        OP(p) = p->op;
    switch (p->op) {
    case ADDP: {
        ⟨commute dataval for each child of ADDP expression 95a⟩
        if (p->x.dataval) OP(p) = CADDRP;
        /* relabel the kids if they're not used in a caddr expression */
        }
        break;
    case INDIRP: {
        q = p;
        ⟨find leaf node of p 95b⟩
        p->x.dataval = constaddr(nindir, q->x.dataval);
        s = datasymbol(p->x.dataval);
        if (s)  p = new_addrgp(s);
        else if (constoffset(p->x.dataval, 0)) OP(p) = CADDRP;
        }
        break;
    case INDIRF: case INDIRD:
        ⟨ label a floating-point constant  94b⟩
        break;
    case ADDRLP: {
        s = SYMBOL(p);
        if (s->temporary && mtemp(s).cse) {
            datavalue = mtemp(s).datavalue;
            ⟨find binding values for symbol s 95c⟩
            }
        }
        break;
    case ADDRGP: {
        s = SYMBOL(p);
        if (s->scope == GLOBAL && s->type && !isfunc(s->type)) {
            datavalue = mglob(s).datavalue;
            ⟨find binding values for symbol s 95c⟩
            }
        }
        break;
    case CNSTI: ⟨label a CNSTI integer constant 94c⟩
        break;
    case CNSTU: ⟨label a CNSTU unsigned constant 94d⟩
        break;
```

```
        default:
            break;
        }
        STATE(p) = FOLD_STATE(OP(p), LKID(p) ? STATE(LKID(p)) : 0,
                              RKID(p) ? STATE(RKID(p)) : 0);
        assert(STATE(p));
        return p;
    }
```

This definition is continued in chunks 107, 116, and 117.
This code is used in chunk 115a.

94a ⟨ *label kids first* 94a⟩≡

```
    LKID(p) = label(LKID(p), relabel);
    RKID(p) = label(RKID(p), relabel);
```

This code is used in chunks 93 and 96b.

Special labels are used for integer and unsigned values of zero, one, and powers of two and for floating-point constants whose values are stored in the data segment.

94b ⟨ *label a floating-point constant* 94b⟩≡

```
    {
        Symbol s = isconstsym(p);
        if (OP(p) == CNST+optype(p->op)) assert(s);
        if (s != 0) {
            OP(p) = CNST+optype(p->op);
            p->syms[0] = s;
        }
    }
```

This code is used in chunk 93.

94c ⟨*label a* CNSTI *integer constant* 94c⟩≡

```
    {
        int i = VALUE(p).i;
        if (i >= 0 && i <= 1)    /* I0I or I1I */
            OP(p) = (I0 + (i<<4))+I;
        /* ISI or P2SI - small shift value */
        else if (i < 8*inttype->size)
            if (ispow2(i)) OP(p) = P2SI;
            else OP(p) = ISI;
        else if (ispow2(i)) OP(p) = P2I;
    }
```

This code is used in chunks 93 and 96b.

94d ⟨*label a* CNSTU *unsigned constant* 94d⟩≡

```
    {
        unsigned i = VALUE(p).u;
        if (i <= 1)                 /* I0U or I1U */
            OP(p) = (I0 + (i<<4))+U;
        /* ISU or P2SU - small shift value */
        else if (i < 8*unsignedtype->size)
            if (ispow2(i)) OP(p) = P2SU;
            else OP(p) = ISU;
        else if (ispow2(i)) OP(p) = P2U;
    }
```

This code is used in chunks 93 and 96b.

95a ⟨*commute* dataval *for each child of* ADDP *expression* 95a⟩≡
```
  if (RKID(p)->op == CNSTI)
      p->x.dataval = constoffset(LKID(p)->x.dataval, VALUE(RKID(p)).i);
  else if (LKID(p)->op == CNSTI)
      p->x.dataval = constoffset(RKID(p)->x.dataval, VALUE(LKID(p)).i);
```
This code is used in chunk 93.

95b ⟨*find leaf node of* p 95b⟩≡
```
  while (q->op == INDIRP && !q->x.dataval) {
      nindir++;
      q = LKID(q);
  }
```
This code is used in chunk 93.

95c ⟨*find binding values for symbol* s 95c⟩≡
```
  {
      if (s->computed) {
          ⟨find base symbol of computed symbol s 96a⟩
          p->x.dataval = constoffset(datavalue, offset);
          if (p->x.dataval) {
              OP(p) = SADDRP;
              p->x.rereduce = 1;
          }
      } else {
          p->x.dataval = datavalue;
          if (constoffset(datavalue, 0)) {
              OP(p) = SADDRP;
              p->x.rereduce = 1;
          }
      }
  }
```
This code is used in chunk 93.

95d ⟨*external declarations* 95d⟩≡
```
  extern int symconvert;
```
This definition is continued in chunks 100e, 106, 115b, and 116d.
This code is used in chunk 112b.

95e ⟨*local declarations* 92⟩+≡
```
  static TN new_addrgp(Symbol);
```

95f ⟨*definitions* 95f⟩≡
```
  static TN new_addrgp(s) Symbol s;
  {
      TN p;
      symconvert++;
      p = NEWTN(ADDRGP, 0, 0, 0);
      SYMBOL(p) = s;
      return p = label(p, 0);
  }
```
This definition is continued in chunks 96c, 97a, 105b, and 108–110.
This code is used in chunks 113 and 115a.

96a ⟨*find base symbol of computed symbol* `s` 96a⟩≡
```
offset = 0;
while (s->computed) {
    offset += maddr(s).offset;
    assert(s = basesymbol(s));
}
```
This code is used in chunk 95c.

`mlcc`'s `label` recursively labels the Tree `p`, but is simpler than `mld`'s labeller.

96b ⟨*tree definitions* 96b⟩≡
```
static TN label(p, relabel) TN p; {
    int nindir = 0, offset = 0;
    TN q;
    Symbol s;

    if (p == 0) return 0;
    else if (STATE(p) && !relabel)
        return p;
    ⟨ label kids first  94a⟩
    if (OP(p) != OVFU && OP(p) != OVFB)   /* special overflow operator */
        OP(p) = p->op;
    switch (p->op) {
       case CNSTI:
        ⟨label a CNSTI integer constant 94c⟩
        break;
    case CNSTU:
        ⟨label a CNSTU unsigned constant 94d⟩
        break;
    case RIGHT:
        if (isstruct(TYPE(p))) OP(p) = RIGHTS;
        else if (!LKID(p))     OP(p) = RIGHTB;
        else if (!RKID(p))     OP(p) = RIGHTC;
        break;
    default:
        break;
    }
    STATE(p) = FOLD_STATE(OP(p), LKID(p) ? STATE(LKID(p)) : 0,
                          RKID(p) ? STATE(RKID(p)) : 0);
    assert(STATE(p));
    return p;
}
```
This definition is continued in chunks 110, 111b, and 114.
This code is used in chunk 113.

`relabel` is applied after commuting operands.

96c ⟨*definitions* 95f⟩+≡
```
static void relabel(p) TN p; {
    STATE(p) = 0;
    label(p, 0);
}
```

`reduce` reduces the Tree or Node `t` and returns the new expression.

97a ⟨*definitions* 95f⟩+≡

```
static TN reduce(t, ty, nt, env) TN t; Type ty; jmp_buf env; {
    int eruleno, newruleno, i, rereduce = 0;
    short *nts;
    TN v[10], p = (TN)0;
    Opcode op = t->op;
    Value val;
    Symbol  s;

    val.u = 0;
    eruleno = FOLD_RULE(STATE(t), nt);
    nts = FOLD_NTS[eruleno];
    FOLD_KIDS(t, eruleno, v);    /* reduce children */
    for (i = 0; nts[i]; i++) {
        v[i] = reduce(v[i], ty, nts[i], env);
        rereduce |= v[i]->x.rereduce;
    }
    t = FOLD_KIDS1(t, eruleno, v); /* update node with reduced children */
    t->x.rereduce = rereduce;
    switch (eruleno) {
#include FOLD_RULES
    case 0:
    default: assert(0);
    }
    if (p) {
        rereduce = p->x.rereduce;
        label(p, 0);
        p->x.rereduce = rereduce;
    }
    return (p ? p : t);
}
```

5.2 Simplification rules

We divide up the specification into rules that are applicable to both Trees and Nodes and those that are unique to each.

97b ⟨*cfold-node.spec* 97b⟩≡

⟨*types* 98b⟩
⟨*goal rules* 98d⟩
⟨*constant rules* 99a⟩
⟨*node constant rules* 100a⟩
⟨*rules* 100b⟩
⟨*node rules* 104e⟩
⟨*default rules* 104f⟩

This code is written to file `cfold-node.spec`.

98a ⟨*cfold-tree.spec* 98a⟩≡
```
⟨types 98b⟩
⟨goal rules 98d⟩
⟨constant rules 99a⟩
⟨rules 100b⟩
⟨tree rules 101d⟩
⟨default rules 104f⟩
⟨default tree rules 106a⟩
```
This code is written to file `cfold-tree.spec`.

We use the `eburs` format for specifying the simplification rules. We factor rules by their operators and by their types. These rules could probably be more concise but they were derived almost verbatim from the original C code, which is probably not the best way to have written them.

The operator types are defined first.

98b ⟨*types* 98b⟩≡
```
<tys>: B B | C C | D D | F F | I I  | S S | P P | U U | V V
<aty>: B B | C C | D D | F F | I I  | S S | U U | V V
<iu>: I I | U U
<dfi>: D D | F F | I I
<all>: D D | F F | I I | V V
```
This definition is continued in chunk 98c.
This code is used in chunks 97b and 98a.

The operands are grouped by their functionality and the number of operands: they include arithmetic (`ao`), bit-wise (`bo`), logical (`lo`), integer (`io`), non-integer (`xo`), add and subtract (`as`), and shift (`sh`).

98c ⟨*types* 98b⟩+≡
```
<ao>: ADD ADD | DIV DIV | MOD MOD | MUL MUL | SUB SUB
<bo>: BAND BAND | BOR BOR | BXOR BXOR
<lo>: EQ EQ | GE GE | GT GT | LE LE | LT LT | NE NE
<io>: ADD ADD | MUL MUL | SUB SUB
<xo>: ADD ADD | DIV DIV | MUL MUL | SUB SUB
<as>: ADD ADD | SUB SUB
<sh>: LSH LSH | RSH RSH
```
The `goal` rules are to first match a constant, then an address, then an arbitrary expression if neither a constant nor address can be matched.

98d ⟨*goal rules* 98d⟩≡
```
goal: const 0 "0"
goal: addr 0
goal: a 1
```
This code is used in chunks 97b and 98a.

The following rules match constants. Some constants, such as zero, one, and powers of two, have special labels so that their values can be identified easily in simplification rules.

99a ⟨*constant rules* 99a⟩≡

```
d:   CNSTD 0 "0"
f:   CNSTF 0
iSh: I0I 0
iSh: I1I 0
iSh: ISI 0
iSh: P2SI
i2p: P2I 0
i2p: P2SI 0
i:   CNSTI 0
i:   i2p 0
i:   iSh 0
p:   CNSTP 0
u:   I0U 0
u:   I1U 0
u:   ISU 0
u:   P2SU 0
u2p: P2U 0
u2p: P2SU 0
u:   CNSTU 0
u:   u2p 0
sc:  CNSTC 0
uc:  CNSTC 0
ss:  CNSTS 0
us:  CNSTS 0
zero: I0I 0
zero: I0U 0
one:  I1I 0
one:  I1U 0
```

This definition is continued in chunk 99b.
This code is used in chunks 97b and 98a.

A `const` can be any type of constant expression.

99b ⟨*constant rules* 99a⟩+≡

```
const: d 0  "0"
const: f 0
const: i 0
const: p 0
const: u 0
const: sc 0
const: uc 0
const: ss 0
const: us 0
```

`caddr` is the non-terminal associated with `CADDRP` and `SADDRP` nodes. A `CADDRP` node is an expression that refers to a link-time constant; the constant values are usually provided in a binding (a.k.a. dataimage) file. The `label` function determines if an address expression refers to such a constant. A `SADDRP` node is analogous to `CADDRP` except the address expression refers to a symbol instead of a constant. Both expressions are used to match and simplify idioms in Modula-3 code.

100a ⟨*node constant rules* 100a⟩≡
```
caddr: CADDRP 0 "0"
caddr: SADDRP 0
addr: SADDRP 0
```
This definition is continued in chunk 100.
This code is used in chunk 97b.

100b ⟨*rules* 100b⟩≡
```
addr: ADDRLP 0 "0"
addr: ADDRFP 0
addr: ADDRGP 0
```
This definition is continued in chunks 100–104.
This code is used in chunks 97b and 98a.

100c ⟨*rules* 100b⟩+≡
```
a: const 0 "0"
a: addr  0
```

100d ⟨*node constant rules* 100a⟩+≡
```
a: caddr 0 "0"
```

We can retrieve a constant from a `caddr` node when its value is fetched by `INDIR`.

100e ⟨*external declarations* 95d⟩+≡
```
extern int exprconvert;
```

100f ⟨*node constant rules* 100a⟩+≡
```
uc: INDIRC(caddr) "%0 = NEWTN(CNST+optype(op), ty, 0, 0), \
                   val = datavalue(%1->x.dataval, optype(op)), \
                   %0s = constant(btot(optype(op)), val), exprconvert++, label(%0, 0)"
sc: INDIRC(caddr)
i:  INDIRI(caddr)
us: INDIRS(caddr)
ss: INDIRS(caddr)
```

The following rules simplify expressions that refer to constants.

101a ⟨*rules* 100b⟩+≡
```
u: <ao>U(u,u) 0  "newconst(optype(op), %v, %0v, %1v <1>OP %2v)"
u: <bo>U(u,u) 0
i: <lo>D(d,d) 0  "newconst(I, %v, %0v, %1v <1>OP %2v)"
i: <lo>F(f,f) 0
i: <lo>I(i,i) 0
i: <lo>U(u,u) 0
d: <xo>D(d,d) 0  "xfoldcnst(optype(op), %v, %0v, %1v, %2v, <1>OP, <1>FUNC)"
f: <xo>F(f,f) 0
i: <io>I(i,i) 0
i: DIVI(i,i)  0  "divtest(%1v, %2v, xfoldcnst(optype(op), %v, %0v, %1v, %2v, DIVOP, div))"
i: MODI(i,i)  0  "divtest(%1v, %2v, xfoldcnst(optype(op), %v, %0v, %1v, %2v, MODOP, div))"
i: BCOMI(i)   0  "newconst(optype(op), %v, %0v, ~%1v)"
u: BCOMU(u)   0
p: ADDP(p,i)  0  "newpconst(P, %v, %0v, (char *)%1v + %2v)"
p: ADDP(p,u)  0
p: ADDP(i,p)  0  "newpconst(P, %v, %0v, %1v + (char *)%2v)"
p: ADDP(u,p)  0
i: SUBP(p,p)  0  "newconst(I, %v, %0v, (char *)%1v - (char *)%2v)"
```

The following rules eliminate unnecessary conversions between types.

101b ⟨*rules* 100b⟩+≡
```
u: CVCU(uc)   0  "newconst(optype(op), %v, %0v, %1v)"
d: CVFD(f)    0
d: CVID(i)    0
u: CVIU(i)    0
i: CVSI(ss)   0
u: CVSU(us)   0
uc: CVUC(u)   0
us: CVUS(u)   0
i: CVCI(sc)   0  "newconst(optype(op), %v, %0v, (%1v &  0200 ? (~0<<8): 0)|(%1v &0377))"
u: CVPU(p)    0  "newconst(optype(op), %v, %0v, (unsigned)%1v)"
d: CVUD(u)    0  "newconst(optype(op), %v, %0v, utod(%1v))"
i: CVUI(u)    0  "overflow(%1v, >, <=, newconst(optype(op), %v, %0v, %1v))"
p: CVUP(u)    0  "newconst(optype(op), %v, %0v, (char *)%1v)"
i: CVDI(d)    0  "xcvtcnst(optype(op), %v, %0v, %1v)"
f: CVDF(d)    0
sc: CVIC(i)   0
ss: CVIS(i)   0
```

Applying unary negation and bit-wise not to a constant produces a new constant.

101c ⟨*rules* 100b⟩+≡
```
d: NEGD(d)    0  "newconst(optype(op), %v, %0v, -%1v)"
f: NEGF(f)    0
i: NEGI(i)    0  "overflow(%1v, ==, !=, newconst(I, %v, %0v, -%1v))"
```

101d ⟨*tree rules* 101d⟩≡
```
i: NOTI(i)    0  "newconst(I, %v, %0v, !%1v)"
```

This definition is continued in chunks 102–104.
This code is used in chunk 98a.

Applying a shift operator to a constant produces a new constant.

102a ⟨*rules* 100b⟩+≡
```
  i: RSHI(i,iSh)  0 "{ if (%1v < 0) \
                       %0 = CNODE(%1v >> %2v | (~0 << (8*TYPE(%1)->size - %2v)), inttype); \
                       else %0 = CNODE(%1v >> %2v, inttype); }"
  i: LSHI(i,iSh)  0 "xlshcnst(%v, %0v, %1v, %2v)"
  u: <sh>U(u,iSh) 0 "newconst(U, %v, %0v, %1v <1>OP %2v)"
```

Some operators are their own inverses; applying them twice yields the original value.

102b ⟨*rules* 100b⟩+≡
```
  a: BCOMU(BCOMU(a)) 0     "%0 = %1"
  a: BCOMU(BCOMU(a)) 0
  a: NEGD(NEGD(a))   0
  a: NEGF(NEGF(a))   0
  a: NEGI(NEGI(a))   0
```

We use many of the special constant labels to help identify identity rules.

102c ⟨*rules* 100b⟩+≡
```
  a: BANDU(a,u)   0     "%0 = (%2v == ~0) ? %1 : (TN)0"
  a: BANDU(u,a)   0     "%0 = (%1v == ~0) ? %2 : (TN)0"
  a: BORU(a,I0U)  0     "%0 = %1"
  a: BXORU(a,I0U) 0
  a: BORU(I0U,a)  0
  a: BXORU(I0U,a) 0
  a: DIV<iu>(a,one)  0
  a: <sh><iu>(a,I0I) 0
  a: DIVU(a,I0U)     0  "/* error */"
  a: MOD<iu>(a,I0I)  0
```

102d ⟨*tree rules* 101d⟩+≡
```
  a: ANDI(I0I,a)  0  "%0 = LKID(t)"
  a: ANDI(i,a)    0  "%0 = cond(%2)"
  a: BANDU(a,I0U) 0  "%0 = NEWTN(RIGHT, unsignedtype, root(%1), CNODE(0, unsignedtype)), \
                     label(%0, 0)"
  a: BANDU(I0U,a) 0
  a: MODI(a,I1I)  0  "%0 = NEWTN(RIGHT, inttype, root(%1), CNODE(0, inttype)), label(%0, 0)"
```

Only trees can include references to FIELD operators.

102e ⟨*tree rules* 101d⟩+≡
```
  a: EQI(FIELD(a),I0I) 0     "%0 = ZEROFIELD(generic(op), LKID(t), RKID(t))"
  a: NEI(FIELD(a),I0I) 0
  a: EQU(FIELD(a),I0U) 0
  a: NEU(FIELD(a),I0U) 0
  a: EQI(I0I,FIELD(a)) 0     "%0 = ZEROFIELD(generic(op), RKID(t), LKID(t))"
  a: NEI(I0I,FIELD(a)) 0
  a: EQU(I0U,FIELD(a)) 0
  a: NEU(I0U,FIELD(a)) 0
```

103a ⟨*tree rules* 101d⟩+≡

```
a: GEU(a,I0U)   0     "geu(%1, 1)"
a: GTU(I0U,a)   0     "geu(%1, 0)"
a: LEU(I0U,a)   0     "geu(%1, 1)"
a: LTU(a,I0U)   0     "geu(%1, 0)"
a: ORI(I0I,a)   0     "%0 = cond(%1)"
a: ORI(i,a)     0     "%0 = CNODE(1, inttype)"
```

103b ⟨*rules* 100b⟩+≡

```
a: GEU(I0U,a)   0     "t->op = EQI"
a: LEU(a,I0U)   0
a: GTU(a,I0U)   0     "t->op = NEI"
a: LTU(I0U,a)   0
```

The following rules attempt to simplify addressing expressions by eliminating or simplifying constant expressions.

103c ⟨*rules* 100b⟩+≡

```
addr: ADDP(addr,i) 0    "%0 = (%2v == 0 ? RETYPE(%1, ty) : addrnode(%1s, %2v, ty))"
addr: ADDP(addr,u) 0
addr: ADDP(i,addr) 0    "%0 = (%1v == 0 ? RETYPE(%2, ty) : addrnode(%2s, %1v, ty))"
addr: ADDP(u,addr) 0
a: ADDP(a,zero)    0    "%0 = RETYPE(%1, ty)"
a: ADDP(zero,a)    0    "%0 = RETYPE(%2, ty)"
a: ADDP(<as>I(a,i),addr) 0 "%0 = SIMP(ADDP, ty, %1, SIMP(<1>+P, ty, %3, %2))"
a: ADDP(<as>I(i,a),addr) 0 "%0 = SIMP(<1>+P, ty, SIMP(ADDP, ty, %1, %3), %2)"
a: ADDP(ADDP(a,addr),i) 0  "%0 = SIMP(ADDP, ty, %1, \
                             ((%3v == 0) ? RETYPE(%2, ty) : addrnode(%2s, %3v, ty)))"
a: ADDP(ADDP(a,addr),u) 0
a: ADDP(i,ADDP(a,addr)) 0  "%0 = SIMP(ADDP, ty, %2, \
                             ((%1v == 0) ? RETYPE(%3, ty) : addrnode(%3s, %1v, ty)))"
a: ADDP(u,ADDP(a,addr)) 0
a: ADDP(ADDP(a,i),i) 0 "%0 = SIMP(ADDP, ty, %1, PLUS(ADD, %2, %3, %2v, %3v))"
a: ADDP(ADDP(a,i),u) 0
a: ADDP(ADDP(a,u),i) 0
a: ADDP(ADDP(a,u),u) 0
a: ADDP(i,ADDP(a,i)) 0 "%0 = SIMP(ADDP, ty, %2, PLUS(ADD, %1, %3, %1v, %3v))"
a: ADDP(i,ADDP(a,u)) 0
a: ADDP(u,ADDP(a,i)) 0
a: ADDP(u,ADDP(a,u)) 0
a: ADDP(ADDI(a,i),ADDP(a,i)) 0 "%0 = SIMP(ADDP, ty, %1, \
                                        SIMP(ADDP, ty, %3, PLUS(ADD, %2, %4, %2v, %4v)))"
a: ADDP(ADDI(a,i),ADDP(a,u)) 0
a: ADDP(ADDI(i,a),ADDP(a,i)) 0 "%0 = SIMP(ADDP, ty, %2, \
                                        SIMP(ADDP, ty, %3, PLUS(ADD, %1, %4, %1v, %4v)))"
a: ADDP(ADDI(i,a),ADDP(a,u)) 0
```

103d ⟨*tree rules* 101d⟩+≡

```
a: ADDP(RIGHTS(a,a),a) 0 "%0 = NEWTN(RIGHT, ty, %1, SIMP(ADDP, ty, %2, %3))"
a: ADDP(RIGHT(a,a),a)  0 "%0 = NEWTN(RIGHT, ty, %1, SIMP(ADDP, ty, %2, %3))"
a: ADDP(RIGHTB,a)      0 "%0 = NEWTN(RIGHT, ty, LKID(LKID(t)), \
                                   SIMP(ADDP, ty, RKID(LKID(t)), %1))"
a: ADDP(RIGHTC(a),a)   0 "%0 = NEWTN(RIGHT, ty, SIMP(ADDP, ty, %1, %3), 0)"
```

Commutativity and associativity rules are applied to eliminate constants in the following rules.

104a ⟨*rules* 100b⟩+≡
```
<iuc>: i i | u u
a: ADDP(<iuc>,a) 0    "COMMUTE(t); relabel(t)"
a: SUBP(a,i) 0        "%0 = SIMP(ADDP, ty, %1, CNODE(-(int)%2v, inttype))"
a: SUBP(a,u) 0
a: SUBP(addr,ADDI(a,i)) 0 "%0 = SIMP(SUBP, ty, SIMP(SUBP, ty, %1, %3), %2)"
a: SUBP(addr,ADDI(i,a)) 0 "%0 = SIMP(SUBP, ty, SIMP(SUBP, ty, %1, %2), %3)"
a: MULI(i,<as>I(a,i)) 0 "%0 = SIMP(<1>I, inttype, \
                           SIMP(MULI, inttype, %1, %2), SIMP(MULI, inttype, %1, %3))"
a: MULI(i,<as>I(i,a)) 0
a: MULI(<as>I(a,i),i) 0 "%0 = SIMP(<1>I, inttype, \
                           SIMP(MULI, inttype, %1, %3), SIMP(MULI, inttype, %2, %3))"
a: MULI(<as>I(i,a),i) 0
```

Multiplications or divisions by a power of two are converted into shifts.

104b ⟨*rules* 100b⟩+≡
```
a: MULI(i2p,a) 0 "%0 = (%1v > 0) ? \
                  SIMP(LSHI, inttype, %2, CNODE(ispow2(%1v), inttype)): (TN)0"
a: MULI(a,i2p) 0 "%0 = (%2v > 0) ? \
                  SIMP(LSHI, inttype, %1, CNODE(ispow2(%2v), inttype)): (TN)0"
a: MULU(u2p,a) 0 "%0 = SIMP(LSHU, unsignedtype, %2, CNODE(ispow2(%1v), inttype))"
a: MULU(a,u2p) 0 "%0 = SIMP(LSHU, unsignedtype, %1, CNODE(ispow2(%2v), inttype))"
a: DIVU(a,u2p) 0 "%0 = SIMP(RSHU, unsignedtype, %1, CNODE(ispow2(%2v), inttype))"
```

104c ⟨*tree rules* 101d⟩+≡
```
a: MODU(a,i2p) 0     "%0 = bittree(BAND, %2, CNODE(%2v - 1, unsignedtype))"
a: ORI(a,a)    0     "t->op = generic(op)"
a: ANDI(a,a)   0
a: NOTI(a)     0
```

104d ⟨*rules* 100b⟩+≡
```
a: EQU(a,a)  0       "t->op = generic(op)+I"
a: NEU(a,a)  0
a: BCOMI(a)  0       "t->op = BCOMU"
```

The following rules guarantee that a symbol inherits the attributes of the symbol whose value it obtains by assignment. These rules are used by `mld` during optimization.

104e ⟨*node rules* 104e⟩≡
```
a: CALL<all>(caddr)         0  "if (s = datasymbol(constoffset(%1->x.dataval,0))) \
                                 { LKID(t) = new_addrgp(s); }"
a: ASGNP(addr,caddr)        0  "inherit_attributes(%1s,%2)"
a: ASGNP(addr,INDIRP(addr)) 0  "if (!msym(%1s).auxtype) \
                                 msym(%1s).auxtype = msym(%2s).auxtype"
```
This definition is continued in chunk 105.
This code is used in chunk 97b.

104f ⟨*default rules* 104f⟩≡
```
a: ASGNP(a,a)     0     "0"
a: ASGN<aty>(a,a) 1     "0"
```

This definition is continued in chunk 105.
This code is used in chunks 97b and 98a.

The target symbol of an assignment inherits its auxilliary type from the RHS of the assignment. If the target is a temporary and represents a common sub-expression, it also inherits the source's binding values.

105a ⟨*local declarations* 92⟩+≡
```
static void inherit_attributes(Symbol, Node);
```

105b ⟨*definitions* 95f⟩+≡
```
static void inherit_attributes(to, from) Symbol to; Node from; {
    if (to->temporary && mtemp(to).cse) {
        mtemp(to).datavalue = from->x.dataval;
    }
}
```

There are many idioms that are unique to Modula-3. In `mill` code, their representation is very close to the machine-level representation. It would be nice to have some mapping that identifies the idioms in a higher-level form and then have a translator convert them into `mill` code idioms.

The following idiom matches the `typecode` field of an object and the NIL check expression.

105c ⟨*node rules* 104e⟩+≡
```
a: RSHI(LSHI(INDIRI(ADDP(INDIRP(addr),i)),i),i)  0    "%0 = typecode_idiom(t,%1,%2,%3,%4)"
a: ASGNC(addr,INDIRC(ADDP(INDIRP(addr),i)))      0    "%0 = nilcheck_idiom(t,%1,%2,%3)"
```

The following rules can only be applied to Nodes in `mld`. They attempt to match Modula-3 method idioms in the intermediate `mill` code.

105d ⟨*node rules* 104e⟩+≡
```
a: CALL<all>(INDIRP(ADDP(INDIRP(INDIRP(addr)),a))) 0    "%0 = method_idiom(t,%1,%2,0,1)"
a: CALL<all>(INDIRP(ADDP(a,INDIRP(INDIRP(addr))))) 0    "%0 = method_idiom(t,%2,%1,0,2)"
a: CALL<all>(INDIRP(ADDP(ADDP(a,INDIRP(INDIRP(addr))),a))) 0 "%0 = method_idiom(t,%2,%1,%3,3)"
a: CALL<all>(INDIRP(a))                            0    "%0 = method_idiom(t,0, %1, 0, 0)"
a: CALLB(INDIRP(ADDP(INDIRP(INDIRP(addr)),a)),a)   0    "%0 = method_idiom(t,%1,%2,0,7)"
a: CALLB(INDIRP(ADDP(a,INDIRP(INDIRP(addr)))),a)   0    "%0 = method_idiom(t,%2,%1,0,8)"
a: CALLB(INDIRP(ADDP(ADDP(a,INDIRP(INDIRP(addr))),a)),a) 0 "%0 = method_idiom(t,%2,%1,%3,9)"
a: CALLB(INDIRP(a),a)                              0    "%0 = method_idiom(t,0,%1,0,0)"
```

The remainder of the rules match any other legal expression with a cost of one. Binary operators are grouped by `bo`; unary operators by `uo`.

105e ⟨*default rules* 104f⟩+≡
```
<bops>: ADD ADD | BAND BAND | BOR BOR | BXOR BXOR | DIV DIV | EQ EQ | GE GE | GT GT | LE LE | LT LT | LSH LSH | MOD MOD | MUL MUL | NE NE | RSH RSH | SUB SUB
<uops>: ARG ARG | BCOM BCOM | CVC CVC | CVD CVD | CVF CVF | CVI CVI | CVP CVP | CVS CVS | CVU CVU | INDIR INDIR | JUMP JUMP | LOAD LOAD  | NEG NEG
```

105f ⟨*default rules* 104f⟩+≡
```
a: <bops><tys>(a,a) 1 "0"
a: OVFB(a,a)        1
a: CALLB(a,a)       1
a: <uops><tys>(a)   1
a: OVFU(a)          1
a: RET<dfi>(a)      1
a: CALL<dfi>(a)     1
a: CALLV(a)         1
a: RETV             1
a: LABELV           1
```

106a ⟨*default tree rules* 106a⟩≡
```
a: COND(a,a)        1 "0"
a: OR(a,a)          1
a: AND(a,a)         1
a: NOT(a)           1
a: RIGHT(a,a)       1
a: RIGHTS(a,a)      1
a: RIGHTB           1
a: RIGHTC(a)        1
a: FIELD(a)         1
```

This code is used in chunk 98a.

106b ⟨*local macro definitions* 106b⟩≡
```
#define LKID(X)    X->kids[0]
#define RKID(X)    X->kids[1]
#define OP(X)      X->x.cop
#define STATE(X)   X->x.cstate
#define CONSTMSG(v) fprint(2,"%s %d\n", opname(op), v)
```

This definition is continued in chunks 111c and 112a.
This code is used in chunks 113 and 115a.

isconstsym determines if **q** refers to a constant symbol and returns the symbol if it does. **constsym** expects **q** to be a reference to a constant symbol and assertion fails if it is not. **iscopyasgn** determines if **q** is a simple assignment of one variable to another.

106c ⟨*external declarations* 95d⟩+≡
```
dclproto(Symbol constsym,(Node));
dclproto(Symbol isconstsym,(Node));
dclproto(int iscopyasgn,(Node));
dclproto(Symbol issimpleuse,(Node));
```

106d ⟨*external declarations* 95d⟩+≡
```
#define isconvertop(op) ((op) == CVC || (op) == CVD || (op) == CVF || (op) == CVI ||  \
                         (op) == CVP || (op) == CVS || (op) == CVU)
```

107a ⟨*node definitions* 93⟩+≡
```
Symbol constsym(q) Node q; {
    Symbol cs;
    cs = isconstsym(q);
    if (!cs) { printdag(q,2); fprint(2,"cop %d\n", OP(q)); }
    assert(cs = isconstsym(q));
    return cs;
}

Symbol isconstsym(q) Node q; {
    Symbol cs;
    while (isconvertop(generic(q->op))) q = q->kids[0];
    if (generic(q->op) == CNST) {
        assert(cs = q->syms[0]);
        return cs;
    } else if ((q->op == INDIRF || q->op == INDIRD) && isaddrop(q->kids[0]->op)) {
        Symbol s = q->kids[0]->syms[0];
        if (s->scope == GLOBAL && s->u.seg == LIT) {
           assert(cs = mglob(s).constsym);
           return cs;
        }
    }
    return (Symbol)0;
}
```

A copy assignment should be an unambiguous definition of its left-hand side.

107b ⟨*node definitions* 93⟩+≡
```
int iscopyasgn(p) Node p; {
    if (generic(p->op) == ASGN &&
        isaddrop(p->kids[0]->op) &&
        issimpleuse(p->kids[1]) != 0) return 1;
    return 0;
}
```

A simple use of a variable may include conversion operators.

107c ⟨*node definitions* 93⟩+≡
```
Symbol issimpleuse(p) Node p; {
    while (isconvertop(generic(p->op))) p = p->kids[0];
    if (generic(p->op) == INDIR && isaddrop(p->kids[0]->op))
        return p->kids[0]->syms[0];
    return (Symbol)0;
}
```

`addrnode` creates a tree for addressing expression p+n, with type ty.

107d ⟨*local declarations* 92⟩+≡
```
dclproto(static TN addrnode,(Symbol, int, Type));
```

108 ⟨*definitions* 95f⟩+≡
```
static TN addrnode(p, n, ty) Symbol p; Type ty; {
    Symbol q = (Symbol)ALLOC(sizeof *q);
    TN e;

    bzero((char *)q, (sizeof *q));;
    q->name = stringd(genlabel(1));
    q->sclass = p->sclass;
    q->scope = p->scope;
    q->type = ty;
    q->defined = 1;
    q->temporary = p->temporary;
    q->generated = p->generated;
    q->computed = 1;
    q->addressed = p->addressed;
    q->ref = 1;
    q->type = ptr(unqual(p->type));
    if (p->scope != LABELS &&
        (p->scope == GLOBAL || p->sclass == STATIC || p->sclass == EXTERN)) {
        q->sclass = p->sclass == AUTO ? STATIC : p->sclass;
        maddr(q).symid = msym(p).id;
        maddr(q).offset = n;
        msym(q).symtab = msym(p).symtab;
        (*IR->address)(q, p, n);
        e = NEWTN(ADDRG+P, ty, 0, 0);
        SYMBOL(e) = q;
        label(e, 0);
#ifdef MLD
        warning("New address expression %s = %s+%d\n", q->name, p->name, n);
#endif
    }
#ifndef MLD
    else {
        Code cp;
        if(p->scope != LABELS && !p->defined)
            addlocal(p);
        cp = code(Address);
        cp->u.addr.sym = q;
        cp->u.addr.base = p;
        cp->u.addr.offset = n;
        if (p->scope == LABELS) e = tree(ADDRG+P, ty, (Tree)0, (Tree)0);
        else e = tree(p->scope == PARAM ? ADDRF+P : ADDRL+P, ty, (Tree)0, (Tree)0);
        SYMBOL(e) = q;
    }
#endif
    return e;
}
```

5.3 Constant folding

The following procedures fold constant expressions.

109a ⟨*local declarations* 92⟩+≡
```
dclproto(static int add,(double, double, double, double, int));
dclproto(static int div,(double, double, double, double, int));
dclproto(static int mul,(double, double, double, double, int));
dclproto(static int sub,(double, double, double, double, int));
```

`add` returns 1 if min <= $x + y$ <= max, 0 otherwise.

109b ⟨*definitions* 95f⟩+≡
```
static int add(x, y, min, max, needconst) double x, y, min, max; {
    int cond = x == 0 || y == 0
    || x < 0 && y < 0 && x >= min - y
    || x < 0 && y > 0
    || x > 0 && y < 0
    || x > 0 && y > 0 && x <= max - y;
    if (!cond && needconst) {
        warning("overflow in constant expression\n");
        cond = 1;
    }
    return cond;
}
```

`div` returns 1 if min <= x/y <= max, 0 otherwise.

109c ⟨*definitions* 95f⟩+≡
```
static int div(x, y, min, max, needconst) double x, y, min, max; {
    int cond;

    &min;
    if (x < 0) x = -x;
    if (y < 0) y = -y;
    cond = y != 0 && (y > 1 || x <= max*y);
    if (!cond && y != 0 && needconst) {
        warning("overflow in constant expression\n");
        cond = 1;
    }
    return cond;
}
```

`mul` returns 1 if min <= $x * y$ <= max, 0 otherwise.

110a ⟨*definitions* 95f⟩+≡

```
static int mul(x, y, min, max, needconst) double x, y, min, max; {
    int cond = x > -1 && x <= 1 || y > -1 && y <= 1
    || x < 0 && y < 0 && -x <= max/-y
    || x < 0 && y > 0 &&  x >= min/y
    || x > 0 && y < 0 &&  x <= min/y
    || x > 0 && y > 0 &&  x <= max/y;
    if (!cond && needconst) {
        warning("overflow in constant expression\n");
        cond = 1;
    }
    return cond;
}
```

`sub` returns 1 if $min <= x - y <= max$, 0 otherwise.

110b ⟨*definitions* 95f⟩+≡

```
static int sub(x, y, min, max, needconst) double x, y, min, max; {
    return add(x, -y, min, max, needconst);
}
```

The following procedures are required by `mlcc`'s front end.

110c ⟨*tree definitions* 96b⟩+≡

```
Tree constexpr(tok) {
        Tree p;

        needconst++;
        p = expr1(tok);
        needconst--;
        return p;
}
int intexpr(tok, n) {
        Tree p = constexpr(tok);

        needconst++;
        if (generic(p->op) == CNST && isint(p->type))
                n = cast(p, inttype)->u.v.i;
        else
                error("integer expression must be constant\n");
        needconst--;
        return n;
}
```

If $u > 1$ and $u == 2^n$, `ispow2` returns n, otherwise return 0.

110d ⟨*tree definitions* 96b⟩+≡

```
int ispow2(u) unsigned u; {
    int n;

    if (u > 1 && (u&(u-1)) == 0)
        for (n = 0; u; u >>= 1, n++)
            if (u&1)
                return n;
    return 0;
}
```

111a ⟨*local declarations* 92⟩+≡

111b ⟨*tree definitions* 96b⟩+≡

Macros are gross, but useful because of C's limited type system. `TN` is defined be `Tree` in `mlcc` and `Node` in `mld`. The following macros are used in both simplifiers.

111c ⟨*local macro definitions* 106b⟩+≡

```
#define ADDOP    +
#define MULOP    *
#define SUBOP    -
#define DIVOP    /
#define RSHOP    >>
#define LSHOP    <<
#define MODOP    %
#define BANDOP   &
#define BOROP    |
#define BXOROP   ^
#define EQOP     ==
#define NEOP     !=
#define LTOP     <
#define LEOP     <=
#define GTOP     >
#define GEOP     >=
#define ADDFUNC  add
#define MULFUNC  mul
#define DIVFUNC  div
#define MODFUNC  div
#define SUBFUNC  sub
#define MIN(T)   (T == D ? -DBL_MAX : (T == F ? -FLT_MAX : INT_MIN))
#define MAX(T)   (T == D ? DBL_MAX : (T == F ? FLT_MAX : INT_MAX))
```

112a ⟨*local macro definitions* 106b⟩+≡
```
#define divtest(L, R, EXP) \
   if (mips_arch && R == -1 && \
       !div((double)L, (double)R, (double)INT_MIN, (double)INT_MAX, 0)) \
       { OP(t) = OVFB; longjmp(env, 1); } \
   else { EXP; }

#define geu(L,V) { warning("result of unsigned comparison is constant\n"); \
           p = NEWTN(RIGHT, inttype, root(L), CNODE(V, inttype)); label(p, 0); }

#define newpconst(T,TMP,RES,EXP) \
{ p = NEWTN(CNST+ttob(btot(T)), ty, 0, 0); TMP = EXP; \
  SYMBOL(p) = (NODE ? constant(btot(T), val) : 0); RES = TMP; label(p, 0);  }

#define overflow(L,OP1,OP2,EXP) \
    if (needconst && L OP1 INT_MIN) warning("overflow in constant expression\n"); \
    if (needconst || L OP2 INT_MIN) { EXP; } else { OP(t) = OVFU; longjmp(env, 1); }

#define xcvtcnst(T,TMP,RES,EXP) \
    if (needconst && (EXP < MIN(T) || EXP > MAX(T))) \
        warning("overflow in constant expression\n"); \
    if (needconst || EXP >= MIN(T) && EXP <= MAX(T)) { newconst(T, TMP, RES, EXP); } \
    else { OP(t) = OVFU; longjmp(env, 1); }

#define xfoldcnst(T,TMP,P,L,R,O,FUNC) \
    if (FUNC((double)L,(double)R,(double)MIN(T),(double)MAX(T), needconst)) { \
        newconst(T, TMP, P, L O R) } \
    else { OP(t) = OVFB; longjmp(env, 1); }

#define xlshcnst(TMP,P,L,R) \
    if (mul((double)L, (double)(1<<R), (double)INT_MIN, (double)INT_MAX, needconst)) \
        { newconst(I, TMP, P, L << R); } \
    else { OP(t) = OVFB; longjmp(env, 1); }
```

112b ⟨*simp.h* 112b⟩≡
```
#ifndef INCLUDE_SIMPH
#define INCLUDE_SIMPH
⟨external declarations 95d⟩
#endif
```
This code is written to file `simp.h`.

The `Tree` simplifier defines those macros that are specific to `Tree`s.

113 ⟨*tree-simp.c* 113⟩≡
```
#include "c.h"
#include <setjmp.h>
#include <simp.h>

#ifdef MIPS
int mips_arch = 1;
#else
int mips_arch = 0;
#endif
int needconst;        /* >=1 if parsing a constant expression */
#define ALLOC             talloc
#define ADDPSIMP(x)       0
#define CNODE(x,y)        consttree(x,y)
#define COMMUTE(p)        { TN _x = LKID(p); LKID(p) = RKID(p); RKID(p) = _x; }
#define NEWTN(op, ty, l, r)     tree(op, ty, (Tree)l, (Tree)r)
#define NODE                    0
#define PLUS(op, l, r, x, y)    ((*optree['+'])(op, l, r))
#define RETYPE(p, ty)           retype(p, ty)
#define SIMP(op, ty, l, r)      simplify(op, ty, (Tree)l, (Tree)r)
#define SYMBOL(p)               (p->u.sym)
#define TN                      Tree
#define TYPE(p)                 (p->type)
#define VALUE(p)                (p->u.v)
#define ZEROFIELD(OP,L,R) eqtree(OP, bittree(BAND, LKID(L), \
          CNODE(fieldmask(L->u.field)<<fieldright(L->u.field), unsignedtype)), R);
#define newconst(T,TMP,RES,EXP) \
{ p = tree(CNST+ttob(btot(T)), btot(T), 0, 0); TMP = EXP; \
  p->u.sym = 0; RES = TMP; label(p, 0); }

#define FOLD_RULES  "tree.rules"
#define FOLD_STATE  tree_state
#define FOLD_RULE   tree_rule
#define FOLD_KIDS   tree_kids
#define FOLD_KIDS1  tree_kids1
#define FOLD_NTS    tree_nts
dclproto(extern int tree_state,(int, int, int));
dclproto(extern int tree_rule,(int, int));
dclproto(extern TN *tree_kids,(TN, int, TN[]));
dclproto(extern TN tree_kids1,(TN, int, TN[]));
extern short *tree_nts[];
⟨local macro definitions 106b⟩
⟨local declarations 92⟩
⟨definitions 95f⟩
⟨tree definitions 96b⟩
```

This code is written to file `tree-simp.c`.

`simplify` builds a `Tree` for `l op r` and returns the resulting expression.

114 ⟨*tree definitions* 96b⟩+≡

```
TN simplify(op, ty, l, r) Type ty; TN l, r; {
    TN tmp, res = (TN)0;
    int lbl;
    jmp_buf env;

    if (optype(op) == 0)
        op += ttob(ty);
    tmp = NEWTN(op, ty, l, r);
    while (res == 0) {
        res = (TN)0;
        setjmp(env);
        label(tmp, 1);    /* label & reduce tree */
        res = reduce(tmp, ty, 1, env);
    }
    return res;
}
```

The Node simplifier defines those macros that are specific to Nodes.

115a ⟨*node-simp.c* 115a⟩≡

```
#include "c.h"
#include <setjmp.h>
#include <m3.h>
#include <simp.h>

extern int needconst;
extern int mips_arch;

#define ALLOC           alloc
#define ADDPSIMP(x)     fprint(2,"ADDP%d ",x), printdag(p, 2)
#define CNODE(x,y)      constnode(x,y)
#define COMMUTE(p)      { TN _x = LKID(p); LKID(p) = RKID(p); RKID(p) = _x; }
#define NEWTN(op,ty,k0,k1) newnode(op, k0, k1, 0)
#define NODE            1
#define PLUS(op, l, r, x, y) CNODE(x + y, \
        (TYPE(l) == unsignedtype || TYPE(r) == unsignedtype) ? unsignedtype : inttype)
#define RETYPE(p, ty)   p
#define SIMP(op, ty, l, r) node_simplify(op, ty, l, r)
#define SYMBOL(p)       (p->syms[0])
#define TN              Node
#define TYPE(p)         (btot(p->op))
#define VALUE(p)        ((p)->syms[0]->u.c.v)
#define newconst(T,TMP,RES,EXP) \
{ TMP = EXP; p = new_constant_node(T, constant(btot(T), val)); \
  RES = TMP; label(p, 0);  p->x.rereduce = 1; }

#define FOLD_RULES   "node.rules"
#define FOLD_STATE   node_state
#define FOLD_RULE    node_rule
#define FOLD_KIDS    node_kids
#define FOLD_KIDS1   node_kids1
#define FOLD_NTS     node_nts
dclproto(extern int node_state,(int, int, int));
dclproto(extern int node_rule,(int, int));
dclproto(extern TN *node_kids,(TN, int, TN[]));
dclproto(extern TN node_kids1,(TN, int, TN[]));
extern short *node_nts[];
int symconvert = 0;
int exprconvert = 0;

dclproto(static TN node_simplify,(int, Type, TN, TN));
dclproto(TN dag_simplify,(TN, Type));

⟨local macro definitions 106b⟩
⟨local declarations 92⟩
⟨definitions 95f⟩
⟨node definitions 93⟩
```

This code is written to file node-simp.c.

115b ⟨*external declarations* 95d⟩+≡

```
dclproto(Node constnode,(unsigned, Type));
```

116a ⟨*node definitions* 93⟩+≡

```
Node constnode(val, ty) unsigned val; Type ty; {
    Node p;
    Value v;

    v.u = val;
    p = newnode(CNST + ttob(ty), 0, 0, constant(ty, v));
    relabel(p);
    return p;
}
```

`dag_simplify` simplifies an expression by labelling and reducing the input expression; multiple reductions may be necessary if the expression is relabelled during a reduction pass.

116b ⟨*node definitions* 93⟩+≡

```
TN dag_simplify(p, ty) TN p; Type ty; {
    TN res = (TN)0;
    int lbl;
    jmp_buf env;

    while (res == 0 || res->x.rereduce) {
        if (res && res->x.rereduce) printdag(res, 2);
        res = (TN)0;
        setjmp(env);
        label(p, 1);    /* label & reduce tree */
        res = reduce(p, ty, 1, env);
        if (res && isconstsym(res)) res->x.rereduce = 0;
    }
    return res;
}
```

`node_simplify` builds a `Node` for `l op r`, simplifying and folding constants, if possible.

116c ⟨*node definitions* 93⟩+≡

```
static TN node_simplify(op, ty, l, r) Type ty; TN l, r; {
    TN tmp;

    if (optype(op) == 0) {
        assert(ty);
        op += ttob(ty);
    }
    tmp = NEWTN(op, ty, l, r);
    return dag_simplify(tmp, ty);
}
```

`new_constant_node` returns a new constant node for `Symbol sym` with Node type `T`.

116d ⟨*external declarations* 95d⟩+≡

```
dclproto(Node new_constant_node,(int, Symbol));
```

117 ⟨*node definitions* 93⟩+≡

```
Node new_constant_node(T, sym) Symbol sym;
{
    Node p;
    Symbol s;
    if (T == F || T == D) {
        if (sym->u.c.loc == NULL) {
            s = genident(STATIC, btot(T), GLOBAL);
            (*IR->defsymbol)(s);
            defglobal(s, LIT);
            (*IR->defconst)(T, sym->u.c.v);
            sym->u.c.loc = s;
            mglob(s).constsym = sym;
            msym(s).announced = 1;
        }
        p = newnode(INDIR+T, newnode(ADDRGP, 0, 0, sym->u.c.loc), 0, 0);
    } else p = newnode(CNST+T, 0, 0, sym);
    p->count = 1;
    return p;
}
```

6 mld: A mill linker and optimizer

program is mld's main driver procedure. It performs mld's startup functions (opening and reading mill files and resolving references to external symbols), then lays out data, applies the requested optimizations, and finally generates code for the entire program.

118a ⟨*local declarations* 118a⟩≡

```
static int nproc = 0, indataseg = 0, indircall = 0;
```

This definition is continued in chunks 121–23, 127, 129c, 133, 134, 137, and 139.
This code is used in chunks 144 and 145.

118b ⟨*main program* 118b⟩≡

```
void program(void) {
     int fileid, procid, gsym, i, cgnid;
     int totsegmentsizes[] = { 0, 0, 0, 0, 0, 0 };
     BaseTable *bt, *mainmodule, *exitmodule = 0;
     Symbol p, mainsym, exitsym;
     ProcIndex pi;
     MTable rootfns;

     fail = &mldfail;
     newtable(&rootfns, 8, sizeof(Symbol), &permanent);
     printetime("");     /* prime the timing pump */

     ⟨open and read modules; install global symbols 122a⟩
     ⟨ open and parse binding file 119b⟩
     ⟨identify reachable modules 120a⟩

     ⟨read optimization file 129a⟩
     ⟨read inlining file 128c⟩
     if (bflag) {
         int binding_address = RUNTIME_ADDRESS;
         ⟨ compute sizes of abstract data segments and initialize real segments 119a⟩
     }
     /* Emit startup code */
     if (bflag)  b_startup(nproc, &rootfns, aoutflag);
     printetime("Startup\t\t");

     ⟨eliminate unused modules 120b⟩
     ⟨layout data 123a⟩

     /* Announce unresolved global symbols */
     doexternals(amillfd);

     ⟨walk call graph 121a⟩
     ⟨ apply inlining 119c⟩
     ⟨ generate code 119d⟩
     ⟨ emit executable program 119e⟩

     ⟨ close open files 119f⟩
     ⟨memory instrumentation 138b⟩
     method_summary();

 /*   dump_cons_counters((Printer)fprint, (void *)2); */
 }
```

This code is used in chunk 144c.

119a ⟨*compute sizes of abstract data segments and initialize real segments* 119a⟩≡

```
gentable(bmilltab, fileid, 0) {
    bt = tableentry(BaseTable **, bmilltab, fileid);
    if (bt->visited)
        for (i = 0; i < 6; i++)
            totsegmentsizes[i] += bt->pbt.segmentsizes[i];
    else if (mapflag)
        warning("Module %s unreachable\n", bt->filename);
}
b_dataseginit(totsegmentsizes, binding_address);
```

This code is used in chunks 118b and 140.

119b ⟨*open and parse binding file* 119b⟩≡

```
if (dataflag) parse_dataimage(dataimage_file);
printetime("Parse dataimage\t");
```

This code is used in chunk 118b.

119c ⟨*apply inlining* 119c⟩≡

```
if (inlineflag != 0) {
    doinlining(rootfns);
    printetime("Inlining\t\t");
};
```

This code is used in chunk 118b.

119d ⟨*generate code* 119d⟩≡

```
generate_code(rootfns);
printetime("Generate code\t");
```

This code is used in chunk 118b.

119e ⟨*emit executable program* 119e⟩≡

```
if (bflag) {
    b_aout(bmillfd, aoutflag);
    printetime("Emit binary\t");
    b_summary();
}
```

This code is used in chunk 118b.

119f ⟨*close open files* 119f⟩≡

```
if (amillfd) close(amillfd);
if (bmillfd) close(bmillfd);
gentable(bmilltab, fileid, 0) {
    bt = tableentry(BaseTable **, bmilltab, fileid);
    fileClose(bt);
}
```

This code is used in chunks 118b, 140, and 141b.

6.1 Module and procedure reachability

Any module reachable from main or from the startup assembly code is added to a set of root functions.

120a ⟨*identify reachable modules* 120a⟩≡
```
assert(mainsym = lookupsym(string("main"), &mainmodule));
addtotable(Symbol *, (&rootfns), mainsym);
if (libcflag) {
    assert(exitsym = install_symbol("exit", &exitmodule));
    mfunc(exitsym)->ncalls += 1;
    addtotable(Symbol *, (&rootfns), exitsym);
}
resolvemodule(mainmodule);
if (bflag) b_resolveasmsyms(aoutflag);
if (exitmodule != 0) resolvemodule(exitmodule);
    fprint(2, "total procs %d in data segment %d called indirectly %d\n",
               nproc, indataseg, indircall);
printetime("Reachable modules");
```

This code is used in chunks 118b and 140.

Any indirectly callable procedure in a reachable module, i.e., the procedure has had its address taken, is also reachable.

120b ⟨*eliminate unused modules* 120b⟩≡
```
{
    int xmodule;
    gentable(bmilltab, fileid, 0) {
        bt = tableentry(BaseTable **, bmilltab, fileid);
        if (bt->visited) {
            xmodule = instringtable(Xfiletable, bt->filename);
if (xmodule) fprint(2,"**Don't opt %s\n", bt->filename);
            gentable(bt->procindex, procid, 0) {
                pi = tableentry(ProcIndex *, bt->procindex, procid);
                p = idxtable(Symbol *, bt->glbsymtab, symid(pi.id));
                mfunc(p)->noopt = xmodule | instringtable(Xproctable, p->name);
                if (indircalledproc(p))
                    addtotable(Symbol *, (&rootfns), p);
                mfunc(p)->procid = procid;
            }
        } else {
            if (mapflag)
                fprint(2,"%s -- unused module\n", bt->filename ? bt->filename : "???");
            if (bt->arena) mlcc_deallocate(&(bt->arena));
        }
    }
}
printetime("Zap unused modules");
```

This code is used in chunks 118b and 140.

Walk the callgraph marking all functions reachable from all root functions.

121a ⟨*walk call graph* 121a⟩≡
```
cgnid = 1;
gentable(rootfns, i, 0) {
    cgnid = walkcallgraph(tableentry(Symbol *, rootfns, i), cgnid, cgnid, inlineflag);
}
inlinestats();
printetime("Walk call graph\t");
```

This code is used in chunk 118b.

`resolvemodule` recursively marks as reachable any modules on which the given module is dependent. A module `m` is dependent on a module `n` if `n` exports a symbol that `m` imports.

121b ⟨*local declarations* 118a⟩+≡
```
dclproto(void resolvemodule,(BaseTable *));
```

121c ⟨*definitions* 121c⟩≡
```
void resolvemodule(bt) BaseTable *bt; {
    int gsym;
    Symbol p;
    BaseTable *m;

    if (bt->filename == 0)
        install_module(bt);
    bt->visited = 1;
    gentable(bt->glbsymtab, gsym, 1) {
        p = tableentry(Symbol *, bt->glbsymtab, gsym);
        if (!p->computed  && p->sclass == EXTERN) {
            m = resolveextsym(&p);
            if (m && !m->visited) {
                if (dbgflag) fprint(2,"%s depends on %s\n", bt->filename, m->filename);
                resolvemodule(m);
            }
        }
    }
}
```
This definition is continued in chunks 121e, 122c, 132c, 137, 138, and 144a.
This code is used in chunks 144 and 145.

`install_symbol` returns a resolved symbol. If the symbol has not been resolved, it installs the module in which the symbol is declared, then returns the symbol.

121d ⟨*external declarations* 121d⟩≡
```
dclproto(Symbol install_symbol,(char *, BaseTable **));
```
This definition is continued in chunks 128d, 132b, and 133b.
This code is used in chunk 144b.

121e ⟨*definitions* 121c⟩+≡
```
Symbol install_symbol(name, bt) char *name; BaseTable **bt; {
    Symbol sym;
    char *symname = string(name);
    if ((sym = lookupsym(symname, bt)) == 0) {
        if (*bt) install_module(*bt);
        sym = lookupsym(symname, bt);
    }
    return sym;
}
```

6.2 Identifying external symbols

Each module is opened and read once during initialization. All symbols exported by a module are installed in a global symbol table for lookup during symbol resolution.

122a ⟨*open and read modules; install global symbols* 122a⟩≡
```
{
    fprint(2,"BaseTable size %d\n", tablesize(bmilltab));
    gentable(bmilltab, fileid, 0) {
        bt = tableentry(BaseTable **, bmilltab, fileid);
        if (bt->filename != 0)
            install_module(bt);
    }
    printetime("Read modules\t");
}
```
This code is used in chunks 118b and 140.

122b ⟨*local declarations* 118a⟩+≡
```
dclproto(static void install_module,(BaseTable *));
```

122c ⟨*definitions* 121c⟩+≡
```
static void install_module(bt) BaseTable *bt;
{
    Symbol p;
    int gsym;

    bt->optvarct = 0;
    breadModuleHdr(bt);
    emitBaseTable(amillfd, *bt);
    breadProgram(bt);
    emitProcIndex(amillfd, bt->procindex);
    gentable(bt->glbsymtab, gsym, 1) {
        p = tableentry(Symbol *, bt->glbsymtab, gsym);
        if (p->generated)
            p->name = stringd(genlabel(1));
        if (!p->computed) {
            if (p->sclass != EXTERN) resolvesym(p, bt);
            if (p->type && isfunc(p->type)) {
                if (msym(p).dataseg) indataseg++;
                if (mfunc(p)->indircall) indircall++;
                if (p->sclass != EXTERN) nproc++;
            }
        }
        ⟨count globals which may be used in optimization 122d⟩
    }
    emitCallgraph(amillfd, bt);
if (mapflag) fprint(2,"%s Globals %d\n", bt->filename, bt->optvarct);
}
```

122d ⟨*count globals which may be used in optimization* 122d⟩≡
```
if (isoptvar(p)) bt->optvarct++;
```

This code is used in chunk 122c.

6.3 Data Layout

The data in each reachable module is emitted by calling the target-specific backend.

123a ⟨*layout data* 123a⟩≡

```
gentable(bmilltab, fileid, 0) {
    bt = tableentry(BaseTable **, bmilltab, fileid);
    if (bt->visited) {
        breadData(bt);
        doglobals(bt);
    }
}
printetime("Layout data\t");
```

This code is used in chunks 118b and 140.

`doglobals` announces all of the global symbols in a module to the backend. Initialized symbols are announced first, then uninitialized symbols.

123b ⟨*local declarations* 118a⟩+≡

```
dclproto(static Symbol dodata,(BaseTable *, int, int));
dclproto(static void doglobal,(Symbol));
dclproto(static void doglobals,(BaseTable *));
```

124a ⟨*data definitions* 124a⟩≡

```
static void doglobals(bt) BaseTable *bt; {
    int      i;
    unsigned short   j, k;
    List     l;
    Symbol   p, q, cs;
    int      lo, hi;
    int      sz;
    static int rdataoff = 0, dataoff = 0, bssoff = 0;

    gentable(bt->dataGlobs, i, 0) {
        j = symid(tableentry(unsigned short *, bt->dataGlobs, i));
        p = idxtable(Symbol *, bt->glbsymtab, j);
        assert(p);
        swtoseg(p->u.seg);
        doglobal(p);
        lo = mglob(p).dataloc;
        if (intablerange(bt->dataGlobs, i+1)) {
            k = idxtable(unsigned short *, bt->dataGlobs, i+1);
            q = idxtable(Symbol *, bt->glbsymtab, symid(k));
            hi = mglob(q).dataloc;
        }
        else hi = tablesize(bt->datatab);
        cs = dodata(bt, lo, hi);
        ⟨install constant-literal global 124b⟩
    }
    gentable(bt->glbsymtab, j, 1){
        p = tableentry(Symbol *, bt->glbsymtab, j);
        if (p->sclass != TYPEDEF && !msym(p).announced) {
            if (p->computed)     /* COMPUTED GLOBAL */
                doaddress(p);
            else (*IR->defsymbol)(p);
            msym(p).announced = 1;
        }
    }
}
```

This definition is continued in chunks 125–27.
This code is used in chunks 144 and 145.

124b ⟨*install constant-literal global* 124b⟩≡

```
if (p->u.seg == LIT && p->type && isscalar(p->type) && cs) {
    mglob(p).constsym = cs;
    if (cs->u.c.loc == 0) cs->u.c.loc = p;
}
```

This code is used in chunk 124a.

`doglobal` announces a global variable to the target-specific backend.

125 ⟨*data definitions* 124a⟩+≡
```
static void doglobal(p) Symbol p; {

    if (p->computed)         /* COMPUTED GLOBAL */
        doaddress(p);
    else
        (*IR->defsymbol)(p);
    if (p->scope == CONSTANTS)
        defglobal(p, LIT);
    else if ((p->scope == GLOBAL || p->scope >= LOCAL) && !isfunc(p->type))
        defglobal(p, p->u.seg);
    else {
        emitSymbol(2, p, "");
        error("mld: Invalid global symbol\n");
    }
    msym(p).announced = 1;
}
```

`dodata` emits all of the data items in a specified range into the data segments. It does so by directing the backend to allocate space for the data items.

126a ⟨*data definitions* 124a⟩+≡

```
static Symbol dodata(bt, lo, hi) BaseTable *bt; {
    int i, l;
    DataItem d;
    char *s;
    Value v;
    Symbol addrsym, basesym, constsym;

    if (hi - lo == 0) return 0;

    for (i = lo; i < hi; i++) {
        constsym = 0;
        d = idxtable(DataItem *, bt->datatab, i);
        switch(d.ty) {
            case C: (*IR->defconst)(C, (v.uc = d.x.uc, v));
                    constsym = constant(chartype, v); break;
            case I:
            case U: (*IR->defconst)(d.ty, (v.u = d.x.u, v));
                    constsym = constant(btot(d.ty), v); break;
            case S: (*IR->defconst)(S, (v.us = d.x.us, v));
                    constsym = constant(shorttype, v);  break;
            case F: (*IR->defconst)(F, (v.f = d.x.f, v));
                    constsym = constant(floattype, v);  break;
            case D:
                ⟨ announce a double 126b⟩
                break;
            case P: (*IR->defconst)(P, (v.p = d.x.p, v)); break;
            case X:
                ⟨ announce a symbol's address 127a⟩
                break;
            case Y:
                s = mapstridx(bt, d.x.us);
                l = mapidx2len(bt, d.x.us);
                (*IR->defstring)(l, s); break;
            case Z: (*IR->space)(d.x.u); break;
            default: error("mld: dodata: unknown type\n"); break;
        }
    }
    return constsym;
}
```

126b ⟨ *announce a double* 126b⟩≡

```
((unsigned *)&v.d)[0] = d.x.u;
i++;
d = idxtable(DataItem *, bt->datatab, i);
((unsigned *)&v.d)[1] = d.x.u;
(*IR->defconst)(D, v);
constsym = constant(doubletype, v);
```

This code is used in chunk 126a.

127a ⟨*announce a symbol's address* 127a⟩≡
```
addrsym = idxtable(Symbol *, bt->glbsymtab, symid(d.x.us));
if (addrsym->computed)
    doaddress(addrsym);
else if (!msym(addrsym).announced)
    (*IR->defsymbol)(addrsym);
/* ??? this is a hack ??? */
if (msym(addrsym).dataseg || apflag) {
    (*IR->defaddress)(addrsym);
    msym(addrsym).dataseg = 1;
}  else (*IR->defconst)(P, (v.p = 0, v));
```
This code is used in chunk 126a.

The global symbol `indirproc` is the default target of an indirect procedure call.

127b ⟨*external definitions* 127b⟩≡
```
extern char *indirproc;
```
This definition is continued in chunk 127d.
This code is used in chunks 144 and 145.

127c ⟨*local declarations* 118a⟩+≡
```
char *indirproc;
```

127d ⟨*external definitions* 127b⟩+≡
```
dclproto(extern int chmod,(char *, int));
dclproto(extern int dup,(int));
dclproto(extern char *strstr,(char *,char *));
```

127e ⟨*local declarations* 118a⟩+≡
```
dclproto(static void defcompsym,(Symbol));
dclproto(static void doaddress,(Symbol));
```

127f ⟨*data definitions* 124a⟩+≡
```
static void doaddress(p) Symbol p; {
    Symbol basesym, comp;
    BaseTable *bt;

    comp = idxtable(Symbol *, (*msym(p).symtab), symid(maddr(p).symid));
    if (comp->sclass == STATIC)    /* Base symbol already resolved */
        tableentry(Symbol *, (*msym(p).symtab), symid(maddr(p).symid)) = comp;
    else if (basesym = lookupsym(comp->name, &bt))    /* Base symbol already resolved */
        tableentry(Symbol *, (*msym(p).symtab), symid(maddr(p).symid)) = comp = basesym;
    defcompsym(comp);
    (*IR->address)(p, comp, maddr(p).offset);
}

static void defcompsym(s) Symbol s; {
    if (s->computed)
        defcompsym(idxtable(Symbol *, (*msym(s).symtab), symid(maddr(s).symid)));
    else if (!(msym(s).announced))
        (*IR->defsymbol)(s);
}
```

6.4 Optimizations

`inlinecallgraph` applies inlining to call sites heuristically or guided by a profile. Link-time flags determine the technique used.

128a ⟨*mld local declarations* 128a⟩≡
```
dclproto(static void doinlining,(MTable));
```
This definition is continued in chunk 129b.
This code is used in chunk 144c.

128b ⟨*mld definitions* 128b⟩≡
```
static void doinlining(rootfns) MTable rootfns; {
    int i;
    Symbol rootfn;
    gentable(rootfns, i, 0) {
        rootfn = tableentry(Symbol *, rootfns, i);
        inlinecallgraph(inlineflag, rootfn, orderflag);
    }
    inlinestats();
}
```
This definition is continued in chunks 130a and 131c.
This code is used in chunk 144c.

An inlining file contains caller/callee pairs that uniquely identify call sites for inlining.

128c ⟨*read inlining file* 128c⟩≡
```
if (Inlinefp) {
    char caller[256], callee[256];
    int  id, count;
    BaseTable *bt;
    Symbol s;
    CallSiteNode *site;

    while(fscanf(Inlinefp, "%s%s%d%d", caller, callee, &id, &count) != EOF) {
        fprintf(stderr, "Inline site %s %s %d %d\n", caller, callee, id, count);
        s = lookupsym(string(caller), &bt);
        if (s) {
            site = (CallSiteNode *)alloc(CALLSITENODESZ);
            if (site->callee = lookupsym(string(callee), &bt)) {
                site->id = id;
                site->count = count;
                if (emptytable(mfunc(s)->inlinesites))
                    newtable(&(mfunc(s)->inlinesites), 4,
                             sizeof(CallSiteNode *), &permanent);
                addtotable(CallSiteNode **, &(mfunc(s)->inlinesites), site);
                if (Optfp) addtotable(Symbol *, &optprocs, s);
            } else warning("External call site %s to %s\n", caller, callee);
        } else warning("Cannot inline unknown function %s\n", caller);
    }
    fclose(Inlinefp);
}
```
This code is used in chunk 118b.

An optimization file contains procedures that have been identified as “hot” in an execution profile, i.e., they execute the highest percentage of total instructions. `mld` targets global optimizations at these “hot” procedures.

128d ⟨*external declarations* 121d⟩+≡
```
MTable optprocs;
```

129a ⟨*read optimization file* 129a⟩≡
```
if (Optfp) {
    char name[256];
    BaseTable *bt;
    Symbol s;

    newtable(&optprocs, 15, sizeof(Symbol), &permanent);
    while(fscanf(Optfp, "%s", name) != EOF) {
        fprintf(stderr, "Hot procedure %s\n", name);
        s = lookupsym(string(name), &bt);
        if (s) { addtotable(Symbol *, &optprocs, s); }
        else warning("Unknown function %s\n", name);
    }
    fclose(Optfp);
} else cleartable(optprocs, &permanent);
```
This code is used in chunk 118b.

6.5 Code Generation

`generate_code` generates code for all those procedures marked as reachable. If the `orderflag` indicates that the procedures should be emitted in a particular order, `generate_code` calls `auxgenerate_code` to walk the call graph in the specified order. If a random order is indicated, `generate_code` chooses an order and emits the procedures. The default order is each procedure by its placement in a module. If the flag for generating all procedures (`apflag`) is set, all procedures are emitted, even if they are not marked as reachable. `auxgenerate_code`

129b ⟨*mld local declarations* 128a⟩+≡
```
dclproto(static void generate_code,(MTable));
dclproto(static void auxgenerate_code,(Symbol, int));
```

129c ⟨*local declarations* 118a⟩+≡
```
static long readproctime = 0;
static long genproctime  = 0;
```

130a ⟨*mld definitions* 128b⟩+≡
```
static void generate_code(rootfns) MTable rootfns;{
    int fileid, procid;
    BaseTable *bt;
    Symbol fn;
    ProcIndex pi;
    int i, j;
    int start, end, delta;
    int pstart, pend, pdelta;

    if (orderflag != 0) {
        gentable(rootfns, i, 0) {
            auxgenerate_code(tableentry(Symbol *, rootfns, i), orderflag);
        }
    }
    if (clusterflag && Optfp)
        gentable(optprocs, i, 0) {
            fn = tableentry(Symbol *, optprocs, i);
            if (⟨fn not yet emitted 130b⟩) {
                ⟨read and generate code for procedure fn 132a⟩
            }
        }
    ⟨initialize module order 131a⟩
    for (fileid = start; fileid != end; fileid += delta) {
        bt = tableentry(BaseTable **, bmilltab, fileid);
        if (bt->visited) {
            ⟨initialize file order 131b⟩
            for (procid = pstart; procid != pend; procid += pdelta) {
                pi = tableentry(ProcIndex *, bt->procindex, procid);
                fn = idxtable(Symbol *, bt->glbsymtab, symid(pi.id));
                if (⟨fn not yet emitted 130b⟩) {
                    ⟨read and generate code for procedure fn 132a⟩
                 }
                /* procedure never called or not reachable from main */
                else if (!mfunc(fn)->emitted && mapflag)
                    fprint(2, "%s never called.\n", fn->name);
            }
        }
    }
    _printetime("Read procedures", readproctime);
    _printetime("Generate procedures", genproctime);
}
```

130b ⟨fn *not yet emitted* 130b⟩≡
```
(!mfunc(fn)->emitted &&
(apflag || indircalledproc(fn) ||
(mfunc(fn)->ncalls > 0 && mfunc(fn)->cgnid != 0)))
```
This code is used in chunk 130a.

131a ⟨*initialize module order* 131a⟩≡
```
if ((randomflag & 0x01) == 0) {
    start = 0;
    end = tablesize(bmilltab);
    delta = 1;
} else {
    start = tablesize(bmilltab) - 1;
    end = -1;
    delta = -1;
}
```
This code is used in chunk 130a.

131b ⟨*initialize file order* 131b⟩≡
```
if ((randomflag & 0x02) == 0) {
    pstart = 0;
    pend = tablesize(bt->procindex);
    pdelta = 1;
} else {
    pstart = tablesize(bt->procindex) - 1;
    pend = -1;
    pdelta = -1;
}
```
This code is used in chunk 130a.

131c ⟨*mld definitions* 128b⟩+≡
```
static void auxgenerate_code(fn, order) Symbol fn; {
    assert(fn);
    mfunc(fn)->onstack = 1;
    if (!mfunc(fn)->emitted && !mfunc(fn)->cached) {
        if (order == 1)
            ⟨generate nested descendents 131d⟩
        ⟨read and generate code for procedure fn 132a⟩
        if (order == 2)
            ⟨generate nested descendents 131d⟩
    }
    mfunc(fn)->onstack = 0;
}
```

131d ⟨*generate nested descendents* 131d⟩≡
```
{
    Symbol callee;
    CallGraphNode cgn;
    int i;
    gentable(mfunc(fn)->callgraph, i, 0) {
        cgn = tableentry(CallGraphNode *, mfunc(fn)->callgraph, i);
        callee = idxtable(Symbol *, (*msym(fn).symtab), symid(cgn.fnid));
        if (callee != fn && callee->sclass != EXTERN && !mfunc(callee)->onstack)
            auxgenerate_code(callee, order);
    }
}
```
This code is used in chunk 131c.

132a ⟨*read and generate code for procedure* `fn` 132a⟩≡

```
breadProcedure(fn);
readproctime += getetime();
doprocedure(fn);
genproctime += getetime();
mlcc_deallocate(&(mfunc(fn)->arena));
assert(arenasize(mfunc(fn)->arena) == 0);
mfunc(fn)->arena = 0;
mfunc(fn)->cached = 0;
```

This code is used in chunks 130a and 131c.

Each *module* that `mld` accesses is represented by a `BaseTable`. Modules may be stored individually in `mill` object files or with other modules in archive files.

132b ⟨*external declarations* 121d⟩+≡

```
dclproto(BaseTable *newBaseTable,(int, int, char *));
```

132c ⟨*definitions* 121c⟩+≡

```
BaseTable *newBaseTable(fd, offset, name) char *name; {
    BaseTable *bt = (BaseTable *) alloc(BASETBLSZ);
    BZERO(bt, (*bt));
    bt->fd = fd;
    bt->offset = offset;
    bt->pathname = name;
    bt->filename = name;
    return bt;
}
```

6.6 Command-line options

`stabinit` processes any flags intended for the backend. All `mill` object and archive files are opened and added to the global table of modules to be linked.

`mld` accepts too many switches. Here's what they mean.

`-allproc`	Generate code for *all* procedures.
	Default: generate only reachable procedures.
`-aout`	Generate a complete executable program.
	Default: generate an assembly file.
`-bu`	Generate code for procedures in callgraph bottom-up, post order
`-td`	Generate code for procedures in callgraph top-down, pre order
`-data=`*file*	Read binding information from *file*
`-dbg`	Emit debugging to standard error.
`-Cp`	Cluster hot procedures onto consecutive pages.
`-Hf`*file*	Hot procedures listed in *file*
`-If`*file*	Inline individual call sites named in *file*.
`-Ih`	Apply heuristic inlining algorithm.
`-Ii`	Attempt to inline converted indirect call sites.
`-Il`*N*	Set inlining size limit to *N* intermediate instructions.
`-Ip`*proc*	Inline procedure *proc* at all possible call sites.
`-m3`	Indicates linking a Modula-3 program.
`-m3O`	Indicates type analysis for Modula-3 program is applied.
`-map`	Generate an address map. Only valid with `-laout`.
`-ma`*file*	Generate ASCII `mill` code to *file*.
`-mem`	Generate summary of dynamic memory usage.
`-nc`	Do not convert method to procedure calls.
	Applies to `m3` programs only.
`-pad`	Pad optimized procedures with nops to approximate alignment of unoptimized proc.
`-O`	Apply optimizations (constant prop).
`-Oc`	Apply optimizations to callers of hot procedures.
`-Odbg`	Enable debugging during optimization.
`-Of`*file*	Optimize procedures listed in *file*.
`-Oh`	Optimize hot procedures.
`-Os`	Optimization statistics only.
`-ppu`	Permit unsafe initialization of type hierarchy.
`-r`*int*	Emit procedures in a random order.
`-time`	Emit summary of time spent in phases of `mld`.
`-Xcg`	Exclude code generation; Useful with `-Os` for getting analysis statistics.
`-Xm`*file*	Exclude all procedures in *file* from optimization.
`-Xp`*proc*	Exclude procedure *proc* from optimization.

133a ⟨*local declarations* 118a⟩+≡
```
static MTable bmilltab;
static int amillfd, bmillfd = 0;
static int apflag = 0, clusterflag = 0, orderflag = 0;
static int inlineflag = 0, libcflag = 0, timeflag = 0;
static int randomflag = 0;
static char *dataimage_file, *dataoutput_file;
```

133b ⟨*external declarations* 121d⟩+≡
```
extern int aoutflag, mapflag, dbgflag, memflag, dataflag, convertflag, indirflag;
extern int codegenflag, m3flag, m3Oflag;
extern int optflag, optcallerflag, optstatflag, optdbgflag, padflag;
extern int ppflag;
```

133c ⟨*local declarations* 118a⟩+≡
```
int aoutflag, mapflag, dbgflag, memflag, dataflag, convertflag, indirflag;
int codegenflag, m3flag, m3Oflag, optflag, optcallerflag, optstatflag, optdbgflag, padflag;
int ppflag;
```

134 ⟨*local declarations* 118a⟩+≡

```
static MTable Xproctable, Xfiletable;
MTable Inlinetable;
static FILE *Inlinefp = 0, *Optfp;
dclproto(void stabinit,(char *, int, char *[]));
```

135 ⟨*stabinit definitions* 135⟩≡

```
void stabinit(srcfile, argc, argv) char *srcfile, *argv[]; {
    int i, idx, fd, offset;
    char *amillfile = 0, *name;
    int bmillfilect, fatalerr = 0, skip;
    BaseTable *bt;
    extern unsigned (*emitter) ARGS((Node, int));

    aoutflag = mapflag = dbgflag = memflag = dataflag = 0;
    ppflag = 0;
    convertflag = codegenflag = 1;
    indirflag = m3flag = m3Oflag = 0;
    optflag = optcallerflag = optstatflag = optdbgflag = padflag =  0;
    newtable(&Xproctable, 8, sizeof(char *), &permanent);
    newtable(&Xfiletable, 8, sizeof(char *), &permanent);
    newtable(&Inlinetable, 8, sizeof(char *), &permanent);
    indirproc = string("_indirproc");
    fileInit();     /* Initialize file management module */
    bmillfilect = 0;
    for (i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-map") == 0) mapflag = 1;
        else if (strncmp(argv[i], "-ma", 3) == 0) {
            if (strlen(argv[i]) > 4)
                amillfile = string(&(argv[i][3]));
            else amillfile = (char *)"-";
        }
        else if (strncmp(argv[i], "-data=", 6) == 0) {
            dataimage_file = string(&(argv[i][6]));
            dataflag = 1;
        }
        else if (strcmp(argv[i], "-allproc") == 0) apflag = 1;
        else if (strcmp(argv[i], "-aout") == 0)    aoutflag = 1;
        else if (strcmp(argv[i], "-bu") == 0) orderflag = 1;
        else if (strcmp(argv[i], "-binary") == 0)  {
            bflag = 1;
            emitter = emitbin;
        } else if (strncmp(argv[i], "-Cl", 3) == 0) Clonelimit = atoi(&(argv[i][3]));
        else if (strcmp(argv[i], "-Cp") == 0)      clusterflag = 1;
        else if (strcmp(argv[i], "-dbg") == 0)      dbgflag = 1;
        else if (strncmp(argv[i], "-If", 3) == 0) {
            if (inlineflag != 0)
                warning("-If ignored.  Only one of -I[fhp] supported\n");
            else if (Inlinefp = fopen(&(argv[i][3]), "r"))
                inlineflag = 1;
            else warning("Cannot open inline input file %s\n", &(argv[i][3]));
        }
        else if (strcmp(argv[i], "-Ih") == 0) {
            if (inlineflag != 0)
                warning("-Ih ignored.  Only one of -I[fhp] supported\n");
            else inlineflag = 2;
        } else if (strcmp(argv[i], "-Ii") == 0) indirflag = 1;
        else if (strncmp(argv[i], "-Il", 3) == 0) Inlinelimit = atoi(&(argv[i][3]));
        else if (strncmp(argv[i], "-Ip", 3) == 0) {
            if (inlineflag != 0)
```

```
                warning("-Ip ignored.  Only one of -I[fhp] supported\n");
            else {
                name = string(&(argv[i][3]));
                addtotable(char **, &Inlinetable, name);
                inlineflag = 3;
            }
        }
        else if (strcmp(argv[i], "-llibc") == 0)   libcflag = 1;
        else if (strcmp(argv[i], "-mem") == 0)     memflag = 1;
        else if (strcmp(argv[i], "-m3") == 0)      m3flag = 1;
        else if (strcmp(argv[i], "-m30") == 0)     m30flag = 1;
        else if (strcmp(argv[i], "-nc") == 0)      convertflag = 0;
        else if (strcmp(argv[i], "-pad") == 0)     padflag = 1;
        else if (strcmp(argv[i], "-O") == 0)       optflag = 1;
        else if (strcmp(argv[i], "-Oa") == 0)      optflag = 3;
        else if (strcmp(argv[i], "-Oc") == 0)      optcallerflag = 1;
        else if (strncmp(argv[i], "-Of", 3) == 0) {
            if (Optfp = fopen(&(argv[i][3]), "r")) optflag = 2;
            else warning("Cannot open optimization input file %s\n", argv[i][3]);
        }
        else if (strncmp(argv[i], "-Hf", 3) == 0) {
            if (!(Optfp = fopen(&(argv[i][3]), "r")))
                warning("Cannot open hot procedures file %s\n", argv[i][3]);
        }
        else if (strcmp(argv[i], "-Oh") == 0)      optflag = 2;
        else if (strcmp(argv[i], "-Odbg") == 0)    optdbgflag = 1;
        else if (strcmp(argv[i], "-Os") == 0)      optstatflag = 1;
        else if (strcmp(argv[i], "-ppu") == 0)     unsafe = 1;
        else if (strncmp(argv[i], "-r", 2) == 0)   randomflag = argv[i][2] - '0';
        else if (strcmp(argv[i], "-td") == 0) orderflag = 2;
        else if (strcmp(argv[i], "-time") == 0)    timeflag = 1;
        else if (strncmp(argv[i], "-Xcg", 4) == 0) codegenflag = 0;
        else if (strncmp(argv[i], "-Xm", 3) == 0) {
            name = string(&(argv[i][3]));
            addtotable(char **, &Xfiletable, name);
        } else if (strncmp(argv[i], "-Xp", 3) == 0) {
            name = string(&(argv[i][3]));
            addtotable(char **, &Xproctable, name);
        }
        else if (strcmp(argv[i], "-o") == 0) {
            char *tmp;

            i += 1;
            tmp = alloc(strlen(argv[i]) + 5);
            strcpy(tmp, argv[i]);
            strcat(tmp, ".data");
            dataoutput_file = string(tmp);
            bmillfd = open(argv[i], O_WRONLY|O_CREAT);
            chmod(argv[i], 00644);
            argv[i] = "";
        }
        else if (strstr(argv[i], ".a") ||
             strstr(argv[i], ".o") ||
             strstr(argv[i], ".io") ||
```

```
            strstr(argv[i], ".mo")) bmillfilect++;
        else warning("mld: Argument %s ignored\n", argv[i]);
    }
    /* Open all bmill files for linking */
    newtable(&bmilltab, bmillfilect, sizeof(BaseTable *), &permanent);
    for (idx = 0, i = 1; i < argc; i++) {
        if (strstr(argv[i], ".o") ||
            strstr(argv[i], ".io") ||
            strstr(argv[i], ".mo")) {
            bt = newBaseTable(0, 0, string(argv[i]));
            if (fileOpen(bt)) {
                updatetable(BaseTable **, (&bmilltab), idx, bt);
                idx++;
            }
        } else if (strstr(argv[i], ".a")) {
            fd = archiveOpen(argv[i]);
            idx = archiveRead(fd, idx, &bmilltab, skip = 0);
        }
    }
    if (amillfile) {
        if (strcmp(amillfile, "-") == 0) amillfd = dup(2);
        else amillfd = open(amillfile, O_WRONLY|O_CREAT);
        if (amillfd == -1) {
            error("mld: Cannot open ASCII MILL file %s\n", amillfile);
            fatalerr++;
        } else chmod(amillfile, 00644);
    }
    if (fatalerr) exit(fatalerr);
    /* Program processed in program() */
}
```

This code is used in chunks 144c and 145a.

Unused stubs required by front end.

137a ⟨*local declarations* 118a⟩+≡

```
dclproto(void stabsym,(Symbol));
dclproto(void stabtype,(Symbol));
```

137b ⟨*definitions* 121c⟩+≡

```
void stabsym(s) Symbol s; {}
void stabtype(s) Symbol s; {}

void compound(a, b, lev) struct swtch *b; {}
Type typename() { return 0; }
```

6.7 Tracing, timing, and instrumentation

`printetime` prints the elapsed wall-clock time since its last call. It is used to report the elapsed times of `mld`'s phases.

137c ⟨*local declarations* 118a⟩+≡

```
static struct timeval etp = { 0, 0 };
dclproto(static void printetime,(char *msg));
dclproto(static void _printetime,(char *msg, int));
dclproto(static int  getetime,(void));
```

138a ⟨*definitions* 121c⟩+≡
```
static void printetime(msg) char *msg; {
    _printetime(msg, getetime());
}
static void _printetime(msg, ttime) char *msg; int ttime; {
    if (timeflag)
        fprintf(stderr, "%s\t%f sec\n", msg, ((float)ttime)/1000000);
}
static int getetime() {
    struct timeval tp;
    struct timezone tzp;
    int ttime = 0, etime;
    dclproto(extern void gettimeofday,(struct timeval *tp, struct timezone *tzp));

    if (timeflag) {
        gettimeofday(&tp, &tzp);
        if (etp.tv_sec != 0) {
            ttime = (tp.tv_sec * 1000000) + tp.tv_usec;
            etime = (etp.tv_sec * 1000000) + etp.tv_usec;
            ttime -= etime;
        }
        etp.tv_sec = tp.tv_sec;
        etp.tv_usec = tp.tv_usec;
    }
    return ttime;
}
```

`memory_usage` reports statistics about `mld`'s dynamic memory usage. Script for summarizing memory usage found in /u/mff/src/mylcc/bin/summem.

138b ⟨*memory instrumentation* 138b⟩≡
```
if (memflag) memory_usage();
```
This code is used in chunk 118b.

138c ⟨*definitions* 121c⟩+≡
```
void memory_usage()
{
    int i, fileid;
    BaseTable *bt;

    i = 0;
    gentable(bmilltab, fileid, 0) {
        bt = tableentry(BaseTable **, bmilltab, fileid);
        if (bt->visited)
            i += breadProgramSize(bt);
    }
    strtablesize();
    fprint(2,"total memory usage %d\n", i);
    fprint(2,"permanent %d\n", arenasize(permanent));
    fprint(2,"label arena %d\n", rlabelarenasize());
    fprint(2,"block arena %d\n", rblockarenasize());
    tabledata();
}
```

6.8 mldpp: A link-time preprocessor

mldpp's **program** template is similar to **mld**'s. After identifying reachable modules and resolving external symbols, **mldpp** calls **LTInit** execute the Modula–3 link-time initialization procedure, which generates a binding file for the complete program.

139 ⟨*local declarations* 118a⟩+≡

```
#define RUNTIME_ADDRESS     0
#define LINKTIME_ADDRESS    1
static  int unsafe = 0;
```

140 ⟨*pp program* 140⟩≡

```
void program(void) {
    int fileid, procid, gsym, i, cgnid, binding_address;
    int totsegmentsizes[] = { 0, 0, 0, 0, 0, 0 };
    int meanprocsize = 0, nprocs = 0;
    int meanmethodsize = 0, nmethods = 0;
    int meanm3procsize = 0, nm3procs = 0;
    BaseTable *bt, *mainmodule, *exitmodule = 0;
    Symbol p, mainsym, exitsym;
    ProcIndex pi;
    MTable rootfns;
    FILE *datafp;

    dclproto(void dispatch,(void));
    dclproto(char *strstr,(char *, char *));

    ppflag = 1;
    fail = &mldfail;
    newtable(&rootfns, 8, sizeof(Symbol), &permanent);
    printetime("");     /* prime the timing pump */

    ⟨open and read modules; install global symbols 122a⟩
    ⟨identify reachable modules 120a⟩

    binding_address = LINKTIME_ADDRESS;
    ⟨ compute sizes of abstract data segments and initialize real segments  119a⟩

    ⟨eliminate unused modules 120b⟩

    /* Announce unresolved global symbols */
    if (mapflag) doexternals(amillfd);

    ⟨layout data 123a⟩
    dispatch();
    printetime("Dispatch\t\t");

    /* link-time preprocessing */
    if (bmillfd) close(bmillfd);
    assert(datafp = fopen(dataoutput_file, "w"));
    LTInit(unsafe, datafp);
    ⟨compute mean size of procedures and methods 141a⟩
    printetime("Link-time init\t");
    fclose(datafp);

    ⟨ close open files  119f⟩
    fflush(stdout);
}
```

This code is used in chunk 145a.

141a ⟨*compute mean size of procedures and methods* 141a⟩≡
```
gentable(bmilltab, fileid, 0) {
    bt = tableentry(BaseTable **, bmilltab, fileid);
    if (bt->visited) {
        gentable(bt->procindex, procid, 0) {
            pi = tableentry(ProcIndex *, bt->procindex, procid);
            p = idxtable(Symbol *, bt->glbsymtab, symid(pi.id));
            if (msym(p).method) {
                nmethods++;
                meanmethodsize += mfunc(p)->dagsize;
            } else if (strstr(p->name, "_map_") || strstr(p->name, "_setup_")) {
                nm3procs++;
                meanm3procsize += mfunc(p)->dagsize;
            } else {
                nprocs++;
                meanprocsize += mfunc(p)->dagsize;
            }
        }
    }
}
printf("# compiler procs:\t%d\tmean size %f\n", nm3procs,
          nm3procs > 0 ? (float)meanm3procsize/nm3procs : 0);
printf("# procedures:\t%d\tmean size %f\n", nprocs,
          nprocs > 0 ? (float)meanprocsize/nprocs : 0);
printf("# methods:\t%d\tmean size %f\n", nmethods,
          nmethods > 0 ? (float)meanmethodsize/nmethods : 0);
```

This code is used in chunk 140.

6.9 `mranlib`: An indexer for `mill` archives

`mranlib` takes an archive of `mill` object files and produces a table of contents for the archive.

141b ⟨*mranlib program* 141b⟩≡
```
void program(void) {
    int fileid, gsym;
    Symbol p;
    ProcIndex pi;
    BaseTable *bt;
    printetime("");     /* prime the timing pump */

    ⟨create table of contents 142a⟩

    ⟨ close open files  119f⟩
    fflush(stdout);
}
```
This code is used in chunk 145b.

142a ⟨*create table of contents* 142a⟩≡

```
{
    static struct arena first = {
        0, 0, 0, &first, 0
    };
    static Arena strarena = &first;     /* set storage */
    static Strings lstrs = { 0, 0, &strarena };
    char *str;
    int extsymct = 0;
    unsigned short idx;
    BaseTable *arbt;

    gentable(bmilltab, fileid, 0) {
        bt = tableentry(BaseTable **, bmilltab, fileid);
        breadModuleHdr(bt);
        breadProgram(bt);
        gentable(bt->glbsymtab, gsym, 1) {
            p = tableentry(Symbol *, bt->glbsymtab, gsym);
            if (⟨ranlib symbol 142b⟩) {
                str = ustringn(p->name, strlen(p->name), &lstrs);
                extsymct++;
            }
        }
    }
    printetime("Read archive\t");
    fprint(2, "Ranlib entries %d\n", extsymct);
    arbt = newBaseTable(0, 0, "ranlib");
    ranlibOpen(bt->fd, arbt, tablesize(bmilltab), extsymct);
    gentable(bmilltab, fileid, 0) {
        bt = tableentry(BaseTable **, bmilltab, fileid);
        gentable(bt->glbsymtab, gsym, 1) {
            p = tableentry(Symbol *, bt->glbsymtab, gsym);
            if (⟨ranlib symbol 142b⟩) {
                idx = ustridx(p->name, strlen(p->name), &lstrs);
                fprint(2,"%s in %d, %d\n", p->name, fileid, idx);
                ranlibEntry(arbt, idx, fileid);
            }
        }
    }
    ranlibStrings(arbt, &lstrs);
    ranlibClose(bt->fd, arbt, tablesize(bmilltab));
    printetime("Write ranlib\t");
}
```

This code is used in chunk 141b.

142b ⟨*ranlib symbol* 142b⟩≡

```
(p->scope == GLOBAL &&
(p->sclass == AUTO ||
(p->sclass == STATIC && p->type && isfunc(p->type))) &&
!(msym(p).typesym))
```

This code is used in chunk 142a.

143 ⟨*mranlib stabinit definitions* 143⟩≡

```
void stabinit(srcfile, argc, argv) char *srcfile, *argv[]; {
    int i, idx, fd, offset;
    char *amillfile = 0, *name;
    int bmillfilect, fatalerr = 0, skip;
    BaseTable *bt;
    extern unsigned (*emitter) ARGS((Node, int));

    indirproc = string("_indirproc");
    fileInit();     /* Initialize file management module */
    bmillfilect = 0;
    for (i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-dbg") == 0)      dbgflag = 1;
        else if (strcmp(argv[i], "-time") == 0)    timeflag = 1;
        else if (strstr(argv[i], ".a")) {
            if (bmillfilect > 0) {
                error("Only one archive can be specified\n");
                exit(1);
            }
            bmillfilect++;
        }  else warning("Argument %s ignored\n", argv[i]);
    }
    newtable(&bmilltab, bmillfilect, sizeof(BaseTable *), &permanent);
    for (idx = 0, i = 1; i < argc; i++) {
        if (strstr(argv[i], ".a")) {
            fd = archiveOpen(argv[i]);
            idx = archiveRead(fd, idx, &bmilltab, skip = 1);
        } else if (strstr(argv[i], ".o") ||
                   strstr(argv[i], ".io") ||
                   strstr(argv[i], ".mo"))
            error("Invalid archive name %s\n", argv[i]);
    }
    if (amillfile) {
        if (strcmp(amillfile, "-") == 0) amillfd = dup(2);
        else amillfd = open(amillfile, O_WRONLY|O_CREAT);
        if (amillfd == -1) {
            error("mld: Cannot open ASCII MILL file %s\n", amillfile);
            fatalerr++;
        } else chmod(amillfile, 00644);
    }
    if (fatalerr) exit(fatalerr);
    /* Program processed in program() */
}
```

This code is used in chunk 145b.

6.10 Boilerplates for mld, mldpp, and mranlib

mldfail issues an error message, but doesn't increment the error count. It is called by the New Jersey Machine-Code toolkit when an error occurs in its library.

144a ⟨*definitions* 121c⟩+≡

```
static void mldfail(char *fmt, ...) {
    va_list ap;

    va_init(ap, fmt);
    fprint(2, "error at %s %x %d: ", crb()->label->name,
               label_location(crb()->label)+lc(), lc());
    vfprint(2, fmt, ap);
    fprint(2,"\n");
    va_end(ap);
}
```

144b ⟨*mld.h* 144b⟩≡

```
#ifndef INCLUDE_MLDH
#define INCLUDE_MLDH
⟨external declarations 121d⟩
#endif
```

This code is written to file mld.h.

144c ⟨*mld.c* 144c⟩≡

```
#include <stdio.h>
#include <c.h>
#include <mld.h>
#include <optmill.h>
#include <resextsym.h>
#include <sys/file.h>
#include <sys/time.h>
#include <m3.h>
#include <analysis.h>
#include <binary.h>
#include <string.h>
#include <tk.h>
#include <decl.h>
#include <aemit.h>
#include <archive.h>

⟨external definitions 127b⟩
⟨mld local declarations 128a⟩
⟨local declarations 118a⟩
⟨data definitions 124a⟩
⟨mld definitions 128b⟩
⟨definitions 121c⟩
⟨stabinit definitions 135⟩
⟨main program 118b⟩
```

This code is written to file mld.c.

145a ⟨*mldpp.c* 145a⟩≡

```
#include <stdio.h>
#include <c.h>
#include <mld.h>
#include <optmill.h>
#include <dataimage.h>
#include <resextsym.h>
#include <string.h>
#include <sys/file.h>
#include <sys/time.h>
#include <binary.h>
#include <lttype.h>
#include <aemit.h>
#include <archive.h>
⟨external definitions 127b⟩
⟨local declarations 118a⟩
⟨data definitions 124a⟩
⟨definitions 121c⟩
⟨stabinit definitions 135⟩
⟨pp program 140⟩
```

This code is written to file `mldpp.c`.

145b ⟨*mranlib.c* 145b⟩≡

```
#include <stdio.h>
#include <c.h>
#include <mld.h>
#include <resextsym.h>
#include <string.h>
#include <sys/file.h>
#include <sys/time.h>
#include <archive.h>
#include <aemit.h>

⟨external definitions 127b⟩
⟨local declarations 118a⟩
⟨data definitions 124a⟩
⟨definitions 121c⟩
⟨mranlib stabinit definitions 143⟩
⟨mranlib program 141b⟩
```

This code is written to file `mranlib.c`.

146 ⟨*header files* 146⟩≡

```
#include <stdio.h>
#include <c.h>
#include <mld.h>
#include <dataimage.h>
#include <resextsym.h>
#include <string.h>
#include <sys/file.h>
#include <sys/time.h>
#include <binary.h>
#include <lttype.h>
#include <aemit.h>
#include <archive.h>
```

This code is written to file `header files`.

6.11 Resolving external symbols

`mld` maintains a meta-global symbol table that resolves the definition of a symbol with its references in external modules.

A global resolution record (`GlbResRec`) contains the name of a symbol, the symbol itself, the module in which it is defined, its abstract relocatable address, and a list of symbols that represent its external references before it is resolved.

147a ⟨*local declarations* 147a⟩≡
```
typedef struct glbsymrec {
    char    *name;        /* name of symbol to resolve */
    Symbol  ressym;       /* Non-null if symbol resolved */
    BaseTable *bt;        /* Non-null if module resolved */
    RAddr     raddr;      /* relocatable address of symbol */
    List      extsyms;    /* Non-null if external symbols not resolved */
    struct    glbsymrec *nxt;    /* next symbol in hash chain */
}*GlbResRec;
#define        GlbResRecSZ    (sizeof(struct glbsymrec))
```
This definition is continued in chunk 147.
This code is used in chunk 151c.

The `glbsymtab` hash table maintains chains of `GlbResRec`s.

147b ⟨*local declarations* 147a⟩+≡
```
#define GSYMSZ    257
#define hashstrname(_s)        ((((int)_s & 0x00000ff0) >> 4) % GSYMSZ)
#define uniquename(n)    stringf("%s_%d", n, genlabel(1))
static GlbResRec glbsymtab[GSYMSZ];
```

`lookupresrec` returns the resolution record associated with a name. It creates one if it doesn't already exist.

147c ⟨*local declarations* 147a⟩+≡
```
dclproto(static GlbResRec lookupresrec,(char *));
```

147d ⟨*definitions* 147d⟩≡
```
static GlbResRec lookupresrec(name) char *name; {
    int     hashid;
    GlbResRec r;

    hashid = hashstrname(name);    /* Find symbol's resolution record */
    for (r = glbsymtab[hashid]; r; r = r->nxt)
        if (name == r->name)
            break;
    if (!r) {
        r = (GlbResRec)alloc(GlbResRecSZ);
        bzero((char *)r, GlbResRecSZ);
        r->name = name;
        r->raddr = addr_new(label_new(name), 0);
        r->nxt = glbsymtab[hashid];
        glbsymtab[hashid] = r;
    }
    return r;
}
```
This definition is continued in chunks 148–51.
This code is used in chunk 151c.

lookupsym returns the symbol that resolves the name **name** and the module in which it is defined, if it is resolved.

148a ⟨*external declarations* 148a⟩≡
```
dclproto(Symbol lookupsym,(char *name, BaseTable **));
```
This definition is continued in chunks 148c and 150e.
This code is used in chunk 151b.

148b ⟨*definitions* 147d⟩+≡
```
Symbol lookupsym(name, bt) char *name; BaseTable **bt;{
    GlbResRec r = lookupresrec(name);
    *bt = r->bt;
    return r->ressym;
}
```

resolveextsym resolves the identity of an external symbol.

148c ⟨*external declarations* 148a⟩+≡
```
dclproto(BaseTable *resolveextsym,(Symbol *));
dclproto(int resolvesym,(Symbol, BaseTable *));
dclproto(void resolvename,(char *, BaseTable *));
dclproto(int nprocs,(void));
```

148d ⟨*definitions* 147d⟩+≡
```
BaseTable *resolveextsym(sym) Symbol *sym; {
    GlbResRec r;
    Symbol p = *sym;
    int id;

    assert(p->sclass == EXTERN);
    r = lookupresrec(p->name);
    if (r->ressym) {    /* Global already resolved */
        id = symid(msym(p).id);
        mergesymbols(r->ressym, tableentry(Symbol *, (*msym(p).symtab), id));
        if (msym(p).symtab)
            updatetable(Symbol *, msym(p).symtab, id, r->ressym);
        p = r->ressym;
    } else {            /* Global not yet resolved */
        if (!msym(p).raddr) msym(p).raddr = r->raddr;
        assert(msym(p).raddr == r->raddr);
        r->extsyms = push((Generic)p, r->extsyms);
    }
    *sym = p;
    return r->bt;
}
```

`resolvesym` resolves external references to symbol `p`. Constants are installed in the `constants` table and thus are excluded from the global symbol table. Static symbols are assigned unique names and unique relocatable addresses. External symbols that have not been resolved are installed in the table for future update. Global symbols are installed in the table `glbsymtab`. Any external references to a global are resolved by replacing the external symbol with its resolved global symbol in the external symbol's symbol table. Thus references to the resolved external symbol refer to the correct symbol.

149 ⟨*definitions* 147d⟩+≡

```
int resolvesym(p, bt) Symbol p; BaseTable *bt; {
    GlbResRec r;
    List    extsyms;
    Symbol    esym;
    int resolved = 0;

    if (p->scope == CONSTANTS && msym(p).symtab != 0) {
        updatetable(Symbol *, msym(p).symtab,
                    (int)symid(msym(p).id), constant(p->type, p->u.c.v));
        return 0;
    } else if (msym(p).typesym == 1) return 0;
    else if (p->generated == 1 || (p->name && num(p->name))) {
        if (!msym(p).raddr) msym(p).raddr = addr_new(label_new(p->name), 0);
        return 0;
    }
    if (p->sclass == STATIC) {
        r = lookupresrec(p->name);
        if (r->ressym) {
            ⟨give static p a unique name 150b⟩
        }
        r->ressym = p;
        if (mapflag && bt && bt->filename)
            fprint(2,"%s in %s\n", p->name, bt->filename);
        if (!msym(p).raddr) msym(p).raddr = addr_new(label_new(p->name), 0);
    } else {
        r = lookupresrec(p->name);
        if (r->ressym) {
            ⟨give multiply defined p a unique name 150c⟩
        } else {
            ⟨resolve external references to p 150d⟩
        }
        r->bt = bt;
    }
    return resolved;
}
```

resolvename is called when the module in which a symbol is defined is known, but the actual symbol is still unknown. This is useful when a library has a table of contents mapping string names to modules.

150a ⟨*definitions* 147d⟩+≡
```
void resolvename(name, bt) char *name; BaseTable *bt; {
    GlbResRec r;
    r = lookupresrec(name);
    if (r->ressym)
        warning("mld: %s multiply defined in: %s and %s.\n", name,
                (r->bt && r->bt->filename) ? r->bt->filename : "???",
                (bt && bt->filename) ? bt->filename : "???");
    r->bt = bt;
}
```

Why the special case for `mldpp`? Because this is a hack that works for Modula–3. All external names are guaranteed to be unique in Modula–3; `mldpp` needs to emit the external names, not those used by `mld` internally so it skips this step.

150b ⟨*give static* `p` *a unique name* 150b⟩≡
```
if (ppflag) warning("mld: %s already defined.\n", p->name);
if (!ppflag) p->name = uniquename(p->name);
r = lookupresrec(p->name);
```

This code is used in chunk 149.

150c ⟨*give multiply defined* `p` *a unique name* 150c⟩≡
```
warning("mld: %s multiply defined in: %s and %s.\n", p->name,
                    (r->bt && r->bt->filename) ? r->bt->filename : "???",
                    (bt && bt->filename) ? bt->filename : "???");
p->name = uniquename(p->name);
if (p->type && isfunc(p->type))
    if (!msym(p).raddr) msym(p).raddr = addr_new(label_new(p->name), 0);
else     /* if a symbol is multiply defined, choose the first definition */
    r->extsyms = push((Generic)p, r->extsyms);
```

This code is used in chunk 149.

150d ⟨*resolve external references to* `p` 150d⟩≡
```
r->ressym = p;
resolved = (r->extsyms != (List)0);
for (extsyms = r->extsyms; extsyms; extsyms = extsyms->link) {
    esym = (Symbol)(extsyms->x);
    mergesymbols(p, esym);
    if (msym(esym).symtab)
        updatetable(Symbol *, msym(esym).symtab,
                    (int)symid(msym(esym).id), p);
}
r->extsyms = 0;    /* <== Reclaim memory here */
if (!msym(p).raddr) msym(p).raddr = r->raddr;
assert(msym(p).raddr == r->raddr);
```

This code is used in chunk 149.

doexternals announces any externals that remain unresolved.

150e ⟨*external declarations* 148a⟩+≡
```
dclproto(int doexternals,(int));
```

151a ⟨*definitions* 147d⟩+≡
```
int doexternals(amillfd) {
    int     i, unresolved;
    GlbResRec r;
    Symbol    p, esym;
    List    extsyms;

    for (unresolved = 0, i = 0; i < GSYMSZ; i++)
        for (r = glbsymtab[i]; r; r = r->nxt) {
            if (r->extsyms) p = (Symbol)r->extsyms->x;
            else  p = (Symbol) 0;
            if (r->ressym && amillfd) emitSymbol(amillfd, r->ressym, "");
            if (r->ressym && r->extsyms) {
                p = r->ressym;
                for (extsyms = r->extsyms; extsyms; extsyms = extsyms->link) {
                    esym = (Symbol)(extsyms->x);
                    warning("Using one copy of multiply defined symbol %s\n", p->name);
                    mergesymbols(p, esym);
                    if (msym(esym).symtab)
                        updatetable(Symbol *, msym(esym).symtab,
                                    (int)symid(msym(esym).id), p);
                }
                r->extsyms = 0;    /* <== Reclaim memory here */
            } else if (!r->ressym && r->extsyms && !p->computed) {
                assert(p);
                if (p->sclass == EXTERN) {
                    if (aoutflag) warning("mld: %s unresolved \n", p->name);
                    unresolved = 1;
                }
            }
        }
    return unresolved;
}
```

151b ⟨*resextsym.h* 151b⟩≡
```
#ifndef INCLUDE_REXTH
#define INCLUDE_REXTH
⟨external declarations 148a⟩
#endif
```

This code is written to file `resextsym.h`.

151c ⟨*resextsym.c* 151c⟩≡
```
#include <c.h>
#include <mld.h>
#include <resextsym.h>
#include <string.h>

⟨local declarations 147a⟩
⟨definitions 147d⟩
```
This code is written to file `resextsym.c`.

6.12 Converting packed `mill` to unpacked `mill` code

Procedures are converted from their packed, on-disk representation in `mill` code into an unpacked, dynamic representation of `Node`s and `Symbol`s before being processed by the optimizer and code generator.

`mill2dags` takes a `mill` function and walks its flow graph, converting the flattened instruction dags associated with each flow-graph node into their dynamic representation. Every intermediate instruction is seen here, so it is a logical place to apply some optimizations. `dag_simplify` applies data-driven simplification to each instruction tree if `optflag` is set.

152a ⟨*external declarations* 152a⟩≡

```
dclproto(extern void mill2dags,(MFunction *mfn, int));
dclproto(extern Node dag_simplify,(Node, Type));
```

This definition is continued in chunks 153d, 154c, and 156a.
This code is used in chunk 158b.

152b ⟨*definitions* 152b⟩≡

```
void mill2dags(mfn, optflag) MFunction *mfn; {
    FGNode  fgnode;
    int     i, hi, low, fgidx, loc;
    Node    olddagn, dagn;
    MTable  glbsymtab, stack, symtab;
    int     ndagroots = 0;

    ⟨initialize symbol and node tables 153a⟩

    gentable(mfn->flowgraph, fgidx, 1) {
        fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
        fgnode->node = (Node)0;
        ndagroots += tablesize(fgnode->dagnodes);
        newtable(&fgnode->dagroots, tablesize(fgnode->dagnodes),
                 sizeof(int), mfn->nodetable.arena);
        if (fgnode->type == CallN && callN(fgnode).mfn) {
            mill2dags(callN(fgnode).mfn, optflag);
        }
        else if (fgnode->type == MultitestN) {
            ⟨ unmill a Multitest flow-graph node 153b⟩
        }

        if (fgnode->labelsym)
            fgnode->node = newnode(LABELV, (Node)0, (Node)0, symfor(fgnode->labelsym));
        gentable(fgnode->dagnodes, i, 0) {
            dagrange(fgnode->dagnodes, i, low, hi);
            dagn = mill2dag(low, hi, &stack, mfn, &loc);
            ⟨ simplify dagn 153c⟩
            if (dagn && rootop(generic(dagn->op)))
                updatetable(int *, &fgnode->dagroots, i, loc);
        }
    }
    mfn->dagroots = ndagroots;
}
```

This definition is continued in chunks 154–58.
This code is used in chunk 158c.

153a ⟨*initialize symbol and node tables* 153a⟩≡
```
newtable(&stack, 16, sizeof(int), &transient);
if (emptytable(mfn->nodetable)) {
    newtable(&mfn->nodetable, mfn->dagtable.maxelem+2,
             sizeof(Node *), mfn->dagtable.arena);
    mfn->nodetable.maxelem = mfn->dagtable.maxelem;
}
```
This code is used in chunks 152b and 155a.

153b ⟨ *unmill a* `Multitest` *flow-graph node* 153b⟩≡
```
Symbol     s;
unsigned short     t;
newtable(&multitestN(fgnode).labelsyms, tablesize(multitestN(fgnode).labels),
         sizeof(Symbol *), &transient);
t = multitestN(fgnode).deflab;
multitestN(fgnode).deflabsym = symfor(t);
gentable(multitestN(fgnode).labels, i, 0) {
    t = tableentry(unsigned short *, multitestN(fgnode).labels, i);
    s = symfor(t);
    updatetable(Symbol *, (&multitestN(fgnode).labelsyms), i, s);
}
```
This code is used in chunk 152b.

153c ⟨ *simplify* `dagn` 153c⟩≡
```
if (dagn && optflag) {
    if (condop(generic(dagn->op))) olddagn = dagn;
    else olddagn = (Node)0;
    dagn = dag_simplify(dagn, (Type)0);
    if (olddagn) {
        dagn = constcondexpr(mfn, fgnode, olddagn, dagn);
        tableentry(Node *, mfn->nodetable, loc) = dagn;
    }
}
```
This code is used in chunk 152b.

`condop` matches conditional operators. `rootop` matches those operators that may be at the roots of instruction trees.

153d ⟨*external declarations* 152a⟩+≡
```
#define condop(op) (op == EQ || op == GE || op == GT || op == LE || op == LT || op == NE)
#define rootop(op) (op == ASGN || condop(op) || op == ARG || op == CALL || op == RET || \
                op == JUMP || op == LABEL)
```

153e ⟨*local declarations* 153e⟩≡
```
#define nodefor(_i) (assert((_i) > 0), idxtable(Node *, mfn->nodetable, (_i)))
#define symfor(_i)  (isglobal(_i) ? (idxtable(Symbol *, mfn->bt->glbsymtab, symid(_i))) : \
                 (idxtable(Symbol *, mfn->localsymtab, _i)))
```
This definition is continued in chunks 153f and 157.
This code is used in chunk 158c.

`mill2dag` converts a range of flattened instructions into their dynamic representation as a single instruction tree (`Node`).

153f ⟨*local declarations* 153e⟩+≡
```
dclproto(static Node mill2dag,(int, int, MTable *, MFunction *, int *));
```

154a ⟨*definitions* 152b⟩+≡
```
static Node mill2dag(low, hi, stack, mfn, loc)
        MTable *stack; MFunction *mfn; int *loc; {
    Node p = 0, k0, k1;
    int i, j;
    MDag mdag;

    for (i = low; i <= hi; i = j) {
        j = unmilldag(i, &mdag, mfn->dagtable, stack);
        if (mdag.opcode != NOP) {
            if (mdag.k0) {
                k0 = nodefor(mdag.k0);
                k0->x.dagid = mdag.k0;
            } else k0 = (Node)0;
            if (mdag.k1) {
                k1 = nodefor(mdag.k1);
                k1->x.dagid = mdag.k1;
            } else k1 = (Node)0;
            if (mdag.opcode == RETV)
                ⟨create a new RETI node 154b⟩
            else p = newnode(mdag.opcode, k0, k1,
                             mdag.sym0 ? symfor(mdag.sym0) : (Symbol)0);
            if (generic(p->op) == ASGN && isaddrop(p->kids[0]->op) &&
                p->kids[0]->syms[0]->temporary == 1 && mtemp(p->kids[0]->syms[0]).cse) {
                p->kids[0]->syms[0]->u.t.cse = p->kids[1];
            }
            if (mdag.sym1) p->syms[1] = symfor(mdag.sym1);
            tableentry(Node *, mfn->nodetable, i) = p;
        } else p = (Node)0;
        *loc = i;
    }
    return p;
}
```

154b ⟨*create a new* RETI *node* 154b⟩≡
```
{
    p = newnode(CNST+I, 0, 0, symfor(GLOBSYM(15)));
    p = newnode(RET+I, p, 0, 0);
}
```
This code is used in chunk 154a.

`convert_site` attempts to convert an indirect call to a direct call. It is called from `walkcallgraph` *before* any procedures have been unmilled. Here's an example of a phase ordering problem. `walkcallgraph` wants to compute the most precise call graph as possible, but indirect calls limit its precision. Data-driven simplification can convert calls sites, but to apply that technique, the call site must be converted into its dynamic representation. `convert_site` makes these transformations and if the site is convertible, it updates the caller's un-MILLed instructions to reflect the direct call.

Initially, it seemed to be a good idea to be able to transform `mill` instructions in both their on-disk and dynamic representations. In retrospect, this choice complicates the implementation and induces redundancy. Transforming only the dynamic representation is much easier and safer.

154c ⟨*external declarations* 152a⟩+≡
```
dclproto(Symbol convert_site,(Symbol, CallGraphNode *));
```

155a ⟨*definitions* 152b⟩+≡
```
Symbol convert_site(caller, cgn) Symbol caller; CallGraphNode *cgn;{
    MFunction *mfn;
    FGNode  fgnode;
    int     i, hi, low, loc;
    Node    dagn, oldn;
    MTable  stack;
    Symbol  callee = 0;

fprint(2,"Attempt to convert site in %s\n", caller->name);
    if (!mfunc(caller)->cached) breadProcedure(caller);
    mfn = &mfn(caller);
    ⟨initialize symbol and node tables 153a⟩
    fgnode = tableentry(FGNode *, mfn->flowgraph, cgn->fgnid);
    assert(fgnode->type == CallN && !callN(fgnode).mfn);
    gentable(fgnode->dagnodes, i, 0) {
        dagrange(fgnode->dagnodes, i, low, hi);
        oldn = mill2dag(low, hi, &stack, mfn, &loc);
        ⟨if dag is indirect CALL node, attempt to convert to direct call 155b⟩
    }
    mlcc_deallocate(&(mfunc(caller)->arena));
    mfunc(caller)->arena = 0;
    mfunc(caller)->cached = 0;
    tfree();
    return callee;
}
```

155b ⟨*if dag is indirect CALL node, attempt to convert to direct call* 155b⟩≡
```
if (oldn && generic(oldn->op) == ASGN) oldn = oldn->kids[1];
if (oldn && generic(oldn->op) == CALL) {
printdag(oldn,2);
    dagn = dag_simplify(oldn, (Type)0);
    assert(generic(dagn->op) == CALL);
    if (dagn->kids[0]->op == ADDRGP) {
        callee = dagn->kids[0]->syms[0];
        fprint(2,"*Convert call site %s %s \n", caller->name, callee->name);
        ⟨set symbol id of callee in the caller's symbol table 155c⟩
    }
}
```
This code is used in chunk 155a.

155c ⟨*set symbol id of callee in the caller's symbol table* 155c⟩≡
```
if (symid(msym(callee).id) >= tablesize(*msym(caller).symtab)
|| callee != idxtable(Symbol *, (*msym(caller).symtab), symid(msym(callee).id))) {
    cgn->fnid = GLOBSYM(tablesize(*msym(caller).symtab));
    addtotable(Symbol *, msym(caller).symtab, callee);
} else cgn->fnid = msym(callee).id;
```

This code is used in chunk 155b.

`constcondexpr` checks whether the given conditional expression is constant valued. If it is, the flow graph is modified appropriately and any flow-graph nodes found to be unreachable are eliminated.

156a ⟨*external declarations* 152a⟩+≡
```
dclproto(Node constcondexpr,(MFunction *mfn, FGNode, Node, Node));
```

156b ⟨*definitions* 152b⟩+≡
```
Node constcondexpr(mfn, fgnode, oldn, newn)
        MFunction *mfn; FGNode fgnode; Node oldn, newn; {
    int val;
    FGNode target, other, truesucc, falsesucc;
    short idx0, idx1;
    if (newn->op == CNSTI) {
        val = newn->syms[0]->u.c.v.i;
        idx0 = idxtable(short *, fgnode->succs, 0);
        idx1 = idxtable(short *, fgnode->succs, 1);
        truesucc = issucc(mfn, fgnode->id + idx0, oldn->syms[0]);
        if (truesucc)
            falsesucc = idxtable(FGNode *, mfn->flowgraph, fgnode->id + idx1);
        else {
            truesucc = issucc(mfn, fgnode->id + idx1, oldn->syms[0]);
            falsesucc = idxtable(FGNode *, mfn->flowgraph, fgnode->id + idx0);
        }
        assert(truesucc && falsesucc);
        if (val == 1) {
            ⟨ unconditional jump to TRUE branch 156c⟩
        } else if (val == 0) {
            ⟨ unconditional jump to FALSE branch 156d⟩
        }
        else assert(0);
        cleartable(fgnode->succs, fgnode->succs.arena);        /* remove other successor */
        addtotable(short *, &(fgnode->succs), srelIndex(fgnode->id, target->id));
        /* delete old predecessor from other's predecessor table */
        deletepred(mfn, other, fgnode);
        /* if successor is unreachable, clear its dagtable */
        if (emptytable(other->preds)) {
            fprint(2,"Flow Graph Node %d unreachable\n", other->id);
            ⟨eliminate all flowgraph nodes dominated by other 157a⟩
        }
    }
    return newn;
}
```

156c ⟨ *unconditional jump to TRUE branch* 156c⟩≡
```
newn = newnode(JUMPV, newnode(ADDRGP, 0, 0, oldn->syms[0]), 0, 0);
fgnode->type = JumpN;
target = truesucc;
other = falsesucc;
```
This code is used in chunk 156b.

156d ⟨ *unconditional jump to FALSE branch* 156d⟩≡
```
target = falsesucc;
other = truesucc;
```
This code is used in chunk 156b.

157a ⟨*eliminate all flowgraph nodes dominated by* `other` 157a⟩≡
```
{
    int fgidx;
    FGNode fgnode;
    gentable(mfn->flowgraph, fgidx, 1) {
        fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
        if (member(&fgnode->dominators, other->id)) {
            fprint(2,"eliminate %d\n", fgnode->id);
/*            eliminatenode(mfn, fgnode->id); */
            cleartable(fgnode->dagroots, fgnode->dagroots.arena);
        }
    }
}
```
This code is used in chunk 156b.

`issucc` returns the successor with given label if it is the node with id `idx`.

157b ⟨*local declarations* 153e⟩+≡
```
dclproto(static FGNode issucc,(MFunction *, int, Symbol));
```

157c ⟨*definitions* 152b⟩+≡
```
static FGNode issucc(mfn, idx, label) MFunction *mfn; Symbol label; {
    Symbol succlab;
    FGNode succ;
    succ = idxtable(FGNode *, mfn->flowgraph, idx);
    if (succ->labelsym) {
        succlab = idxtable(Symbol *, mfn->localsymtab, succ->labelsym);
        if (succlab == label)
            return succ;
    }
    return (FGNode)0;
}
```

`deletepred` deletes `neighbor` from `node`'s predecessor table; fail if `neighbor` is not in `node`'s predecessor table.

157d ⟨*local declarations* 153e⟩+≡
```
dclproto(static void deletepred,(MFunction *, FGNode, FGNode));
```

158a ⟨*definitions* 152b⟩+≡
```
static void deletepred(mfn, node, neighbor) MFunction *mfn; FGNode node, neighbor; {
    short idx = srelIndex(node->id, neighbor->id);
    int i, j, found = 0;
    FGNode pred;
    gentable(node->preds, i, 0) {
        pred = idxtable(FGNode *, mfn->flowgraph,
                        node->id + tableentry(short *, node->preds, i));
        if (pred == neighbor)  {
            gentable(node->preds, j, i) {
                if (j < node->preds.maxelem)
                    tableentry(short *, node->preds, j) =
                        tableentry(short *, node->preds, j+1);
            }
            node->preds.maxelem -= 1;
            found = 1;
            break;
        }
    }
    assert(found);
}
```

158b ⟨*unmill.h* 158b⟩≡
  ⟨*external declarations* 152a⟩

This code is written to file `unmill.h`.

158c ⟨*unmill.c* 158c⟩≡
```
#include <c.h>
#include <unmill.h>
#include <optmill.h>
#include <set.h>
#include <aemit.h>
```
  ⟨*local declarations* 153e⟩
  ⟨*definitions* 152b⟩

This code is written to file `unmill.c`.

6.13 mld's interface to code generator and emitter

`mld` provides all of the front-end services required by the target-dependent backends and appears to be like any other front end, even though it is generating code for the entire program instead of a single module. The following procedures implement the front-end services required by `mld`'s backends; its implementation parallels `mlcc`'s frontend. `emitcode()` and `gencode()` are called by the backends for each function `fn`. `fn` must be in an un-milled representation before calling `IR->function()` to generate and emit code.

159a ⟨*decl declarations* 159a⟩≡
```
#define NARGS 100

dclproto(extern void reset,(void));
extern int dataflag, mapflag, dbgflag;    /* mld flags */

Symbol  cfunc = 0;        /* current function */
char    *fname;           /* Dummy variables for c.h */
Symbol  retv;
```

This definition is continued in chunks 159b, 161a, 165c, and 167b.
This code is used in chunk 170.

`dobegin/doend` begin and end a scope; `doend` calls `function` to generate code for the given function.

159b ⟨*decl declarations* 159a⟩+≡
```
dclproto(static void dobegin,(void));
dclproto(static void doblockbeg,(FGNode));
dclproto(static void doblockend,(FGNode));
dclproto(static void doend,(Symbol));
```

160a ⟨*decl definitions* 160a⟩≡

```
void defglobal(p, seg) Symbol p; {
    swtoseg(p->u.seg = seg);
    if (p->sclass != STATIC)
        (*IR->export)(p);
    if (level == GLOBAL && glevel && IR->stabsym) {
        (*IR->stabsym)(p);
        swtoseg(p->u.seg);
    }
    (*IR->global)(p);
}

static void dobegin(void) {
    level = PARAM;
}

static void doblockbeg(fgnode) FGNode fgnode; {
    assert(level >= PARAM);
    ++level;
    blockbeg(&(fgnode->env));
}

static void doblockend(entryn) FGNode entryn; {
    assert(entryn);
    assert(level >= LOCAL);
    level--;
    blockend(&(entryn->env));
}
```

This definition is continued in chunks 160–62, 166, and 168a.
This code is used in chunk 170.

160b ⟨*decl definitions* 160a⟩+≡

```
static void doend(fn) Symbol fn; {
    int i = 0;
    Symbol caller[NARGS], callee[NARGS];

    gentable(mfn(fn).caller, i, 0) {
        assert(i < NARGS);
        caller[i] = idxtable(Symbol *, mfn(fn).localsymtab,
                             tableentry(unsigned short *, mfn(fn).caller, i));
        callee[i] = idxtable(Symbol *, mfn(fn).localsymtab,
                             tableentry(unsigned short *, mfn(fn).callee, i));
    }
    caller[i] = callee[i] = (Symbol)0;

    assert(level == PARAM);
    swtoseg(CODE);
    if (fn->sclass != STATIC) (*IR->export)(fn);
    cfunc = fn;
    (*IR->function)(fn, caller, callee,
                    mfunc(fn)->leaf ? 0 : tablesize(mfunc(fn)->callgraph));
    outflush();
}
```

dolocals announces local variables to the backend.

161a ⟨*decl declarations* 159a⟩+≡
```
dclproto(static void dolocals,(MTable, MTable));
```

161b ⟨*decl definitions* 160a⟩+≡
```
static void dolocals(localsymids, symtab) MTable localsymids, symtab; {
    Symbol s;
    int i;

    gentable(localsymids, i, 0) {
        s = idxtable(Symbol *, symtab,
                     ((int)tableentry(unsigned short *, localsymids, i)));
        if (!msym(s).announced) {
            if (s->generated)
                s->name = stringd(genlabel(1));
            if (s->computed)         /* COMPUTED LOCAL */
                (*IR->address)(s, idxtable(Symbol *, symtab, maddr(s).symid),
                               maddr(s).offset);
            else if (s->scope >= LOCAL)      /* LOCAL */
                (*IR->local)(s);
            else if (s->scope != PARAM)    /* LABEL or EXTERN */
                (*IR->defsymbol)(s);
            msym(s).announced = 1;
        }
    }
}
```

doprocedure calls `genprocedure], which converts a function in [[mill` code to its dynamic representation, and then frees the arena used by the function.

161c ⟨*external declarations* 161c⟩≡
```
dclproto(void doprocedure,(Symbol));
dclproto(void genprocedure,(Symbol, MTable, Arena *));
```
This code is used in chunk 169d.

161d ⟨*decl definitions* 160a⟩+≡
```
void doprocedure(fn) Symbol fn; {
    MTable context;

    cleartable(context, &transient);
    genprocedure(fn, context, &mfunc(fn)->arena);
    tfree();             /* deallocate any transient data */
    deallocate(FUNC);
    assert(arenasize(transient) == 0);
    mfunc(fn)->emitted = 1;
}
```

If any optimization flags are set, the procedure is analyzed and transformed before code is generated for the procedure. `dodagnodes` calls `dag_simplify` once, which will simplify constant expressions provided in a binding file. If `fn` is Modula–3 code, we do type analysis here, which will presumably expose opportunities for constant propagation.

`context` provides the calling context for `fn`. For a default call to `fn`, `context` is empty. If `fn` has been cloned and specialized or has been inlined by a caller, `context` contains the calling context.

162 ⟨*decl definitions* 160a⟩+≡

```
void genprocedure(fn, context, arena) Symbol fn; MTable context; Arena *arena; {
    unsigned pc, loc;
    int optimize_callee = 0, i;
    MFunction *mfn = &mfn(fn);

    if (mfunc(fn)->emitted) return;
    mill2dags(&mfn(fn), dataflag & !mfunc(fn)->noopt);

    ⟨prepend context to mfn 163⟩

    ⟨check if fn's callee selected for optimization 164a⟩

    if (m3Oflag && ⟨fn selected for optimization 164d⟩)
        m3type_analysis(&mfn(fn), arena);
    if (optflag == 1 || optstatflag || ⟨fn selected for optimization 164d⟩)
        constant_propagation(&mfn(fn), arena);
    if (optimize_callee) {
        ⟨report call site information 165a⟩
    }
    if (dbgflag) {
        fprint(2,"GEN ***%s in %s\n", fn->name, mfn(fn).bt->filename);
        emitProcedure(2, fn);
    }
    if (codegenflag) {
        dobegin();
        doend(fn);
    }
    if (mapflag && bflag) {
        loc = label_location(msym(fn).raddr->label);
        pc = b_pc();
        fprint(2, "%s: %x %x %x %d\n", fn->name, loc, symeaddr(fn),
            pc - symeaddr(fn), mfunc(fn)->dagsize);
    }
    mfunc(fn)->emitted = 1;
}
```

163 ⟨*prepend* context *to* mfn 163⟩≡

```
if (!emptytable(context)) {
    FGNode fgnode;
    int i, argloc;
    extern Node asgnnode(Symbol, Node);

    fgnode = idxtable(FGNode *, mfn->flowgraph, 1);
    if (emptytable(fgnode->dagroots))
        newtable(&fgnode->dagroots, tablesize(context), sizeof(int), arena);
    gentable(context, i, 0) {
        Node asgnn, argn, arg = tableentry(Node *, context, i);
        Symbol formal;
        if (arg) {
           if (optdbgflag) {
               fprint(2, "%d:\n", i);
               printdag(arg, 2);
           }
           argloc = tablesize(mfn->nodetable);
           argn = copynode(0, arg);
           formal = idxtable(Symbol *, mfn->localsymtab, (i+1)*2);
           if (formal->scope != CONSTANTS) {
               asgnn = asgnnode(formal, argn);
               if (formal->scope == PARAM)
                   asgnn->kids[0]->op = ADDRFP;
               else if (formal->scope >= LOCAL)
                   asgnn->kids[0]->op = ADDRLP;
               else assert(0);
               if (optdbgflag) printdag(asgnn, 2);
               addtotable(Node *, &(mfn->nodetable), asgnn);
               addtotable(int *, &fgnode->dagroots, argloc);
           }
        }
    }
    mfn->dagroots += tablesize(fgnode->dagroots);
}
```

This code is used in chunk 162.

164a ⟨*check if* `fn`*'s callee selected for optimization* 164a⟩≡
```
{
    CallGraphNode *cgn;
    Symbol callee;
    int j;

    if (optflag == 3) optimize_callee = 1;
    else if (!emptytable(optprocs)) {
        gentable(mfunc(fn)->callgraph, j, 0) {
            cgn = &(tableentry(CallGraphNode *, mfunc(fn)->callgraph, j));
            assert(callee = idxtable(Symbol *, (*msym(fn).symtab), symid(cgn->fnid)));
            ⟨check if callee is converted call site 164b⟩
            if (msym(callee).typeval != 0)
                fprint(2,"@@ %s has known type %d\n", callee->name, msym(callee).typeval);
            optimize_callee |= (insymtable(callee, optprocs) | msym(callee).typeval != 0);
        }
    }
}
```
This code is used in chunk 162.

164b ⟨*check if* `callee` *is converted call site* 164b⟩≡
```
if (callee->name == indirproc) {
    FGNode calln;
    Node callsite;
    calln = idxtable(FGNode *, mfn(fn).flowgraph, cgn->fgnid);
    assert(callsite = idxtable(Node *, mfn(fn).nodetable, callN(calln).dagn));
    if (generic(callsite->op) == ASGN) callsite = callsite->kids[1];
    assert (generic(callsite->op) == CALL);
    if (isaddrop(callsite->kids[0]->op)) callee = callsite->kids[0]->syms[0];
    if (callee->name != indirproc) fprint(2,"@@ Converted call site %s\n", callee->name);
}
```
This code is used in chunks 164a and 165a.

164c ⟨*new* `fn` *selected for optimization* 164c⟩≡
```
(!mfunc(fn)->noopt && (insymtable(fn, optprocs) || !emptytable(context)))
```
This code is written to file `new [[fn]] selected for optimization`.

164d ⟨`fn` *selected for optimization* 164d⟩≡
```
(!mfunc(fn)->noopt && (insymtable(fn, optprocs) ||
    (optimize_callee && optcallerflag) || !emptytable(context)))
```
This code is used in chunk 162.

165a ⟨*report call site information* 165a⟩≡
```
{
    CallGraphNode *cgn;
    Symbol callee;
    int j;

    gentable(mfunc(fn)->callgraph, j, 0) {
        cgn = &(tableentry(CallGraphNode *, mfunc(fn)->callgraph, j));
        assert(callee = idxtable(Symbol *, (*msym(fn).symtab), symid(cgn->fnid)));
        ⟨check if callee is converted call site 164b⟩
        if ((optflag == 3  && callee->sclass != EXTERN) || insymtable(callee, optprocs)) {
            fprint(2,"@@ %s calls %s knowntype %d\n",
                      fn->name, callee->name, msym(callee).typeval);
            specialize(fn, callee, cgn, arena);
        }
    }
}
```
This code is used in chunk 162.

The padflag specifies whether nops should be emitted to pad a procedure's text. We do this on the MIPS so that an optimized procedure's length is approximately the same as its unoptimized version. This is necessary to avoid poor VM performance.

165b ⟨*pad* 165b⟩≡
```
if (padflag && !bflag) {
    static int nops = 0;
    if (nops < convertible + symconvert + exprconvert)
        print(".set noreorder\n");
    for (; nops < convertible + symconvert + exprconvert; nops++)
        print("nop\nnop\n");
    print (".set reorder\n");
}
```
This code is written to file pad.

emitcode is a stub procedure that calls emitmillfn. emitmillfn emits code for the given function by walking the function's flow graph and calling emit for each instruction tree rooted in the flow-graph node. Instruction Nodes are linked in dodagnodes. emitmillfn also reconstructs switch statements encoded in Multitest nodes.

165c ⟨*decl declarations* 159a⟩+≡
```
dclproto(static void emitmillfn,(MFunction *mfn));
```

166a ⟨*decl definitions* 160a⟩+≡
```
void emitcode() {
    emitmillfn(&mfn(cfunc));
}
static void emitmillfn(mfn) MFunction *mfn; {
    FGNode fgnode;
    int i, j, fgidx;
    int  k;
    Symbol s;

    gentable(mfn->flowgraph, fgidx, 1) {
        fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
        if (fgnode->codelist)
            emit(fgnode->node);
        if (fgnode->type == MultitestN) {
            /* Table head is defined in dodagnodes() */
            k = idxtable(int *, multitestN(fgnode).values, 0);
            gentable(multitestN(fgnode).labels, i, 0) {
                while (k++ < (j = idxtable(int *, multitestN(fgnode).values, i)))
                    (*IR->defaddress)(multitestN(fgnode).deflabsym);
                (*IR->defaddress)(idxtable(Symbol *, multitestN(fgnode).labelsyms, i));
            }
        }
        else if (fgnode->type == CallN && callN(fgnode).mfn) {
            if (!bflag) print("/* >%s */\n", callN(fgnode).mfn->sym->name);
            emitmillfn(callN(fgnode).mfn);
            if (!bflag) print("/* <%s */\n", callN(fgnode).mfn->sym->name);
        }
    }
}
```

166b ⟨*decl definitions* 160a⟩+≡
```
void finalize() {
}
```

`gencode` generates code for the procedure prologue, which maps a caller's representation of parameters to the callee's representation. It then calls `genmillfn` to generate code for the body of the procedure.

166c ⟨*decl definitions* 160a⟩+≡
```
void gencode(lcaller, lcallee) Symbol lcaller[], lcallee[]; {
    FGNode  fgnode;
    int     i, argct;
    Node    head = 0, tail, p, tmp;
    Symbol  caller, callee;

    fgnode = idxtable(FGNode *, mfn(cfunc).flowgraph, 1);
    assert(emptytable(fgnode->dagnodes));    /* should be an empty table */
    ⟨generate dags for procedure prologue 167a⟩
    genmillfn(&mfn(cfunc));
}
```

The procedure prologue determines if actual arguments are stored in registers or on the stack. The appropriate code for accessing them is generated and placed in the first flow graph node of the procedure.

167a ⟨*generate dags for procedure prologue* 167a⟩≡
```
for (i = 0; (caller = lcallee[i]) && (callee = lcaller[i]); i++) {
    tmp = caller2callee(caller, callee);
    if (tmp) {
        p = tmp->link;
        tmp->link = 0;
        reset();
        ⟨add p to forest of nodes 169b⟩
    }
}
fgnode->node = head;
if (fgnode->node)
    fgnode->codelist = 1;
```

This code is used in chunk 166c.

`genmillfn` generates code for the given function by walking the function's flow graph; at each change in scope or flow control, the appropriate backend functions are called.

167b ⟨*decl declarations* 159a⟩+≡
```
dclproto(static void genmillfn,(MFunction *mfn));
```

168a ⟨*decl definitions* 160a⟩+≡
```
static void genmillfn(mfn) MFunction *mfn; {
    FGNode fgnode;
    int fgidx;
    Node head, tail;

    head = tail = (Node)0;
    gentable(mfn->flowgraph, fgidx, 1) {
        fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
        ⟨link fgnode's roots into a forest 169c⟩
        switch(fgnode->type) {
        case MultitestN: /* emitcode() emits multitest stmt addresses */
        case ArgN:
        case CallN:
        case EntryN:
        case ExitN:
        case JumpN:
        case RetN:
        case StmtsN:
        case TestN:
            if (fgnode->type == EntryN ||
               (fgnode->type == CallN && !emptytable(fgnode->localsymids))) {
                doblockbeg(fgnode);
                dolocals(fgnode->localsymids, mfn->localsymtab);
            }
            ⟨announce temporary symbols 168b⟩
            ⟨generate code for nodes rooted in fgnode 168c⟩
            ⟨generate code for inlined call site 169a⟩

            if (fgnode->type == ExitN)
                doblockend(idxtable(FGNode *, mfn->flowgraph, exitN(fgnode).entryNid));
            else if (fgnode->type == CallN && !emptytable(fgnode->localsymids))
                doblockend(fgnode);
            break;
        case UnknownN: default:
            fatal("mld:", "Unknown flow graph node type.\n", 0);
        }
    }
}
```

Local symbols must be announced before temporaries; they can overlap.

168b ⟨*announce temporary symbols* 168b⟩≡
```
dolocals(fgnode->tempsymids, mfn->localsymtab);
```

This code is used in chunk 168a.

168c ⟨*generate code for nodes rooted in* `fgnode` 168c⟩≡
```
if (fgnode->codelist && head) {
    fgnode->node = gen(head);
    head = tail = (Node) 0;
} else fgnode->node = 0;
```

This code is used in chunk 168a.

Embedded `mill` functions appear at call sites where inlining has been applied. The code for embedded `mill` functions is generated when the `Call` node of the call site is processed.

169a ⟨*generate code for inlined call site* 169a⟩≡
```
if (fgnode->type == CallN && callN(fgnode).mfn) {
     if (dbgflag) fprint(2,"**GENMILLFN %d\n", fgidx);
     genmillfn(callN(fgnode).mfn);
}
```
This code is used in chunk 168a.

169b ⟨*add* `p` *to forest of nodes* 169b⟩≡
```
if (!head) head = tail = p;
else {
     for (; tail && tail->link; tail = tail->link);
     assert (tail);
     tail->link = p;
}
```
This code is used in chunks 167a and 169c.

169c ⟨*link* `fgnode`*'s roots into a forest* 169c⟩≡
```
{
     int i, rootid;
     Node p;

     if (fgnode->node) {
          p = fgnode->node;
          ⟨add p to forest of nodes 169b⟩
     }
     gentable(fgnode->dagroots, i, 0) {
          rootid = tableentry(int *, fgnode->dagroots, i);
          if ((p = tableentry(Node *, mfn->nodetable, rootid)) != 0
                && rootop(generic(p->op)) && p->x.dagid == 0) {
               if (!fgnode->node) fgnode->node = p;
               ⟨add p to forest of nodes 169b⟩
          }
     }
}
```
This code is used in chunk 168a.

169d ⟨*decl.h* 169d⟩≡
```
⟨external declarations 161c⟩
```
This code is written to file `decl.h`.

170 ⟨*decl.c* 170⟩≡
```
#include <c.h>
#include <unmill.h>
#include <analysis.h>
#include <binary.h>
#include <m3.h>
#include <mld.h>
#include <simp.h>
#include <optmill.h>
#include <decl.h>
#include <aemit.h>

⟨decl declarations 159a⟩
⟨decl definitions 160a⟩
```
This code is written to file decl.c.

7 Link-time binding files

To guide link-time optimization, `mld` accepts a command line option specifying the name of a binding file that includes link-time information about variables and their values. For a Modula-3 program, the binding file is generated automatically by the link-time preprocessor, `mldpp`, and contains the constant-valued representation of the program's type hierarchy. It is not necessary for a binding file to be generated automatically – any variables with values known to be constant at run time and whose values are known at link time may be specified in the file.

The contents of a binding file resemble assignment statements in 'C'. The following EBNF grammar specifies the syntax of a binding file.[1] Strings and symbols in all capitals are terminals; all other symbols are non-terminals.

A binding file contains zero or more assignments of values to global variables. The value of a global variable may be an integer, floating-point number, string, the contents of a structure or union, or the value of another global variable. Explicit type information is not necessary. `mld` uses the type data provided in the `mill` object files to type check the assignments; the type of a global variable must match the type of the assigned value. Type conversion via casting is not permitted.

171a ⟨*binding grammar* 171a⟩≡
```
Binding-File => { Binding }
Binding      => [ Qualifiers ] Global-Variable "=" Value ";"
Qualifiers      => { "*" }
Value           => INTEGER
                |  FLOATING-POINT
                |  String
                |  Global-Variable
                |  Structure
                |  Union
                |  Memory-Image
```

This definition is continued in chunks 171c and 172a.
This code is written to file `binding grammar`.

The contents of a pointer-valued global variable may be initialized by qualifying the global variable with the indirection operator, "*". For example, the following statement assigns the value 100 to the contents of `global_count_ptr`. `global_count_ptr` must be of type `int *`.

171b ⟨*binding example* 171b⟩≡
```
*global_count_ptr = 100;
```

This definition is continued in chunk 172c.
This code is written to file `binding example`.

A global variable of a structure or union type is initialized by assigning values to the fields of the structure or union. Any number of a structure's fields may be initialized, but only one field of a union may be initialized.

171c ⟨*binding grammar* 171a⟩+≡
```
Structure       => "{" { Field-Assignment } "}"
Union           => "{" Field-Assignment "}"
Field-Assignment => Type-Field "=" Value ";"

Global-Variable => IDENTIFIER
String          => "\"" IDENTIFIER "\""
Type-Field      => IDENTIFIER
```

[1] In EBNF, | denotes alternation, `[...]` optional fields, and `{...}` Kleene closure.

A `Memory-Image` initializes a block of bytes. No type checking is performed when assigning a memory block to a global variable or field of a stucture or union, so beware! Any errors due to overlapping values, incorrectly sized blocks, or invalid alignment will not be flagged. The block is specified by a sequence of `byte offset`, `value` pairs. The `offset` specifies the byte offset within the block at which `value` is written.

172a ⟨*binding grammar* 171a⟩+≡

```
Memory-Image     => "[" { Offset-Assignment } "]"
Offset-Assignment => INTEGER ":" Value ";"
```

In the following example, `_null_TC`, `_root_TC` and `_RTHeap_TC` are all global variables of type `_TYPE *`. `_TYPE` is the run-time type representation of a Modula-3 type. For the purposes of this example, its definition is:

172b ⟨*TYPE definition* 172b⟩≡

```
typedef struct _tc {
    int typecode;
    int lastSubTypeTC;
    int dataOffset;
    int dataSize;
    int dataAlignment;
    int methodOffset;
    int methodSize;
  } _TYPE;
```

This code is written to file `TYPE definition`.

172c ⟨*binding example* 171b⟩+≡

```
*null_TC = {
    name = "$null$";
    typecode = 0;
    dataSize = 0;
    dataAlignment = 0;
    dataOffset = 0;
}

*root_TC = {
    typecode = 2;
    lastSubTypeTC = 29;
    dataSize = 4;
    dataAlignment = 4;
    dataOffset = 0;
    methodSize = 4;
    methodOffset = 0;
}

*RTHeap_TC = {
    typecode= 24;
    dataSize = 8;
    dataAlignment = 4;
    dataOffset = 4;
    methodSize = 8;
    methodSize = 8;
}
```

7.1 Parsing a binding file

The binding-file syntax is recognized by a recursive-descent parser. The initialized value of a global variable is represented by the `DataValue` structure.

173a ⟨*exported data declarations* 173a⟩≡

```
typedef struct datavalue {
    int offset;
    unsigned short nquals;
    char datatype;
    union {
        int i;
        float f;
        char *s;
        Symbol sym;
        struct datavalue *fields;    /* struct or union fields */
    } u;
    struct datavalue *next;
}*DataValue;
```

This code is used in chunk 182b.

173b ⟨*exported macro definitions* 173b⟩≡

```
#define datasymbol(d) ((d && d->datatype == ID) ? d->u.sym : (Symbol)0)
```

This code is used in chunk 182b.

173c ⟨*exported procedure declarations* 173c⟩≡

```
dclproto(extern int parse_dataimage,(char *));
```

This definition is continued in chunk 180a.
This code is used in chunk 182b.

173d ⟨*local declarations* 173d⟩≡

```
static char *datafile;
static Type stringtype;
```

This definition is continued in chunks 174b, 175c, 177–79, and 181b.
This code is used in chunk 182c.

174a ⟨*definitions* 174a⟩≡

```
int parse_dataimage(filename) char *filename;{
    extern int infd;
    int tok;

    fileClose(0);
    if ((infd = open(filename, O_RDONLY)) == -1) {
        error("Cannot open data image file %s\n", filename);
        return 0;
    }
    stringtype = ptr(chartype);
    datafile = filename;
    inputInit();
    dataimage();
    expect(EOI);
    close(infd);
    return 1;
}
```

This definition is continued in chunks 175–82.
This code is used in chunk 182c.

174b ⟨*local declarations* 173d⟩+≡

```
dclproto(static void dataimage,(void));
```

175a ⟨*definitions* 174a⟩+≡
```
static void dataimage() {
    int done = 0, nquals, i;
    Type type;
    Symbol sym;
    BaseTable *bt;
    DataValue val;

    t = gettok();
    while(!done) {
        nquals = 0;
        type = 0;
        switch (t) {
        case '*':
            nquals = indirection();
            /* fall thru */
        case ID:
            expect(ID);
            sym = install_symbol(token, &bt);
            if (!sym) error("Undefined global symbol %s\n", token);
            else type = dereference(sym->type, nquals);
            expect('=');
            val = datatypevalue(type);
            expect(';');
            if (sym && val) {
                val->nquals = nquals;
                mglob(sym).datavalue = val;
            }
            ⟨dump dataimage 175b⟩
            break;
        default: done = 1;
            break;
        }
    }
}
```

175b ⟨*dump dataimage* 175b⟩≡
```
if (dbgflag && sym) {
    for (i = 0; i < nquals; i++) fprint(2,"*");
    fprint(2,"%s = ", sym->name);
}
if (dbgflag && val) dumpdatavalue(val, 0);
```
This code is used in chunk 175a.

`datatypevalue` attempts to consume a value of type `type`.

175c ⟨*local declarations* 173d⟩+≡
```
dclproto(static DataValue datatypevalue,(Type));
```

176a ⟨*definitions* 174a⟩+≡
```
static DataValue datatypevalue(type) Type type; {
    int gett = 1, nquals = 0;
    char *s, str[256];
    DataValue val = 0, flds;
    Symbol sym;
    BaseTable *bt;
    Type symtype;

    bzero(str,256);
    switch (t) {
    case ICON:
        ⟨ expect an integer 176b⟩
        break;
    case FCON:
        ⟨ expect a floating point value 176c⟩
        break;
    case SCON:
        ⟨ expect a string 177a⟩
        break;
    case '*':    /* dereferenced symbol, "*p", "**p" */
        nquals = indirection();
        /* fall thru */
    case ID:
        ⟨ expect a symbol 177b⟩
        break;
    case '{' :
        ⟨ expect a structure or union 177c⟩
        break;
    case '[' :
        ⟨ expect a memory image 177d⟩
        break;
    default:
        expect(ID);
        break;
    }
    if (gett) t = gettok();
    return val;
}
```

176b ⟨ *expect an integer* 176b⟩≡
```
if (!type || isint(type)) {
    val = newvalue(ICON);
    val->u.i = tsym->u.c.v.i;
} else typeerr("integer", type);
```
This code is used in chunk 176a.

176c ⟨ *expect a floating point value* 176c⟩≡
```
if (!type || isfloat(type)) {
    val = newvalue(FCON);
    val->u.f = tsym->u.c.v.f;
} else typeerr("float", type);
```
This code is used in chunk 176a.

177a ⟨ *expect a string* 177a⟩≡
```
if (!type || eqtype(type, stringtype, 1)) {
    val = newvalue(SCON);
    val->u.s = string(tsym->u.c.v.p);
} else typeerr("char *", type);
```
This code is used in chunk 176a.

177b ⟨ *expect a symbol* 177b⟩≡
```
expect(ID);
gett = 0;
sym = install_symbol(token, &bt);
if (!sym) error("Undefined global symbol %s\n", token);
else  {
    symtype = dereference(sym->type, nquals);
    if (!type || (symtype &&
       (isfunc(symtype) || isfunc(type) || eqtype(symtype, type, 1)))) {
        if (nquals == 0) {
            val = newvalue(ID);
            val->u.sym = sym;
        } else val = constaddr(nquals, mglob(sym).datavalue);
    }
    else typeerr(sym->type ? typestring(sym->type, str) : "NIL", type);
}
```
This code is used in chunk 176a.

177c ⟨ *expect a structure or union* 177c⟩≡
```
if (!type || isstruct(type))
    val = newvalue(isunion(type) ? UNION : STRUCT);
else typeerr("struct or union", type);
t = gettok();
flds = fields(type);
expect('}');
gett = 0;
if (val) val->u.fields = flds;
```
This code is used in chunk 176a.

177d ⟨ *expect a memory image* 177d⟩≡
```
val = newvalue(STRUCT);
t = gettok();
val->u.fields = offsets();
expect(']');
gett = 0;
```
This code is used in chunk 176a.

`indirection` consumes a string of `*`'s.

177e ⟨*local declarations* 173d⟩+≡
```
dclproto(static int indirection,(void));
```

177f ⟨*definitions* 174a⟩+≡
```
static int indirection(void) {
    int nquals = 1;
    while ((t = gettok()) == '*')
        nquals++;
    return nquals;
}
```

`fields` consumes the field(s) of a structure or union. `offsets` consumes the initialized bytes of a memory image.

178a ⟨*local declarations* 173d⟩+≡
```
dclproto(static DataValue fields,(Type));
dclproto(static DataValue offsets,(void));
```

178b ⟨*definitions* 174a⟩+≡
```
static DataValue fields(type) Type type; {
    DataValue firstval = 0, val;
    Field fld;
    Type ty;
    int nquals, i;

    while(t == ID || t == '*') {
        ty = 0;
        val = 0;
        nquals = 0;
        if (t == '*') nquals = indirection();
        expect(ID);
        if (type) {
            fld = fieldref(string(token), type);     /* check field reference */
            if (!fld) error("Unknown field: %s of type %t\n", token, type);
            else ty = dereference(fld->type, nquals);
        }
        expect('=');
        val = datatypevalue(ty);
        expect(';');
        if (ty && val) {
            val->nquals = nquals;
            val->offset = fld->offset;
            val->next = firstval;
            firstval = val;
        }
    }
    return firstval;
}
```

179a ⟨*definitions* 174a⟩+≡
```
static DataValue offsets() {
    DataValue firstval = 0, val;
    int offset;

    while(t == ICON) {
        val = 0;
        expect(ICON);
        offset = tsym->u.c.v.i;
        expect(':');
        val = datatypevalue(0);
        expect(';');
        if (val) {
            val->nquals = 0;
            val->offset = offset;
            val->next = firstval;
            firstval = val;
        }
    }
    return firstval;
}
```

179b ⟨*local declarations* 173d⟩+≡
```
dclproto(static Type dereference,(Type, int));
dclproto(static DataValue newvalue,(int));
dclproto(static void typeerr,(char *, Type));
```

`newvalue` allocates and returns a new `DataValue`. `dereference` returns the base type of `*^nquals(type)`. `typerr` reports a type mismatch error.

179c ⟨*definitions* 174a⟩+≡
```
static DataValue newvalue(datatype) {
    DataValue val = (DataValue)alloc(sizeof(*val));
    BZERO(val, *val);
    val->datatype = datatype;
    return val;
}
```

179d ⟨*definitions* 174a⟩+≡
```
static Type dereference(type, nquals) Type type; {
    int i;
    for (i = 0; i < nquals; i++, type = type->type)
        if (type->op != POINTER) {
            typeerr("pointer to ?", type);
            type = 0;
            break;
        }
    return type;
}
```

179e ⟨*definitions* 174a⟩+≡
```
static void typeerr(s, t) char *s; Type t; {
    warning("Type error: expecting %s found ", s);
    if (t) fprint(2, "%t\n", t);
    else fprint(2, "NIL\n");
}
```

7.2 Using binding values

180a ⟨*exported procedure declarations* 173c⟩+≡

```
dclproto(extern DataValue constaddr,(int, DataValue));
dclproto(extern DataValue constoffset,(DataValue, int));
dclproto(extern Value datavalue,(DataValue, int));
```

`constoffset` returns the `DataValue` bound to the address `val+offset`, if it exists.

180b ⟨*definitions* 174a⟩+≡

```
DataValue constoffset(val, offset) DataValue val; {
    DataValue field;
    if (val == (DataValue)0)
        return val;
    if (offset >= 0 && (val->datatype == STRUCT || val->datatype == UNION)) {
        for (field = val->u.fields; field; field = field->next)
            if (offset == field->offset)
                return field;
    } else if (offset == 0)
        return val;
    return (DataValue)0;
}
```

`constaddr` returns the `DataValue` bound to the address `*^nindir(val)`, where `*^nindir` represents `nindir` indirections of the address `val`. if it exists.

180c ⟨*definitions* 174a⟩+≡

```
DataValue constaddr(nindir, val) DataValue val; {
    if (val == (DataValue)0)
        return val;
    if (nindir == val->nquals)
        return val;
    else if (nindir > val->nquals) {
        if (val->nquals == 0) {
            if (nindir == 1) return val;
        } else {
            nindir -= val->nquals;
            if (val->datatype == ID) /* refers to symbol */
                return constaddr(nindir,
                       (val->u.sym->scope == GLOBAL ?
                       mglob(val->u.sym).datavalue : mtemp(val->u.sym).datavalue));
        }
    }
    return (DataValue)0;
}
```

`datavalue` returns the `Value` of type `ty` represented by the `DataValue` `dataval`.

181a ⟨*definitions* 174a⟩+≡

```
Value datavalue(dataval, ty) DataValue dataval; {
    Value val;
    dataval = constoffset(dataval, 0);
    assert(dataval);
    switch(ty) {
    case C:
        assert(dataval->datatype == ICON);
        val.uc = (unsigned char)dataval->u.i;
        break;
    case I:
        assert(dataval->datatype == ICON);
        val.i = dataval->u.i;
        break;
    case F:
        if (dataval->datatype == ICON)
            val.f = (float)dataval->u.i;
        else if (dataval->datatype == FCON)
            val.f = dataval->u.f;
        else assert(0);
        break;
    case D:
        if (dataval->datatype == ICON)
            val.f = (double)dataval->u.i;
        else if (dataval->datatype == FCON)
            val.d = (double)dataval->u.f;
        else assert(0);
        break;
    case P:
        assert(dataval->datatype == ID);
        val.p = (char *)dataval->u.sym;
        break;
    }
    return val;
}
```

181b ⟨*local declarations* 173d⟩+≡

```
dclproto(void   dumpdatavalue,(DataValue, int));
```

182a ⟨*definitions* 174a⟩+≡

```
static char *space[6] = { "", "  ", "    ", "      ", "        ", "          " };
void dumpdatavalue(val, s) DataValue val; int s; {
    int i;
    DataValue fld;

    s = (s < 6) ? s : 5;
    if (!val) {
        fprint(2,"%s**no image**\n", space[s]);
        return;
    }
    switch(val->datatype) {
    case ICON: fprint(2, "%s%d;\n", space[s], val->u.i); break;
    case FCON: fprint(2, "%s%x;\n", space[s], val->u.f); break;
    case SCON: fprint(2, "%s%s;\n", space[s], val->u.s); break;
    case ID:   fprint(2, "%s%s;\n", space[s], val->u.sym->name); break;
    case STRUCT:
        fprint(2, "%s{\n", space[s]);
        for (fld = val->u.fields; fld; fld = fld->next) {
            fprint(2, space[s]);
            for (i = 0; i < fld->nquals; i++) fprint(2,"*");
            fprint(2, "%d: ", fld->offset);
            dumpdatavalue(fld, s + 1);
        }
        fprint(2, "%s}\n", space[s]);
        break;
    default: assert(0);
    }
}
```

182b ⟨*dataimage.h* 182b⟩≡

```
#ifndef INCLUDE_DATAIMAGEH
#define INCLUDE_DATAIMAGEH

⟨exported data declarations 173a⟩
⟨exported macro definitions 173b⟩
⟨exported procedure declarations 173c⟩
#endif
```

This code is written to file `dataimage.h`.

182c ⟨*dataimage.c* 182c⟩≡

```
#include <c.h>
#include <mld.h>
/* #include <sys/file.h> */
#include <resextsym.h>
#define O_RDONLY 0

⟨local declarations 173d⟩
⟨definitions 174a⟩
```

This code is written to file `dataimage.c`.

8 Inlining

This module implements the inlining code transformation and walks the program's call graph to find all reachable procedures.

8.1 Identifying reachable procedures

`walkcallgraph` walks the call graph in depth-first order, tagging as reachable those procedures called by the function `fn`, and inlines procedures at call sites marked as inlineable. Inlining is performed on the `mill]ed (on-disk) representation of a program.`
`This simplifies the implementation because copying a [[mill` function in its on-disk representation is trivial; copying an inlined procedure requires a few memory block-copy operations. This choice increases the size of `mld`, however, because it requires that `mill` instructions be transformed in both their on-disk and in-memory representations.

There are three ways to select inlineable procedures. If the `-Ih` option is chosen, inlineable procedures are chosen based on the heuristics in `inlineheuristic`. A procedure is not inlineable if it is self-recursive, takes variadic arguments, or if it can potentially create an inlining cycle in the call graph. Individual call sites, i.e. a caller, callee, call site triple, may be chosen for inlining using the `-If`*F* option, where *F* is a file containing a list of call sites. This option is useful for selecting frequently executed call sites identified in profiles. The `-Ip`*P* chooses an individual procedure *P* for inlining and attempts to inline it at all of its call sites. `inlineproc` determines if an individual procedure is inlineable. `walkcallgraph` determines which call sites for a specific procedure are inlineable.

183a ⟨*external declarations* 183a⟩≡

```
extern MTable Inlinetable;
dclproto(int walkcallgraph,(Symbol, int, int, int));
```

This definition is continued in chunks 185–87, 192c, 205b, and 207a.
This code is used in chunk 210b.

183b ⟨*local declarations* 183b⟩≡

```
static int totalsites = 0;
```

This definition is continued in chunks 187a, 188b, 190–93, 195b, 202c, 203b, 206b, 207c, and 209e.
This code is used in chunk 210c.

184a ⟨*definitions* 184a⟩≡

```
int walkcallgraph(caller, cgnid, rootid, inlineflag) Symbol caller; int cgnid; {
    int i, j, ct;
    CallGraphNode *cgn;
    CallSiteNode *site;
    Symbol callee;

    assert(caller);
    if (mfunc(caller)->cgnid > 0) return cgnid;
    if (dbgflag) fprint(2, "Visit %s %d\n", caller->name, cgnid);
    mfunc(caller)->cgnid = cgnid;
    cgnid += 1;
    mfunc(caller)->onstack = 1;
    if (inlineflag == 1) {
        ⟨evaluate individual sites 185a⟩
    }
    gentable(mfunc(caller)->callgraph, i, 0) {
        cgn = &(tableentry(CallGraphNode *, mfunc(caller)->callgraph, i));
        ⟨visit children 184b⟩
        evalcallsite(caller, callee, cgn, rootid, inlineflag == 1 ? 0 : inlineflag, 1);
    }
    mfunc(caller)->onstack = 0;
    totalsites += i;
/*    emitDotCallgraph(fn); */

    return cgnid;
}
```

This definition is continued in chunks 186–88, 190c, 193a, 194, 197a, 199a, 200, 203–7, and 210a.
This code is used in chunk 210c.

184b ⟨*visit children* 184b⟩≡

```
callee = idxtable(Symbol *, (*msym(caller).symtab), symid(cgn->fnid));
if (!mfunc(callee)->cgnid)
    cgnid = walkcallgraph(callee, cgnid, rootid, inlineflag);
```

This code is used in chunks 184a and 185a.

185a ⟨*evaluate individual sites* 185a⟩≡

```
gentable(mfunc(caller)->inlinesites, i, 0) {
    site = tableentry(CallSiteNode **, mfunc(caller)->inlinesites, i);
    if (dbgflag)
        fprint(2,"Eval call site %s %s %d %d\n",
               caller->name, site->callee->name, site->id, site->count);
    ct = site->id;
    gentable(mfunc(caller)->callgraph, j, 0) {
        cgn = &(tableentry(CallGraphNode *, mfunc(caller)->callgraph, j));
        assert(callee = idxtable(Symbol *, (*msym(caller).symtab), symid(cgn->fnid)));
        if (callee == site->callee
        || (callee->name == indirproc && site->callee == convert_site(caller, cgn))) ct--;
        if (ct == 0) break;
        cgn = 0;
    }
    if (cgn) {
        ⟨visit children 184b⟩
        evalcallsite(caller, callee, cgn, rootid, inlineflag, 1);
    } else warning("No callsite %s %s %d.\n", caller->name, site->callee->name, site->id);
}
```

This code is used in chunk 184a.

185b ⟨*external declarations* 183a⟩+≡

```
dclproto(int evalcallsite,(Symbol, Symbol, CallGraphNode *, int, int, int));
```

186a ⟨*definitions* 184a⟩+≡
```
int evalcallsite(caller, callee, cgn, rootid, inlineflag, cycle)
    Symbol caller, callee; CallGraphNode *cgn;{
    int inline;

    if (cgn->ref == -1) return 0;
    inline = 0;
    if (callee->name == indirproc) indircalls++;
    if (callee->name != indirproc
    && caller != callee && callee->type && !variadic(callee->type)
    && !mfunc(callee)->asmcode && callee->sclass != EXTERN) {
        if (inlineflag == 2) inline = inlineheuristic(callee, cgn);
        if (inlineproc(caller, callee, inlineflag)) inline = 4;
    }
    ⟨check that inlining doesn't cause cycle 186b⟩
    cgn->inline = (unsigned char) inline;
    if (inline && dbgflag)
        fprint(2,"\\Inline call site %s %s %d %d\n", caller->name,
               callee->name, mfunc(caller)->dagsize, mfunc(callee)->dagsize);
    if (inline > 0) {
        heuristics[inline]++;
        mfunc(caller)->dagsize += mfunc(callee)->dagsize;
        mfunc(callee)->ninlines += 1;
        mfunc(callee)->ncalls -= 1;
    }
    if (callee->name != indirproc)
        cgn->ref = -1; /* tag direct call sites as visited */
    return cgn->inline;
}
```

186b ⟨*check that inlining doesn't cause cycle* 186b⟩≡
```
if (cycle && inline && mfunc(caller)->cgnid > mfunc(callee)->cgnid
&& mfunc(callee)->cgnid >= rootid && mfunc(callee)->onstack) {
    warning("mld: Up arc from %s to %s at site %d,%d\n", caller->name, callee->name,
        cgn->fgnid, cgn->ref);
    inline = 0;
}
```
This code is used in chunk 186a.

8.2 Inlining heuristics

`inlineheuristic` applies a set of heuristics to the given procedure and returns a flag indicating whether the procedure is inlineable at the given call site. A callee is inlineable if it is a leaf procedure; has one call site; has a static reference count greater than 20000 and a text size less than 400 intermediate instructions; or is called once and has a text size less than 700 intermediate instructions. These heuristics are based on those described in Holler's thesis (Holler 1991).

186c ⟨*external declarations* 183a⟩+≡
```
extern char *indirproc;
extern int Inlinelimit;
```

187a ⟨*local declarations* 183b⟩+≡
```
dclproto(static int inlineheuristic,(Symbol, CallGraphNode *));
dclproto(static int inlineproc,(Symbol, Symbol, int));
static int indircalls = 0, asgnargs = 0, promotetmps = 0;
static int heuristics[] = { 0, 0, 0, 0, 0 };
int Inlinelimit = 1000;
```

`inlineheuristic` applies heuristics to `callee` and returns a flag indicating whether it is inlineable.

187b ⟨*definitions* 184a⟩+≡
```
static int inlineheuristic(callee, cgn) Symbol callee; CallGraphNode *cgn; {
    int heuristic;

    if (mfunc(callee)->leaf) heuristic = 1;
    else if (cgn->ref >= 20000 && mfunc(callee)->dagsize <= 400) heuristic = 2;
    else if (mfunc(callee)->ncalls <= 1 && mfunc(callee)->dagsize <= 700) heuristic = 3;
    else heuristic = 0;
    return heuristic;
}

static int inlineproc(caller, callee, inlineflag) Symbol caller, callee; {
    if (mfunc(callee)->noopt) return 0;
    if (inlineflag == 1 || (inlineflag == 3 && instringtable(Inlinetable, callee->name)))
        if (mfunc(caller)->dagsize + mfunc(callee)->dagsize < Inlinelimit)
            return 1;
        else warning("Call site %s %s rejected -- exceeds source limit (%d)\n",
                caller->name, callee->name, mfunc(caller)->dagsize + mfunc(callee)->dagsize);
    return 0;
}
```

`inlinestats` prints the program's inlining statistics to standard error.

187c ⟨*external declarations* 183a⟩+≡
```
dclproto(void inlinestats,(void));
```

187d ⟨*definitions* 184a⟩+≡
```
void inlinestats(void) {
    int tot;
    tot = heuristics[1] + heuristics[2] + heuristics[3];
    fprint(2,"total call sites %d; inlineable %d indirect calls %d\n",
        totalsites, tot, indircalls);
    fprint(2,"leaf %d refct %d called once %d individual\n", heuristics[1], heuristics[2],
        heuristics[3], heuristics[4]);
    fprint(2,"total procs %d generated procs %d\n", totprocs, genprocs);
    fprint(2,"total args %d actual %d temp %d asgn %d promote %d\n",
        totargs, actargs, tempargs, asgnargs, promotetmps);
    fprint(2,"clones %d reused clones %d\n", clonect, Clonedups);
}
```

8.3 Inlining

`inlinecallgraph` calls `auxinlinecallgraph` which recursively descends the call graph beginning at the function `fn`, generating code for descendents of `fn` and applying inlining where possible.

187e ⟨*external declarations* 183a⟩+≡
```
dclproto(void inlinecallgraph,(int, Symbol, int));
```

188a ⟨*definitions* 184a⟩+≡
```
void inlinecallgraph(flag, fn, order) Symbol fn; {
    MFunction *mfn = (MFunction *)0;
    Symbol s = fn;
    if (flag == 1)
        auxinlinecallgraph1(fn, 0, order);
    else {
        mfn = auxinlinecallgraph(fn, 0, &mfunc(fn)->arena, 0);
        ⟨deallocate s if not inlined 189c⟩
    }
}
```

auxinlinecallgraph recursively descends the call graph from function `fn`, inlining functions in bottom-up, post order. It is called when aggressive, heuristic inlining is applied to nested descendents in the call graph. For example, if `A` inlines `B`, and `B` inlines `C`, then `A` inlines both `B` and `C`. `auxinlinecallgraph1` can traverse the call graph either bottom-up or top-down, and applies inlining only at individual sites specified in an inlining file.

Both procedures generates and emits code for `fn` if `fn` is ever called directly or can be called indirectly. If `fn` will be inlined elsewhere in callgraph, a copy is made and this copy is returned. After the last copy of `fn` has been inlined, its arena is deallocated.

188b ⟨*local declarations* 183b⟩+≡
```
dclproto(static MFunction *auxinlinecallgraph,(Symbol, int, struct arena **, int));
static int totprocs = 0;
```

188c ⟨*definitions* 184a⟩+≡
```
static MFunction *auxinlinecallgraph(fn, inlining, arena, tab)
    Symbol fn; struct arena **arena; {

    MFunction *mfn = (MFunction *)0;
    MTable mfntable;

    assert(fn);
    mfunc(fn)->onstack = 1;
    ⟨allocate a new arena for fn 188d⟩
    newtable(&mfntable, tablesize(mfunc(fn)->callgraph), sizeof(mfn), &mfunc(fn)->arena);
    if (!mfunc(fn)->emitted && !mfunc(fn)->cached) {
        totprocs++;
        ⟨inline nested descendents first 189a⟩
        breadProcedure(fn);
        ⟨inline inlineable call sites in this function 189b⟩
    }
    if (inlining)         /* copy procedure for inlining */
        mfn = bcopyMFunction(&mfn(fn), arena);
    ⟨generate and emit code if called elsewhere 192a⟩
    mfunc(fn)->onstack = 0;
    return mfn;
}
```

188d ⟨*allocate a new arena for* `fn` 188d⟩≡
```
if (!mfunc(fn)->arena) {
    mfunc(fn)->arena = newarena();
}
```
This code is used in chunks 188c and 190c.

189a ⟨*inline nested descendents first* 189a⟩≡
```
{
    int i,j;
    Symbol s;
    CallGraphNode cgn;

    if (dbgflag) {
        for (j = 0; j < tab; j++) fprint(2,"  ");
        fprint(2,"*** %s\n", fn->name);
    }
    gentable(mfunc(fn)->callgraph, i, 0) {
        cgn = tableentry(CallGraphNode *, mfunc(fn)->callgraph, i);
        s = idxtable(Symbol *, (*msym(fn).symtab), symid(cgn.fnid));
        if (s != fn && s->sclass != EXTERN && !mfunc(s)->onstack)
            tableentry(MFunction **, mfntable, i) =
                auxinlinecallgraph(s, cgn.inline, &mfunc(fn)->arena, tab+1);
    }
}
```
This code is used in chunk 188c.

189b ⟨*inline inlineable call sites in this function* 189b⟩≡
```
{
    int i;
    Symbol s;
    CallGraphNode cgn;

    gentable(mfunc(fn)->callgraph, i, 0) {
        cgn = tableentry(CallGraphNode *, mfunc(fn)->callgraph, i);
        s = idxtable(Symbol *, (*msym(fn).symtab), symid(cgn.fnid));
        if (cgn.inline) {
            assert(tableentry(MFunction **, mfntable, i));
            inline(fn, s, tableentry(MFunction **, mfntable, i), cgn.fgnid);
            ⟨deallocate s if not inlined 189c⟩
        }
    }
}
```
This code is used in chunk 188c.

Arena allocation is a good idea for `mld`, because once a procedure has been used in all possible contexts (called directly, indirectly, inlined, etc), its memory can be freed. Deallocation occurs, however, in several places in the source code and premature deallocation has been the source of many bugs.

189c ⟨*deallocate* `s` *if not inlined* 189c⟩≡
```
if(mfunc(s)->ninlines == 0) {
    ⟨deallocate s 190a⟩
}
```
This code is used in chunks 188a and 189b.

190a ⟨*deallocate* `s` 190a⟩≡
```
if (mfunc(s)->arena != 0) {
    mlcc_deallocate(&(mfunc(s)->arena));
    mfunc(s)->cached = 0;
    assert(arenasize(mfunc(s)->arena) == 0);
    mfunc(s)->arena = 0;
    assert(mfunc(s)->arena == 0 && mfunc(s)->cached == 0);
}
```

This code is used in chunks 189–91.

190b ⟨*local declarations* 183b⟩+≡
```
dclproto(static MFunction *auxinlinecallgraph1,(Symbol, int, int));
```

190c ⟨*definitions* 184a⟩+≡
```
static MFunction *auxinlinecallgraph1(fn, tab, order)
    Symbol fn; {

    MFunction *mfn = (MFunction *)0;
    Symbol s;

    assert(fn);
    assert(order == 1 || order == 2);
    mfunc(fn)->onstack = 1;
    ⟨allocate a new arena for fn 188d⟩
    if (!mfunc(fn)->emitted && !mfunc(fn)->cached) {
        totprocs++;
        if (order == 1)
            ⟨generate nested descendents 191a⟩
        breadProcedure(fn);
        ⟨read & inline inlineable call sites in this function 191b⟩
        if (order == 2)
            ⟨generate nested descendents 191a⟩
    }
    ⟨generate and emit code if called elsewhere 192a⟩
    s = fn;
    ⟨deallocate s 190a⟩
    mfunc(fn)->onstack = 0;
    return mfn;
}
```

191a ⟨*generate nested descendents* 191a⟩≡
```
{
    CallGraphNode cgn;
    Symbol s;
    int i;

    gentable(mfunc(fn)->callgraph, i, 0) {
        cgn = tableentry(CallGraphNode *, mfunc(fn)->callgraph, i);
        s = idxtable(Symbol *, (*msym(fn).symtab), symid(cgn.fnid));
        if (s != fn && s->sclass != EXTERN && !mfunc(s)->onstack)
            auxinlinecallgraph1(s, tab+1, order);
    }
}
```
This code is used in chunks 190c and 191b.

When inlining is not applied aggressively, i.e., recursively through nested descendents, we must be sure that the callees of an inlined procedure are generated. After inlining the procedure, we make sure all its callees are reached.

191b ⟨*read & inline inlineable call sites in this function* 191b⟩≡
```
{
    CallGraphNode cgn;
    Symbol s;
    int i;

    gentable(mfunc(fn)->callgraph, i, 0) {
        cgn = tableentry(CallGraphNode *, mfunc(fn)->callgraph, i);
        s = idxtable(Symbol *, (*msym(fn).symtab), symid(cgn.fnid));
        if (cgn.inline) {
            if (!mfunc(s)->cached) {
                Symbol fn;
                breadProcedure(s);
                fn = s;
                ⟨generate nested descendents 191a⟩
            }
            inline(fn, s, bcopyMFunction(&mfn(s), &mfunc(fn)->arena), cgn.fgnid);
            ⟨deallocate s 190a⟩
        }
    }
}
```
This code is used in chunk 190c.

191c ⟨*local declarations* 183b⟩+≡
```
static int genprocs;
```

192a ⟨*generate and emit code if called elsewhere* 192a⟩≡
```
if (!mfunc(fn)->emitted && (mfunc(fn)->ncalls > 0 || indircalledproc(fn))) {
    MFunction savmfn, *newmfn;

    /* if procedure is inlined elsewhere, make a clean copy */
    savmfn = mfn(fn);
    if (mfunc(fn)->ninlines >= 1) {
        newmfn = bcopyMFunction(&mfn(fn), &transient);
        mfn(fn) = *newmfn;
    }
    genprocs++;
    doprocedure(fn);
    mfn(fn) = savmfn;
}
```

This code is used in chunks 188c and 190c.

`readdags` generates the nodes in a dag table. It reads each node individually but does *not* identify children nodes. `unmilldags` also generates the nodes in a dag table, but also maintains a stack so that children nodes can be identified.

192b ⟨*local declarations* 183b⟩+≡
```
#define readdags(dagnodes, dagtable, dag) \
    { int i, dagid, k; Range range; \
    gentable(dagnodes, i, 0) { \
        range = tableentry(Range *, dagnodes, i); \
        for (dagid = range.low; dagid <= range.hi; dagid = k) { \
            k = readdag(dagid, &dag, dagtable);
#define ereaddags    }}}

#define unmilldags(dagnodes, dagtable, dag, stack) \
    { int i, dagid, k; Range range; \
    gentable(dagnodes, i, 0) { \
        range = tableentry(Range *, dagnodes, i); \
        for (dagid = range.low; dagid <= range.hi; dagid = k) { \
            k = unmilldag(dagid, &dag, dagtable, stack);
#define eunmilldags(stack)    }}} reusetable(stack);

#define temp2formal(temp, formal) (formal && ttob(temp->type) == ttob(formal->type))
```

`inline` is the inlining work horse. It inlines the callee function `mfn` at the caller's call site in the flow graph node identified by `fgnid`. It calls `inlineargs` to map the actual to formal arguments; `inlinecaller` to make the appropriate changes in the caller; and `inlinecallee` to make the appropriate changes in the callee.

192c ⟨*external declarations* 183a⟩+≡
```
dclproto(void   inline,(Symbol, Symbol, MFunction *, int));
```

192d ⟨*local declarations* 183b⟩+≡
```
dclproto(static int inlineargs,(MFunction *, FGNode, MFunction *));
dclproto(static void inlinecallee,(Symbol, MFunction *, Symbol, Symbol, int));
dclproto(static void inlinesubcallee,(MFunction *, unsigned int, unsigned int, MTable *));
dclproto(static Symbol inlinecaller,(MFunction *, MFunction *, FGNode));
```

193a ⟨*definitions* 184a⟩+≡
```
void inline(caller, callee, mfn, fgnid) Symbol caller, callee; MFunction *mfn; {
    FGNode fgn, succ;
    Symbol target, retval;
    int argn;

    if (dbgflag)
        fprint(2,"XXX Inline %s(%d) at %d in %s(%d)\n", callee->name,
               tablesize(mfn(callee).localsymtab), fgnid, caller->name,
               tablesize(mfn(caller).localsymtab));
    fgn = idxtable(FGNode *, mfn(caller).flowgraph, fgnid);
    assert(fgn->type == CallN && callN(fgn).mfn == 0);

    /* Get label of call's successor */
    succ = idxtable(FGNode *, mfn(caller).flowgraph,
                    idxtable(short *, fgn->succs, 0) + fgn->id);
    assert(succ->labelsym);
    target = idxtable(Symbol *, mfn(caller).localsymtab, succ->labelsym);

    argn = inlineargs(&mfn(caller), fgn, mfn);
    retval = inlinecaller(&mfn(caller), mfn, fgn);
    inlinecallee(caller, mfn, target, retval, argn);
    mfunc(callee)->ninlines -= 1;
}
```

`convertarg` is true if `op` converts from a character, short, or unsigned to another type.

193b ⟨*local declarations* 183b⟩+≡
```
#define convertarg(_op) \
       ((generic(_op) == CVC || generic(_op) == CVS || generic(_op) == CVU))
```

`inlineargs` looks at each actual argument provided by the caller and determines whether it can replace the corresponding formal parameter in the callee. The callee's local symbol table is updated with the appropriate variables. These are used by `inlinecallee` to transform the callee's text appropriately. It also transforms the caller's code, eliminating the `ARG` nodes and `CALL` node.

193c ⟨*local declarations* 183b⟩+≡
```
static totargs = 0, actargs = 0, tempargs = 0;
```

194 ⟨*definitions* 184a⟩+≡

```
static int inlineargs(callermfn, fgn, mfn)
    FGNode fgn; MFunction *callermfn, *mfn; {

    int argn, fgnid;
    MDag mdag;
    FGNode cfgn;
    Range r;
    MTable stack;

    argn = 0;
    newtable(&stack, 16, sizeof(int), &transient);

    for (fgnid = callN(fgn).argnid; fgnid <= fgn->id; fgnid++) {
        cfgn = idxtable(FGNode *, callermfn->flowgraph, fgnid);
        switch(cfgn->type) {
        case ArgN: case CallN:
            readdags(cfgn->dagnodes, callermfn->dagtable, mdag) {
            switch(generic(mdag.opcode)) {
            case ARG: {
                Type type;
                MDag actualdag;
                Symbol actual, formal, temp;

                ⟨get formal's variable and type 195a⟩
                ⟨get actual's value 195c⟩
                ⟨map actual to formal 196a⟩
                argn++;
                totargs++;
                reusetable(stack);
                break;
            }
            case CALL: {
                unsigned short callbsymid = 0, tempsymid;

                ⟨eliminate CALL node 196b⟩
                reusetable(stack);
                break;
            }
            case ASGN: case EQ: case GE: case GT: case LE: case LT: case NE:
            case RET: case JUMP: case LABEL:
                reusetable(stack);
                break;
            default:
                addtotable(int *, &stack, dagid);
                break;
            }
        } ereaddags;
        default: break;
        }
    }
    return argn;
}
```

Argument evaluation order is target dependent, so we have to be sure that we're looking at the correct formal. We check `IR->left_to_right` to determine the correct offset for the argument in the `callee` symbol table.

195a ⟨*get formal's variable and type* 195a⟩≡

```
if (argn < tablesize(mfn->callee)) {
    int formalid;
    if (IR->left_to_right) formalid = argn;
    else formalid = tablesize(mfn->callee) - argn - 1;
    formal = idxtable(Symbol *, mfn->localsymtab,
                tableentry(unsigned short *, mfn->callee, formalid));
    type = isenum(formal->type) ? formal->type->type : formal->type;
} else {
    formal = (Symbol)0;
    type = (mdag.opcode != ARGB) ? btot(mdag.opcode) : (Type)0;
}
```

This code is used in chunk 194.

`symfor` selects the symbol `_i` from either the local or global symbol table depending on its class. `integraltype` returns the base type of an integral type constructor.

195b ⟨*local declarations* 183b⟩+≡

```
#define symfor(_i, symtab, glbsymtab)  (isglobal(_i) ? \
 (assert(symid(_i) > 0), idxtable(Symbol *, glbsymtab, symid(_i))) : \
 (assert(_i > 0),        idxtable(Symbol *, symtab, _i)))
#define integraltype(_t) (isenum(_t) ? (_t)->type : (_t))
```

The actual argument is buried inside `mill` code. This was a bad design choice; the variable or constant argument should be accessible from its `Arg` flowgraph node.

195c ⟨*get actual's value* 195c⟩≡

```
{
    int id;
    nopdag(dagid, &callermfn->dagtable);
    id = popfromtable(int *, &stack);
    readdag(id, &actualdag, callermfn->dagtable);
    while(convertarg(actualdag.opcode)) {
        id = popfromtable(int *, &stack);
        readdag(id, &actualdag, callermfn->dagtable);
    }
}
if (generic(actualdag.opcode) == CNST)
    actual = symfor(actualdag.sym0, callermfn->localsymtab, callermfn->bt->glbsymtab);
else if (usedag(actualdag.opcode, &actualdag, callermfn->dagtable, &stack))
    actual = symfor(actualdag.sym0, callermfn->localsymtab, callermfn->bt->glbsymtab);
else actual = (Symbol)0;
```

This code is used in chunk 194.

If possible, replace `formal` by `actual`. Otherwise, create a temporary with formal's type, assign the actual to the temporary, and replace the formal by the temporary.

196a ⟨*map actual to formal* 196a⟩≡

```
if (actual2formal(actual, formal)) {
    replacesym(formal, actual, mfn->localsymtab, "actual");
    actargs++;
    setdagrange(cfgn->dagnodes, i, 0, -1);
} else {    /* Create a new temporary of formal's type */
    temp = newtempsym(type, &callermfn->localsymtab, &fgn->localsymids);
    if (basictype(temp->type) && (formal && !formal->addressed))
        temp->sclass = REGISTER;
    r = asgndags(ttob(integraltype(temp->type)), optype(mdag.opcode), msym(temp).id,
            mdag.sym0, mdag.sym1, range, &callermfn->dagtable);
    setdagrange(cfgn->dagnodes, i, r.low, r.hi);
    if (formal) replacesym(formal, temp, mfn->localsymtab, "temp");
    tempargs++;
}
```

This code is used in chunk 194.

In the caller, the `Call` node is eliminated. In the callee, the address of its new "return" label is assigned to a temporary and the temporary is added to its symbol table. Elsewhere, the callee's `Return` node is converted to a `Jump` to this label.

196b ⟨*eliminate CALL node* 196b⟩≡

```
if (mdag.opcode == CALLB) {
    if (tablesize(mfn->callee) > 0)
        callbsymid = tableentry(unsigned short *, mfn->callee, tablesize(mfn->callee)-1)+1;
    else callbsymid = 1;
    tempsymid = newlocalsym(tableentry(Symbol *, mfn->localsymtab, callbsymid),
                &callermfn->localsymtab, &fgn->localsymids);
    nopdag(range.low, &callermfn->dagtable);
    nopdag(dagid, &callermfn->dagtable);
    r = asgndags(P, P, tempsymid, constantsym(btot(P)->size),
    constantsym(btot(P)->align), range, &callermfn->dagtable);
    setdagrange(cfgn->dagnodes, i, r.low, r.hi);
}
else setdagrange(cfgn->dagnodes, i, 0, -1);
```

This code is used in chunk 194.

`inlinecaller` converts a `Call` flow graph node in the caller into an inlined procedure. Any temporary symbol whose definition and use spanned the boundary of the `Call` node must be promoted to a local variable, because the injection of the inlined code will interrupt the scope of the temporary's previous definition and use.

197a ⟨*definitions* 184a⟩+≡
```
static Symbol inlinecaller(callermfn, mfn, fgn) MFunction *callermfn, *mfn; FGNode fgn; {
     int i, j;
     Symbol sym;
     unsigned short symid;
     FGNode lastn, entryn;
     MDag mdag;
     MTable temps, promote;

     ⟨find enclosing Entry node of Call node 197b⟩
     newtable(&temps, 8, SHORTSZ, &transient);

     ⟨find temporary variables' uses that cross codelist boundary 197c⟩
     callN(fgn).mfn = mfn;

     ⟨promote callee's return value if it is a temporary 198c⟩
     return sym;
}
```

The enclosing `Entry` node contains the locals announced (i.e., that become live) in this scope. If any temporaries are promoted, they are added to this `Entry` node's locals.

197b ⟨*find enclosing* `Entry` *node of* `Call` *node* 197b⟩≡
```
entryn = lastEntryN(callermfn->flowgraph, fgn->id);
nopdag(callN(fgn).dagn, &callermfn->dagtable);
```

This code is used in chunk 197a.

197c ⟨*find temporary variables' uses that cross codelist boundary* 197c⟩≡
```
if (!fgn->codelist) {
     lastn = fgn;
     ⟨search backward from Call node for temporaries' definitions 197d⟩
     if (tablesize(temps) > 0) {
          newtable(&promote, tablesize(temps), SHORTSZ, &transient);
          ⟨search forward from Call node for temporaries' uses 198a⟩
          ⟨promote temporaries that span Call node boundary 198b⟩
     }
     fgn->codelist = 1;
}
```
This code is used in chunk 197a.

197d ⟨*search backward from Call node for temporaries' definitions* 197d⟩≡
```
for (i = fgn->id - 1; i > 0 && !lastn->codelist; i--) {
     gentable(lastn->tempsymids, j, 0) {
          insertintable(&temps, tableentry(unsigned short *, lastn->tempsymids, j));
     }
     lastn = tableentry(FGNode *, callermfn->flowgraph, i);
}
```
This code is used in chunk 197c.

198a ⟨*search forward from Call node for temporaries' uses* 198a⟩≡
```
gentable(callermfn->flowgraph, i, fgn->id + 1) {
    lastn = tableentry(FGNode *, callermfn->flowgraph, i);
    readdags(lastn->dagnodes, callermfn->dagtable, mdag) {
        if (mdag.opcode == ADDRLP) {
            sym = idxtable(Symbol *, callermfn->localsymtab, mdag.sym0);
            if (sym->temporary != 0 && intable(temps, mdag.sym0) >= 0)
                insertintable(&promote, mdag.sym0);
        }
    } ereaddags;
    if (lastn->codelist) break;
}
```
This code is used in chunk 197c.

198b ⟨*promote temporaries that span Call node boundary* 198b⟩≡
```
gentable(promote, i, 0) {
    symid = tableentry(unsigned short *, promote, i);
    if (dbgflag) fprint(2, "promote %d\n", symid);
    sym = idxtable(Symbol *, callermfn->localsymtab, symid);
    sym->temporary = 0;
    addtotable(unsigned short *, &entryn->localsymids, symid);
}
promotetmps += tablesize(promote);
```
This code is used in chunk 197c.

198c ⟨*promote callee's return value if it is a temporary* 198c⟩≡
```
if (callN(fgn).retsymid) {
    sym = idxtable(Symbol *, callermfn->localsymtab, callN(fgn).retsymid);
    if (sym->temporary) {
        if (dbgflag) fprint(2, "promote %d\n", symid);
        sym->temporary = 0;
        addtotable(unsigned short *, &entryn->localsymids, msym(sym).id);
    }
}
else sym = (Symbol)0;
```
This code is used in chunk 197a.

`inlinecallee` converts the callee `mfn` into an inlined procedure. The formals in the callee's local symbol table are eliminated and its return value symbol and target label symbol are added to the local symbol table. `inlinesubcallee` is called on `mfn` to recursively convert any inlined procedures nested in the callee. Any references to formal parameters in `mfn` or its nested inlined children are converted to references to the corresponding variable (a formal, local, or constant) in the caller.

199a ⟨*definitions* 184a⟩+≡
```
static void inlinecallee(caller, mfn, target, retval, argn)
        Symbol caller, retval, target; MFunction *mfn; {
    FGNode entryn;
    int i;
    unsigned short retvalid, targetsymid;
    Symbol sym;
    MTable stack;

    newtable(&stack, 16, sizeof(int), &transient);

    cleartable(mfn->caller, mfn->caller.arena);
    entryn = idxtable(FGNode *, mfn->flowgraph, 1);
    gentable(mfn->callee, i, 0) {
        addtotable(unsigned short *, (&entryn->localsymids),
                   tableentry(unsigned short *, mfn->callee, i));
        ⟨check for argument count mismatch 199b⟩
    }
/*    cleartable(mfn->callee, mfn->callee.arena); */

    /* Add return value symbol and target label to local symbol table */
    retvalid = retval ? newlocalsym(retval, &mfn->localsymtab, &entryn->localsymids) : 0;
    targetsymid = target ?
                  newlocalsym(target, &mfn->localsymtab, &entryn->localsymids) : 0;
    inlinesubcallee(mfn, targetsymid, retvalid, &stack);
}
```

199b ⟨*check for argument count mismatch* 199b⟩≡
```
if (i >= argn) {
    warning("Argument count mismatch %s expects %d\n", caller->name, argn);
    sym = idxtable(Symbol *, mfn->localsymtab,
                   tableentry(unsigned short *, mfn->callee, i));
    sym->scope = LOCAL;
}
```
This code is used in chunk 199a.

8.4 Transforming inlined code

After making a copy of a procedure to inline, the procedure must be specialized at the call site by replacing, where possible, a formal by its corresponding actual parameter. `actual2formal` determines whether this replacement is possible. `inlinesubcallee` changes all references to formals to references to its replacement symbol. It replacement may be a local, constant, or another formal. `collapsesyms` recursively follows a chain of variable replacements.

200 ⟨*definitions* 184a⟩+≡

```
static void inlinesubcallee(mfn, targetsymid, retvalid, stack)
        MFunction *mfn; unsigned int targetsymid, retvalid; MTable *stack; {
    int i;
    MDag mdag;
    Symbol s, sym;
    Range r;
    FGNode fgn;
    ⟨collapse symbols and update any computed symbols 201a⟩
    gentable(mfn->flowgraph, i, 1) {
        reusetable(*stack);
        fgn = tableentry(FGNode *, mfn->flowgraph, i);
         readdags(fgn->dagnodes, mfn->dagtable, mdag) {
            switch(generic(mdag.opcode)) {
            case INDIR:
                ⟨change dereference of formal to constant ref, if possible 201b⟩
                addtotable(int *, stack, dagid);
                break;
            case ADDRF:
                ⟨change formal reference to local, if necessary 201c⟩
                addtotable(int *, stack, dagid);
                break;
            case RET:
                nopdag(dagid, &mfn->dagtable);
                ⟨assign return value to appropriate temporary 202a⟩
                /* fall through */
            case ASGN: case EQ: case GE: case GT: case LE: case LT: case NE:
            case JUMP: case LABEL:  reusetable(*stack);
                break;
            default:   addtotable(int *, stack, dagid);
                break;
            }
        } ereaddags;
        switch (fgn->type) {
        case CallN: /* if a nested inlined procedure, make changes recursively */
            if (callN(fgn).mfn)
                inlinesubcallee(callN(fgn).mfn, 0, 0, stack);
            break;
        case RetN: {
            int lastretn = 1;
            ⟨change Return node to Jump node 202b⟩
            break;
            }
        default: break;
        }
    }
}
```

201a ⟨*collapse symbols and update any computed symbols* 201a⟩≡
```
gentable(mfn->localsymtab, i, 1) {
    s = tableentry(Symbol *, mfn->localsymtab, i);
    if (!s->computed) {
        if (msym(s).replacesym)
            tableentry(Symbol *, mfn->localsymtab, i) = collapsesyms(s);
    }
    else {
        sym = idxtable(Symbol *, mfn->localsymtab, maddr(s).symid);
        s->scope = sym->scope;
        s->sclass = sym->sclass;
    }
}
```
This code is used in chunk 200.

201b ⟨*change dereference of formal to constant ref, if possible* 201b⟩≡
```
{
    MDag kiddag;
    int  kidid;

    if (kidid = usedag(mdag.opcode, &kiddag, mfn->dagtable, stack)) {
        if (kiddag.opcode == ADDRFP) {
            sym = idxtable(Symbol *, mfn->localsymtab, kiddag.sym0);
            if (sym->scope == CONSTANTS) {
                kiddag.opcode = CNST+ttob(sym->type);
                writedag(kidid, kiddag, &mfn->dagtable);
                nopdag(dagid, &mfn->dagtable);
            }
        }
    }
}
```
This code is used in chunk 200.

201c ⟨*change formal reference to local, if necessary* 201c⟩≡
```
sym = idxtable(Symbol *, mfn->localsymtab, mdag.sym0);
if (sym->scope >= LOCAL) {
    mdag.opcode = ADDRLP;
    writedag(dagid, mdag, &(mfn->dagtable));
}
```

This code is used in chunk 200.

202a ⟨*assign return value to appropriate temporary* 202a⟩≡
```
if (mdag.opcode != RETV)  {
    if (!retvalid) {
        Symbol retval;
        retval = newtempsym(btot(mdag.opcode), &mfn->localsymtab, &fgn->tempsymids);
        retval->temporary = 1;
        retvalid = msym(retval).id;
    }
    r = asgndags(optype(mdag.opcode), optype(mdag.opcode), retvalid,
        constantsym(btot(mdag.opcode)->size), constantsym(btot(mdag.opcode)->align),
        range, &mfn->dagtable);
    setdagrange(fgn->dagnodes, i, r.low, r.hi);
}
```

This code is used in chunk 200.

202b ⟨*change Return node to Jump node* 202b⟩≡
```
{
    FGNode nextn;
    int j;
    gentable(mfn->flowgraph, j, i+1) {
        nextn = tableentry(FGNode *, mfn->flowgraph, j);
        if (tablesize(nextn->dagnodes) > 0) {
            lastretn = 0;
            break;
        }
    }
}
/* Change Return to Jump if target is not next executable node */
if (!lastretn) {
    int addridx, lblidx;
    assert(targetsymid);
    addridx = newdag(ADDRGP, targetsymid, 0, &(mfn->dagtable));
    lblidx = newdag(JUMPV, 0, 0, &(mfn->dagtable));
    adddagrange(&fgn->dagnodes, addridx, lblidx);
}
cleartable(fgn->succs, fgn->succs.arena);
fgn->type = JumpN;
```
This code is used in chunk 200.

202c ⟨*local declarations* 183b⟩+≡
```
dclproto(static int actual2formal,(Symbol, Symbol));
dclproto(static Symbol collapsesyms,(Symbol));
```

203a ⟨*definitions* 184a⟩+≡
```
static int actual2formal(actual, formal) Symbol actual, formal; {
    if (actual && formal &&
       (!isglobal(msym(actual).id) || actual->scope == CONSTANTS) &&
       basictype(actual->type) &&
       ttob(integraltype(actual->type)) == ttob(integraltype(formal->type)) &&
       !actual->temporary && !actual->addressed && !msym(actual).downarg &&
       !formal->addressed && !msym(formal).written)
        return 1;
    else return 0;
}

static Symbol collapsesyms(s) Symbol s; {
    if (msym(s).replacesym)
        return msym(s).replacesym = collapsesyms(msym(s).replacesym);
    else return s;
}
```

8.5 Miscellaneous procedures

asgndags generates a new `ASGN` dag. `lastEntryN` finds the closest enclosing `Entry` node of the given node, i.e. it determines the entry point for the given node's scope. **newdag** creates a new dag in the given dag table. **newsym** adds a new symbol to the local symbol table. If the symbol is a local, the table of local symbols announced in the given entry node is updated; if it is a temporary, the table of temporaries announced in the given flow graph node is updated. **replacesym** replaces an old symbol by a new symbol in the given symbol table. **usedag** returns a valid dag id if the given dag is a use of a variable.

203b ⟨*local declarations* 183b⟩+≡
```
dclproto(static Range asgndags,(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int,
                                 unsigned int, Range, MTable *));
dclproto(static FGNode lastEntryN,(MTable, int));
dclproto(static int newdag,(unsigned int, unsigned int, unsigned int, MTable *));
dclproto(static void replacesym,(Symbol, Symbol, MTable, char *));
dclproto(static int usedag,(unsigned int, MDag *, MTable, MTable *));
```

204 ⟨*definitions* 184a⟩+≡

```
static Range asgndags(optype, koptype, k0, sym0, sym1, range, dagtable)
    unsigned int optype, koptype, k0, sym0, sym1; Range range; MTable *dagtable; {
    Range r;
    unsigned int cvtop = 0;

    r.low = newdag(ADDRLP, k0, 0, dagtable);
    copyrange(dagtable, range);
    if (optype != koptype) {
        switch(koptype) {
        case C: assert(optype == I || optype == U);
            cvtop = CVC+optype;
            break;
        case S: assert(optype == I || optype == U);
            cvtop = CVS+optype;
            break;
        case I: if (optype == C || optype == S) cvtop = CVI+optype;
            else if (optype == P) {
                newdag(CVIU, 0, 0, dagtable);
                cvtop = CVUP;
            }
            break;
        case U: assert(optype == C || optype == S || optype == I || optype == P);
            if (optype == C || optype == S || optype == P) cvtop = CVU+optype;
            break;
        case F: case D: default: break;
        }
    }
    if (cvtop) newdag(cvtop, 0, 0, dagtable);
    r.hi = newdag(ASGN+ (optype == U ? I : optype), sym0, sym1, dagtable);
    return r;
}
```

205a ⟨*definitions* 184a⟩+≡
```
static FGNode lastEntryN(flowgraph, fgnid) MTable flowgraph; {
    int ct, i;
    FGNode entryn = 0;
    for (ct = 0, i = fgnid; i > 0; i--) {
        entryn = idxtable(FGNode *, flowgraph, i);
        if (entryn->type == EntryN) {
            if (ct == 0) break;
            else ct += 1;
        } else if (entryn->type == ExitN) ct -= 1;
    }
    return entryn;
}

static int newdag(opcode, sym0, sym1, dagtab)
    unsigned int opcode, sym0, sym1; MTable *dagtab; {
    MDag mdag;
    int idx;

    idx = tablesize((*dagtab));
    mdag.opcode = opcode;
    mdag.sym0 = sym0;
    mdag.sym1 = sym1;
    writedag(idx, mdag, dagtab);
    return idx;
}
```

205b ⟨*external declarations* 183a⟩+≡
```
dclproto(extern int newsym,(Symbol, MTable *, FGNode, FGNode));
```

206a ⟨*definitions* 184a⟩+≡
```
int newsym(p, localsymtab, centryn, cfgn)
    Symbol p; MTable *localsymtab; FGNode centryn, cfgn; {
    if (msym(p).id != 0) return msym(p).id;
    if (msym(p).typesym || p->scope == CONSTANTS || p->scope == GLOBAL ||
        (p->scope != LABELS && (p->sclass == STATIC || p->sclass == EXTERN)))
        assert(0);
    if (p->temporary == 1 && p->sclass == REGISTER)
        msym(p).id = newlocalsym(p, localsymtab, &cfgn->tempsymids);
    else
        msym(p).id = newlocalsym(p, localsymtab, &centryn->localsymids);
    return msym(p).id;
}

static void replacesym(old, new, symtab, str) Symbol old, new; MTable symtab; char *str; {
    if (dbgflag)
        fprint(2, "%s %s/%x replaces formal %s/%x\n",
               str, new->name, msym(new).id, old->name, msym(old).id);
    msym(new).replacement = 1;
    assert(ttob(integraltype(old->type)) == ttob(integraltype(new->type)));
    tableentry(Symbol *, symtab, msym(old).id) = new;
    new->addressed |= old->addressed;
    msym(new).written |= msym(old).written;
    msym(old).replacesym = new;
}
static int usedag(opcode, dag, dagtable, stack)
        unsigned int opcode; MDag *dag; MTable dagtable, *stack; {
    int k0 = 0;
    if (generic(opcode) == INDIR) {
        assert(k0 = popfromtable(int *, stack));
        readdag(k0, dag, dagtable);
        if (isaddrop(dag->opcode))
            return k0;
    }
    return 0;
}
```

9 Specialization

A clone's name is constructed from the name of its original parent and the values of the arguments with which it is specialized. `C`n denotes an argument with constant value n; `K`t denotes an argument with known type t; `X` denotes a placeholder for an unknown argument.

`specialize` specializes `callee` in the context of `caller` at the call site `cgn`. The callee's calling context is found by evaluating its arguments at the call site. Type and constant propagation are applied to the caller before specialization, so if any useful context exists, it is associated with the caller's variables.

206b ⟨*local declarations* 183b⟩+≡
```
static int clonect = 0;
int Clonelimit = 200;
int Clonedups = 0;
```

207a ⟨*external declarations* 183a⟩+≡
```
extern int Clonelimit;
extern int Clonedups;
dclproto(void specialize,(Symbol, Symbol, CallGraphNode *, Arena *));
```

207b ⟨*definitions* 184a⟩+≡
```
void specialize(caller, callee, cgn, arena)
    Symbol caller, callee; CallGraphNode *cgn; Arena *arena;
{
    MFunction *mfn = (MFunction *)0;
    Symbol clone = 0;
    MTable cloneargnames, cloneargs;
    char *clonename;
    BaseTable *bt;
    int constargs = 0, knownargs = 0, nargs = 0, i;
    FGNode calln;

    if (variadic(callee->type)) return;
    else nargs = argcount(callee->type);
    newtable(&cloneargnames, nargs, sizeof(char *), arena);
    newtable(&cloneargs, nargs, sizeof(Node), arena);
    assert(calln = idxtable(FGNode *, mfn(caller).flowgraph, cgn->fgnid));
    ⟨report argument information 208⟩
    if (constargs > 0 || knownargs > 0) {
        if (dbgflag) fprint(2,"@@ Specialize %s for %s\n", callee->name, caller->name);
        clonename = callee->name;
        gentable(cloneargnames, i, 0) {
            clonename = stringf("%s%s", clonename, tableentry(char **, cloneargnames, i));
        }
        fprint(2,"@@ Cloning %s\n", clonename);
        if (!(clone = lookupsym(clonename, &bt)) && (clonect < Clonelimit)) {
            ⟨make a clone of callee 209a⟩
            ⟨read clean copy of callee's text 209b⟩
            ⟨generate new callee 209c⟩
            clonect++;
        } else if (clone) {
            fprint(2,"@@ Clone exists\n");
            Clonedups++;
        }
        if (clone) {
            ⟨change call site in caller 209d⟩
        } else fprint(2,"@@ Too many clones\n");
    }
}
```

207c ⟨*local declarations* 183b⟩+≡
```
static char *Xname = "_X";
```

208 ⟨*report argument information* 208⟩≡

```
{
    FGNode argn;
    Node p;
    int nodect, i, val, argct = 0;
    Symbol s;
    char *argname;

    if (IR->left_to_right) argct = 0;
    else argct = nargs - 1;
    for (nodect = callN(calln).argnid; nodect <= cgn->fgnid; nodect++) {
        assert(argn = idxtable(FGNode *, mfn(caller).flowgraph, nodect));
        gentable(argn->dagroots, i, 0) {
            p = idxtable(Node *, mfn(caller).nodetable,
                         tableentry(int *, argn->dagroots, i));
            if (p && generic(p->op) == ARG) {
                argname = Xname;
                if ((s = isconstsym(p->kids[0])) != 0) {
                    constargs++;
                    if (s->type != doubletype)
                        argname = stringf("_C%x", s->u.c.v.u);
                    else argname = stringf("_C%x%x", ((unsigned *)&s->u.c.v.d)[0],
                                           ((unsigned *)&s->u.c.v.d)[1]);
                    updatetable(Node *, &cloneargs, argct, p->kids[0]);
                } else if ((s = issimpleuse(p->kids[0])) && msym(s).typeval != 0) {
                    knownargs++;
                    argname = stringf("_K%d", msym(s).typeval);
                }
                updatetable(char **, &cloneargnames, argct, argname);
                if (IR->left_to_right) argct++;
                else argct--;
            }
        }
    }
    if (dbgflag)
        fprint(2,"@@ argct %d constargs %d knownargs %d \n", nargs, constargs, knownargs);
}
```

This code is used in chunk 207b.

If the `callee` has already been specialized with the same arguments, we use the existing clone. Otherwise, we create a `clone` of the callee and install it in the global symbol table. The values of a clone's calling context are included in its name; we install that name in the global symbol table so that a simple lookup determines if such a clone already exists. The clone is installed as an external so that the resolution module does not give a new unique name to a static cloned function.

209a ⟨*make a* `clone` *of* `callee` 209a⟩≡
```
{
    MFuncSym mfuncsym;
    int sclass;

    clone = (Symbol)yyalloc((sizeof *clone), permanent);
    bcopy((char *)callee, (char *)clone, sizeof *clone);
    clone->name = clonename;
    msym(clone).raddr = 0;
    sclass = clone->sclass;
    clone->sclass = AUTO;
    (*IR->defsymbol)(clone);
    resolvesym(clone, mfn(caller).bt);
    clone->sclass = sclass;
    mfuncsym = (MFuncSym)yyalloc(sizeof *mfuncsym, permanent);
    bcopy((char *)mfunc(callee), (char *)mfuncsym, sizeof *mfuncsym);
    mfunc(clone) = mfuncsym;
}
```
This code is used in chunk 207b.

A clean copy of `callee` must be read if the callee has already been emitted (and thus dirtied) or if it is no longer cached. The clean copy is made in `caller`'s arena.

209b ⟨*read clean copy of* `callee`*'s text* 209b⟩≡
```
if (mfunc(callee)->emitted || !mfunc(callee)->cached) {
    breadProcedure(callee);
}
mfn = bcopyMFunction(&mfn(callee), arena);
mfunc(clone)->emitted = 0;
bcopy((char *)mfn, (char *)&mfn(clone), (sizeof *mfn));
mfn(clone).sym = clone;
```
This code is used in chunk 207b.

209c ⟨*generate new callee* 209c⟩≡
```
genprocedure(clone, cloneargs, arena);
```
This code is used in chunk 207b.

209d ⟨*change call site in* `caller` 209d⟩≡
```
{
    Node p;
    assert (p = idxtable(Node *, mfn(caller).nodetable, callN(calln).dagn));
    if (generic(p->op) == ASGN) p = p->kids[1];
    assert (generic(p->op) == CALL && isaddrop(p->kids[0]->op));
    p->kids[0]->syms[0] = clone;
}
```
This code is used in chunk 207b.

209e ⟨*local declarations* 183b⟩+≡
```
static int  argcount(Type);
```

210a ⟨*definitions* 184a⟩+≡

```
static int argcount(ty) Type ty; {
    if (isfunc(ty) && !ty->u.f.oldstyle && ty->u.f.proto) {
        int i;
        for (i = 0; ty->u.f.proto[i]; i++)
            ;
        if (ty->u.f.proto[i-1] != voidtype) return i;
    }
    return 0;
}
```

210b ⟨*optmill.h* 210b⟩≡

⟨*external declarations* 183a⟩

This code is written to file `optmill.h`.

210c ⟨*optmill.c* 210c⟩≡

```
#include <c.h>
#include <simp.h>
#include <mld.h>
#include <unmill.h>
#include <optmill.h>
#include <resextsym.h>
#include <decl.h>
#include <bemit.h>
```

⟨*local declarations* 183b⟩
⟨*definitions* 184a⟩

This code is written to file `optmill.c`.

10 Data-flow analysis

This module contains procedures that compute the information required by intra-procedural, data-flow algorithms.

10.1 Definitions and uses

`defuse_analysis` analyzes the text of function `fn` and computes all of the definition and use data necessary for computing reaching definitions. Not all dags or symbols are significant during analysis. Only the *roots* of dags represent relevant statements and some globals (e.g., type symbols) are not relevant. The statement (`smap`) and variable maps (`vmap`) map the `mill` identifiers of statements and variables to their identifiers in the analysis code. The maps `ismap` and `ivmap` compute the inverse mapping. We use `s` and `v` to be consistent with standard data-flow analysis terminology. These maps are returned in a `DefUseInfo` structure and can then be used by other analysis procedures.

Potentially aliased variables are identified in `defuse_analysis`. The function `aliasfn` is applied to the set of relevant symbols in `ivmap` to determine if a variable is aliased. Since aliases are language specific, a language-specific function is applied to each symbol.

211a ⟨*external declarations* 211a⟩≡
```
typedef struct defuseinfo {
    MTable smap, ismap, lvmap, vmap, ivmap;
} DefUseInfo;
```

This definition is continued in chunks 211, 214–17, 225c, 229a, 231, 232b, 235a, 237e, 240, 245d, 248b, 250b, and 251d.
This code is used in chunk 253b.

211b ⟨*external declarations* 211a⟩+≡
```
typedef struct defuse {
    int id, ivid, typeval;
    struct set adefs, udefs, auses, uuses, svdefs;
} DefUse;
#define DEFUSESZ sizeof(struct defuse)
```

211c ⟨*local declarations* 211c⟩≡
```
static int analdbgflag = 1;
```
This definition is continued in chunks 214d, 216g, 217c, 220c, 222, 223, 227a, 228d, 235c, 237, 241a, 249, 251b, and 253c.
This code is used in chunk 253d.

211d ⟨*external declarations* 211a⟩+≡
```
typedef int (*SymbolFn)(Symbol);
dclproto(void init_analysis_maps,(MFunction *, DefUseInfo *, Arena *));
dclproto(DefUseInfo defuse_analysis,(MFunction *, SymbolFn, Arena *));
```

212a ⟨*definitions* 212a⟩≡
```
DefUseInfo defuse_analysis(mfn, aliasfn, arena)
     MFunction *mfn; SymbolFn aliasfn; Arena *arena; {
     struct set aliases;
     DefUseInfo duinfo;

     if (analdbgflag) fprint(2,"def/use...\n");
     init_analysis_maps(mfn, &duinfo, arena);
     if (tablesize(duinfo.ismap) * tablesize(duinfo.ivmap) > OPTLIMIT) {
         warning("Cannot analyze %s; procedure too big: %d statements, %d variables\n",
                 mfn->sym->name, tablesize(duinfo.ismap), tablesize(duinfo.ivmap));
         cleartable(duinfo.smap, arena);
         return duinfo;
     }
     init_aliases(mfn, &duinfo, &aliases, aliasfn);
     defs_and_uses(mfn, &duinfo, &aliases);
     return duinfo;
}
```

This definition is continued in chunks 212b, 215, 216h, 218a, 221, 223, 224a, 226a, 228e, 229b, 232, 235–38, 240e, 241b, and 248–52.
This code is used in chunk 253d.

Each statement and variable has an associated `defuse` (`du`) node that contains its true identifer and its definitions and uses sets. `init_analysis_maps` identifies all those statements and variables that are relevant to optimization. `defuse_analysis` uses those maps to find all definitions and uses of relevant variables.

212b ⟨*definitions* 212a⟩+≡
```
void init_analysis_maps(mfn, duinfo, arena)
   MFunction *mfn; DefUseInfo *duinfo; Arena *arena; {
     int ivmap_size, localct, globalct;

     ⟨compute ivmap_size 213c⟩
     ⟨create smap/ismap for statements 212c⟩
     ⟨create lvmap/vmap/ivmap for locals and globals 213d⟩
     var_used_tag = (unsigned short)((short) var_used_tag - 1);
}
```

212c ⟨*create* `smap/ismap` *for statements* 212c⟩≡
```
{
     FGNode fgnode;
     int    fgidx;
     int    i, rootid, mapid = 1;
     DefUse du;

     newtable(&duinfo->smap,  tablesize(mfn->nodetable)+1, sizeof(int), arena);
     newtable(&duinfo->ismap, mfn->dagroots + 1, DEFUSESZ, arena);
     for (i = 1; i <= tablesize(mfn->nodetable); i++)
         updatetable(int *, &duinfo->smap, i, 0);
     gentable(mfn->flowgraph, fgidx, 1) {
         fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
         ⟨install roots in ismap 213a⟩
     }
}
```
This code is used in chunk 212b.

213a ⟨*install roots in* `ismap` 213a⟩≡

```
{
    Node p;
    gentable(fgnode->dagroots, i, 0) {
        rootid = tableentry(int *, fgnode->dagroots, i);
        if (rootid != 0) {
            assert ((p = idxtable(Node *, mfn->nodetable, rootid))
                     && rootop(generic(p->op)));
            updatetable(int *, &duinfo->smap, rootid, mapid);
            du.id = rootid;
            du.ivid = 0;
            du.typeval = 0;
            ⟨initialize sets for statement 213b⟩
            updatetable(DefUse *, &duinfo->ismap, mapid, du);
            mapid++;
            }
    }
}
```

This code is used in chunk 212c.

213b ⟨*initialize sets for statement* 213b⟩≡

```
init_defuse_sets(&du, ivmap_size);
```

This code is used in chunk 213a.

213c ⟨*compute* `ivmap_size` 213c⟩≡

```
{
    int idx;
    Symbol p;

    localct = globalct = 0;
    ⟨tag globals used in this function 214e⟩
    gentable(mfn->localsymtab, idx, 1) {
        p = tableentry(Symbol *, mfn->localsymtab, idx);
        if (p->scope != LABELS) localct++;
    }
    if (optdbgflag)
      fprint(2,"dags %d globs %d locals %d vsize %d\n", mfn->dagroots,
        globalct, localct, (globalct + localct)*mfn->dagroots);
    ivmap_size = localct + globalct + 1;
}
```

This code is used in chunk 212b.

213d ⟨*create* `lvmap/vmap/ivmap` *for locals and globals* 213d⟩≡

```
{
    int idx, mapid = 1;
    Symbol p;
    DefUse du;
    newtable(&duinfo->lvmap, localct + 1, sizeof(int), arena);
    newtable(&duinfo->vmap,  globalct + 1, sizeof(int), arena);
    newtable(&duinfo->ivmap, localct + globalct + 1, DEFUSESZ, arena);
    ⟨update lvmap and ivmap with locals 214a⟩
    ⟨update vmap and ivmap with globals 214c⟩
}
```

This code is used in chunk 212b.

Not all local symbols are included in the set of variables necessary for analysis; those that are used are installed in `lvmap`. Their definitions and uses sets are in the first entries of `ivmap`.

214a ⟨*update* `lvmap` *and* `ivmap` *with locals* 214a⟩≡
```
gentable(mfn->localsymtab, idx, 1) {
    p = tableentry(Symbol *, mfn->localsymtab, idx);
    if (p->scope != LABELS) {
/*        msym(p).oid = du.id = msym(p).id; <== this won't work if symbol
is a replacement symbol in inlined code */
        msym(p).oid = du.id = idx;
        updatetable(int *, &duinfo->lvmap, idx, mapid);
        ⟨initialize sets for variable 214b⟩
        updatetable(DefUse *, &duinfo->ivmap, mapid, du);
        mapid++;
    } else updatetable(int *, &duinfo->lvmap, idx, 0);
}
```
This code is used in chunk 213d.

214b ⟨*initialize sets for variable* 214b⟩≡
```
init_defuse_sets(&du, tablesize(duinfo->ismap));
```
This code is used in chunk 214.

Most globals are omitted from analysis; the `vmap` specifies which globals are included. Those that are included are placed in the `ivmap` following the locals.

214c ⟨*update* `vmap` *and* `ivmap` *with globals* 214c⟩≡
```
gentable(mfn->bt->glbsymtab, idx, 1) {
    p = tableentry(Symbol *, mfn->bt->glbsymtab, idx);
    if (isoptvar(p) && msym(p).oid == var_used_tag) {
        msym(p).oid = du.id = GLOBSYM(idx);
        updatetable(int *, &duinfo->vmap, idx, mapid);
        ⟨initialize sets for variable 214b⟩
        updatetable(DefUse *, &duinfo->ivmap, mapid, du);
        mapid++;
    } else updatetable(int *, &duinfo->vmap, idx, 0);
}
```
This code is used in chunk 213d.

214d ⟨*local declarations* 211c⟩+≡
```
static unsigned short var_used_tag = SHRT_MAX;
```

214e ⟨*tag globals used in this function* 214e⟩≡
```
{
    Args   args;
    args.tag.i = 0;
    args.tag.tag = var_used_tag;
    match_all_nodes(mfn, global_ref, identitynode, &args);
    globalct = args.tag.i;
}
```
This code is used in chunk 213c.

`isoptvar` determines if a variable should be included in the `vmap`. Constants, type symbols, and functions that never appear on the right-hand side are excluded.

214f ⟨*external declarations* 211a⟩+≡
```
dclproto(int isoptvar,(Symbol));
```

215a ⟨*definitions* 212a⟩+≡
```
int isoptvar(s) Symbol s; {
    if (msym(s).typesym != 0 || s->scope == LABELS
       || s->scope == CONSTANTS || s->type == 0
       || s->u.seg == LIT || (isfunc(s->type) && !(indircalledproc(s))))
        return 0;
    return 1;
}
```

The function's `ismap` contains all the dag roots in `mfn`. `defs_and_uses` generates these roots and then derives their definitions and uses.

215b ⟨*external declarations* 211a⟩+≡
```
dclproto(void defs_and_uses,(MFunction *, DefUseInfo *, Set));
```

215c ⟨*definitions* 212a⟩+≡
```
void defs_and_uses(mfn, duinfo, aliases) MFunction *mfn; DefUseInfo *duinfo; Set aliases; {
    int    idx, mapid;
    DefUse *du;
    Node   p;
    char   *tab = "";

    gentable(duinfo->ismap, idx, 1) {
        du = &(tableentry(DefUse *, duinfo->ismap, idx));
        assert(p = idxtable(Node *, mfn->nodetable, du->id));
        if (optdbgflag) {
            ⟨print statement header 216a⟩
            fprint(2,"\n");
        }
        defs(p, idx, du, duinfo, aliases);
        uses(p, idx, du, duinfo, aliases);
    }
    ⟨def/use sanity check 215d⟩
}
```

215d ⟨*def/use sanity check* 215d⟩≡
```
{
    int    idx, mapid;
    DefUse *du;
    Node   p, q;
    char   *tab = "\t";

   if (optdbgflag)
        gentable(duinfo->ismap, idx, 1) {
            du = &(tableentry(DefUse *, duinfo->ismap, idx));
            assert(p = idxtable(Node *, mfn->nodetable, du->id));
            q = p;
            ⟨print statement header 216a⟩
            fprint(2,"\n");
            ⟨print statement's udefs 216b⟩
            ⟨print statement's adefs 216c⟩
            ⟨print statement's uuses 216d⟩
            ⟨print statement's auses 216e⟩
        }
}
```
This code is used in chunk 215c.

216a ⟨*print statement header* 216a⟩≡
```
fprint(2,"%s%d:%d %s ", tab, du->id, idx, opname(generic(p->op)));
```
This code is used in chunks 215 and 228e.

216b ⟨*print statement's* `udefs` 216b⟩≡
```
print_defuses(idx, du, duinfo, DEFS, UNAMBIG, mfn);
```
This code is used in chunk 215d.

216c ⟨*print statement's* `adefs` 216c⟩≡
```
print_defuses(idx, du, duinfo, DEFS, AMBIG, mfn);
```
This code is used in chunk 215d.

216d ⟨*print statement's* `uuses` 216d⟩≡
```
print_defuses(idx, du, duinfo, USES, UNAMBIG, mfn);
```
This code is used in chunk 215d.

216e ⟨*print statement's* `auses` 216e⟩≡
```
print_defuses(idx, du, duinfo, USES, AMBIG, mfn);
```
This code is used in chunk 215d.

216f ⟨*external declarations* 211a⟩+≡
```
#define symfor(_i)  (isglobal(_i) ? (idxtable(Symbol *, mfn->bt->glbsymtab, symid(_i))) : \
                   (idxtable(Symbol *, mfn->localsymtab, _i)))
```

We use a terribly conservative estimate of the set of variables that could be aliased. A better approximation would be by type but first we want to get it right.

216g ⟨*local declarations* 211c⟩+≡
```
dclproto(static void init_aliases,(MFunction *, DefUseInfo *, Set, SymbolFn));
```

216h ⟨*definitions* 212a⟩+≡
```
static void init_aliases(mfn, duinfo, aliases, aliasfn)
    MFunction *mfn; DefUseInfo *duinfo; Set aliases; SymbolFn aliasfn; {
    DefUse du;
    Symbol p;
    int idx;

    if (analdbgflag) fprint(2,"Aliases\n");
    initset(aliases, tablesize(duinfo->ivmap));
    gentable(duinfo->ivmap, idx, 1) {
        du = tableentry(DefUse *, duinfo->ivmap, idx);
        p = symfor(du.id);
        if (aliasfn(p)) {
            insert(aliases, idx);
        }
    }
    if (optdbgflag) ⟨alias sanity check 217a⟩
}
```

217a ⟨*alias sanity check* 217a⟩≡
```
{
    int i;
    gentable(duinfo->ivmap, i, 1) {
        if (member(aliases, i)) {
            p = symfor(tableentry(DefUse *, duinfo->ivmap, i).id);
/*            emitSymbol(2, p, "\t"); */
        }
    }
}
```
This code is used in chunk 216h.

10.1.1 Defs

217b ⟨*external declarations* 211a⟩+≡
```
#define ivmapid(_i)  (isglobal(_i) ? (idxtable(int *, duinfo->vmap, symid(_i))) : \
                      (idxtable(int *, duinfo->lvmap, _i)))
#define ivsym(_i)    (symfor(tableentry(DefUse *, duinfo->ivmap, _i).id))
```

217c ⟨*local declarations* 211c⟩+≡
```
dclproto(static void defs,(Node, int, DefUse *, DefUseInfo *, Set));
```

218a ⟨*definitions* 212a⟩+≡
```
static void defs(p, idx, du, duinfo, aliases)
    Node p; DefUse *du; DefUseInfo *duinfo; Set aliases;
{
    Node q = p;
    char *tab = "";

    if (generic(p->op) == ASGN) {
        if (isaddrop(p->kids[0]->op)) {
            q = p->kids[0];
            ⟨unambiguous var definition 218b⟩
        } else if (generic(p->kids[0]->op) == INDIR) {
            ⟨ambiguous type definition 219d⟩
        } else if (generic(p->kids[0]->op) == LSH ||
                   generic(p->kids[0]->op) == RSH) {
            ⟨unambiguous array definition 219e⟩
        } else if (p->kids[0]->op == ADDP ||
                   p->kids[0]->op == SUBP) {
            ⟨unambiguous field definition 219f⟩
        } else {
            ⟨ambiguous type definition 219d⟩
        }
    } else if (p->op == ARGP) {
        if (isaddrop(p->kids[0]->op)) {
            q = p->kids[0];
            if (isptr(q->syms[0]->type)) {
                ⟨ambiguous definition 219g⟩
            } else {
                ⟨ambiguous var definition 219c⟩
            }
        } else if (p->kids[0]->op == INDIRP) {
            ⟨ambiguous definition 219g⟩
        } else {
            ⟨ambiguous definition 219g⟩
        }
    }

}
```

218b ⟨*unambiguous var definition* 218b⟩≡
```
{
    Symbol s = q->syms[0];
    ⟨print var's udef 224b⟩
    newdef(du, idx, duinfo->ivmap, ivmapid(msym(s).oid), UNAMBIG);
    assert(du->ivid = ivmapid(msym(s).oid));
    ⟨insert statement in variable's udefs 219b⟩
    if (⟨assignment to replacement symbol 219a⟩) {
        ⟨ambiguous definition 219g⟩
    }
}
```
This code is used in chunk 218a.

When a function is inlined, all `VAR` parameters are assigned to temporaries, which are then substituted for the formal parameters in the inlined code. The `VAR` parameters are still treated like addressed arguments, i.e. the assignment is considered an ambiguous definition of the variable.

219a ⟨*assignment to replacement symbol* 219a⟩≡
```
  (p->op == ASGNP && msym(s).replacement == 1)
```
This code is used in chunk 218b.

219b ⟨*insert statement in variable's* `udefs` 219b⟩≡
```
  newdefuse(&tableentry(DefUse *, duinfo->ivmap, du->ivid), idx, DEFS, UNAMBIG);
```
This code is used in chunk 218b.

219c ⟨*ambiguous var definition* 219c⟩≡
```
  {
      Symbol s = q->syms[0];
      int ivid = ivmapid(msym(s).oid);
      if (ivid != 0) {
          ⟨print var's adef 224c⟩
          newdef(du, idx, duinfo->ivmap, ivid, AMBIG);
          du->ivid = 0;
      }
  }
```
This code is used in chunk 218a.

219d ⟨*ambiguous type definition* 219d⟩≡
```
  if (optdbgflag) fprint(2, "ADef %c\n", TYPENAMES[optype(p->op)]);
  uset(&(du->adefs), aliases);
  du->ivid = 0;
  ⟨insert statement in aliased variables' adefs 220a⟩
```
This code is used in chunk 218a.

219e ⟨*unambiguous array definition* 219e⟩≡
```
  if (optdbgflag) fprint(2, "ARRAY? ADef %c\n", TYPENAMES[optype(p->op)]);
  uset(&(du->adefs), aliases);
  du->ivid = 0;
  ⟨insert statement in aliased variables' adefs 220a⟩
```
This code is used in chunk 218a.

219f ⟨*unambiguous field definition* 219f⟩≡
```
  if (optdbgflag) fprint(2, "FIELD? ADef %c\n", TYPENAMES[optype(p->op)]);
  uset(&(du->adefs), aliases);
  du->ivid = 0;
  ⟨insert statement in aliased variables' adefs 220a⟩
```
This code is used in chunk 218a.

219g ⟨*ambiguous definition* 219g⟩≡
```
  if (optdbgflag) fprint(2, "UNKNOWN ADef\n");
  uset(&(du->adefs), aliases);
  du->ivid = 0;
  ⟨insert statement in aliased variables' adefs 220a⟩
```
This code is used in chunk 218.

220a ⟨*insert statement in aliased variables'* `adefs` 220a⟩≡
```
{
    int ivid;
    gentable(duinfo->ivmap, ivid, 1) {
        if (member(aliases, ivid)) {
            ⟨insert statement in variable's adefs 220b⟩
        }
    }
}
```
This code is used in chunk 219.

220b ⟨*insert statement in variable's* `adefs` 220b⟩≡
```
newdefuse(&tableentry(DefUse *, duinfo->ivmap, ivid), idx, DEFS, AMBIG);
```
This code is used in chunk 220a.

10.1.2 Uses

220c ⟨*local declarations* 211c⟩+≡
```
dclproto(static void uses,(Node, int, DefUse *, DefUseInfo *, Set));
```

221 ⟨*definitions* 212a⟩+≡

```
static void uses(p, idx, du, duinfo, aliases)
    Node p; DefUse *du; DefUseInfo *duinfo; Set aliases;
{
    Node q = p;
    char *tab = "";

    if (!p) return;
    switch(generic(p->op)) {
    case LABEL: case CNST: break;
    case ADDRL: case ADDRF: case ADDRG:
        ⟨unambiguous var use 222a⟩
        break;
    case CALL: case RET:
        if (p->kids[0] == NULL) break;
        /* fall through */
    case BCOM:
    case CVC:  case CVD:  case CVF: case CVI:
    case CVP:  case CVS:  case CVU:
    case JUMP: case NEG:
        uses(p->kids[0], idx, du, duinfo, aliases);
        break;
    case INDIR:
        if (isaddrop(p->kids[0]->op)) {
            q = p->kids[0];
            ⟨unambiguous var use 222a⟩
        } else {
            ⟨ambiguous type use 222c⟩
            uses(p->kids[0], idx, du, duinfo, aliases);
        }
        break;
    case ARG:
        uses(p->kids[0], idx, du, duinfo, aliases);
        break;
    case ASGN:
        if (!isaddrop(p->kids[0]->op))
            uses(p->kids[0], idx, du, duinfo, aliases);
        uses(p->kids[1], idx, du, duinfo, aliases);
        break;
    case EQ:   case GE:  case GT:   case LE:  case LT: case NE:
    case ADD: case BAND: case BOR: case BXOR: case DIV:
    case LSH: case MOD:  case MUL: case RSH:  case SUB:
        uses(p->kids[0], idx, du, duinfo, aliases);
        uses(p->kids[1], idx, du, duinfo, aliases);
        break;
    default: error("Invalid op on RHS %s\n", opname(p->op));
        break;
    }
}
```

Some variables that appear in the text of a procedure do not appear in the procedure's local symbol table nor in the module's global symbol table. They are recognized by their optimization ids `oid`'s, which are not equal to the current procedure's `oid`. They are the result of applying data-driven optimizations (e.g., replacing a method invocation by a direct procedure call). Since they do not appear in the original text, they are omitted from global analysis.

222a ⟨*unambiguous var use* 222a⟩≡
```
{
     int ivid;
     Symbol s = q->syms[0];

     if (msym(s).oid != var_used_tag + 1) {
          ivid = ivmapid(msym(s).oid);
          if (ivid != 0) {
               ⟨print var's uuse 224d⟩
               insert(&(du->uuses), ivid);
               ⟨insert statement in variable's uuses 222b⟩
          }
     }
}
```
This code is used in chunk 221.

222b ⟨*insert statement in variable's* `uuses` 222b⟩≡
```
newdefuse(&tableentry(DefUse *, duinfo->ivmap, ivid), idx, USES, UNAMBIG);
```
This code is used in chunk 222a.

222c ⟨*ambiguous type use* 222c⟩≡
```
if (optdbgflag) fprint(2, "%sAUse %c\n", tab, TYPENAMES[optype(p->op)]);
uset(&(du->auses), aliases);
⟨insert statement in aliased variables' auses 222d⟩
```
This code is used in chunk 221.

222d ⟨*insert statement in aliased variables'* `auses` 222d⟩≡
```
{
     int ivid;
     gentable(duinfo->ivmap, ivid, 1) {
          if (member(aliases, ivid)) {
               ⟨insert statement in variable's auses 222e⟩
          }
     }
}
```
This code is used in chunk 222c.

222e ⟨*insert statement in variable's* `auses` 222e⟩≡
```
newdefuse(&tableentry(DefUse *, duinfo->ivmap, ivid), idx, USES, AMBIG);
```
This code is used in chunk 222d.

222f ⟨*local declarations* 211c⟩+≡
```
dclproto(static void newdefuse,(DefUse *, int, int, int));
```

223a ⟨*definitions* 212a⟩+≡
```
static void newdefuse(du, sidx, def, unambig) DefUse *du; {
    if (def) {
        if (unambig) insert(&du->udefs, sidx);
        else insert(&du->adefs, sidx);
    } else {
        if (unambig) insert(&du->uuses, sidx);
        else insert(&du->auses, sidx);
    }
}
```

10.2 Managing def and use sets

223b ⟨*local declarations* 211c⟩+≡
```
dclproto(static void init_defuse_sets,(DefUse *, int));
```

223c ⟨*definitions* 212a⟩+≡
```
static void init_defuse_sets(du, size) DefUse *du; {
    initset(&(du->adefs), size);
    initset(&(du->udefs), size);
    initset(&(du->auses), size);
    initset(&(du->uuses), size);
}
```

223d ⟨*local declarations* 211c⟩+≡
```
dclproto(static void newdef,(DefUse *, int, MTable, int, int));
```

223e ⟨*definitions* 212a⟩+≡
```
static void newdef(du, sidx, ivmap, vidx, unambig) DefUse *du; MTable ivmap; {
    DefUse *vdu, tmp;

    tmp = idxtable(DefUse *, ivmap, vidx);
    vdu = &(tableentry(DefUse *, ivmap, vidx));
    if (unambig) {
        insert(&(du->udefs), vidx);
        insert(&(vdu->udefs), sidx);
    }
    else {
        insert(&(du->adefs), vidx);
        insert(&(vdu->adefs), sidx);
    }
}
```

223f ⟨*local declarations* 211c⟩+≡
```
dclproto(static void print_defuses,(int, DefUse *, DefUseInfo *, int, int, MFunction *));
#define DEFS 1
#define USES 0
#define UNAMBIG 1
#define AMBIG   0
```

224a ⟨*definitions* 212a⟩+≡

```
static void print_defuses(idx, du, duinfo, def, unambig, mfn)
  DefUse *du; DefUseInfo *duinfo; MFunction *mfn; {
    int id;
    Symbol s;
    Set duset;
    char   *tab = "\t";
    DefUse *vdu;

    gentable(duinfo->ivmap, id, 1) {
        vdu = &(tableentry(DefUse *, duinfo->ivmap, id));
        if (def) {
            if (unambig) duset = &(du->udefs);
            else duset = &(du->adefs);
        } else {
            if (unambig) duset = &(du->uuses);
            else duset = &(du->auses);
        }
        if (member(duset, id)) {
            s = ivsym(id);
            if (def) {
                if (unambig) {
                    ⟨print var's udef 224b⟩
                    assert(member(&vdu->udefs, idx));
                } else {
                    ⟨print var's adef 224c⟩
                    assert(member(&vdu->adefs, idx));
                }
            } else {
                if (unambig) {
                    ⟨print var's uuse 224d⟩
                    assert(member(&vdu->uuses, idx));
                } else {
                    ⟨print var's ause 225a⟩
                    assert(member(&vdu->auses, idx));
                }
            }
        }
    }
}
```

224b ⟨*print var's* `udef` 224b⟩≡

```
{ if (optdbgflag) fprint(2, "%sUDef %s ", tab, s->name);
  ⟨print var's type 225b⟩ }
```

This code is used in chunks 218b and 224a.

224c ⟨*print var's* `adef` 224c⟩≡

```
{ if (optdbgflag) fprint(2, "%sADef %s ", tab, s->name);
  ⟨print var's type 225b⟩ }
```

This code is used in chunks 219c and 224a.

224d ⟨*print var's* `uuse` 224d⟩≡

```
{ if (optdbgflag) fprint(2, "%sUUse %s ", tab, s->name);
  ⟨print var's type 225b⟩ }
```

This code is used in chunks 222a and 224a.

225a ⟨*print var's* `ause` 225a⟩≡
```
{ if (optdbgflag) fprint(2, "%sAUse %s ", tab, s->name);
  ⟨print var's type 225b⟩ }
```
This code is used in chunk 224a.

225b ⟨*print var's type* 225b⟩≡
```
{
    if (optdbgflag) {
        fprint(2, "%s", tab);
        if (s->type) printtype(s->type, 2);
        else fprint(2,"\n");
    }
}
```
This code is used in chunks 224 and 225a.

10.3 Computing `gen` and `kill` sets

The `gen` and `kill` sets are computed for each flow graph node. They represent the set of definitions that are generated and killed in each block.

225c ⟨*external declarations* 211a⟩+≡
```
typedef struct genkillinfo {
    int size;
    MTable gen, kill;
} GenKillInfo;
dclproto(GenKillInfo genkill_analysis,(MFunction *, DefUseInfo *, Arena *));
```

226a ⟨*definitions* 212a⟩+≡
```
GenKillInfo genkill_analysis(mfn, duinfo, arena)
    MFunction *mfn; DefUseInfo *duinfo; Arena *arena; {
    int idx, didx;
    FGNode fgnode;
    Set genB, killB;
    GenKillInfo gkinfo;
    int gksize;

    if (analdbgflag) fprint(2,"gen/kill...\n");
    ⟨compute gksize, size of gen/kill sets 226b⟩
    newtable(&gkinfo.gen, tablesize(mfn->flowgraph), SETSZ, arena);
    newtable(&gkinfo.kill, tablesize(mfn->flowgraph), SETSZ, arena);
    gentable(mfn->flowgraph, idx, 1) {
        fgnode = tableentry(FGNode *, mfn->flowgraph, idx);
        genB = &tableentry(Set, gkinfo.gen, idx);
        killB = &tableentry(Set, gkinfo.kill, idx);
        ⟨initialize gen and kill sets 226c⟩
        gentable(fgnode->dagroots, didx, 0) {
            int sidx, isid;
            DefUse *sdu, *vdu;
            ⟨for each root in fgnode 233a⟩ {
                ⟨get def/use info for root 226d⟩
                ⟨compute sdefs, set of all (root,v) 227b⟩
                ⟨compute vdefs, set of all (s,v) 227c⟩
                ⟨gen = gen - ( (s,v) | root is an unambiguous def of v ) 227d⟩
                ⟨gen = gen U ( root | root is a definition of v ) 227e⟩
                ⟨kill = kill U ( (s,v) | root is an unambiguous def of v ) 228a⟩
                ⟨kill = kill - ( root | root is a definition of v ) 228b⟩
            }
        }
        ⟨ gen/kill sanity check  228c⟩
    }
    return gkinfo;
}
```

226b ⟨*compute* `gksize`, *size of* `gen/kill` *sets* 226b⟩≡
```
gksize = tablesize(duinfo->ismap) * tablesize(duinfo->ivmap);
gkinfo.size = gksize;
if (optdbgflag)
   fprint(2,"|S| = %d, |V| = %d, |S|*|V| = %d\n",
          tablesize(duinfo->ismap), tablesize(duinfo->ivmap), gksize);
```

This code is used in chunk 226a.

226c ⟨*initialize* `gen` *and* `kill` *sets* 226c⟩≡
```
initset(genB, gksize);
initset(killB, gksize);
```

This code is used in chunk 226a.

226d ⟨*get def/use info for* `root` 226d⟩≡
```
sdu = &tableentry(DefUse *, duinfo->ismap, isid);
```

This code is used in chunks 226a and 232c.

svmapid maps a statement, variable pair into a unique index, which is size(**ivmap**) * statement index + variable index. **sunmapid** and **vunmapid** compute the inverse mappings.

227a ⟨*local declarations* 211c⟩+≡

```
#define svmapid(sidx, vidx) ((tablesize(duinfo->ivmap) * sidx) + vidx)
#define sunmapid(i)         (i/tablesize(duinfo->ivmap))
#define vunmapid(i)         (i%tablesize(duinfo->ivmap))
```

sdefs of **root** is the set of all variables defined by **root**.

227b ⟨*compute* **sdefs**, *set of all* **(root,v)** 227b⟩≡

```
{
    int i;
    inittempset(&sdu->svdefs, gkinfo.size);
    gentable(duinfo->ivmap, i, 1) {
        if (member(&sdu->udefs, i) || member(&sdu->adefs, i))
            tinsert(&sdu->svdefs, svmapid(isid, i));
    }
    installset(&sdu->svdefs);
    if (optdbgflag) {
        fprint(2,"\nSDEFS\n");
        print_svset(&sdu->svdefs, gkinfo.size, duinfo, mfn);
    }
}
```

This code is used in chunk 226a.

vdefs of **v** is the set of all statements **s** that define **v**.

227c ⟨*compute* **vdefs**, *set of all* **(s,v)** 227c⟩≡

```
if (sdu->ivid != 0) {
    int i;
    DefUse t = idxtable(DefUse *, duinfo->ivmap, sdu->ivid);

    vdu = &tableentry(DefUse *, duinfo->ivmap, sdu->ivid);
    inittempset(&vdu->svdefs, gkinfo.size);
    gentable(duinfo->ismap, i, 1) {
        if (member(&vdu->adefs, i) || member(&vdu->udefs, i))
            tinsert(&vdu->svdefs, svmapid(i, sdu->ivid));
    }
    installset(&vdu->svdefs);
    if (optdbgflag) {
        fprint(2,"VDEFS\n");
        print_svset(&vdu->svdefs, gkinfo.size, duinfo, mfn);
    }
}
```

This code is used in chunk 226a.

227d ⟨*gen = gen -* (*(s,v)* | *root is an unambiguous def of v*) 227d⟩≡

```
if (sdu->ivid != 0)
    diffset(genB, &vdu->svdefs);
```

This code is used in chunk 226a.

227e ⟨*gen = gen U* (*root* | *root is a definition of v*) 227e⟩≡

```
uset(genB, &sdu->svdefs);
```

This code is used in chunk 226a.

228a ⟨*kill = kill U ((s,v) | root is an unambiguous def of v)* 228a⟩≡
```
if (sdu->ivid != 0)
    uset(killB, &vdu->svdefs);
```

This code is used in chunk 226a.

228b ⟨*kill = kill - (root | root is a definition of v)* 228b⟩≡
```
diffset(killB, &sdu->svdefs);
```

This code is used in chunk 226a.

228c ⟨ `gen/kill` *sanity check* 228c⟩≡
```
{
    MTable    stack;
    int j;

    if (optdbgflag) {
        newtable(&stack, 10, sizeof(int), arena);
        emitFlowgraphnode(2, fgnode, mfn->dagtable, &stack, "");
        for (j = 0; j < 2; j++) {
            fprint(2,"%s\n", j == 0 ? "GEN" : "KILL");
            print_svset((j == 0 ? genB : killB), gksize, duinfo, mfn);
        }
    }
}
```
This code is used in chunk 226a.

228d ⟨*local declarations* 211c⟩+≡
```
dclproto(static void print_svset,(Set, int, DefUseInfo *, MFunction *));
```

228e ⟨*definitions* 212a⟩+≡
```
static void print_svset(s, gksize, duinfo, mfn)
  Set s; DefUseInfo *duinfo; MFunction *mfn; {
    char *tab = "";
    DefUse *du;
    int i, idx, vid;
    Node p;

    for (i = 1; i <= gksize; i++) {
        if (member(s, i)) {
            idx = sunmapid(i);
            vid = vunmapid(i);
            du = &tableentry(DefUse *, duinfo->ismap, idx);
            assert(p = idxtable(Node *, mfn->nodetable, du->id));
            ⟨print statement header 216a⟩
            fprint(2,"%s\n", symfor(tableentry(DefUse *, duinfo->ivmap, vid).id)->name);
        }
    }
}
```

10.4 Computing the fixed point

The fixed point computation is the same for all forward data flow analysis problems. `fwd_df_analysis` computes the fixed point over the `gen` and `kill` sets in `gkinfo` for the function `mfn`.

229a ⟨*external declarations* 211a⟩+≡

```
typedef struct inoutinfo {
    MTable in, out;
} InOutInfo;
dclproto(InOutInfo fwd_df_analysis,(MFunction *, DefUseInfo *, GenKillInfo *, Arena *));
```

229b ⟨*definitions* 212a⟩+≡

```
InOutInfo fwd_df_analysis(mfn, duinfo, gkinfo, arena)
  MFunction *mfn; DefUseInfo *duinfo; GenKillInfo *gkinfo; Arena *arena; {
    FGNode    fgnode;
    int       changed, fgidx, loopct = 0;
    Set       inB, outB;
    InOutInfo inoutinfo;
    struct set save, temp;

    if (analdbgflag) fprint(2,"in/out...\n");
    newtable(&inoutinfo.in, tablesize(mfn->flowgraph), SETSZ, arena);
    newtable(&inoutinfo.out, tablesize(mfn->flowgraph), SETSZ, arena);
    initset(&save, gkinfo->size);
    initset(&temp, gkinfo->size);
    gentable(mfn->flowgraph, fgidx, 1) {
        ⟨initialize in and out sets 229c⟩
    }
    changed = 1;
    while (changed) {
        changed = 0;
        gentable(mfn->flowgraph, fgidx, 1) {
            fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
            inB = &tableentry(Set, inoutinfo.in, fgidx);
            outB = &tableentry(Set, inoutinfo.out, fgidx);
            ⟨initialize inB to out of first predecessor 230a⟩
            ⟨inB = ( U (p ∈ pred(fgnode)) out(p) ) 230b⟩
            ⟨save current outB 230c⟩
            ⟨outB = genB U (inB - killB) 230d⟩
            ⟨check if outB changed 230e⟩
        }
        loopct++; if (optdbgflag) fprint(2,"."); if (loopct > 10) optdbgflag = 1;
    }
    if (optdbgflag)fprint(2,"loopct %d\n", loopct);
    ⟨in/out sanity check 231a⟩
    return inoutinfo;
}
```

229c ⟨*initialize* `in` *and* `out` *sets* 229c⟩≡

```
initset(&(tableentry(Set, inoutinfo.in, fgidx)), gkinfo->size);
initset(&(tableentry(Set, inoutinfo.out, fgidx)), gkinfo->size);
```

This code is used in chunk 229b.

230a ⟨*initialize* `inB` *to* `out` *of first predecessor* 230a⟩≡
```
if (tablesize(fgnode->preds) != 0) {
     int predid;
     predid = fgnode->id + tableentry(short *, fgnode->preds, 0);
     copyset(inB, &tableentry(Set, inoutinfo.out, predid));
}
```
This code is used in chunk 229b.

230b ⟨`inB` = (*U* (`p` ∈ `pred(fgnode)`) `out(p)`) 230b⟩≡
```
{
     int predid, i;
     gentable(fgnode->preds, i, 1) {
          predid = fgnode->id + tableentry(short *, fgnode->preds, i);
          uset(inB, &(tableentry(Set, inoutinfo.out, predid)));
     }
}
```
This code is used in chunk 229b.

230c ⟨*save current* `outB` 230c⟩≡
```
copyset(&save, outB);
```
This code is used in chunk 229b.

230d ⟨`outB` = `genB` *U* (`inB` - `killB`) 230d⟩≡
```
{
     copyset(&temp, inB);
     diffset(&temp, &tableentry(Set, gkinfo->kill, fgidx));
     uset(&temp, &tableentry(Set, gkinfo->gen, fgidx));
     copyset(outB, &temp);
}
```
This code is used in chunk 229b.

230e ⟨*check if* `outB` *changed* 230e⟩≡
```
{ int chtmp = !eqsets(outB, &save);
     changed |= chtmp;
     if (chtmp && optdbgflag) {
          fprint(2,"outB of %d changed %d %d %d\n",
                    fgidx, save.charsize, outB->charsize, gkinfo->size);
          fprint(2,"SAV\n");
          print_svset(&save, gkinfo->size, duinfo, mfn);
          fprint(2,"OUT\n");
          print_svset(outB, gkinfo->size, duinfo, mfn);
     }
}
```

This code is used in chunk 229b.

231a ⟨in/out *sanity check* 231a⟩≡
```
{
    MTable stack;
    int fgidx;

    if (optdbgflag)
        gentable(mfn->flowgraph, fgidx, 1) {
            newtable(&stack, 10, sizeof(int), arena);
            fprint(2,"IN\n");
            print_svset(&(tableentry(Set, inoutinfo.in, fgidx)),
                          gkinfo->size, duinfo, mfn);
            emitFlowgraphnode(2, tableentry(FGNode *, mfn->flowgraph, fgidx),
                              mfn->dagtable, &stack, "");
            fprint(2,"OUT\n");
            print_svset(&(tableentry(Set, inoutinfo.out, fgidx)),
                        gkinfo->size, duinfo, mfn);
        }
}
```
This code is used in chunk 229b.

10.5 Ud-chains and du-chains

Ud-chains and du-chains tend to be sparse so we use lists instead of sets to represent them. The chains are indexed by variable. For the ud chains, each element is a list of `Link`s. Each link contains a list of statement identifiers that represent the definitions which reach the given use. The du chains are constructed similarly although their elements contain the list of uses reached by the given definition.

231b ⟨*external declarations* 211a⟩+≡
```
typedef struct sidlink {
    int sid;
    struct sidlink *next;
}*Sidlink;
typedef struct link {
    int sid;
    Sidlink defs_or_uses;
    struct link *next;
}*Link;
typedef struct chains {
    MTable ud_chain, du_chain;
}Chains;
```

231c ⟨*external declarations* 211a⟩+≡
```
dclproto(Sidlink newsidlink,(int));
dclproto(Link newlink,(int));
```

232a ⟨*definitions* 212a⟩+≡
```
Sidlink newsidlink(sid) {
    Sidlink slink = (Sidlink)talloc(sizeof(*slink));
    slink->sid = sid;
    slink->next = 0;
    return slink;
}
Link newlink(isid) {
    Link link = (Link)talloc(sizeof(*link));
    link->sid = isid;
    link->defs_or_uses = 0;
    link->next = 0;
    return link;
}
```

232b ⟨*external declarations* 211a⟩+≡
```
dclproto(Chains ud_du_chains,(MFunction *, DefUseInfo *,
                              GenKillInfo *, InOutInfo *, Arena *));
```

232c ⟨*definitions* 212a⟩+≡
```
Chains ud_du_chains(mfn, duinfo, gkinfo, inoutinfo, arena)
  MFunction *mfn; DefUseInfo *duinfo; GenKillInfo *gkinfo;
  InOutInfo *inoutinfo; Arena *arena; {
    FGNode    fgnode;
    int       changed, fgidx, didx;
    Chains    uddu;
    struct set D;

    if (analdbgflag) fprint(2,"ud/du...\n");
    newtable(&uddu.ud_chain, tablesize(duinfo->ivmap), sizeof(Link), arena);
    newtable(&uddu.du_chain, tablesize(duinfo->ivmap), sizeof(Link), arena);
    gentable(mfn->flowgraph, fgidx, 1) {
        D = tableentry(Set, inoutinfo->in, fgidx);
        fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
        gentable(fgnode->dagroots, didx, 0) {
            int sidx, isid;
            ⟨for each root in fgnode 233a⟩ {
                DefUse *sdu, vdu;
                int ivid;

                ⟨get def/use info for root 226d⟩
                gentable(duinfo->ivmap, ivid, 1) {
                    if (member(&(sdu->uuses), ivid) || member(&(sdu->auses), ivid)) {
                        ⟨ud(v,root) = ( (s,v) | (s,v) ∈ D ) 233e⟩
                        ⟨D = D - ( (s,v) | root is an unambiguous def of v ) 233b⟩
                     }
                }
               ⟨D = D U ( (root, v) | root is a definition of v) 233c⟩
            }
        }
    }
    ⟨ud/du-chain sanity check 234c⟩
    return uddu;
}
```

233a ⟨*for each* `root` *in* `fgnode` 233a⟩≡
```
  sidx = tableentry(int *, fgnode->dagroots, didx);
  if (sidx != 0 && (isid = tableentry(int *, duinfo->smap, sidx)) != 0)
```
This code is used in chunks 226a and 232c.

233b ⟨`D` = `D - ( (s,v)` | *root is an unambiguous def of v* `)` 233b⟩≡
```
  if (sdu->ivid != 0) {
      vdu = idxtable(DefUse *, duinfo->ivmap, sdu->ivid);
      diffset(&D, &vdu.svdefs);
  }
```
This code is used in chunk 232c.

233c ⟨`D` = `D` *U* `( (root, v)` | *root is a definition of v*`)` 233c⟩≡
```
  uset(&D, &sdu->svdefs);
```
This code is used in chunk 232c.

233d ⟨*create* `ud(v, root)` 233d⟩≡
```
  link = newlink(isid);
```
This code is used in chunk 233e.

233e ⟨`ud(v,root)` = `( (s,v)` | `(s,v)` ∈ *D* `)` 233e⟩≡
```
  {
      int sid;
      Link link;

      ⟨create ud(v, root) 233d⟩
      gentable(duinfo->ismap, sid, 1) {
          if (member(&D, svmapid(sid, ivid))) {
              Sidlink slink;
              Symbol s;

              slink = newsidlink(sid);
              s = symfor(tableentry(DefUse *, duinfo->ivmap, ivid).id);
              if (optdbgflag)
                  fprint(2,"def %d reaches use %d, %s\n",
                         tableentry(DefUse *, duinfo->ismap, sid).id, sidx, s->name);
              slink->next = link->defs_or_uses;
              link->defs_or_uses = slink;
              ⟨create du(sid,v) chain from sid to isid 233f⟩
          }
      }
      link->next = tableentry(Link *, uddu.ud_chain, ivid);
      updatetable(Link *, &(uddu.ud_chain), ivid, link);
  }
```
This code is used in chunk 232c.

233f ⟨*create* `du(sid,v)` *chain from* `sid` *to* `isid` 233f⟩≡
```
  {
      Link link = tableentry(Link *, uddu.du_chain, ivid);
      ⟨find sid in v's du chain 234a⟩
      ⟨insert isid in du(sid, v) 234b⟩
  }
```
This code is used in chunk 233e.

234a ⟨*find* sid *in* v's *du chain* 234a⟩≡

```
{
    for (; link; link = link->next)
        if (link->sid == sid) break;
    if (link == 0) {
        link = newlink(sid);
        link->next = tableentry(Link *, uddu.du_chain, ivid);
        updatetable(Link *, &(uddu.du_chain), ivid, link);
    }
}
```

This code is used in chunk 233f.

234b ⟨*insert* isid *in* du(sid, v) 234b⟩≡

```
{
    Sidlink slink;

    slink = newsidlink(isid);
    slink->next = link->defs_or_uses;
    link->defs_or_uses = slink;
}
```

This code is used in chunk 233f.

234c ⟨ud/du-chain *sanity check* 234c⟩≡

```
{
    int idx;
    Link l;
    Sidlink s;

    if (optdbgflag) {
       gentable(uddu.ud_chain, idx, 1) {
        Symbol sym = symfor(tableentry(DefUse *, duinfo->ivmap, idx).id);
        l = tableentry(Link *, uddu.ud_chain, idx);
        for (; l; l = l->next)
            for (s = l->defs_or_uses; s; s = s->next)
                fprint(2,"def %d reaches use %d, %s\n",
                    tableentry(DefUse *, duinfo->ismap, s->sid).id,
                    tableentry(DefUse *, duinfo->ismap, l->sid).id, sym->name);

        }
        fprint(2,"****\n");
        gentable(uddu.du_chain, idx, 1) {
            Symbol sym = symfor(tableentry(DefUse *, duinfo->ivmap, idx).id);
            l = tableentry(Link *, uddu.du_chain, idx);
            for (; l; l = l->next)
                for (s = l->defs_or_uses; s; s = s->next)
                    fprint(2,"def %d reaches use %d, %s\n",
                        tableentry(DefUse *, duinfo->ismap, l->sid).id,
                        tableentry(DefUse *, duinfo->ismap, s->sid).id, sym->name);

        }
    }
}
```

This code is used in chunk 232c.

10.6 Reaching definitions

`compute_reaching_defs` computes all the information necessary for deriving `ud-` and `du-`chains. Before computing the analysis info, a context is prepended to the function. All `MFunction`s have an implicit context: if `mfn` is a top-level procedure, its context contains the values of its parameters; if `mfn` is an inlined procedure, its context contains the values of the locals that were originally parameters to the inlined callee.

235a ⟨*external declarations* 211a⟩+≡
```
dclproto(int compute_reaching_defs,(MFunction *, SymbolFn, Arena *));
```

235b ⟨*definitions* 212a⟩+≡
```
int compute_reaching_defs(mfn, aliasfn, arena)
    MFunction *mfn; SymbolFn aliasfn; Arena *arena; {
    OptInfo *optinfo;
    MTable context;
    unsigned short calleeid;
    Symbol callee;
    int i;
    Node p;

    newtable(&context, tablesize(mfn->callee), sizeof(Node), arena);
    gentable(mfn->callee, i, 0) {
        calleeid = idxtable(unsigned short *, mfn->callee, i);
        assert(callee = idxtable(Symbol *, mfn->localsymtab, calleeid));
        p = newnode(ADDRFP, NULL, NULL, callee);
        if (callee->scope >= LOCAL) p->op = ADDRLP;
        p = newnode(INDIR+(isunsigned(callee->type) ? I :
                    ttob(callee->type)), p, NULL, NULL);
        updatetable(Node *, &context, i, p);
    }
    if (mfn->optinfo == 0) {
        optinfo = newoptinfo(arena);
        ⟨prepend context to mfn ??⟩
        optinfo->duinfo = defuse_analysis(mfn, aliasfn, arena);
        if (optstatflag || emptytable(optinfo->duinfo.smap)) return 0;
        optinfo->gkinfo = genkill_analysis(mfn, &optinfo->duinfo, arena);
        optinfo->inoutinfo = fwd_df_analysis(mfn, &optinfo->duinfo,
                                 &optinfo->gkinfo, arena);
        optinfo->uddu = ud_du_chains(mfn, &optinfo->duinfo, &optinfo->gkinfo,
                                 &optinfo->inoutinfo, arena);
        mfn->optinfo = optinfo;
    }
    return 1;
}
```

10.7 Dominators

We use the algorithm on pg. 671 of the Dragon book (Aho, Sethi, and Ullman 1986) for computing the set of dominators of each node in the flow graph. Dominators are used when unreachable nodes are identified in the graph; any node dominated by an unreachable node is also unreachable. The algorithm is implemented almost verbatim from the pseudo-code provided in the text.

235c ⟨*local declarations* 211c⟩+≡
```
dclproto(static void dominators,(MFunction *));
```

236a ⟨*definitions* 212a⟩+≡
```
static void dominators(mfn) MFunction *mfn; {
    int i, N, fgidx, changed;
    FGNode fgnode;
    struct set allN, save;

    N = tablesize(mfn->flowgraph);
    initset(&save, N);
    ⟨initialize universe allN 236b⟩
    ⟨initialize dominator sets 236c⟩
    changed = 1;
    while (changed) {
        changed = 0;
        gentable(mfn->flowgraph, fgidx, 2) {
            fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
            ⟨patch predecessors 237a⟩
            ⟨dom(n) = n ⋃(⋂(p ∈ pred(n)) dom(p) ) 236d⟩
        }
    }
    ⟨dominator sanity check 237b⟩
}
```

236b ⟨*initialize universe* `allN` 236b⟩≡
```
inittempset(&allN, N);
for (i = 1; i < N; i++) tinsert(&allN, i);
installset(&allN);
```
This code is used in chunk 236a.

The root of the flowgraph dominates itself. All other dominator sets are conservatively initialized to the universe.

236c ⟨*initialize dominator sets* 236c⟩≡
```
gentable(mfn->flowgraph, fgidx, 1) {
    fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
    initset(&fgnode->dominators, N);
    if (fgidx == 1) insert(&fgnode->dominators, 1);
    else copyset(&fgnode->dominators, &allN);
}
```
This code is used in chunk 236a.

236d ⟨`dom(n)` $= n \bigcup(\bigcap($`p` $\in$ `pred(n)`$)$ `dom(p)` $)$ 236d⟩≡
```
{
    int predid, i;
    FGNode pred;

    copyset(&save, &fgnode->dominators);
    gentable(fgnode->preds, i, 0) {
        predid = fgnode->id + tableentry(short *, fgnode->preds, i);
        pred = idxtable(FGNode *, mfn->flowgraph, predid);
        xset(&fgnode->dominators, &pred->dominators);
    }
    insert(&fgnode->dominators, fgidx);
    changed |= (!eqsets(&fgnode->dominators, &save));
}
```
This code is used in chunk 236a.

237a ⟨*patch* `predecessors` 237a⟩≡
```
if (emptytable(fgnode->preds) && fgnode->type == JumpN &&
    tableentry(FGNode *, mfn->flowgraph, fgidx - 1)->type == RetN) {
    newtable(&fgnode->preds, 1, sizeof(short), mfn->flowgraph.arena);
    addtotable(short *, &fgnode->preds, -1);
}
```
This code is used in chunk 236a.

237b ⟨*dominator sanity check* 237b⟩≡
```
if (optdbgflag) {
    gentable(mfn->flowgraph, fgidx, 1) {
        fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
        fprint(2, "Node %d dominated by ", fgnode->id);
        for (i = 1; i < N; i++)
            if (member(&fgnode->dominators, i)) fprint(2," %d", i);
        fprint(2,"\n");
    }
}
```
This code is used in chunk 236a.

11 Constant propagation

`constant_propagation` implements simple constant propagation. First the function is analyzed and then the transformation is applied. We use a simple function to determine if a variable is in the alias set: if it is addressed, it is considered a potential alias.

237c ⟨*local declarations* 211c⟩+≡
```
dclproto(static int isaddressed,(Symbol));
```

237d ⟨*definitions* 212a⟩+≡
```
static int isaddressed(p) Symbol p; {
    return p->addressed;
}
```

237e ⟨*external declarations* 211a⟩+≡
```
dclproto(void constant_propagation,(MFunction *, Arena *));
```

237f ⟨*local declarations* 211c⟩+≡
```
#define OPTLIMIT SHRT_MAX
```

238a ⟨*definitions* 212a⟩+≡
```
void constant_propagation(mfn, arena) MFunction *mfn; Arena *arena; {
    int fgidx;
    FGNode fgnode;
    if (analdbgflag) fprint(2,"%s > >\n", mfn->sym->name);
    if (compute_reaching_defs(mfn, isaddressed, arena)) {
        dominators(mfn);
        constant_propagation_xform(mfn);
        gentable(mfn->flowgraph, fgidx, 1) {
            fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
            if (fgnode->type == CallN && callN(fgnode).mfn) {
                MFunction *savmfn;
                MTable context;

                if (optdbgflag) fprint(2, "***>");
                ⟨setup context for inlined callee 239a⟩
                savmfn = mfn;
                mfn = callN(fgnode).mfn;
                ⟨prepend context to mfn ??⟩
                constant_propagation(mfn, arena);
                mfn = savmfn;
            }
        }
        ⟨free all sets 243b⟩
        ⟨eliminate unnecessary jumps 240a⟩
    }
    ⟨delete context 238b⟩
    if (analdbgflag) fprint(2,"< < %s\n", mfn->sym->name);
}
```

Any context provided is only used for computing analysis information, i.e., the calling context of a clone or the initial values of parameters are not required in the generated code and can be omitted.

238b ⟨*delete context* 238b⟩≡
```
{
    FGNode fgnode;
    fgnode = tableentry(FGNode *, mfn->flowgraph, 1);
    cleartable(fgnode->dagroots, arena);
}
```
This code is used in chunk 238a.

The inlining transformation doesn't leave enough info behind to easily find the values of the arguments to an inlined procedure. We have to go through some maneuvers to find the arguments' values.

239a ⟨*setup context for inlined* `callee` 239a⟩≡
```
{
     int calleeidx;
     MFunction *calleemfn  = callN(fgnode).mfn;
     Symbol callee;

     newtable(&context, tablesize(calleemfn->callee), sizeof(Node), arena);
     gentable(calleemfn->callee, calleeidx, 0) {
         unsigned short calleeid;
         calleeid = idxtable(unsigned short *, calleemfn->callee, calleeidx);
         assert(callee = idxtable(Symbol *, calleemfn->localsymtab, calleeid));
         ⟨find callee's context 239b⟩
     }
}
```
This code is used in chunk 238a.

This is gross! For each (old) argument, a pass is made over the `Node`s that contain the calling context of the inlined procedure. On each pass, we look for assignments to the old argument and use the assigned value as its calling context.

239b ⟨*find* `callee`*'s context* 239b⟩≡
```
{
     int nodect, rootid, i, formalid;
     FGNode argn;
     Node p;

     for (nodect = callN(fgnode).argnid; nodect <= fgnode->id; nodect++) {
         assert(argn = idxtable(FGNode *, mfn->flowgraph, nodect));
         gentable(argn->dagroots, i, 0) {
             rootid = tableentry(int *, argn->dagroots, i);
             if (rootid && (p = idxtable(Node *, mfn->nodetable, rootid))
              && generic(p->op) == ASGN && isaddrop(p->kids[0]->op)
              && p->kids[0]->syms[0] == callee && isconstsym(p->kids[1])) {
                 p = copynode(0, p->kids[1]);
                 if (optdbgflag) {
                     fprint(2,"***>Calling context of %s is:\n", callee->name);
                     printdag(p,2);
                 }
                 if (IR->left_to_right) formalid = calleeidx;
                 else formalid = tablesize(mfn->callee) - calleeidx - 1;
                 updatetable(Node *, &context, formalid, p);
             }
         }
     }
}
```
This code is used in chunk 239a.

Constant propagation and unreachable-code elimination can result in unnecessary jumps. A post-optimization walk over the flowgraph eliminates these jumps.

240a ⟨*eliminate unnecessary jumps* 240a⟩≡
```
gentable(mfn->flowgraph, fgidx, 1) {
    fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
    if (fgnode->type == JumpN && !emptytable(fgnode->succs)) {
        short succ = idxtable(short *, fgnode->succs, 0) + fgnode->id;
        int i, emptypath = 1;
        if (succ <= fgidx) emptypath = 0;
        for (i = fgidx+1; i < succ; i++) {
            FGNode succn;
            succn = tableentry(FGNode *, mfn->flowgraph, i);
            if (!emptytable(succn->dagroots) ||
                (succn->type == CallN && callN(succn).mfn != 0)) {
                emptypath = 0;
                break;
            }
        }
        ⟨eliminate jump 240b⟩
    }
}
```
This code is used in chunk 238a.

This is a hack. In a `JumpN` node, the last dag in the dagroot table is always a `JUMPV`. Decrementing the size of the flow graph node's dagroot table essentially eliminates the jump.

240b ⟨*eliminate jump* 240b⟩≡
```
if (emptypath) {
    if (optdbgflag) fprint(2,"Eliminate jump at node %d\n", fgnode->id);
    fgnode->dagroots.maxelem -= 1;
}
fgnode->type = StmtsN;
```
This code is used in chunk 240a.

The algorithm for constant propagation is derived from the description in the Dragon book and from Hanson's class notes. Since the dataflow analysis information can be used both by type analysis and constant propagation, we bundle it together into one data structure.

240c ⟨*external declarations* 211a⟩+≡
```
typedef struct optinfo {
    DefUseInfo  duinfo;
    GenKillInfo gkinfo;
    InOutInfo   inoutinfo;
    Chains      uddu;
} OptInfo;
```
240d ⟨*external declarations* 211a⟩+≡
```
dclproto(OptInfo *newoptinfo,(Arena *));
```
240e ⟨*definitions* 212a⟩+≡
```
OptInfo *newoptinfo(arena) Arena *arena;
{
    OptInfo *optinfo = (OptInfo *)yyalloc(sizeof(OptInfo), (*arena));
    bzero((char *)optinfo, sizeof(OptInfo));
    return optinfo;
}
```

We are interested in propagating constant expressions derived from specialization, the source text, and constant-valued pointers provided in a binding file. Assignments of constants and constant-valued pointers to variables are placed on a worklist.

241a ⟨*local declarations* 211c⟩+≡
```
static void constant_propagation_xform(MFunction *);
```

241b ⟨*definitions* 212a⟩+≡
```
static void constant_propagation_xform(mfn)
    MFunction *mfn; {
    Sidlink Q, Qtail, s;
    int (*constfn)(Node);
    DefUseInfo *duinfo;
    Chains *uddu;

    if (analdbgflag) fprint(2,"constant propagation...\n");
    constfn = const_expr;
    assert(mfn->optinfo != 0);
    duinfo = &mfn->optinfo->duinfo;
    uddu = &mfn->optinfo->uddu;
    ⟨append all s: x = c in fn to Q 247c⟩
    ⟨Q sanity check 247b⟩
    while (Q != 0) {
        int ivid, conststmt;
        Link du = 0;
        Sidlink use;
        Symbol x, c;
        Node p;

        ⟨s = pop(Q) 246c⟩
        ⟨find du(s,x) 247a⟩
        ⟨for each use(x,s') ∈ du(s,x) 243c⟩ {
            if (optdbgflag)
                fprint(2, "use of %s at %d\n", x->name,
                       tableentry(DefUse *, duinfo->ismap, use->sid).id);
            if (⟨use(x,s') is unambiguous 243d⟩ && (x->type && isscalar(x->type))) {
                Link ud = 0;
                DataValue dv;

                ⟨find ud(x,s') 243e⟩
                if (isconstasgn(p)) ⟨propagate simple constant 242a⟩
                else if (dv = isconstptr(p)) ⟨propagate constant-valued pointer 242b⟩
                else assert(0);
            }
        }
        if (x->sclass != REGISTER)
            ⟨eliminate unused statement s 245c⟩
    }
    if (analdbgflag) fprint(2,"...constant propagation\n");
}
```

242a ⟨*propagate simple constant* 242a⟩≡
```
{
     Node constq;

     c = constsym(p->kids[1]);
     conststmt = 1;
     ⟨conststmt = true if ∀ def(x,s'') ∈ ud(x,s'), s'': x = c 244c⟩
     if (conststmt) {
          fprint(2,"propagate %s at %d to %s at %d\n",
                    c->name, tableentry(DefUse *, duinfo->ismap, s->sid).id,
                    x->name, tableentry(DefUse *, duinfo->ismap, use->sid).id);
          ⟨change s':u = w o x to s': u = w o c 246a⟩
          ⟨du(s'',x) = du(s'',x) - ( use(x,s') ) 245a⟩
          ⟨ud(x,s') = ∅ 245b⟩
     }
}
```
This code is used in chunk 241b.

242b ⟨*propagate constant-valued pointer* 242b⟩≡
```
{
     int defcount;
     ⟨count defs in ud(x,s') 244b⟩
     if (optdbgflag) fprint(2,"Found dataval use of %s %d\n", x->name, defcount);
     if (defcount == 1) {
          ⟨associate p's dataval with use 243a⟩
     }
}
```
This code is used in chunk 241b.

If p has a link-time binding value, associate its binding with its use.

243a ⟨*associate* p*'s dataval with use* 243a⟩≡
```
{
    Node savp, p;
    Args args;
    int sid;

    sid = tableentry(DefUse *, duinfo->ismap, use->sid).id;
    savp = p = idxtable(Node *, mfn->nodetable, sid);
    args.datasym.s = x;
    args.datasym.d = dv;
    p = subst(p, p, use_of_ptrx, identitynode, &args);
    p = dag_simplify(p, (Type)0);
    if (savp != p) {
        if (condop(generic(savp->op))) {
            FGNode fgnode;
            ⟨find p's flowgraph node 246b⟩
            p = constcondexpr(mfn, fgnode, savp, p);
        }
        updatetable(Node *, &(mfn->nodetable), sid, p);
    }
    if (⟨p is s: x = c 247d⟩) {
        int sid = use->sid;
        if (optdbgflag)
            fprint(2,"new s: x = c at %d\n",
                   tableentry(DefUse *, duinfo->ismap, use->sid).id);
        ⟨append s: x = c to Q 248a⟩
        if (optdbgflag) printdag(p, 2);
    }
 }
```
This code is used in chunk 242b.

243b ⟨*free all sets* 243b⟩≡
```
freesets();
```
This code is used in chunk 238a.

243c ⟨*for each* use(x,s') ∈ du(s,x) 243c⟩≡
```
if (du)
    for (use = du->defs_or_uses; use; use = use->next)
```
This code is used in chunk 241b.

243d ⟨use(x,s') *is unambiguous* 243d⟩≡
```
member(&(tableentry(DefUse *, duinfo->ivmap, ivid).uuses), use->sid)
```
This code is used in chunk 241b.

243e ⟨*find* ud(x,s') 243e⟩≡
```
for (ud = tableentry(Link *, uddu->ud_chain, ivid); ud ; ud = ud->next)
    if (ud->sid == use->sid) break;
assert(ud);
```
This code is used in chunk 241b.

244a ⟨*count defs in* du(s',x) 244a⟩≡
```
{
     Link ud = du;
     ⟨count defs in ud(x,s') 244b⟩
}
```

This code is written to file count defs in [[du(s',x)]].

244b ⟨*count defs in* ud(x,s') 244b⟩≡
```
{
     Sidlink def;
     for (defcount = 0, def = ud->defs_or_uses; def; def = def->next)
         if (def->sid != 0) defcount++;
}
```

This code is used in chunks 242b and 244a.

244c ⟨conststmt = *true if* ∀ def(x,s'') ∈ ud(x,s'), s'': x = c 244c⟩≡
```
{
     Sidlink def;
     constq = tableentry(Node *, mfn->nodetable,
                         tableentry(DefUse *, duinfo->ismap, s->sid).id)->kids[1];
     for (def = ud->defs_or_uses; def && conststmt; def = def->next) {
         if (def->sid != 0 && def->sid != s->sid) {
             Symbol newc;
             constq = idxtable(Node *, mfn->nodetable,
                               tableentry(DefUse *, duinfo->ismap, def->sid).id);
             if (generic(constq->op) == ASGN) {
                 constq = constq->kids[1];
                 newc = isconstsym(constq);
                 if (optdbgflag)
                     fprint(2, "def %d reaches with value %s\n",
                           tableentry(DefUse *, duinfo->ismap, def->sid).id,
                           newc ? newc->name : "NIL");
                 conststmt = member(&(tableentry(DefUse *, duinfo->ivmap, ivid).udefs), def->sid) &&
                             newc && newc == c;
             } else conststmt = 0;
         }
     }
}
```
This code is used in chunk 242a.

245a ⟨du(s'',x) = du(s'',x) - (use(x,s')) 245a⟩≡
```
{
    Sidlink def, u;
    Link du;
    int usid = use->sid;
    for (def = ud->defs_or_uses; def; def = def->next) {
        for (du = tableentry(Link *, uddu->du_chain, ivid); du ; du = du->next)
            if (du->sid == def->sid) break;
        assert(du);
        for (u = du->defs_or_uses; u; u = u->next)
            if (u->sid == usid) break;
        assert(u);
        u->sid = 0;
    }
}
```

This code is used in chunk 242a.

245b ⟨ud(x,s') = ∅ 245b⟩≡
```
ud->defs_or_uses = (Sidlink) 0;
```

This code is used in chunk 242a.

245c ⟨*eliminate unused statement* s 245c⟩≡
```
{
    Sidlink use;
    int used = 0, sid;
    if (du)
        for (use = du->defs_or_uses; use; use = use->next)
            if (use->sid != 0) used = 1;
    if (!used) {
        sid = tableentry(DefUse *, duinfo->ismap, s->sid).id;
        tableentry(Node *, mfn->nodetable, sid) = 0;
        fprint(2,"Eliminating stmt %d\n", sid);
    }
}
```
This code is used in chunk 241b.

Once a simple constant or constant-valued pointer has been propagated, it can reveal other transformations. `dag_simplify` is called to simplify binding values or constant-valued conditional expressions revealed by the propagation. Any new constant assignments are added to the work list.

245d ⟨*external declarations* 211a⟩+≡
```
dclproto(Node dag_simplify,(Node, Type));
```

246a ⟨*change s':u = w o x to s': u = w o c* 246a⟩≡
```
  {
      Node savp, p;
      Args args;
      int sid;

      sid = tableentry(DefUse *, duinfo->ismap, use->sid).id;
      savp = p = idxtable(Node *, mfn->nodetable, sid);
      args.s = x;
      p = subst(p, constq, use_of, copyconst, &args);
      p = dag_simplify(p, (Type)0);
      if (savp != p) {
          if (condop(generic(savp->op))) {
              FGNode fgnode;
              ⟨find p's flowgraph node 246b⟩
              p = constcondexpr(mfn, fgnode, savp, p);
          }
          updatetable(Node *, &(mfn->nodetable), sid, p);
      }
      if (⟨p is s: x = c 247d⟩) {
          int sid = use->sid;
          if (optdbgflag)
              fprint(2,"new s: x = c at %d\n",
                  tableentry(DefUse *, duinfo->ismap, use->sid).id);
          ⟨append s: x = c to Q 248a⟩
          if (optdbgflag) printdag(p, 2);
      }
   }
```
This code is used in chunk 242a.

246b ⟨*find* p*'s flowgraph node* 246b⟩≡
```
  {
      int fgidx, minleaf, r, maxroot;
      gentable(mfn->flowgraph, fgidx, 1) {
          fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
          if (!isemptytable(fgnode->dagnodes)) {
              dagrange(fgnode->dagnodes, 0, minleaf, r);
              dagrange(fgnode->dagnodes, fgnode->dagnodes.maxelem, r, maxroot);
              if (sid <= maxroot && sid >= minleaf) break;
          }
      }
      assert(fgnode && sid <= maxroot && sid >= minleaf);
  }
```
This code is used in chunks 243a and 246a.

246c ⟨s *= pop(*Q*)* 246c⟩≡
```
  s = Q;
  Q = Q->next;
```
This code is used in chunk 241b.

247a ⟨*find* du(s,x) 247a⟩≡
```
{
    int sid;

    sid = tableentry(DefUse *, duinfo->ismap, s->sid).id;
    assert(p = idxtable(Node *, mfn->nodetable, sid));
    x = p->kids[0]->syms[0];
    ivid = ivmapid(msym(x).oid);
    for (du = tableentry(Link *, uddu->du_chain, ivid); du ; du = du->next)
        if (du->sid == s->sid) break;
}
```
This code is used in chunk 241b.

247b ⟨Q *sanity check* 247b⟩≡
```
{
    Sidlink tmp;
    if (optdbgflag) {
        fprint(2, "S: X = C\n");
        for (tmp = Q; tmp; tmp = tmp->next)
            fprint(2,"%d\n", tableentry(DefUse *, duinfo->ismap, tmp->sid).id);
    }
}
```
This code is used in chunk 241b.

247c ⟨*append all* s: x = c *in* fn *to* Q 247c⟩≡
```
{
    int sid;
    Node p;

    Q = 0;
    gentable(duinfo->ismap, sid, 1) {
        int nid;
        nid = tableentry(DefUse *, duinfo->ismap, sid).id;
        assert(p = idxtable(Node *, mfn->nodetable, nid));
        if (⟨p is s: x = c 247d⟩) {
            ⟨append s: x = c to Q 248a⟩
        }
    }
}
```
This code is used in chunk 241b.

247d ⟨p *is* s: x = c 247d⟩≡
```
constfn(p)
```

This code is used in chunks 243a, 246a, and 247c.

248a ⟨*append* `s: x = c` *to* `Q` 248a⟩≡

```
{
    Sidlink slink;
    slink = newsidlink(sid);
    if (!Q) {
        Q = Qtail = slink;
    } else {
        Qtail->next = slink;
        Qtail = slink;
    }
}
```

This code is used in chunks 243a, 246a, and 247c.

11.1 Functions on `Nodes`

`match_all_nodes` is the lame equivalent of a LISP-style maplist function. It applies `matchfn` to each `Node` in the function `mfn`. If `matchfn` succeedes, `nodefn` is applied to the node using the arguments in `Args`. `nodefn` and `Args` should really be a closure.

248b ⟨*external declarations* 211a⟩+≡

```
typedef union {
    struct {
        int i,tag;
    } tag;
    Symbol s;
    struct {
        Symbol s;
        DataValue d;
    } datasym;
} Args;
typedef int (*MatchFn)(Node, Args *);
typedef Node (*NodeFn)(Node, Node);
dclproto(void match_all_nodes,(MFunction *, MatchFn, NodeFn, Args *));
```

248c ⟨*definitions* 212a⟩+≡

```
void match_all_nodes(mfn, matchfn, nodefn, args)
    MFunction *mfn; MatchFn matchfn; NodeFn nodefn; Args *args; {
    int    fgidx, i, rootid;
    FGNode fgnode;
    Node   p;

    gentable(mfn->flowgraph, fgidx, 1) {
        fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
        gentable(fgnode->dagroots, i, 0) {
            rootid = tableentry(int *, fgnode->dagroots, i);
            if (rootid != 0) {
                assert(p = idxtable(Node *, mfn->nodetable, rootid));
                subst(p, p, matchfn, nodefn, args);
            }
        }
    }
}
```

isconstasgn determines if the statement rooted at **p** is an unambiguous assignment of a constant to a variable.

249a ⟨*local declarations* 211c⟩+≡
```
dclproto(static int const_expr,(Node));
dclproto(static int isconstasgn,(Node));
dclproto(static DataValue isconstptr,(Node));
```

249b ⟨*definitions* 212a⟩+≡
```
static int const_expr(p) Node p; {
    return (isconstasgn(p) || isconstptr(p));
}
static int isconstasgn(p) Node p; {
    if (generic(p->op) == ASGN &&
       (generic(p->kids[0]->op) == ADDRL || generic(p->kids[0]->op) == ADDRF) &&
       !(p->kids[0]->syms[0]->computed))
        if (isconstsym(p->kids[1])) return 1;
    return 0;
}
static DataValue isconstptr(p) Node p; {
    if (generic(p->op) == ASGN &&
       (generic(p->kids[0]->op) == ADDRL || generic(p->kids[0]->op) == ADDRF))
        return p->kids[1]->x.dataval;
    return 0;
}
```

subst applies the matching function **matchfn** to its first argument **p** and the arguments **args**. If a match is found, it applies the transformation function **nodefn** to **p** and **q** and returns its result. One use of **subst** is to substitute **q** for **p** under the conditions required by **matchfn**. Another is simply to identify a node of interest and return its value.

249c ⟨*local declarations* 211c⟩+≡
```
dclproto(static Node subst,(Node, Node, MatchFn, NodeFn, Args *));
```

250a ⟨*definitions* 212a⟩+≡
```
static Node subst(p, q, matchfn, nodefn, args)
    Node p,q; MatchFn matchfn; NodeFn nodefn; Args *args;
{
    if (!p || !q) return p;
    if (matchfn(p, args))
        return nodefn(p, q);
    switch(generic(p->op)) {
    /* 0 kids */
    case ADDRL: case ADDRF: case ADDRG:
    case LABEL: case CNST: break;
    /* 1 kid */
    case CALL: case RET:
        if (p->kids[0] == NULL) break;
        /* fall through */
    case ARG:  case BCOM:
    case CVC:  case CVD: case CVF: case CVI:
    case CVP:  case CVS: case CVU: case INDIR:
    case JUMP: case NEG:
        p->kids[0] = subst(p->kids[0], q, matchfn, nodefn, args);
        break;
    /* 2 kids */
    case ASGN:
    case EQ:   case GE:   case GT:   case LE:   case LT:  case NE:
    case ADD:  case BAND: case BOR:  case BXOR: case DIV:
    case LSH:  case MOD:  case MUL:  case RSH:  case SUB:
        p->kids[0] = subst(p->kids[0], q, matchfn, nodefn, args);
        p->kids[1] = subst(p->kids[1], q, matchfn, nodefn, args);
        break;
    default: error("Invalid op %s\n", opname(p->op));
        break;
    }
    return p;
}
```

250b ⟨*external declarations* 211a⟩+≡
```
dclproto(Node identitynode,(Node, Node));
dclproto(Node copyconst,(Node, Node));
dclproto(Node copynode,(Node, Node));
```

`copyconst` returns a new copy of the constant node in `q` and converts it to the type of `p`, if necessary.

250c ⟨*definitions* 212a⟩+≡
```
Node copyconst(p, q) Node p, q; {
    if (q && generic(q->op) == CNST && optype(p->op) != optype(q->op)) {
        Node newp;
        newp = new_constant_node(optype(p->op), constant(btot(optype(p->op)),
                                 q->syms[0]->u.c.v));
        return newp;
    }
    else return copynode(p, q);
}
```

`copynode` returns a copy of its second argument by recursively calling itself. `identitynode` returns its second argument.

251a ⟨*definitions* 212a⟩+≡
```
Node copynode(x, p) Node x, p; {
    Node newp = p;
    if (p) {
        newp = newnode(p->op,
                       copynode(x,p->kids[0]), copynode(x,p->kids[1]),
                       p->syms[0]);
        newp->count = p->count;
        newp->syms[1] = p->syms[1];
    }
    return newp;
}
Node identitynode(p, q) Node p, q; {
    return p;
}
```

251b ⟨*local declarations* 211c⟩+≡
```
dclproto(static int use_of,(Node, Args *));
dclproto(static int use_of_ptrx,(Node, Args *));
dclproto(static int global_ref,(Node, Args *));
```

251c ⟨*definitions* 212a⟩+≡
```
static int use_of(p, args) Node p; Args *args; {
    if (generic(p->op) == INDIR) {
        if (isaddrop(p->kids[0]->op) && p->kids[0]->syms[0] == args->s)
            return 1;
    }
    return 0;
}
static int use_of_ptrx(p, args) Node p; Args *args; {
    if (generic(p->op) == INDIR) {
        if (isaddrop(p->kids[0]->op) && p->kids[0]->syms[0] == args->datasym.s) {
            p->kids[0]->x.dataval = args->datasym.d;
            return 1;
        }
    }
    return 0;
}
static int global_ref(p, args) Node p; Args *args; {
    if (generic(p->op) == ADDRG && msym(p->syms[0]).oid != args->tag.tag
        && isoptvar(p->syms[0])) {
emitSymbol(2,p->syms[0],"\t");
        msym(p->syms[0]).oid = args->tag.tag;
        args->tag.i++;
        return 1;
    }
    return 0;
}
```

11.2 Miscellaneous

251d ⟨*external declarations* 211a⟩+≡
```
int insymtable(Symbol, MTable);
```

252a ⟨*definitions* 212a⟩+≡
```
int insymtable(s, stab) Symbol s; MTable stab; {
   int i;
   gentable(stab, i, 0) {
        if (s == tableentry(Symbol *, stab, i))
            return 1;
    }
    return 0;
}
```

Transformations often identify unreachable flow graph nodes. When a node is eliminated from the flow graph, any optimization information associated with statements in the nodes can be eliminated, which may reveal other opportunities for eliminating dead code. `eliminatenode` cleans up the `ud` and `du` chains of the given `mill` function and attempts to eliminate any code found to be dead.

252b ⟨*unused external declarations* 252b⟩≡
```
dclproto(void eliminatenode,(MFunction *, int));
```
This code is written to file `unused external declarations`.

252c ⟨*unused definitions* 252c⟩≡
```
void eliminatenode(mfn, fgnid) MFunction *mfn; int fgnid; {
    int dagid, isid, ivid;
    FGNode fgnode;
    DefUse sdu;
    Link l;
    fgnode = idxtable(FGNode *, mfn->flowgraph, fgnid);
    if (mfn->optinfo != 0) {
        gentable(fgnode->dagroots, dagid, 0) {
            int dagroot = tableentry(int *, fgnode->dagroots, dagid);
            if (dagroot != 0) {
                isid = idxtable(int *, mfn->optinfo->duinfo.smap, dagroot);
                sdu = idxtable(DefUse *, mfn->optinfo->duinfo.ismap, isid);
                if (sdu != 0) {
                    gentable(mfn->optinfo->duinfo.ivmap, ivid, 1) {
                        if (member(&sdu.udefs, ivid) || member(&sdu.adefs, ivid)) {
                            l = idxtable(Link *, uddu.du_chain, ivid);
                            ⟨delete def or use of s 253a⟩
                        } else if (member(&sdu.uuses, ivid) || member(&sdu.auses, ivid)) {
                            l = idxtable(Link *, uddu.ud_chain, ivid);
                            ⟨delete def or use of s 253a⟩
                        }
                    }
                }
            }
        }
    }
    cleartable(fgnode->dagroots, fgnode->dagroots.arena);
}
```
This code is written to file `unused definitions`.

253a ⟨*delete* `def` *or* `use` *of* `s` 253a⟩≡
```
{
    SidLink s;
    if (l->sid == isid) {
        l->sid = 0;
        l->defs_or_uses = 0;
    }
    for (s = l->defs_or_uses; s; s = s->next)
        if (s->sid == isid) s->sid = 0;
}
```
This code is used in chunk 252c.

253b ⟨*analysis.h* 253b⟩≡
```
#ifndef INCLUDE_ANALYSISH
#define INCLUDE_ANALYSISH
⟨external declarations 211a⟩
#endif
```

This code is written to file `analysis.h`.

253c ⟨*local declarations* 211c⟩+≡

253d ⟨*analysis.c* 253d⟩≡
```
#include <c.h>
#include <mld.h>
#include <analysis.h>
#include <set.h>
#include <simp.h>
#include <unmill.h>
#include <aemit.h>

⟨local declarations 211c⟩
⟨definitions 212a⟩
```

This code is written to file `analysis.c`.

12 Analysis and optimization of Modula–3 programs

12.1 Generating binding files for Modula–3 programs

`mld` initializes a Modula–3 program's type hierarchy just as the Modula–3 runtime system does, but instead of initializing the hierarchy in memory, `mld` emits a binding file that describes the initialized type hierarchy. The initialization source is derived from the SRC V2.11 run-time system (`runtime/src/generic/RTType.m3`).

This module assumes that the run-time addresses of symbols are mapped to *consecutive* link-time addresses. The current implementation of `mld/mclib` guarantees this is true for all data segments, but not for text segments.

The following changes were made to the M3 driver program, which produces the driver file `_m3main.c`: the variable `_modules` was changed from `_PRIVATE` to `_EXTERNAL` and two new variables, `_ntypes` and `_nmodules`, were added to provide access to the number of types and modules of the program pre-processed by `mldpp`.

`rt_object` produces the run-time address of the global variable `_name`.

254a ⟨*local declarations* 254a⟩≡
```
#define rt_object(_type, _name)    ((_type) rt_addr(_name))
#define LAST_Typecode 0xFFFFF
```
This definition is continued in chunks 254–58, 260c, 261d, 264c, and 267a.
This code is used in chunk 267d.

`RT0u_modules` is an initialized array that contains all modules use by the program; `RT0u_types` is an uninitialized array that contains the type hierarchy after initialization.

254b ⟨*local declarations* 254a⟩+≡
```
static _LINK_INFO **RT0u_modules;
static _TYPE **RT0u_types;
static int RT0u_nModules;
static int RT0u_nTypes;
```

The four basic type descriptors are `null`, `text`, `root`, and `uroot`. The following descriptors represent these types.

254c ⟨*local declarations* 254a⟩+≡
```
static _TYPE *_t8daa22a1_TC, *_t97b236b1_TC, *_td28c3ed0_TC, *_td2983fc3_TC;
static _TYPE *rootobj, *urootobj;
```

`LTInit` initializes the type hierarchy then prints the data image of the initialized types to standard output.

254d ⟨*external declarations* 254d⟩≡
```
void LTInit(int, FILE *);
```
This code is used in chunk 267c.

255a ⟨*definitions* 255a⟩≡
```
void LTInit (unsafe, fp) FILE *fp;{
    unsigned rtaddr, ltaddr;

    RT0u_modules = rt_object(_LINK_INFO **, "_modules") + 1;
    RT0u_nModules = *rt_object(int *, "_nmodules");
    RT0u_nTypes = *rt_object(int *, "_ntypes");
    RT0u_types = (_TYPE **)talloc((RT0u_nTypes + 10) * sizeof(_TYPE *));
    bzero((char *)RT0u_types, (RT0u_nTypes + 10) * sizeof(_TYPE *));

    LTRegisterTypes ();
    LTIdentifyOpaques ();
    LTFindChildren ();
    LTAssignTypecodes ();
    LTFixSizes ();
    LTSetupObject(rootobj, unsafe);
    LTSetupObject(urootobj, unsafe);
    LTShowDataImage (fp);
}
```
This definition is continued in chunks 255–62 and 265–67.
This code is used in chunk 267d.

255b ⟨*initialize* `selfLink`*s for root objects* 255b⟩≡
```
init_selfLink("_t8daa22a1_TC");
init_selfLink("_t97b236b1_TC");
init_selfLink("_td28c3ed0_TC");
init_selfLink("_td2983fc3_TC");
```
This code is written to file `initialize [[selfLink]]s for root objects`.

255c ⟨*local declarations* 254a⟩+≡
```
void init_selfLink(char *);
```

255d ⟨*definitions* 255a⟩+≡
```
void init_selfLink(objname) char *objname; {
    _TYPE **objt, *obj;
    objt = rt_object( _TYPE **, objname);
    obj = *objt;
    if (obj->selfLink == 0) {
        fprint(2,"Initializing selfLink of %s to %x\n", objname, objt);
        obj->selfLink = objt;
    }
}
```

`LTRegisterTypes` assigns a temporary type code to each type defined in the program, choosing a single type descriptor for any type used in more than one module.

255e ⟨*local declarations* 254a⟩+≡
```
static void LTRegisterTypes (void);
```

256a ⟨*definitions* 255a⟩+≡
```
static void LTRegisterTypes ()  {
    _TYPE *t, *old_t, **tp;
    _LINK_INFO *module;
    int cnt = 0, i;

    for (i = 0; i < RTOu_nModules; i++) {
        module = RTOu_modules[i];
        tp = module->type_cells;
        for (; tp && *tp != 0; tp++) {
            t = *tp;
            old_t = 0;
            if (t->selfLink == 0) warning("%s has NIL self link\n", t->name);
            else  old_t = *(t->selfLink);
            if (old_t == 0) { /* this is the first typecell with the given uid */
                if (t->selfLink) *(t->selfLink) = t;
                t->typecode = cnt;
                RTOu_types[cnt] = t;
                cnt++;
            } else if (old_t->name == 0 && t->name != 0) {
                /* keep the new one with the name */
                if (t->selfLink) *(t->selfLink) = t;
                t->typecode = old_t->typecode;
                RTOu_types[t->typecode] = t;
            }
        }
    }
    assert(cnt <= RTOu_nTypes);
    RTOu_nTypes = cnt;
}
```

`LTIdentifyOpaques` checks that all opaque types are revealed at most once. `LTBadPair` reports the presence of an invalid revelation.

256b ⟨*local declarations* 254a⟩+≡
```
static void LTIdentifyOpaques (void);
static void LTBadPair (char *, _TYPE_PAIR);
static void LTPutPair (char *, _TYPE_PAIR);
```

257a ⟨*definitions* 255a⟩+≡

```
static void LTIdentifyOpaques () {
    _TYPE_PAIR *pairs;
    int i;

    for (i = 0; i < RT0u_nModules; i++) {
      pairs = RT0u_modules[i]->exported_full_revelations;
        if (pairs != 0) {
            while (pairs->lhs != 0) {
                if (*(pairs->lhs) != 0) {
                    LTBadPair ("multiple revelations for opaque type: ", *pairs);
                    *(pairs->lhs) = *(pairs->rhs);
                    warning("Picking RHS\n");
                } else if (*(pairs->rhs) == 0)
                    LTBadPair ("cannot identify two opaque types: ", *pairs);
                else { /* identify the opaque and concrete */
                    *(pairs->lhs) = *(pairs->rhs);
                    LTPutPair ("Reveal: ", *pairs);
                }
                pairs++;
            }
        }
    }
}
```

257b ⟨*definitions* 255a⟩+≡

```
static void LTBadPair (msg, p) char *msg; _TYPE_PAIR p; {
    LTPutPair(msg, p);
    fatal("LTType.c", "bad revelations\n", 0);
}

static void LTPutPair (msg, p) char *msg; _TYPE_PAIR p; {
    fprint(2, "%s _t%x", msg, p.lhs_id);
    LTPutType (*(p.lhs));
    fprint(2, " and _t%x", p.rhs_id);
    LTPutType (*(p.rhs));
    fprint(2, "\n");
}
```

`LTFindChildren` finds the children, or immediate subtypes, of each type and links together the children into a list of siblings. The parent type points to the first child on the list.

257c ⟨*local declarations* 254a⟩+≡

```
static void LTFindChildren(void);
```

258a ⟨*definitions* 255a⟩+≡
```
static void LTFindChildren() {
    _TYPE *child, *parent;
    int i;

    for (i = 0; i < RT0u_nTypes; i++) {
        child = RT0u_types[i];
        assert(child);
        if (child->parentLink != 0) {
            parent = *(child->parentLink);
            assert(parent);
            child->parent = parent;
            child->sibling = parent->children;
            parent->children = child;
        }
    }
}
```

`LTAssignTypecodes` walks the type hierarchy beginning at the `root` and `uroot` types and assigns a unique typecode to each type in pre-order.

258b ⟨*local declarations* 254a⟩+≡
```
static void LTAssignTypecodes (void);
static void LTAssignObjectTypecode (_TYPE *, int *);
```

259a ⟨*definitions* 255a⟩+≡
```
static void LTAssignTypecodes () {
    _TYPE *t;
    _TYPE *u, *null, *text;
    int next_typecode, i;

    /* first, use the compiler & the temporary typecodes to find the
       types with reserved typecodes */
    _t8daa22a1_TC = *rt_object( _TYPE **, "_t8daa22a1_TC");
    _t97b236b1_TC = *rt_object( _TYPE **, "_t97b236b1_TC");
    _td28c3ed0_TC = *rt_object( _TYPE **, "_td28c3ed0_TC");
    _td2983fc3_TC = *rt_object( _TYPE **, "_td2983fc3_TC");

    null = RT0u_types[_t8daa22a1_TC->typecode];
    text = RT0u_types[_t97b236b1_TC->typecode];
    rootobj = RT0u_types[_td28c3ed0_TC->typecode];
    urootobj = RT0u_types[_td2983fc3_TC->typecode];

    /* reset the temporary typecodes */
    for( i = 0; i <  RT0u_nTypes; i++) {
        t = RT0u_types[i];
        t->typecode = LAST_Typecode;    /* LAST_Typecode = 0xFFFFF on MIPS */
    }

    /* assign the fixed typecodes */
    null->typecode = 0;
    null->lastSubTypeTC = 0;
    text->typecode = 1;
    text->lastSubTypeTC = 1;
        next_typecode = 2;

    /* assign the OBJECT typecodes */
    LTAssignObjectTypecode(rootobj, &next_typecode);
    LTAssignObjectTypecode(urootobj, &next_typecode);

    ⟨assign the remaining REF typecodes 259b⟩
    assert(next_typecode == RT0u_nTypes);

    ⟨shuffle the typecells into their correct slots 260a⟩
}
```

259b ⟨*assign the remaining REF typecodes* 259b⟩≡
```
for (i = 0; i < RT0u_nTypes; i++) {
    t = RT0u_types[i];
    if (t->typecode == LAST_Typecode) {
        t->typecode = next_typecode;
        t->lastSubTypeTC = next_typecode;
        next_typecode++;
    }
}
```

This code is used in chunk 259a.

260a ⟨*shuffle the typecells into their correct slots* 260a⟩≡
```
for (i = 0; i < RTOu_nTypes; i++) {
    t = RTOu_types[i];
    while (t->typecode != i) {
        u = RTOu_types[t->typecode];
        RTOu_types[t->typecode] = t;
        t = u;
    }
    RTOu_types[i] = t;
}
```
This code is used in chunk 259a.

260b ⟨*definitions* 255a⟩+≡
```
static void LTAssignObjectTypecode (t, next) _TYPE *t; int *next; {
    _TYPE *u;

    assert(t->typecode == LAST_Typecode);
    t->typecode = *next;
    *next += 1;
    u = t->children;
    while (u != 0) {
              LTAssignObjectTypecode (u, next);
        u = u->sibling;
    }
    t->lastSubTypeTC = *next - 1;
}
```

`LTFixSizes` computes the data and method sizes and offsets for each object type. There are lots of architecture dependencies in here!

260c ⟨*local declarations* 254a⟩+≡
```
static void LTFixSizes (void);
static void LTFixObjectSizes (_TYPE *);
```

260d ⟨*definitions* 255a⟩+≡
```
static void LTFixSizes () {
    _TYPE *t;
    int i;

    /* make sure that all the REF types are some multiple of header words */
    for (i = 0; i < RTOu_nTypes; i++) {
        t = RTOu_types[i];
                    /* ???? 0 replaces RTO.NilTypecode */
        if (t->typecode != 0 && t->parent == 0  && t->children == 0) {
            ⟨compute size of t 260e⟩
        }
    }
    LTFixObjectSizes (rootobj);
    LTFixObjectSizes (urootobj);
}
```

This is a hack. `sizeof(unsigned)` replaces `RTHeapRep.Header`, which is probably architecture dependent!

260e ⟨*compute size of* `t` 260e⟩≡
```
t->dataSize = roundup (t->dataSize, sizeof(unsigned));
```
This code is used in chunks 260d and 261a.

261a ⟨*definitions* 255a⟩+≡
```
static void LTFixObjectSizes (t) _TYPE *t; {
    _TYPE *u;

    u = t->parent;
    if (u != 0) {
        t->dataOffset = roundup(u->dataSize, t->dataAlignment);
        t->dataSize += t->dataOffset;
        t->dataAlignment = max (t->dataAlignment, u->dataAlignment);
        t->methodOffset = u->methodSize;
        t->methodSize +=  t->methodOffset;
    }
    ⟨compute size of t 260e⟩
    ⟨allocate default methods 261c⟩
    ⟨compute sizes of children 261b⟩
}
```

261b ⟨*compute sizes of children* 261b⟩≡
```
u = t->children;
while (u != 0) {
    LTFixObjectSizes (u);
    u = u->sibling;
}
```
This code is used in chunk 261a.

261c ⟨*allocate default methods* 261c⟩≡
```
t->defaultMethods = (_ADDRESS)alloc(t->methodSize);
bzero(t->defaultMethods, t->methodSize);
```
This code is used in chunk 261a.

`LTSetupObject` recursively walks over the type hierarchy in post order, computing the method values for the type object `t`. If `unsafe` is unset, a type method `m` is initialized to the procedure `p` if and only if `m` is initialized to `p` in `t`. If `unsafe` is set, a type method `m` is initialized to the procedure `p` in `bindings(t,m)` if and only if `|bindings(t,m)| = 1`.

261d ⟨*local declarations* 254a⟩+≡
```
typedef struct methodinfo {
    struct set v;
    MTable methods;
}MethodInfo;

static MethodInfo LTSetupObject (_TYPE *, int);
#define METHODID(off)    ((off) / sizeof(_ADDRESS))
```

262a ⟨*definitions* 255a⟩+≡

```
static MethodInfo LTSetupObject (t, unsafe) _TYPE *t; {
    _TYPE *u;
    _TYPE ***methodtypes = 0, ***types;
    _ADDRESS defaults;
    int   off, stmethodsize, id;
    BaseTable *bt;
    struct set subtv;
    MethodInfo retval;
    char  miname[30];
    Symbol method;
    int msize;

    u = t->parent;
    ⟨copy parent's default methods 262b⟩
    *(int *)t->defaultMethods = t->typecode;

    msize = METHODID(t->methodSize);
    initset(&retval.v, msize);     /* vector of this type's overrides */
    initset(&subtv, msize);        /* vector of subtype's overrides */

    newtable(&retval.methods, msize, sizeof(Symbol), &permanent);
    updatetable(Symbol *, &(retval.methods), msize, 0);

    ⟨lookup supertypes that type t overrides 263a⟩
    if (methodtypes) {
        for (types = methodtypes; *types; types++) {
            _TYPE *st;

            st = **types;
            sprintf(miname,"_t%08x_t%08x_methods", t->selfID, st->selfID);
            if (method = install_symbol(string(miname), &bt)) {
                void  **methodvals, **methods;

                assert(methodvals = rt_object(void **, miname));
                ⟨compute t's offset in supertype st's method image 263c⟩
            }
        }
    }
    ⟨initialize method images of subtypes 264a⟩
    ⟨eliminate overridden methods from type's default method image 264b⟩
    uset(&retval.v, &subtv);
    return retval;
}
```

262b ⟨*copy parent's default methods* 262b⟩≡

```
if (u != 0) bcopy(u->defaultMethods, t->defaultMethods, u->methodSize);
```

This code is used in chunk 262a.

263a ⟨*lookup supertypes that type t overrides* 263a⟩≡
```
sprintf(miname, "_t%08x_methodType", t->selfID);
if (method = install_symbol(string(miname), &bt))
    methodtypes = rt_object(_TYPE ***, miname);
```

This code is used in chunk 262a.

263b ⟨*check if method is overridden* 263b⟩≡
```
if (method) {
    int idx;

    msym(method).method = 1;
    if (t != st) fprint(2,"# type %d override of %d:%s in supertype %d\n",
            t->typecode, st->methodOffset+off, method->name, st->typecode);
    *((Symbol *)(defaults + st->methodOffset + off)) = method;
    idx = METHODID(st->methodOffset + off);
    insert(&retval.v, idx);
    updatetable(Symbol *, &(retval.methods), idx, method);
}
```
This code is used in chunk 263c.

263c ⟨*compute* `t`*'s offset in supertype* `st`*'s method image* 263c⟩≡
```
stmethodsize = st->methodSize - st->methodOffset;
defaults = (_ADDRESS)t->defaultMethods;
for (methods = methodvals, off = 0;
    off < stmethodsize; methods++, off += sizeof(_ADDRESS)) {
    method = *(Symbol *)methods;
    ⟨check if method is overridden 263b⟩
}
```
This code is used in chunk 262a.

264a ⟨*initialize method images of subtypes* 264a⟩≡

```
u = t->children;
while (u != 0) {
    Symbol subtm;
    MethodInfo stval;

    stval = LTSetupObject(u, unsafe);
    uset(&subtv, &stval.v);

    /* Determine if a subtype overrides a method undefined in t */
    for (id = 0; id < msize && id < tablesize(stval.methods); id++) {
        subtm = idxtable(Symbol *, stval.methods, id);
        if (subtm != 0) {
            /* if type's method is not overridden, update with subtype's method */
            if (id >= tablesize(retval.methods)
             || idxtable(Symbol *, retval.methods, id) == 0) {
                updatetable(Symbol *, &retval.methods, id, subtm);
            } else if (subtm != idxtable(Symbol *, retval.methods, id)) {
                updatetable(Symbol *, &retval.methods, id, 0);
            }
        }
    }
    u = u->sibling;
}
```

This code is used in chunk 262a.

264b ⟨*eliminate overridden methods from type's default method image* 264b⟩≡

```
defaults = (_ADDRESS)t->defaultMethods;
for (id = 0; id < msize; id++) {
    if (member(&subtv, id)) {
        off = sizeof(_ADDRESS)*id;
        if (unsafe && *((Symbol *)(defaults + off)) == 0) {
            fprint(2,"# type %d has UNSAFE subtype override of %d\n", t->typecode, off);
            *((Symbol *)(defaults + off)) = idxtable(Symbol *, retval.methods, id);
        } else {
            fprint(2,"# type %d has subtype override of %d\n", t->typecode, off);
            *((Symbol *)(defaults + off)) = (Symbol)0;
        }
    }
}
```

This code is used in chunk 262a.

`LTShowDataImage` prints the data image of the initialized type hierarchy to standard output. `LTShowMethods` prints an initialized array of the methods for a particular type.

264c ⟨*local declarations* 254a⟩+≡

```
static void LTShowDataImage (FILE *);
static void LTShowMethods (_TYPE *, char *, FILE *);
```

265 ⟨*definitions* 255a⟩+≡

```
static void LTShowDataImage (fp) FILE *fp;{
    _TYPE *t;
    int isObject, i;
    _TYPE_PAIR *pairs;

    fprintf(fp, "RT0u__nModules = %d;\n", RT0u_nModules);
    fprintf(fp, "RT0u__nTypes = %d;\n", RT0u_nTypes);

    for (i = 0; i < RT0u_nTypes; i++) {
        t = RT0u_types[i];
        isObject = 0;
        fprintf(fp, "*_t%08x_TC = {\n", t->selfID);
        if (t->name) fprintf(fp, "  name = \"%s\";\n", t->name);
        if (t->brand) fprintf(fp, "  brand = \"%s\";\n", t->brand);
        fprintf(fp, "  typecode= %d;\n", t->typecode);
        fprintf(fp, "  lastSubTypeTC = %d;\n", t->lastSubTypeTC);

        ⟨emit relatives 266a⟩
        fprintf(fp, "  traced = %d;\n", t->traced);
        fprintf(fp, "  dataSize = %d;\n", t->dataSize);
        fprintf(fp, "  dataAlignment = %d;\n", t->dataAlignment);
        fprintf(fp, "  dataOffset = %d;\n", t->dataOffset);
        if (isObject) {
            fprintf(fp, "  methodSize= %d;\n", t->methodSize);
            fprintf(fp, "  methodOffset= %d;\n", t->methodOffset);
        }
        fprintf(fp, "  nDimensions = %d;\n", t->nDimensions);
        fprintf(fp, "  elementSize = %d;\n", t->elementSize);

        ⟨emit default method image 266b⟩
        fprintf(fp, "};\n");
    }
    ⟨reveal all opaque types explicitly 266c⟩
    fprintf(fp, "RT0u__types = [\n");
    for (i = 0; i < RT0u_nTypes; i++) {
        t = RT0u_types[i];
        fprintf(fp, "    %d : _t%08x_TC;\n", i * 4, t->selfID);
    }
    fprintf(fp, "];\n");
    fflush(stdout);
}
```

266a ⟨*emit relatives* 266a⟩≡
```
if (t->parent != 0) {
    fprintf(fp, "  parent = _t%08x_TC;\n", t->parent->selfID);
    isObject = 1;
}
if (t->children != 0) {
    fprintf(fp, "  children = _t%08x_TC;\n", t->children->selfID);
    isObject = 1;
}
if (t->sibling != 0) {
    fprintf(fp, "  sibling = _t%08x_TC;\n", t->sibling->selfID);
    isObject = 1;
}
```
This code is used in chunk 265.

266b ⟨*emit default method image* 266b⟩≡
```
if (isObject) LTShowMethods(t, "  ", fp);
if (t->setupProc) fprintf(fp, "  setupProc = %s;\n", ((Symbol)t->setupProc)->name);
if (t->mapProc)   fprintf(fp, "  mapProc = %s;\n",   ((Symbol)t->mapProc)->name);
if (t->initProc)  fprintf(fp, "  initProc = %s;\n",  ((Symbol)t->initProc)->name);
```
This code is used in chunk 265.

266c ⟨*reveal all opaque types explicitly* 266c⟩≡
```
for (i = 0; i < RT0u_nModules; i++) {
    pairs = RT0u_modules[i]->exported_full_revelations;
    if (pairs != 0) {
        while (pairs->lhs != 0) {
            assert(*(pairs->lhs) != 0);
            fprintf(fp, "*_t%08x_TC = *_t%08x_TC;\n", pairs->lhs_id, pairs->rhs_id);
            pairs++;
        }
    }
}
```

This code is used in chunk 265.

266d ⟨*definitions* 255a⟩+≡
```
static void LTShowMethods (t, space, fp) _TYPE *t; char *space; FILE *fp;{
    _ADDRESS defaults;
    Symbol dfault;
    int off;

    if (t->defaultMethods && t->methodSize > sizeof(_ADDRESS)) {
        fprintf(fp, "%s*defaultMethods = [\n", space, t->selfID);
        defaults = t->defaultMethods;
        for (off = sizeof(_ADDRESS); off < t->methodSize; off += sizeof(_ADDRESS)) {
            dfault = *((Symbol *)(defaults + off));
            if (dfault != 0)
                fprintf(fp, "%s%s%d: %s;\n", space, space, off, dfault->name);
            else
                fprintf(fp, "%s%s# %d: method overridden by subtype\n", space, space, off);
        }
        fprintf(fp, "%s];\n", space);
    }
}
```

`LTPutType` prints out the header of a type.

267a ⟨*local declarations* 254a⟩+≡
```
static void LTPutType (_TYPE *);
```

267b ⟨*definitions* 255a⟩+≡
```
static void LTPutType (t) _TYPE *t; {
    fprint (2,"[");
    if (t->name != 0) fprint(2, "%s", t->name);
    else fprint (2,"<anonymous>");
    fprint (2,"  ");

    if (t->brand != 0) {
        fprint (2,"\"");
        fprint(2, "%s", t->brand);
        fprint (2,"\"");
    } else fprint (2,"<no brand>");
    fprint (2,"]");
}
```

267c ⟨*lttype.h* 267c⟩≡
```
#ifndef INCLUDE_LTTYPEH
#define INCLUDE_LTTYPEH
⟨external declarations 254d⟩
#endif
```

This code is written to file `lttype.h`.

267d ⟨*lttype.c* 267d⟩≡
```
#include <stdio.h>
#include <c.h>
#include <M3Runtime.h>
#include <resextsym.h>
#include <binary.h>
#include <set.h>
#include <lttype.h>
#include <mld.h>

⟨local declarations 254a⟩
⟨definitions 255a⟩
```
This code is written to file `lttype.c`.

12.2 Optimization analysis for Modula-3 programs

This module contains procedures that analyze and transform Modula-3 programs. Analyses include identifying an object's runtime type and recognition of method calls. Transformations include conversion of method calls into direct procedure calls, propagation and simplification of expressions that refer to known (runtime) types, and elimination of NIL checks for object's whose runtime values are provably non-NIL.

12.2.1 Type analysis

Type analysis is applied to all Modula-3 procedures, excluding those generated by the `m3` compiler and `main`. Type analysis is similar to constant propagation. Instead of discovering all assignments of a constant to a variable, we look for assignments of new objects to variables. When an object is created, its runtime type is known and can be propagated to uses of the object. Expressions that refer to an object's runtime type may be simplified or eliminated.

268a ⟨*m3 external declarations* 268a⟩≡

```
dclproto(void m3type_analysis,(MFunction *, Arena *));
```

This definition is continued in chunks 273, 275d, and 276d.
This code is used in chunk 277b.

268b ⟨*m3 definitions* 268b⟩≡

```
void m3type_analysis(mfn, arena) MFunction *mfn; Arena *arena; {
    OptInfo *optinfo;
    fprint(2,"%s: ncalls %d\n", mfn->sym->name, mfunc(mfn->sym)->ncalls);
    if (compute_reaching_defs(mfn, m3aliased_var, arena))
        m3type_propagation(mfn);
}
```

This definition is continued in chunks 269, 271, 272b, and 274–76.
This code is used in chunk 277c.

For type propagation, we want to discover assignments of a new object to a variable or of an object with known type to a variable; `has_known_type` identifies such assignments. After identifying the initial set of assignments, we propagate the type information through the procedure; when new assignments are revealed, they are also propagated. During propagation, we attempt to simplify or eliminate any expressions that refer to the object's type. We also determine if the procedure returns an object with a known type; this information is used inter-procedurally during analysis of the callers of the current procedure.

268c ⟨*m3 local declarations* 268c⟩≡

```
dclproto(static void m3type_propagation,(MFunction *));
```

This definition is continued in chunks 271–73 and 276b.
This code is used in chunk 277c.

269 ⟨*m3 definitions* 268b⟩+≡

```
static void m3type_propagation(mfn) MFunction *mfn;  {
    Sidlink Q, Qtail, s;
    int (*constfn)(Node);
    DefUseInfo *duinfo;
    Chains *uddu;

    fprint(2,"m3 type propagation...\n");
    assert(mfn->optinfo);
    duinfo = &mfn->optinfo->duinfo;
    uddu = &mfn->optinfo->uddu;

    constfn = has_known_type;
    ⟨append all s: x = c in fn to Q ??⟩
    ⟨Q sanity check ??⟩
    while (Q != 0) {
        int ivid, sid, type;
        Link du = 0;
        Sidlink use;
        Symbol x;
        Node p;
        FGNode fgnode;
        DefUse *defdu;

        ⟨s = pop(Q) ??⟩
        defdu = &tableentry(DefUse *, duinfo->ismap, s->sid);
        sid = defdu->id;
        ⟨find p's flowgraph node ??⟩
        ⟨find du(s,x) ??⟩
        ⟨find type of object 270d⟩
        ⟨for each use(x,s') ∈ du(s,x) ??⟩ {
            DefUse *usedu;

            usedu = &tableentry(DefUse *, duinfo->ismap, use->sid);
            fprint(2, "use of %s at %d\n", x->name, usedu->id);
            if (⟨use(x,s') is unambiguous ??⟩) {
                Link ud = 0;
                Node defq, useq;
                int typeval = defdu->typeval;

                ⟨find ud(x,s') ??⟩
                ⟨announce defs that reach this use 270e⟩
                useq = idxtable(Node *, mfn->nodetable, usedu->id);
                if (⟨use is assignment y = x 270a⟩ && typeval != 0) {
                     int sid = use->sid;
                    ⟨propagate type 270c⟩
                    ⟨append s: x = c to Q ??⟩
                } else if (⟨use is return value 270b⟩  && typeval != 0) {
                    ⟨tag ReturnN node with type value 272c⟩
                } else if (typeval != 0) {
                    ⟨try to simplify expression, when type is known 272d⟩
                }
            }
        }
```

```
    }
    ⟨determine if return type of fn is known 273a⟩
    fprint(2,"...m3 type propagation\n");
  }
```

270a ⟨*use is assignment* `y = x` 270a⟩≡

```
  iscopyasgn(useq)
```

This code is used in chunk 269.

270b ⟨*use is return value* 270b⟩≡

```
  (generic(useq->op) == RET && useq->kids[0] && issimpleuse(useq->kids[0]))
```

This code is used in chunk 269.

270c ⟨*propagate type* 270c⟩≡

```
  usedu->typeval = typeval;
  fprint(2,"Propagate type %d to %d\n", typeval, usedu->id);
```

This code is used in chunk 269.

270d ⟨*find type of object* 270d⟩≡

```
  {
    Node argn;
    Symbol ty;

    if (defdu->typeval != 0)
        fprint(2,"Object type known %d\n", defdu->typeval);
    else if (isnewobject(p)) {
        int rootid = tableentry(int *, fgnode->dagroots, 0);
        assert ((argn = idxtable(Node *, mfn->nodetable, rootid))
                 && argn->op == ARGI);
        ty = isconstsym(argn->kids[0]);
        if (ty != 0) {
            fprint(2,"Create object type %d\n", ty->u.c.v.i);
            defdu->typeval = ty->u.c.v.i;
        } else fprint(2,"Create object type UNKNOWN\n");
    } else if (!(defdu->typeval = has_known_return_type(p)))
        fprint(2,"Object type UNKNOWN\n");
  }
```

This code is used in chunk 269.

270e ⟨*announce defs that reach this use* 270e⟩≡

```
  {
    Sidlink def;
    DefUse defdu;
    for (def = ud->defs_or_uses; def; def = def->next) {
        defdu = idxtable(DefUse *, duinfo->ismap, def->sid);
        defq = idxtable(Node *, mfn->nodetable, defdu.id);
        fprint(2, "def %d reaches by %s type %d\n",
                  defdu.id, opname(defq->op), defdu.typeval);
        typeval = (typeval != defdu.typeval) ? 0 : typeval;
    }
  }
```

This code is used in chunk 269.

Variables generated by the Modula-3 compiler are excluded from the set of possibly aliased variables because we know they will have no effect on analysis. For the remaining variables, we apply the very conservative heuristic that if the variable's address is taken, it is a potential alias of all other addressed variables.

271a ⟨*m3 local declarations* 268c⟩+≡
```
dclproto(static int m3aliased_var,(Symbol));
dclproto(static int rootnode,(Node, Args *));
```

271b ⟨*m3 definitions* 268b⟩+≡
```
static int m3aliased_var(p) Symbol p; {
    char *s;
    if (!p->addressed) return 0;
    else if (m3generated_name(p->name)) return 0;
    else return p->addressed;
}

static int rootnode(p, args) Node p; Args *args; {
    if (rootop(generic(p->op))) return 1;
    else return 0;
}
```

`has_known_type` determines if the value on the right-hand side of an assignment has a known type; expressions that create a new object or are the results of functions with known types qualify.

271c ⟨*m3 local declarations* 268c⟩+≡
```
dclproto(static int isnewobject,(Node));
dclproto(static unsigned has_known_return_type,(Node));
dclproto(static int has_known_type,(Node));
```

271d ⟨*m3 definitions* 268b⟩+≡
```
static int has_known_type(p) Node p; {
    return (isnewobject(p) || has_known_return_type(p));
}

static unsigned has_known_return_type(p) Node p; {
    if (generic(p->op) == ASGN && generic(p->kids[1]->op) == CALL &&
        p->kids[1]->kids[0]->op == ADDRGP &&
        msym(p->kids[1]->kids[0]->syms[0]).typeval != 0) {
            fprint(2,"Known return type %d\n", msym(p->kids[1]->kids[0]->syms[0]).typeval);
        return msym(p->kids[1]->kids[0]->syms[0]).typeval;
    }
    return 0;
}

static int isnewobject(p) Node p; {
    static char *allocname = 0;
    if (allocname == 0) allocname = string("RTHeap__Allocate");
    if (generic(p->op) == ASGN && generic(p->kids[1]->op) == CALL &&
        p->kids[1]->kids[0]->op == ADDRGP &&
        p->kids[1]->kids[0]->syms[0]->name == allocname) {
        fprint(2,"New object call at %d\n", p->kids[1]->x.dagid);
        return 1;
    }
    return 0;
}
```

272a ⟨*m3 local declarations* 268c⟩+≡
```
dclproto(static int m3generated_name,(char *));
```

272b ⟨*m3 definitions* 268b⟩+≡
```
static int m3generated_name(name) char *name; {
    dclproto(extern char *strstr,(char *,char *));
    char *s, *t;
    if (strncmp(name, "_t", 2) == 0) {
        s = &(name[2]);
        if (strstr(s, "_TC") ||
            strstr(s, "_map") ||
            strstr(s, "_tc")) return 1;
        else if (strncmp(name, "_try", 4) ||
                 strncmp(name, "_txt", 4) ||
                 strncmp(name, "_type_info", 10)  == 0 ||
                 strncmp(name, "_type_cells", 11)  == 0 )
            return 1;
        else return 0;
    } else if (strcmp(name, "_M3__nil_check_char") == 0 ||
               strncmp(name, "_map_", 5)  == 0 ||
               strncmp(name, "_init_", 6)  == 0 ||
               strncmp(name, "_fp_data", 8)  == 0 ||
               strncmp(name, "_exported", 9) == 0 ||
               strncmp(name, "_imported", 9) == 0 ||
               strncmp(name, "_proc_info", 10)  == 0 ||
               strncmp(name, "_raises", 6) == 0 ||
               strncmp(name, "_FINALLY", 8) == 0)
        return 1;
    else return 0;
}
```

272c ⟨*tag* `ReturnN` *node with type value* 272c⟩≡
```
{
    FGNode fgnode;
    Node p = useq;
    int sid = usedu->id;
    fprint(2,"Return type known %d\n", typeval);
    ⟨find p's flowgraph node ??⟩
    assert(fgnode->type == RetN);
    returnN(fgnode).typeval = typeval;
}
```
This code is used in chunk 269.

272d ⟨*try to simplify expression, when type is known* 272d⟩≡
```
Node savq = useq;

msym(x).typeval = typeval;
printdag(useq, 2);
useq = dag_simplify(useq, (Type)0);
if (savq != useq)
    updatetable(Node *, &(mfn->nodetable), usedu->id, useq);
msym(x).typeval = 0;
```
This code is used in chunk 269.

273a ⟨*determine if return type of* `fn` *is known* 273a⟩≡
```
{
    FGNode fgnode;
    int fgidx, set = 0;
    unsigned typeval = 0;
    gentable(mfn->flowgraph, fgidx, 1) {
        fgnode = tableentry(FGNode *, mfn->flowgraph, fgidx);
        if (fgnode->type == RetN) {
            if (!set) {
                typeval = returnN(fgnode).typeval;
                set = 1;
            } else if (typeval != returnN(fgnode).typeval)
                typeval = 0;
        }
    }
    if (typeval != 0) {
        fprint(2,"RETURN TYPE IS ALWAYS %d\n", typeval);
        msym(mfn->sym).typeval = typeval;
    }
}
```
This code is used in chunk 269.

12.2.2 Identifying and converting method calls

273b ⟨*m3 local declarations* 268c⟩+≡
```
static int methodcalls = 0, indircalls = 0;
static int anon = 0, unknown = 0;
static int leaftypes = 0, nonleaftypes = 0;
int convertible = 0;
```

273c ⟨*m3 external declarations* 268a⟩+≡
```
extern int convertible;
```

`method_idiom` is called from the `iburg`-generated expression simplifier. It catagorizes each indirect procedure call as either a method call or as a standard indirect procedure call. Method calls are divided into four groups: `anonymous`, meaning the type of the method's object is anonymous; `unknown`, meaning the type of the method's object is known but does not have a known representation at link-time; `leaf`, meaning the object type is a leaf in the type hierarchy; and `non-leaf`, meaning the object type is an internal node in the type hierarchy.

273d ⟨*m3 external declarations* 268a⟩+≡
```
dclproto(Node method_idiom,(Node, Node, Node, Node, int));
```

273e ⟨*m3 local declarations* 268c⟩+≡
```
#define TYPECODE_OFFSET 0
#define LASTSUBT_OFFSET 4
#define METHOD_OFFSET 52
#define anon_name(_obj) ((_obj) ? (_obj->name) : "ANON")
```

274 ⟨*m3 definitions* 268b⟩+≡

```
Node method_idiom(callnode, symnode, othernode1, othernode2, code)
  Node callnode, symnode, othernode1, othernode2; {
    extern Symbol cfunc;
    Symbol obj, type, method, c1, c2;
    DataValue typeval, defmethodval, methodval;
    Value typecodeval, lastsubtypeval;
    BaseTable *bt;
    int methodoff;

    if (code == 0) {
        indircalls++;
        if (dbgflag) fprint(2,"other indirect call in %s:\n", anon_name(cfunc));
    } else {
        methodcalls++;
        obj = symnode->syms[0];
        fprint(2,"method call in %s %d: obj %s auxtype %s typeval %d\n",
                 anon_name(cfunc), code, anon_name(obj),
        obj ? anon_name(msym(obj).auxtype) : "ANON", msym(obj).typeval);

        if (!obj || !msym(obj).auxtype) anon++;
        else if (msym(obj).auxtype) {
            char *typename;

            typename = m3typename(msym(obj).auxtype->name, "_TC");
            if ((type = lookupsym(string(typename), &bt)) &&
               (typeval = constaddr(1, mglob(type).datavalue))) {

                typecodeval = datavalue(constoffset(typeval, TYPECODE_OFFSET), I);
                ⟨count leaf and non-leaf types 275c⟩

                defmethodval = constoffset(typeval, METHOD_OFFSET);
                if (defmethodval && (c1 = isconstsym(othernode1)) &&
                    (!othernode2 || (c2 = isconstsym(othernode2)))) {

                    methodoff = c1->u.c.v.i;
                    if (code == 3 || code == 6 || code == 9)
                        methodoff += c2->u.c.v.i;
                    if (dbgflag) fprint(2,"method offset %d\n", methodoff);
                    if (methodval = constoffset(defmethodval, methodoff)) {
                        ⟨convert method to direct call 275a⟩
                    }
                }
            }
            else unknown++;
        }
    }
    ⟨print debug info 275b⟩
    return callnode;
}
```

275a ⟨*convert method to direct call* 275a⟩≡
```
method = (Symbol)(datavalue(methodval, P).p);
if (optdbgflag) fprint(2, "method %s\n", method ? method->name : "*unknown method*");
if (method && convertflag) {
    callnode->kids[0] = newnode(ADDRGP, 0, 0, method);
    convertible++;
}
```
This code is used in chunk 274.

275b ⟨*print debug info* 275b⟩≡
```
if (dbgflag) {
    fprint(2,"Node 1: "); if (othernode1) printdag(othernode1, 2); else fprint(2,"\n");
    fprint(2,"Node 2: "); if (othernode2) printdag(othernode2, 2); else fprint(2,"\n");
}
```
This code is used in chunk 274.

This is a hack, because we're using explicit constants to identify the fields of binding values. The right way to do this is use the type of a type descriptor to find the offset to the required field.

275c ⟨*count leaf and non-leaf types* 275c⟩≡
```
lastsubtypeval = datavalue(constoffset(typeval, LASTSUBT_OFFSET), I);
if (typecodeval.i == lastsubtypeval.i) {
    leaftypes++;
    if (dbgflag) fprint(2,"leaf type %s\n", type->name);
} else {
    nonleaftypes++;
    if (dbgflag) fprint(2,"nonleaf type %s\n", type->name);
}
```
This code is used in chunk 274.

`typecode_idiom` attempts to convert a reference to an object's typecode to its runtime value. This is possible if `obj`'s runtime type has been inferred.

275d ⟨*m3 external declarations* 268a⟩+≡
```
dclproto(Node typecode_idiom,(Node, Node, Node, Node, Node));
dclproto(Node nilcheck_idiom,(Node, Node, Node, Node));
```

275e ⟨*m3 definitions* 268b⟩+≡
```
Node typecode_idiom(res, obj, offset, shift1, shift2)
    Node res, obj, offset, shift1, shift2; {
    Symbol objsym, offsym, shsym1, shsym2;

    offsym = isconstsym(offset);
    shsym1 = isconstsym(shift1);
    shsym2 = isconstsym(shift2);
    if (isaddrop(obj->op) && offsym != 0 && shsym1 != 0 && shsym2 != 0 &&
        offsym->u.c.v.i == -4 && shsym1->u.c.v.i == 11 && shsym2->u.c.v.i == 12) {
        objsym = obj->syms[0];
        if (msym(objsym).typeval != 0) {
           fprint(2,"TYPECODE %s at %d of %d\n",
                  objsym->name, res->x.dagid, msym(objsym).typeval);
           return constnode(msym(objsym).typeval, inttype);
        }
    }
    return res;
}
```

`nilcheck_idiom` attempts to eliminate a `NIL` check of an object, which is possible if `obj`'s runtime type is known.

276a ⟨*m3 definitions* 268b⟩+≡
```
Node nilcheck_idiom(res, nilcheckchar, obj, offset)
    Node res, nilcheckchar, obj, offset; {
    static char *nilcheckname;
    Symbol offsym = isconstsym(offset), objsym;
    if (!nilcheckname) nilcheckname = string("_M3__nil_check_char");
    if (nilcheckchar->op == ADDRGP
        && nilcheckchar->syms[0]->name == nilcheckname
        && offsym->u.c.v.i == -1
        && isaddrop(obj->op)) {
        objsym = obj->syms[0];
        if (msym(objsym).typeval != 0) {
            fprint(2,"NILCHECKB %s of %d\n", objsym->name, msym(objsym).typeval);
            return constnode(0, inttype);
        }
    }
    return res;
}
```

276b ⟨*m3 local declarations* 268c⟩+≡
```
dclproto(static char *m3typename,(char *, char *));
```

276c ⟨*m3 definitions* 268b⟩+≡
```
static char *m3typename(name, suffix) char *name, *suffix; {
    char *typename;
    typename = talloc(strlen(name) + strlen(suffix));
    strcpy(typename, name);
    strcat(typename, suffix);
    return typename;
}
```

`method_summary` provides a summary of the types of indirect procedure calls.

276d ⟨*m3 external declarations* 268a⟩+≡
```
dclproto(void method_summary,(void));
```

276e ⟨*m3 definitions* 268b⟩+≡
```
void method_summary() {
    fprint(2,"expressions converted %d\n", exprconvert);
    fprint(2,"method calls %d : convertible %d: anon %d unknown %d leaf types %d non-leaf types %d\n",
        methodcalls, convertible, anon, unknown, leaftypes, nonleaftypes);
    fprint(2,"indirect calls %d converted calls %d\n", indircalls, symconvert);
}
```

277a ⟨ivmap *sanity check* 277a⟩≡
```
gentable(duinfo.lvmap, i, 1) {
    if (tableentry(int *, duinfo.lvmap, i) != 0) {
        s = idxtable(Symbol *, mfn->localsymtab, i);
        fprint(2,"local %s %s\n", s->name, anon_name(msym(s).auxtype));
    }
}
gentable(duinfo.vmap, i, 1) {
    if (tableentry(int *, duinfo.vmap, i) != 0) {
        s = idxtable(Symbol *, mfn->bt->glbsymtab, i);
        fprint(2,"global %s %s\n", s->name, anon_name(msym(s).auxtype));
    }
}
```
This code is written to file `[[ivmap]] sanity check`.

277b ⟨*m3.h* 277b⟩≡
```
#ifndef INCLUDE_M3H
#define INCLUDE_M3H
⟨m3 external declarations 268a⟩
#endif
```

This code is written to file `m3.h`.

277c ⟨*m3.c* 277c⟩≡
```
#include <c.h>
#include <mld.h>
#include <analysis.h>
#include <dataimage.h>
#include <m3.h>
#include <resextsym.h>
#include <unmill.h>
#include <simp.h>

⟨m3 local declarations 268c⟩
⟨m3 definitions 268b⟩
```
This code is written to file `m3.c`.

13 Machine-independent binary emission

This section implements the back-end support procedures that emit a binary representation of code and data. Machine-dependent code for emitting binary appears in the files *target*-`bemit.nw`.

`emitbin` emits one or more binary instructions generated for node `p`. `emitbin` is the binary analog of `emitasm`. The prologue and epilogue of `emitbin` are target independent. The rules for matching instructions are not; they appear in the target-dependent files.

278a ⟨*emitbin prologue* 278a⟩≡

```
unsigned emitbin(p, nt) Node p; {
     int rulenum, i;
     short *nts;
     Node kids[10];
     register unsigned lhs;
     unsigned u, v[10];
     char *fmt;
     int omitmove = 0;

     p = reuse(p, nt);
     rulenum = getrule(p, nt);
     nts = IR->x._nts[rulenum];
     fmt = IR->x._templates[rulenum];
     if (IR->x._isinstruction[rulenum] && p->x.emitted)
         return ((unsigned)p->syms[RX]);
     if (*fmt != '#') {
         if (*fmt == '?') {
             assert(p->kids[0]);
             if (p->syms[RX] == p->kids[0]->syms[RX])
                 omitmove = 1;
         }
         (*IR->x._kids)(p, rulenum, kids);
         for (i = 0; nts[i]; i++)
             v[i] = emitbin(kids[i], nts[i]);
     }
     switch (rulenum) {
```

This code is written to file `emitbin prologue`.

278b ⟨*emitbin epilogue* 278b⟩≡

```
     default:    assert(0);
     }
     return lhs;
}
```

This code is written to file `emitbin epilogue`.

278c ⟨*local declarations* 278c⟩≡

```
dclproto(static void b_address,(Symbol, Symbol, int));
dclproto(static unsigned b_def,(Symbol));
dclproto(static void b_defaddress,(Symbol));
dclproto(static void b_end,(Symbol));
dclproto(static void b_entry,(Symbol));
dclproto(static void b_global,(Symbol));
```

This definition is continued in chunks 279–84.
This code is written to file `local declarations`.

`b_address` initializes a symbol `q` to be located at `n` offset bytes from the address of symbol `p`. The relocatable address of `q` is initialized to the relocatable address of `p` plus the given offset.

279a ⟨*definitions* 279a⟩≡
```
static void b_address(q, p, n) Symbol q, p; {
    q->x.offset = p->x.offset + n;
    if (p->scope == GLOBAL || p->sclass == STATIC || p->sclass == EXTERN) {
        assert(msym(p).raddr);
        if (msym(q).raddr != (RAddr)0)
            assert(msym(q).raddr->label == msym(p).raddr->label &&
                   msym(q).raddr->offset ==  msym(p).raddr->offset + n);
        else
            msym(q).raddr = addr_new(msym(p).raddr->label, msym(p).raddr->offset + n);
    }
}
```
This definition is continued in chunks 279–85.
This code is written to file `definitions`.

`b_def` defines symbol `s` at the current location counter in the current relocatable block.

279b ⟨*definitions* 279a⟩+≡
```
static unsigned b_def(s) Symbol s; {
    if (!msym(s).raddr)
        msym(s).raddr = addr_new(label_new(s->name), 0);
    if (!block_defined(msym(s).raddr->label->block))
        label_define(msym(s).raddr->label, msym(s).raddr->offset);
    else  warning("Attempt to redefine %s\n", s->name);
    if (location_known(msym(s).raddr))
        symeaddr(s) = location(msym(s).raddr);
    else symeaddr(s) = msym(s).raddr->offset;
    if (mapflag) fprint(2,"%s: %x offset %d\n",
        s->name, symeaddr(s), msym(s).raddr->label->offset + msym(s).raddr->offset);
    return symeaddr(s);
}
```

`b_defaddress` emits the relocatable address of symbol `p`. `UseSymAddrs` is set if we're emitting link-time addresses in data segments.

279c ⟨*local declarations* 278c⟩+≡
```
static int UseSymAddrs = 0;
```

279d ⟨*definitions* 279a⟩+≡
```
static void b_defaddress(p) Symbol p; {
    if (UseSymAddrs) {
        if (p->type && isfunc(p->type))
            emitu((unsigned)p);
        else emit_raddr(msym(p).raddr);
    } else emit_raddr(msym(p).raddr);
    if (mapflag) fprint(2,"relocate %s\n", p->name);
}
static void b_end(s1) Symbol s1; {
}
```

`b_entry` begins a new function by creating a new relocatable block in the text segment; we should assert that `cseg` = `text`.

280a ⟨*definitions* 279a⟩+≡
```
static void b_entry(f) Symbol f; {
    b_newtextblock(max(64,mfunc(f)->dagsize), f->name);
    segments[cseg].names[segments[cseg].crb] = f->name;
}
```

280b ⟨*local declarations* 278c⟩+≡
```
static int cseg = 0;
static RBlock textcrb = 0;
```

280c ⟨*definitions* 279a⟩+≡
```
static void b_global(p) Symbol p; {
    b_symbol(p->u.seg, p);
}

unsigned b_pc() {
    RBlock savcrb = crb();
    unsigned pc;

    set_block(textcrb);
    assert(cur_pc_known());
    pc = cur_pc();
    set_block(savcrb);
    return pc;
}
```

`use_reg` asserts that `u` is a register and returns its register number.

280d ⟨*local declarations* 278c⟩+≡
```
dclproto(static unsigned use_reg,(unsigned));
```

280e ⟨*definitions* 279a⟩+≡
```
static unsigned use_reg(u) unsigned u; {
    Symbol s = (Symbol) u;
    assert(s->x.regnode);
    return s->x.regnode->number;
}
```

280f ⟨*local macro definitions* 280f⟩≡
```
#define register_symbol(p)    (((Symbol)p)->x.regnode)
```
This code is written to file `local macro definitions`.

13.1 Segments

A `Segment` is a named array of relocatable blocks located at the base address `baseaddr`. Data segments usually contain only one relocatable block, because their maximum size is precomputed at startup; text segments contain multiple relocatable blocks, one per procedure. `crb` is the index of the current relocatable block in `relocblocks`, an array of relocatable blocks. `maxcrb` is the maximum index of a relocatable block. `closures` a table of closures that refer to relocatable addresses in `relocblocks`.

`totsize` is the total size of the segment. `flag` contains any flags set for the segment.

281a ⟨*local declarations* 278c⟩+≡

```
typedef struct segment{
    char name[6];
    unsigned baseaddr;
    unsigned crb;
    unsigned maxcrb;
    int totsize;
    int flag;
    RBlock *relocblocks;
    MTable *closures;
    char **names;
} Segment;
extern Segment segments[];
extern int segmentmap[];

dclproto(static void b_datasegment,(int, unsigned, int));
dclproto(static void b_newtextblock,(int, char *));
dclproto(static int b_segsize,(int));
```

`b_datasegment` initializes the data segment `id`. If `use_lt_address` is set, the link-time address of the data segment is used as its base address. This is necessary for executing link-time algorithms that refer to data in the initialized data segments. In particular, the Modula-3 link-time type initialization in `mldpp` refers to initialized data.

281b ⟨*definitions* 279a⟩+≡

```
static void b_datasegment(id, baseaddr, use_lt_address) unsigned baseaddr; {
    int elems;

    elems = segments[id].maxcrb+1;
    segments[id].closures = (MTable *)alloc(elems * sizeof(MTable));
    newtable(&(segments[id].closures[0]), 0, sizeof(RClosure *), &permanent);
    segments[id].relocblocks = (RBlock *)alloc(elems * sizeof(RBlock *));
    bzero((char *)segments[id].relocblocks, elems * sizeof(RBlock *));
    segments[id].names = (char **)alloc(elems * sizeof(char *));
    segments[id].relocblocks[0] = block_new(segments[id].totsize);
    segments[id].relocblocks[0]->label->name = segments[id].name;
    if (use_lt_address) {
        ⟨compute the link-time address of the segment 282a⟩
    } else
        segments[id].baseaddr = baseaddr;
}
```

The link-time address of the segment is derived by emitting one word into the segment and then calling `lcaddr()` to return the address of the current location counter. To guarantee that the entire segment has been layed out in contiguous memory, we check the assertion that the base address and ending address are the appropriate distance apart.

282a ⟨*compute the link-time address of the segment* 282a⟩≡
```
set_block(segments[id].relocblocks[0]);
org(0);
emitu(0);
org(0);
segments[id].baseaddr = lcaddr();
org(segments[id].totsize-1);
assert(lcaddr() == segments[id].baseaddr+segments[id].totsize-1);
org(0);
```
This code is used in chunk 281b.

`b_newtextblock` creates a new relocatable block in the text segment with the given `size` and `name`.

282b ⟨*definitions* 279a⟩+≡
```
static void b_newtextblock(size, name) char *name; {
    int crb;

    assert(cseg == text);
    segments[text].crb += 1;
    crb = segments[text].crb;
    assert(crb < segments[text].maxcrb);
    newtable(&(segments[text].closures[crb]), 0, sizeof(RClosure *), &permanent);
    segments[text].relocblocks[crb] = block_new(roundup(size << 1, 4));
    segments[text].relocblocks[crb]->label->name = name;
    set_block(segments[text].relocblocks[crb]);
    set_address(segments[text].relocblocks[crb],
                textcrb ? b_pc() : segments[text].baseaddr);
    textcrb = segments[text].relocblocks[crb];
}
```

`b_segsize` returns the size of the segment in bytes.

282c ⟨*definitions* 279a⟩+≡
```
static int b_segsize(seg) int seg; {
    int sz, i;
    for (sz = 0, i = 0; segments[seg].relocblocks[i]; i++)
        sz += block_size(segments[seg].relocblocks[i]);
    return sz;
}
```

282d ⟨*emit segment sizes* 282d⟩≡
```
if (memflag) {
    for (i = 0; i < nsegs; i++)
        fprint(2,"%s size %d\n", segments[i].name, b_segsize(i));
}
```
This code is written to file `emit segment sizes`.

13.2 Assembly Code

282e ⟨*local declarations* 278c⟩+≡
```
dclproto(static Symbol b_newsym,(char *, int));
```

`b_newsym` creates a new symbol for assembly labels.

283a ⟨*definitions* 279a⟩+≡

```
static Symbol b_newsym(name, class) char *name; {
    Symbol sym = (Symbol) alloc(sizeof(struct symbol));
    BaseTable *bt = 0;

    BZERO(sym, struct symbol);
    sym->name = string(name);
    sym->scope = GLOBAL;
    sym->sclass = class;
    if (class != EXTERN) {
        resolvesym(sym, bt);
        msym(sym).u.func = (MFuncSym)alloc(MFNSYMSZ);
        BZERO(msym(sym).u.func, *(msym(sym).u.func));
        mfunc(sym)->asmcode = 1;
        mfunc(sym)->emitted = 1;
        mfunc(sym)->cached = 1;
    }
    return sym;
}
```

283b ⟨*local declarations* 278c⟩+≡

```
dclproto(void resolvemodule,(BaseTable *bt));
```

`b_resolveasmsyms` adds external symbols defined in the assembly code to the global symbol table and resolves references to external symbols that appear in the assembly code.

283c ⟨*definitions* 279a⟩+≡

```
void b_resolveasmsyms(aoutflag) {
    int i, j;
    BaseTable *bt;
    Symbol sym;

    ⟨initialize target-specific assembly code ??⟩
    if (aoutflag) {
        for (i = 0; Asmcode[i].symdef != (char *)0; i++) {
            sym = b_newsym(Asmcode[i].symdef, AUTO);
            for (j = 0; Asmcode[i].insts[j].op != LASTOP; j++) {
                if (Asmcode[i].insts[j].symname) {
                    sym = install_symbol(Asmcode[i].insts[j].symname, &bt);
                    if (sym == 0) {
                        sym = b_newsym(Asmcode[i].insts[j].symname, EXTERN);
                        bt = resolveextsym(&sym);
                    }
                    Asmcode[i].insts[j].symptr = sym;
                    if (bt && !bt->visited) resolvemodule(bt);
                }
            }
        }
    }
}
```

`b_summary` prints a summary of data collected during emission of binary code.

283d ⟨*definitions* 279a⟩+≡

```
void b_summary() {
/* fprint(2,"tk_reloc_addrs %d tk_undef_closures %d\n", tk_reloc_addrs, tk_undef_closures); */
}
```

13.3 Support for link-time initialization

`rt_addr` returns the run-time address of the global symbol with the given name.

284a ⟨*definitions* 279a⟩+≡
```
unsigned rt_addr(name) char *name; {
    BaseTable *bt;
    Symbol sym;

    assert(sym = install_symbol(name, &bt));
    assert(location_known(msym(sym).raddr));
    fprint(2,"rtaddr %s %x\n", name, location(msym(sym).raddr));
    return location(msym(sym).raddr);
}
```

13.4 Support for New Jersey Machine-Code Toolkit

This section implements the application procedures required by the New Jersey Machine-Code Toolkit.

`mc_alloc_closure` allocates a new closure that relocates an instruction in `dest_block` and adds it to the `closures` associated with the current relocatable block.

284b ⟨*definitions* 279a⟩+≡
```
RClosure mc_alloc_closure(size_in_bytes, dest_block, dest_lc)
    unsigned size_in_bytes; RBlock dest_block; unsigned dest_lc; {
    RClosure cl;

    cl = (RClosure)mc_alloc(size_in_bytes, RClosure_pool);
    assert(cl && dest_block);
    assert(dest_block == segments[cseg].relocblocks[segments[cseg].crb]);
    addtotable(RClosure *, &(segments[cseg].closures[segments[cseg].crb]), cl);
    return cl;
}
```

`dispatch` resolves all relocatable addresses and relocates instructions by applying all the closures associated with each segment. This is just too cool.

284c ⟨*definitions* 279a⟩+≡
```
void dispatch() {
    int seg, rb, i;
    RClosure cl;

    for (seg = 0; seg < nsegs; seg++)
        for (rb = 0; segments[seg].relocblocks && segments[seg].relocblocks[rb]; rb++)
            gentable(segments[seg].closures[rb], i, 0) {
                cl = tableentry(RClosure *, segments[seg].closures[rb], i);
                apply_closure(cl, cl_emitm, &tkwarning);
            }
}
```

`tkwarning` prints a warning generated by the toolkit.

284d ⟨*local declarations* 278c⟩+≡
```
dclproto(static void tkwarning,(char *, ...));
```

285a ⟨*definitions* 279a⟩+≡
```
static void tkwarning VARARGS((char *fmt, ...),(fmt, va_alist),char *fmt; va_dcl) {
        va_list ap;

        va_init(ap, fmt);
        if (wflag == 0) {
                error("");
                vfprint(2, fmt, ap);
        }
        va_end(ap);
}
```

Each binary emitter must export the following procedures. Their prototypes appear in `binary.h`.

285b ⟨*binary.h* 285b⟩≡
```
#ifndef INCLUDE_BINARYH
#define INCLUDE_BINARYH

dclproto(void b_aout,(int, int));
dclproto(unsigned b_pc,(void));
dclproto(void b_dataseginit,(int [], int));
dclproto(void b_resolveasmsyms,(int));
dclproto(void b_startup,(int, MTable *, int));
dclproto(void b_summary,(void));
dclproto(void dispatch,(void));
dclproto(unsigned rt_addr,(char *));

#endif
```
This code is written to file `binary.h`.

286a ⟨*local macro definitions* 286a⟩≡
This code is written to file `local macro definitions`.

`b_seg` sets the current segment to `seg`.

286b ⟨*local declarations* 286b⟩≡
```
dclproto(static void b_seg,(int));
```
This definition is continued in chunks 286 and 287b.
This code is written to file `local declarations`.

286c ⟨*definitions* 286c⟩≡
```
static void b_seg(seg) {
    assert(seg < 6);
    cseg = segmentmap[seg];
    assert(segments[cseg].relocblocks[segments[cseg].crb]);
    set_block(segments[cseg].relocblocks[segments[cseg].crb]);
    if (seg == text) textcrb = segments[cseg].relocblocks[segments[cseg].crb];
    if (mapflag) fprint(2, "%s:\n", segments[cseg].name);
}
```
This definition is continued in chunks 286 and 287.
This code is written to file `definitions`.

`b_symbol` defines the symbol `p` in segment `seg`.

286d ⟨*local declarations* 286b⟩+≡
```
dclproto(static void b_symbol,(int, Symbol));
```

286e ⟨*definitions* 286c⟩+≡
```
static void b_symbol(seg, p) Symbol p; {
    assert(p);
    b_seg(seg);
    align(p->type->align);
    b_def( p);
}
```

13.5 Segments for SPARC and Intel486

The `SPARC` and `Intel486` use the same segments: `data`, `bss`, and `text`, so they share the code for initializing those segments. The `MIPS` segments are more complex; they are initialized in the `MIPS`-specific code.

`segmentmap` maps abstract segments to real ones.

286f ⟨*local declarations* 286b⟩+≡
```
enum { text = 0, data, bss, nsegs };
/*              Segments  CODE, BSS, DATA, LIT, SYM */
int segmentmap[] = { -1, text, bss, data, text, data };
#define SYM LIT+1
#define HEADERSZ     sizeof(struct exec)
Segment segments[] = {
    { ".text", TXTADDR, -1, 0, 0, 0, 0 },
    { ".data", 0, 0, 1, 0, 0, 0 },
    { ".bss", 0, 0, 1, 0, 0, 0 } };
Segment segments[];
```

`b_dataseginit` creates the `data` and `bss` segments. These segments follow the `text` segment in memory; therefore their base addresses cannot be set until code for the entire program has been generated. `b_aout` sets their base addresses.

287a ⟨*definitions* 286c⟩+≡

```
void b_dataseginit(segmentsizes, use_lt_address) int segmentsizes[]; {

    UseSymAddrs = use_lt_address;
    fail = tkwarning;

    segments[data].totsize =
        roundup(segmentsizes[DATA] + segmentsizes[LIT] + segmentsizes[SYM], 16);
    b_datasegment(data, segments[data].baseaddr, use_lt_address);
    if (use_lt_address)
        set_address(segments[data].relocblocks[0], segments[data].baseaddr);
    fprint(2,"data %x, %x\n", segments[data].baseaddr, segments[data].totsize);

    segments[bss].totsize += segmentsizes[BSS];
    b_datasegment(bss, segments[bss].baseaddr, use_lt_address);
    if (use_lt_address)
        set_address(segments[bss].relocblocks[0], segments[bss].baseaddr);
    fprint(2,"bss %x, %x\n", segments[bss].baseaddr, segments[bss].totsize);
}
```

`use_const` returns the value of constant symbol `u`.

287b ⟨*local declarations* 286b⟩+≡

```
dclproto(static int use_const,(unsigned));
```

287c ⟨*definitions* 286c⟩+≡

```
static int use_const(u) unsigned u; {
    Symbol s = (Symbol) u;

    assert(s->scope == CONSTANTS);
    switch (ttob(unqual(s->type))) {
    case C: return s->u.c.v.uc;
    case S: return s->u.c.v.us;
    case U: return s->u.c.v.u;
    case I: return s->u.c.v.i;
    case P: return (int)s->u.c.v.p;
    default: assert(0);
    }
    return 0;
}
```

14 MIPS binary emission procedures

This section implements the target-dependent, back-end procedures that emit binary representations of code and data on the MIPS.

14.1 Code generation specification

The format for the binary code generator is the same on all three targets. There is a prologue and epilogue for `emitbin`. Sandwiched in the middle are the rules that map assembly-language templates to calls to binary-encoding procedures.

288a ⟨*definitions* 288a⟩≡
```
⟨emitbin prologue ??⟩
%}
⟨special rules 288c⟩
⟨generic rules 289⟩
%{
⟨emitbin epilogue ??⟩
```

This definition is continued in chunks 291–94, 296–99, and 301.
This code is used in chunk 303b.

288b ⟨*local declarations* 288b⟩≡
```
#define isreg(_x) register_symbol(_x)
#define reg(_x)   use_reg(_x)
```

This definition is continued in chunks 291–93, 295, 297, and 298a.
This code is used in chunk 303b.

The shell script `mkemit` takes the following rules and the assembly code generator for the MIPS as inputs and produces a new binary code generator by replacing each assembly template on the left side of a rule with the binary template on the right. If a left side is omitted, its right-hand side is inherited from the last non-empty rule; for these rules, `mkemit` substititutes the opcode of the assembly template for the name of the binary encoding procedure in the binary template.

288c ⟨*special rules* 288c⟩≡
```
addr: acon "%0"         addrcon(0, use_sym(%0))
"# arg\n"               (*IR->x._kids)(p, eruleno, kids), (*IR->x.emit2)(p), 0
"# argb %0\n"
"# asgnb %0 %1\n"
"# read register\n"
"# reg\n"
"# ret\n"
"# write register\n"
"%0"                    %0
"$%0"
"%1($%0)"               addrcon(reg(%0),use_sym(%1))
"($%0)"                 addrcon(reg(%0),0)
"%a"                    (unsigned)%a
"%a+%F($sp)"            addrcon(r_sp,b_offset(%a->x.offset + framesize))
"%a:\n"                 b_def((Symbol)%a)
```
This code is used in chunk 288a.

289 ⟨*generic rules* 289⟩≡

```
"addu $%c,$%0,%1\n"     (isreg(%1) ? addu(reg(%c),reg(%0),reg(%1)) : addiu(reg(%c),reg(%0),(short)use_const(%1))), %c
"and $%c,$%0,%1\n"      (isreg(%1) ? and(reg(%c),reg(%0),reg(%1)) : (u = use_const(%1), ea_base(u)==r_at ? (addiu(r_at,r_at,(short)ea_offset(u)), and(reg(%c),reg(%0),r_at)) : andi(reg(%c),reg(%0),u))), %c
"xor $%c,$%0,%1\n"
"or $%c,$%0,%1\n"
"subu $%c,$%0,%1\n"     (isreg(%1) ? subu(reg(%c),reg(%0),reg(%1)) : addiu(reg(%c),reg(%0),-(short)use_const(%1))), %c
"sll $%c,$%0,%1\n"      (isreg(%1) ? sllv(reg(%c),reg(%0),reg(%1)) : sll(reg(%c),reg(%0),use_sym(%1))), %c
"sra $%c,$%0,%1\n"
"srl $%c,$%0,%1\n"
"la $%c,%0\n"           addiu(reg(%c),ea_base(%0),(short)ea_offset(%0)), %c
"l.d $f%c,%0\n"         l_d(reg(%c),(short)ea_offset(%0),ea_base(%0)), nop(), %c
"l.s $f%c,%0\n"         lwc1(reg(%c),(short)ea_offset(%0),ea_base(%0)), nop(), %c
"s.d $f%1,%0\n"         s_d(reg(%1),(short)ea_offset(%0),ea_base(%0)), %1
"s.s $f%1,%0\n"         swc1(reg(%1),(short)ea_offset(%0),ea_base(%0)), %1
"lb $%c,%0\n"           lb(reg(%c),(short)ea_offset(%0),ea_base(%0)), nop(), %c
"lbu $%c,%0\n"
"lh $%c,%0\n"
"lhu $%c,%0\n"
"lw $%c,%0\n"
"sh $%1,%0\n"           sh(reg(%1),(short)ea_offset(%0),ea_base(%0)), %1
"sb $%1,%0\n"           sb(reg(%1),(short)ea_offset(%0),ea_base(%0)), %1
"sw $%1,%0\n"
```

This definition is continued in chunks 290 and 291a.
This code is used in chunk 288a.

290 ⟨*generic rules* 289⟩+≡

```
"div $%c,$%0,$%1\n"     tested_div(reg(%c),reg(%0),reg(%1)), %c
"divu $%c,$%0,$%1\n"
"rem $%c,$%0,$%1\n"
"remu $%c,$%0,$%1\n"
"mul $%c,$%0,$%1\n"     mul(reg(%c),reg(%0),reg(%1)),%c
"not $%c,$%0\n"         nor(reg(%c), 0,reg(%0)), %c
"negu $%c,$%0\n"        subu(reg(%c),0,reg(%0)), %c
"move $%c,$%0\n"        mov(reg(%c),reg(%0)),%c
"add.s $f%c,$f%0,$f%1\n"        add_s(reg(%c),reg(%0),reg(%1)), nop(), %c
"add.d $f%c,$f%0,$f%1\n"
"div.s $f%c,$f%0,$f%1\n"
"div.d $f%c,$f%0,$f%1\n"
"mul.s $f%c,$f%0,$f%1\n"
"mul.d $f%c,$f%0,$f%1\n"
"sub.s $f%c,$f%0,$f%1\n"
"sub.d $f%c,$f%0,$f%1\n"
"mov.d $f%c,$f%0\n"             mov_d(reg(%c),reg(%0)), %c
"mov.s $f%c,$f%0\n"
"neg.d $f%c,$f%0\n"
"neg.s $f%c,$f%0\n"
"sll $%c,$%0,24; sra $%c,$%c,24\n""     sll(reg(%c),reg(%0),24), sra(reg(%c),reg(%c),24), %c
"sll $%c,$%0,16; sra $%c,$%c,16\n"      sll(reg(%c),reg(%0),16), sra(reg(%c),reg(%c),16), %c
"and $%c,$%0,0xff\n"    andi(reg(%c),reg(%0),(unsigned)0xff), %c
"and $%c,$%0,0xffff\n"  andi(reg(%c),reg(%0),(unsigned)0xffff), %c
"cvt.d.s $f%c,$f%0\n"   cvt_d_s(reg(%c),reg(%0)),nop(), %c
"cvt.s.d $f%c,$f%0\n"
"mtc1 $%0,$f%c; cvt.d.w $f%c,$f%c\n"    mtc1(reg(%0),reg(%c)),nop(), cvt_d_w(reg(%c),reg(%c)), nop(), %c
"trunc.w.d $f2,$f%0,$%c; mfc1 $%c,$f2\n"        trunc(reg(%c),2,reg(%0)), %c
"jal %0\n"      (isreg(%0) ? jalr(reg(%0),31) : jal(msym((Symbol)%0).raddr)), nop(), 31
"b %0\n"        b(msym((Symbol)%0).raddr),nop(), 0
"j $%0\n"       jr(reg(%0)),nop(), %0
```

291a ⟨*generic rules* 289⟩+≡
```
"bne $%0,$%1,%a\n"      bne(reg(%0),reg(%1),msym(%a).raddr), nop(), 0
"beq $%0,$%1,%a\n"
"bge $%0,$%1,%a\n"      bge(reg(%0),reg(%1),msym(%a).raddr), 0
"bgeu $%0,$%1,%a\n"
"blt $%0,$%1,%a\n"
"bltu $%0,$%1,%a\n"
"bleu $%0,$%1,%a\n"
"ble $%0,$%1,%a\n"
"bgt $%0,$%1,%a\n"
"bgtu $%0,$%1,%a\n"
"c.eq.d $f%0,$f%1; bc1t %a\n"   c_eq_d(reg(%0),reg(%1)),nop(),bc1t(msym(%a).raddr),nop(), 0
"c.eq.s $f%0,$f%1; bc1t %a\n"
"c.le.d $f%0,$f%1; bc1t %a\n"
"c.le.s $f%0,$f%1; bc1t %a\n"
"c.lt.d $f%0,$f%1; bc1t %a\n"
"c.lt.s $f%0,$f%1; bc1t %a\n"
"c.eq.d $f%0,$f%1; bc1f %a\n"   c_eq_d(reg(%0),reg(%1)),nop(),bc1f(msym(%a).raddr),nop(),0
"c.eq.s $f%0,$f%1; bc1f %a\n"
"c.le.d $f%0,$f%1; bc1f %a\n"
"c.le.s $f%0,$f%1; bc1f %a\n"
"c.lt.d $f%0,$f%1; bc1f %a\n"
"c.lt.s $f%0,$f%1; bc1f %a\n"
""              0
"%"             0
```

14.2 Symbols and registers

The following functions use unsigned integers to represent effective addresses. This happens to work on the MIPS, but not on the other targets. These functions would be better implemented by toolkit constructors, but since this was the first target, this was a hack.

291b ⟨*local declarations* 288b⟩+≡
```
dclproto(static unsigned addrcon,(int, unsigned));
dclproto(static unsigned use_const,(unsigned));
dclproto(static unsigned use_sym,(unsigned));
```

`addrcon` returns the effective address of register `reg` and effective address `ea`.

291c ⟨*definitions* 288a⟩+≡
```
static unsigned addrcon(reg, ea) unsigned ea; { /* ea must be an effective address */
    if (ea_base(ea) == 0)
        return ea_addr(ea_offset(ea), reg);
    else if (reg != 0) {
        addu(r_at, ea_base(ea), reg);
        return ea_addr(ea_offset(ea), r_at);
    }
    return ea;
}
```

`b_bigconst` loads the low-order 16 bits of a constant into `$at` and returns the effective address `c[15:32]($at)`, where `c[15:32]` are the constant's high-order bits.

292a ⟨*local macro definitions* 292a⟩≡
```
#define b_bigconst(i)  (lui(r_at, (unsigned)(ea_offset((unsigned)(i>>16) + \
                        (ea_offset(i) & 0x8000 ? 1 : 0)))),\
                        ea_addr(ea_offset(i), r_at))
```

This definition is continued in chunk 296b.
This code is used in chunk 303b.

`use_const` returns the value of constant symbol `u`.

292b ⟨*definitions* 288a⟩+≡
```
static unsigned use_const(u) unsigned u; {
    Symbol s = (Symbol) u;

    assert(s->scope == CONSTANTS);
    switch (ttob(unqual(s->type))) {
    case C: return ea_offset(s->u.c.v.uc);
    case S: return ea_offset(s->u.c.v.us);
    case U: if (s->u.c.v.u <= SHRT_MAX) return ea_offset(s->u.c.v.u);
        else return b_bigconst(s->u.c.v.u);
    case I: return b_offset(s->u.c.v.i);
    case P: if ((unsigned)s->u.c.v.p <= SHRT_MAX) return ea_offset((unsigned)s->u.c.v.p);
        else return b_bigconst((unsigned)s->u.c.v.p);
    default: assert(0);
    }
    return 0;
}
```

`use_sym` returns the effective address of symbol `u`. If `u` is a constant, its value is returned. If it is a register, its register number is returned. Otherwise the address of `u` is loaded and its effective address is returned.

292c ⟨*definitions* 288a⟩+≡
```
static unsigned use_sym(u) unsigned u; {
    Symbol s = (Symbol) u;

    if (register_symbol(s)) return use_reg(u);
    else if (s->scope == CONSTANTS) return use_const(u);
    else return la(msym(s).raddr);
}
```

292d ⟨*local declarations* 288b⟩+≡
```
dclproto(static unsigned b_offset,(int));
dclproto(static void b_seg,(int));
dclproto(static void b_symbol,(int, Symbol));
```

`b_offset` returns the constant `i` if it is a short integer, otherwise an instruction to compute its value is emitted and its low-order 16 bits are returned.

292e ⟨*definitions* 288a⟩+≡
```
static unsigned b_offset(i) {
    if (i >= SHRT_MIN && i <= SHRT_MAX)
        return ea_offset(i);
    else return b_bigconst(i);
}
```

b_seg sets the current segment to **seg**.

293a ⟨*definitions* 288a⟩+≡

```
static void b_seg(seg) {
    assert(seg < 6);
    cseg = segmentmap[seg];
    if (cseg == data && sdataonly)
        cseg = sdata;
    else if (cseg == bss && sbssonly)
        cseg = sbss;
    assert(segments[cseg].relocblocks[segments[cseg].crb]);
    set_block(segments[cseg].relocblocks[segments[cseg].crb]);
    if (seg == text) textcrb = segments[cseg].relocblocks[segments[cseg].crb];
}
```

b_symbol defines the symbol **p** in segment **seg**. It attempts to place the symbol in a "small" data segment.

293b ⟨*definitions* 288a⟩+≡

```
static void b_symbol(seg, p) Symbol p; {
    int gnum;
    assert(p);
    if (seg == DATA) {
        if (sdataonly) cseg = sdata;
        else if (p->type->size > gnum || isarray(p->type)) cseg = data;
        else {     /* if there is room in the sdata section, define symbol there  */
            set_block(segments[sdata].relocblocks[segments[sdata].crb]);
            if ((p->type->size + lc()) < segments[sdata].totsize)
                cseg = sdata;
            else cseg = data;
        }
        set_block(segments[cseg].relocblocks[segments[cseg].crb]);
    }
    if (seg == BSS) {
        b_seg(BSS);
        align(4);
    } else if (p->type->align > 1)
        align(p->type->align);
    b_def(p);
}
```

14.3 Segments

segmentmap maps abstract to real segments.

293c ⟨*local declarations* 288b⟩+≡

```
enum { text = 0, rdata, data, sdata, sbss, bss, nsegs };
/*              Segments  CODE, BSS, DATA, LIT, SYM */
#define SYM LIT+1
int segmentmap[] = { -1, text, bss, data, rdata, rdata };
#define HEADERSZ     roundup(FILHSZ + AOUTHSZ + (SCNHSZ * nsegs), 16)
Segment segments[] = {
    { ".text", USRTEXT + HEADERSZ, -1, 0, 0, STYP_TEXT, 0 }, { ".rdata", USRDATA, 0, 1, 0, STYP_RDATA, 0 },
    { ".data", 0, 0, 1, 0, STYP_DATA, 0 }, { ".sdata", 0, 0, 1, -SHRT_MIN, STYP_SDATA, 0 },
    { ".sbss", 0, 0, 1, SHRT_MAX, STYP_SBSS, 0 }, { ".bss", 0, 0, 1, 0, STYP_BSS, 0 } };
static int sdataonly = 0, sbssonly = 0;
```

b_dataseginit initializes the sizes and addresses of MIPS data segments.

294a ⟨*definitions* 288a⟩+≡
```
void b_dataseginit(segmentsizes, use_lt_address) int segmentsizes[]; {

    UseSymAddrs = use_lt_address;

    ⟨initialize rdata 294b⟩

    /* Allocate all data from small regions if possible */
    if (sdataonly = (segmentsizes[DATA] <= segments[sdata].totsize))
        warning("mld: data allocated from sdata region.\n");
    else segments[data].totsize = roundup(segmentsizes[DATA],4);

    ⟨initialize data 294c⟩
    ⟨initialize sdata 294d⟩
    ⟨initialize sbss 294e⟩
    ⟨initialize bss 295a⟩
}
```

294b ⟨*initialize* rdata 294b⟩≡
```
segments[rdata].totsize = roundup(segmentsizes[LIT] + segmentsizes[SYM], 16);
b_datasegment(rdata, segments[rdata].baseaddr, use_lt_address);
set_address(segments[rdata].relocblocks[0], segments[rdata].baseaddr);
```
This code is used in chunk 294a.

294c ⟨*initialize* data 294c⟩≡
```
b_datasegment(data, segments[rdata].baseaddr +
                    segments[rdata].totsize, use_lt_address);
set_address(segments[data].relocblocks[0], segments[data].baseaddr);
```

This code is used in chunk 294a.

$gp = the address of the end of sdata.

294d ⟨*initialize* sdata 294d⟩≡
```
b_datasegment(sdata, segments[data].baseaddr +
                     segments[data].totsize, use_lt_address);
set_address(segments[sdata].relocblocks[0], segments[sdata].baseaddr);
set_register(segments[sdata].relocblocks[0], r_gp,
segments[sdata].totsize);
```

This code is used in chunk 294a.

$gp = the address of the start of sbss.

294e ⟨*initialize* sbss 294e⟩≡
```
b_datasegment(sbss, segments[sdata].baseaddr +
                    segments[sdata].totsize, use_lt_address);
set_address(segments[sbss].relocblocks[0], segments[sbss].baseaddr);
set_register(segments[sbss].relocblocks[0], r_gp, 0);
```
This code is used in chunk 294a.

295a ⟨*initialize* `bss` 295a⟩≡

```
if (sbssonly = (segmentsizes[BSS] <= segments[sbss].totsize))
    warning("mld: bss data allocated from sbss region.\n");
else segments[bss].totsize += segmentsizes[BSS];
b_datasegment(bss, segments[sbss].baseaddr +
                   roundup(segments[sbss].totsize, 4), use_lt_address);
set_address(segments[bss].relocblocks[0], segments[bss].baseaddr);
```

This code is used in chunk 294a.

14.4 Synthesized instructions

The MIPS code generator emits synthesized assembly instructions which are assembled by the native assembler. For each machine instruction, the `tookit` generator provides a procedure that emits the binary encoding of the instruction. Currently, the `tookit` does not support synthesized instructions that refer to explicit labels. Until this feature is supported, instructions that refer to labels implemented by hand.

`la()` implements the load address assembly instruction. If the relocatable address of the symbol is unknown, a closure is created to emit the appropriate instruction when the address becomes known.

295b ⟨*local declarations* 288b⟩+≡

```
dclproto(static unsigned la,(RAddr));
dclproto(static void la_app,(RClosure, Emitter, FailCont));
dclproto(static void la_relocfn,(RClosure, RelocCallback, void *));
```

295c ⟨*local declarations* 288b⟩+≡

```
typedef struct la_closure {
  ClosureHeader h;
  ClosureLocation loc;
  struct { RAddr reloc; } v;
}*la_Closure;

static int lblrefs = 0;
```

`la` loads the address of the relocatable address `ra`. A closure is created if the relocatable address is unknown. Otherwise, its effective address is returned.

296a ⟨*definitions* 288a⟩+≡
```
static unsigned la(RAddr reloc) {
    la_Closure _c;
    unsigned loc, addr;
    int offset;

    lblrefs++;
    if (!location_known(reloc)) {
        static struct closure_header _h = { la_app, la_relocfn, 1, sizeof *_c };
        _c = (la_Closure) mc_alloc_closure(sizeof *_c, crb(), lc());
        _c->loc.dest_block = crb();  /* not(!) initialized by alloc_closure! */
        _c->loc.dest_lc    = lc();
        _c->h = &_h;
        _c->v.reloc = reloc;
        return b_bigconst(0);
    } else {
        loc = location(reloc);
        if (addr = block_regaddr(reloc->label->block)) {
            offset = loc - addr;
            if (offset >= SHRT_MIN && offset <= SHRT_MAX)
                return ea_addr(ea_offset(loc - addr), block_reg(reloc->label->block));
            else return b_bigconst(loc);
        } else return b_bigconst(loc);
    }
}
```

296b ⟨*local macro definitions* 292a⟩+≡
```
#define xfunct(_funct)    (_funct & 0x0000003f)
#define xoffset(_o)    (_o & 0x0000ffff)
#define xop(_op)    (_op & 0xfc000000)
#define xrt(_rt)    ((_rt >> 16) & 0x0000001f)
```

296c ⟨*definitions* 288a⟩+≡
```
static void la_relocfn(RClosure c, RelocCallback f, void *closure) {
  la_Closure _c = (la_Closure) c;
  (*f)(closure, _c->v.reloc);
  }
```

297a ⟨*definitions* 288a⟩+≡

```
static void la_app(RClosure c, Emitter emitter, FailCont fail) {
    la_Closure _c = (la_Closure)c;
    unsigned src, dst, newdst, loc, curlc;

    src = location(_c->v.reloc);
curlc = loc = _c->loc.dest_lc;
block_copy((unsigned char *)&dst, _c->loc.dest_block, curlc, sizeof(dst));
    assert(xop(dst) == o_lui && (xrt(dst) == r_at || xrt(dst) == r_gp) && xoffset(dst) == 0);
    newdst = dst | (ea_offset(src >> 16) + (ea_offset(src) & 0x8000 ? 1 : 0));
    curlc += 4;
block_copy((unsigned char *)&dst, _c->loc.dest_block, curlc, sizeof(dst));
    if (xfunct(dst) == o_addu) {
        curlc += 4;
block_copy((unsigned char *)&dst, _c->loc.dest_block, curlc, sizeof(dst));
    }
    dst = dst | ea_offset(src);
cl_emitm(_c->loc.dest_block, curlc, dst, sizeof(dst));
cl_emitm(_c->loc.dest_block, loc, newdst, sizeof(newdst));
}
```

`b_emit` emits one instruction into the instruction stream.

297b ⟨*local declarations* 288b⟩+≡

```
dclproto(static unsigned b_emit,(unsigned, int, int, int, int));
```

297c ⟨*definitions* 288a⟩+≡

```
static unsigned b_emit(op, rs, rt, rd, immed) unsigned op;{
    switch (op) {
    case o_mtc1_d: mtc_d(rt, rd); break;
    case o_mfc1_d: mfc_d(rt, rd); break;
    case o_l_d:
        lwc1(rt,  immed, rs);
        lwc1(rt+1, immed+4, rs);
        break;
    case o_s_d:
        swc1(rt, immed, rs);
        swc1(rt+1, immed+4, rs);
        break;
    default:
        emitu(op | (rs<<21) | (rt<<16) | (rd<<11) | immed);
        break;
    }
    return rd;
}
```

14.5 Assembly and startup code

297d ⟨*local declarations* 288b⟩+≡

```
dclproto(static Symbol b_newsym,(char *, int));
static Symbol gpsym;
```

297e ⟨*initialize target-specific assembly code* 297e⟩≡

```
gpsym = b_newsym("_global_ptr", AUTO);
```

This code is written to file `initialize target-specific assembly code`.

`b_assemble` assembles the instructions represented by the assembly code in `acode`.

298a ⟨*local declarations* 288b⟩+≡
```
dclproto(static void b_assemble,(struct asmcode, int, MTable *));
```

298b ⟨*definitions* 288a⟩+≡
```
static void b_assemble(acode, nprocs, rootfns) struct asmcode acode; MTable *rootfns; {
    Symbol    sym, symuse;
    int i, offset;
    BaseTable *bt;

    assert(sym = install_symbol(string(acode.symdef), &bt));
    mfunc(sym)->cgnid = nprocs+1;
    b_def(sym);
    for (i = 0; acode.insts[i].op != (unsigned)-1; i++) {
        if (acode.insts[i].symname) {
            assert(symuse = acode.insts[i].symptr);
            switch ((unsigned)acode.insts[i].op) {
            case o_jal:
                addtotable(Symbol *, rootfns, symuse);
                jal(msym(symuse).raddr);
                mfunc(symuse)->ncalls += 1;
                break;
            case o_b:   b(msym(symuse).raddr); break;
            case o_beq:
                beq(acode.insts[i].rs, acode.insts[i].rt,msym(symuse).raddr);
                break;
            case o_bne:
                bne(acode.insts[i].rs, acode.insts[i].rt,msym(symuse).raddr);
                break;
            case o_ble: blez(acode.insts[i].rs, msym(symuse).raddr); break;
            case o_bgt: bgtz(acode.insts[i].rs, msym(symuse).raddr); break;
            case o_blt: bltz(acode.insts[i].rs, msym(symuse).raddr); break;
            case o_bge: bgez(acode.insts[i].rs, msym(symuse).raddr); break;
            case o_addui: case o_lb:  case o_lbu: case o_lh:  case o_lhu: case o_lw:
            case o_sb:    case o_sh:  case o_sw:  case o_lwl: case o_lwr: case o_swl:
            case o_swr:   case o_l_s: case o_l_d: case o_s_s: case o_s_d:
                offset = use_sym((unsigned)symuse);
                b_emit(acode.insts[i].op, acode.insts[i].rs,
                       acode.insts[i].rt, acode.insts[i].rd, offset);
                break;
            default: assert(0);
            }
        }
        else {
            if (acode.insts[i].op == o_b)
                offset = ((b_pc() & 0x0fffffff) >> 2) + acode.insts[i].immed;
            else offset = ea_offset(acode.insts[i].immed);
            b_emit(acode.insts[i].op, acode.insts[i].rs, acode.insts[i].rt, acode.insts[i].rd, offset);
        }
    }
}
```

`b_startup` emits the executable's startup code. Derived from /usr/lib/crt0.o.

299 ⟨*definitions* 288a⟩+≡

```
void b_startup(nprocs, rootfns, aoutflag) MTable *rootfns; {
    int totdatasz, i, j, elems;
    BaseTable *bt;


    static char *syscalls[] = { "access", "chmod", "close", "creat",
        "dup", "dup2", "execve", "fstat", "getdirentries",
        "getpagesize", "getpid", "getrlimit", "getrusage", "gettimeofday",
        "kill", "link", "lseek", "lstat", "mprotect", "open", "read",
        "setrlimit", "sigblock", "sigsetmask", "stat", "syscall",
        "write", "unlink", (char *)0 };
    static int syscallids[] = {
        SYS_access, SYS_chmod, SYS_close, SYS_creat, SYS_dup,
        SYS_dup2, SYS_execve,         SYS_fstat, SYS_getdirentries,
        SYS_getpagesize, SYS_getpid, SYS_getrlimit, SYS_getrusage, SYS_gettimeofday,
        SYS_kill, SYS_link, SYS_lseek, SYS_lstat, SYS_mprotect, SYS_open, SYS_read,
        SYS_setrlimit, SYS_sigblock, SYS_sigsetmask, SYS_stat, SYS_syscall,
        SYS_write, SYS_unlink, 0 };

    if (aoutflag) {
        b_symbol(DATA, install_symbol("environ", &bt));
        (*IR->space)(4);
        b_symbol(DATA, install_symbol("errno", &bt));
        (*IR->space)(4);
    }
    ⟨initialize text 300a⟩
    cseg = text;
    b_newtextblock(1024, string("asmcode"));
    if (aoutflag) {
        ⟨compute total size of data segments 300b⟩
        ⟨set sbrk address _end == end 300c⟩
        emitu(segments[rdata].baseaddr + totdatasz);

        b_seg(CODE);
        for (i = 0; Asmcode[i].symdef != 0; i++)    /* emit assembled assembly code */
            if (strcmp(Asmcode[i].symdef, "_std_syscall") == 0)
                for (j = 0; syscalls[j]; j++) {
                    Asmcode[i].symdef = syscalls[j];
                    b_newsym(Asmcode[i].symdef, AUTO);
                    Asmcode[i].insts[0].immed = syscallids[j];
                    b_assemble(Asmcode[i], nprocs, rootfns);
                }
            else b_assemble(Asmcode[i], nprocs, rootfns);
    }
    ⟨assign $gp to sdata/sbss boundary address 300d⟩
}
```

300a ⟨*initialize* `text` 300a⟩≡

```
segments[text].maxcrb = nprocs + 1;
elems = segments[text].maxcrb+1;
segments[text].closures = (MTable *)alloc(elems * sizeof(MTable));
bzero((char *)segments[text].closures, elems * sizeof(MTable));
segments[text].relocblocks = (RBlock *)alloc(elems * sizeof(RBlock *));
bzero((char *)segments[text].relocblocks, elems * sizeof(RBlock *));
segments[text].names = (char **)alloc(elems * sizeof(char *));
```

This code is used in chunk 299.

300b ⟨*compute total size of data segments* 300b⟩≡

```
totdatasz = roundup(segments[rdata].totsize + segments[data].totsize +
                    segments[sdata].totsize + segments[sbss].totsize +
                    segments[bss].totsize, 4096);
```

This code is used in chunk 299.

300c ⟨*set* `sbrk` *address _end == end* 300c⟩≡

```
b_symbol(DATA, install_symbol(string("_end"), &bt));
b_def(b_newsym("end", AUTO));
```

This code is used in chunk 299.

300d ⟨*assign* `$gp` *to sdata/sbss boundary address* 300d⟩≡

```
set_block(segments[sbss].relocblocks[0]);
org(0);
b_def(gpsym);
```

This code is used in chunk 299.

14.6 Executable format

`b_aout` produces an executable a.out file. Some of the code is just black magic. Since `mld` emits the entire executable, it can initialize fields to whatever values that work.

301 ⟨*definitions* 288a⟩+≡

```
  void b_aout(fd, aoutflag)  {
      int i, j, zerobufsz, idatasz;
      struct filehdr fh;
      struct aouthdr ah;
      struct scnhdr sh[nsegs];
      int nscns, scnptr;
      char *zerobuf;

      assert(fd);
      ⟨emit segment sizes ??⟩
      segments[text].totsize = b_segsize(text);

      if (aoutflag) dispatch();
      ⟨initialize file header 302a⟩

      segments[text].totsize = roundup(segments[text].totsize, 16);
      bzero((char *)&ah, AOUTHSZ);
      ah.magic = ZMAGIC;
      ah.vstamp = 0x020a;        /* ?? */

      ⟨set base address and size of text segment 302b⟩
      ⟨set base address and size of data segments 302c⟩

      ah.bss_start = ah.data_start + ah.dsize;
      ah.bsize = 0;

      ah.gprmask = 0xf3fffffe;    /* ?? */
      ah.gp_value = label_location(msym(gpsym).raddr->label);
      write(fd, (char *)&ah, AOUTHSZ);

      scnptr = HEADERSZ;
      for (i = 0; i < nsegs; i++) {
          if (segments[i].totsize == 0)
              continue;
          bzero((char *)&sh[i], SCNHSZ);
          strcpy(sh[i].s_name, segments[i].name);
          sh[i].s_paddr = sh[i].s_vaddr = segments[i].baseaddr;
          sh[i].s_size = segments[i].totsize;
          if (i < sbss) {
              sh[i].s_scnptr = scnptr;
              scnptr += segments[i].totsize;
              if (i == text) scnptr += ah.tsize - (segments[text].totsize + HEADERSZ);
          }
          sh[i].s_flags = segments[i].flag;
          write(fd, (char *)&sh[i], SCNHSZ);
      }
      lseek(fd, HEADERSZ, SEEK_SET);
      for (i = 0; i < nsegs; i++) {
          if (segments[i].totsize == 0)
```

```
            continue;
        if (i == text) {
            ⟨write text segment to fd 302d⟩
        } else if (i < sbss) {
            ⟨write data segment to fd 303a⟩
        }
    }
    write(fd, zerobuf, ah.dsize - idatasz);
}
```

302a ⟨*initialize file header* 302a⟩≡

```
for (nscns = 0, i = 0; i < nsegs; i++)
    if (segments[i].totsize > 0) nscns++;
bzero((char *)&fh, FILHSZ);
fh.f_magic = MIPSELMAGIC;
fh.f_nscns = nscns;
fh.f_opthdr = AOUTHSZ;
fh.f_flags = F_RELFLG | F_EXEC | F_LNNO | F_LSYMS;
write(fd, (char *)&fh, FILHSZ);
```

This code is used in chunk 301.

302b ⟨*set base address and size of* `text` *segment* 302b⟩≡

```
ah.text_start = USRTEXT;
ah.entry = segments[text].baseaddr;
ah.tsize = roundup(segments[text].totsize + HEADERSZ, 4096);
zerobufsz = ah.tsize - (segments[text].totsize + HEADERSZ);
```

This code is used in chunk 301.

302c ⟨*set base address and size of* `data` *segments* 302c⟩≡

```
ah.data_start = segments[rdata].baseaddr;
idatasz = segments[rdata].totsize + segments[data].totsize + segments[sdata].totsize;
ah.dsize = roundup(idatasz + segments[sbss].totsize + segments[bss].totsize, 4096);
zerobufsz = max(zerobufsz, ah.dsize - idatasz);
zerobuf = malloc(zerobufsz);
bzero(zerobuf, zerobufsz);
```

This code is used in chunk 301.

302d ⟨*write text segment to* `fd` 302d⟩≡

```
int totsz = 0;
unsigned addr;
```

This code is used in chunk 301.

303a ⟨*write data segment to* `fd` 303a⟩≡
```
int loc = tell(fd);
unsigned bsize = segments[i].relocblocks[0]->size;
assert(loc == sh[i].s_scnptr);
block_write(fd, segments[i].relocblocks[0], 0, bsize);
if (segments[i].totsize - bsize > 0)
    write(fd, zerobuf, segments[i].totsize - bsize);

assert(tell(fd) == sh[i].s_scnptr);
addr = segments[text].baseaddr;
for (j = 0; j <= segments[text].crb; j++) {
    unsigned bsize = block_size(segments[text].relocblocks[j]);
    assert(label_location(segments[text].relocblocks[j]->label) == addr);
    block_write(fd, segments[text].relocblocks[j], 0, bsize);
    totsz += bsize;
    addr = label_location(segments[text].relocblocks[j]->label) + bsize;
}
assert(roundup(totsz, 16) == segments[text].totsize);
write(fd, zerobuf, ah.tsize - (totsz + HEADERSZ));
```
This code is used in chunk 301.

303b ⟨*mips.binary* 303b⟩≡
```
%{
#include <stdio.h>
#include <c.h>
#include <mld.h>
#include <mclib.h>
#include <mldtk.h>
#include <tk.h>
#include </usr/include/mips/vmparam.h>
#include </usr/include/signal.h>
#include <filehdr.h>
#include <aouthdr.h>
#include <scnhdr.h>
#include <syscall.h>
#include <mips-config.h>
#include <mips-asmcode.h>
#include <resextsym.h>
#include <binary.h>

⟨local declarations 288b⟩
⟨local macro definitions 292a⟩
⟨definitions 288a⟩

%}
```
This code is written to file `mips.binary`.

15 SPARC binary emission procedures

This section implements the SPARC-specific back-end support procedures that emit a binary representation of code and data.

304a ⟨*local macro definitions* 304a⟩≡
This definition is continued in chunks 304b and 316f.
This code is used in chunk 318d.

15.1 Code generation specification

304b ⟨*local macro definitions* 304a⟩+≡
```
#define use_name(_c)    atoi(((Symbol)_c)->x.name)
#define reg(_x)         use_reg(_x)
```

304c ⟨*local declarations* 304c⟩≡
```
static Symbol divsym, udivsym, remsym, uremsym, mulsym, umulsym;
```
This definition is continued in chunks 307–9, 313, 314, and 316c.
This code is used in chunk 318d.

The format for the binary code generator is the same on all three targets. See the MIPS code-generation specification in Section 14.1 for a detailed description.

304d ⟨*definitions* 304d⟩≡
```
#define RegOrImm_Instance reg_or_imm_Instance
⟨emitbin prologue ??⟩
%}
⟨special rules 304e⟩
⟨generic rules 305a⟩
%{
⟨emitbin epilogue ??⟩
```
This definition is continued in chunks 308, 310, and 314–17.
This code is used in chunk 318d.

304e ⟨*special rules* 304e⟩≡
```
call: ADDRGP "%a"       new_relocfun(msym(%a).raddr)
call: addr "%0"         new_addrfun((address__Instance *)%0)
base: con13 "%0"        new_absolutea(use_const(%0))
rc: con13 "%0"          new_imode(use_const(%0))
base: reg "%%%0"        new_indirecta(reg(%0))
rc: reg "%%%0"          new_rmode(reg(%0))
rc5: reg "%%%0"         new_rmode(reg(%0))
rc5: CNSTI "%a"         new_imode(use_const((unsigned)%a))
stk: ADDRFP "set %a,%%%c\n"     set(%a->x.offset, reg(%c)), %c
stk: ADDRLP "set %a,%%%c\n"     set(%a->x.offset, reg(%c)), %c
"%a"                    (unsigned) %a  /* Symbol */
"%0"                    (unsigned) %0 /* (address__Instance *) */
"# reg\n"               (*IR->x._kids)(p, eruleno, kids), (*IR->x.emit2)(p), 0
"# ret\n"
"# read register\n"
"# write register\n"
"# LOADD\n"
"# NEGD\n"
"# ARGD\n"
"# ARGF\n"
"# ASGNB\n"
```

This code is used in chunk 304d.

305a ⟨*generic rules* 305a⟩≡
```
"set %0,%%g1\nstb %%%1,[%%fp+%%g1]\n"   set(%a->x.offset, 1), stb(reg(%1), indexa(r_fp, 1)), 0
"set %0,%%g1\nsth %%%1,[%%fp+%%g1]\n"
"set %0,%%g1\nst %%%1,[%%fp+%%g1]\n"
"set %0,%%g1\nstd %%f%1,[%%fp+%%g1]\n"
"set %0,%%g1\nst %%f%1,[%%fp+%%g1]\n"   set(%a->x.offset, 1), stf(reg(%1), indexa(r_fp, 1)), 0
"set %a,%%%c\nadd %%%c,%%fp,%%%c\n"     set(%a->x.offset, reg(%c)), add(reg(%c),rmode(r_fp),reg(%c)), %c
"add %0,%%fp,%%%c\n"    add(r_fp, imode(((Symbol)%0)->x.offset), reg(%c)), %c
"set %0,%%%c\n"         (((Symbol)%0)->scope == CONSTANTS ? set(use_const(%0), reg(%c)) : (setr(msym((Symbol)%0).raddr, reg(%c)))), %c
"set %a,%%%c\n"         setr(msym(%a).raddr, reg(%c)), %c
"%%%fp+%a"              new_dispa(r_fp, %a->x.offset)
"%%fp+%0"               new_dispa(r_fp, ((Symbol)%0)->x.offset)
"%%fp+%%%0"             new_indexa(r_fp, reg(%0))
"mov %%%0,%%%c\n"               mov(rmode(reg(%0)), reg(%c)), %c
"faddd %%f%0,%%f%1,%%f%c\n"     faddd(reg(%0), reg(%1), reg(%c)), %c
"fadds %%f%0,%%f%1,%%f%c\n"
"fdivd %%f%0,%%f%1,%%f%c\n"
"fdivs %%f%0,%%f%1,%%f%c\n"
"fmuld %%f%0,%%f%1,%%f%c\n"
"fmuls %%f%0,%%f%1,%%f%c\n"
"fsubd %%f%0,%%f%1,%%f%c\n"
"fsubs %%f%0,%%f%1,%%f%c\n"
```
This definition is continued in chunks 305–7.
This code is used in chunk 304d.

305b ⟨*generic rules* 305a⟩+≡
```
"fcmped %%f%0,%%f%1; nop; fbue" fcmped(reg(%0), reg(%1)), nop(), fbue(msym(%a).raddr), 0
"fcmpes %%f%0,%%f%1; nop; fbue"
"fcmped %%f%0,%%f%1; nop; fbuge"        fcmped(reg(%0), reg(%1)), nop(), fbuge(msym(%a).raddr), 0
"fcmpes %%f%0,%%f%1; nop; fbuge"
"fcmped %%f%0,%%f%1; nop; fbug" fcmped(reg(%0), reg(%1)), nop(), fbug(msym(%a).raddr), 0
"fcmpes %%f%0,%%f%1; nop; fbug"
"fcmped %%f%0,%%f%1; nop; fbule"        fcmped(reg(%0), reg(%1)), nop(), fbule(msym(%a).raddr), 0
"fcmpes %%f%0,%%f%1; nop; fbule"
"fcmped %%f%0,%%f%1; nop; fbul" fcmped(reg(%0), reg(%1)), nop(), fbul(msym(%a).raddr), 0
"fcmpes %%f%0,%%f%1; nop; fbul"
"fcmped %%f%0,%%f%1; nop; fbne" fcmped(reg(%0), reg(%1)), nop(), fbne(msym(%a).raddr), 0
"fcmpes %%f%0,%%f%1; nop; fbne"
"%0 %a; nop\n"                  nop(), 0
```

306a ⟨*generic rules* 305a⟩+≡

```
"fnegs %%f%0,%%f%c\n"           fnegs(reg(%0), reg(%c)), %c
"fmovs %%f%0,%%f%c\n"
"fdtos %%f%0,%%f%c\n"
"fstod %%f%0,%%f%c\n"
"neg %%%0,%%%c\n"               neg2(reg(%0), reg(%c)), %c
"not %%%0,%%%c\n"               not2(reg(%0), reg(%c)), %c
"%%%0+%%%1"     new_indexa(reg(%0), reg(%1))
"%%%0+%1"       new_dispa(reg(%0), use_const(%1))
"%%%0-%1"       new_dispa(reg(%0), -(use_const(%1)))
"sll %%%0,24,%%%c; sra %%%c,24,%%%c\n"  sll(reg(%0), imode(24), reg(%c)), sra(reg(%c), imode(24), reg(%c)),%c
"sll %%%0,16,%%%c; sra %%%c,16,%%%c\n"  sll(reg(%0), imode(16), reg(%c)), sra(reg(%c), imode(16), reg(%c)),%c
"st %%%0,[%%sp+4*%c+68]\n"      st(reg(%0), dispa(r_sp, 4*use_name(%c)+68)), 0
"st %%%0,[%%sp+64]; ld [%%sp+64],%%f%c; fitod %%f%c,%%f%c\n"    st(reg(%0),dispa(r_sp,64)), ldf(dispa(r_sp,64),reg(%c)), fitod(reg(%c),reg(%c)), %c
"fdtoi %%f%0,%%f0; st %%f0,[%%sp+64]; ld [%%sp+64],%%%c\n"      fdtoi(reg(%0),0), stf(0,dispa(r_sp,64)), ld(dispa(r_sp,64),reg(%c)), %c
"set 0xffff,%%g1; and %%%0,%%g1,%%%c\n"         set(0xffff, 1), and(reg(%0), rmode(1), reg(%c)), %c
```

306b ⟨*generic rules* 305a⟩+≡

```
"call .div,2; nop\n"    call__(msym(divsym).raddr), nop(), 2
"call .udiv,2; nop\n"   call__(msym(udivsym).raddr), nop(), 2
"call .rem,2; nop\n"    call__(msym(remsym).raddr), nop(), 2
"call .urem,2; nop\n"   call__(msym(uremsym).raddr), nop(), 2
"call .mul,2; nop\n"    call__(msym(mulsym).raddr), nop(), 2
"call .umul,2; nop\n"   call__(msym(umulsym).raddr), nop(), 2
"orn %%%0,%1,%%%c\n"    orn(reg(%0), *(RegOrImm_Instance *)%1, reg(%c)), %c
"add %%%0,%1,%%%c\n"
"and %%%0,%1,%%%c\n"
"andn %%%0,%1,%%%c\n"
"or %%%0,%1,%%%c\n"
"sll %%%0,%1,%%%c\n"
"sra %%%0,%1,%%%c\n"
"srl %%%0,%1,%%%c\n"
"sub %%%0,%1,%%%c\n"
"xor %%%0,%1,%%%c\n"
"xnor %%%0,%1,%%%c\n"
"and %%%0,0xff,%%%c\n"  and(reg(%0), imode(0xff), reg(%c)), %c
```

307a ⟨*generic rules* 305a⟩+≡

```
"cmp %%%0,%1; ble %a; nop\n"    cmp(reg(%0), *(RegOrImm_Instance *)%1), ble(msym(%a).raddr), nop(), 0
"cmp %%%0,%1; bleu %a; nop\n"   cmp(reg(%0), *(RegOrImm_Instance *)%1), bleu(msym(%a).raddr), nop(), 0
"cmp %%%0,%1; bl %a; nop\n"     cmp(reg(%0), *(RegOrImm_Instance *)%1), bl(msym(%a).raddr), nop(), 0
"cmp %%%0,%1; blu %a; nop\n"    cmp(reg(%0), *(RegOrImm_Instance *)%1), blu(msym(%a).raddr), nop(), 0
"cmp %%%0,%1; bne %a; nop\n"    cmp(reg(%0), *(RegOrImm_Instance *)%1), bne(msym(%a).raddr), nop(), 0
"cmp %%%0,%1; be %a; nop\n"     cmp(reg(%0), *(RegOrImm_Instance *)%1), be(msym(%a).raddr), nop(), 0
"cmp %%%0,%1; bge %a; nop\n"    cmp(reg(%0), *(RegOrImm_Instance *)%1), bge(msym(%a).raddr), nop(), 0
"cmp %%%0,%1; bgeu %a; nop\n"   cmp(reg(%0), *(RegOrImm_Instance *)%1), bgeu(msym(%a).raddr), nop(), 0
"cmp %%%0,%1; bg %a; nop\n"     cmp(reg(%0), *(RegOrImm_Instance *)%1), bg(msym(%a).raddr), nop(), 0
"cmp %%%0,%1; bgu %a; nop\n"    cmp(reg(%0), *(RegOrImm_Instance *)%1), bgu(msym(%a).raddr), nop(), 0
"stb %%%1,[%0]\n"       stb(reg(%1), *(address__Instance *)%0), 0
"sth %%%1,[%0]\n"
"st %%%1,[%0]\n"
"st %%f%1,[%0]\n"       stf(reg(%1), *(address__Instance *)%0), 0
"ldsb [%0],%%%c\n"      ldsb(*(address__Instance *)%0, reg(%c)), %c
"ldsh [%0],%%%c\n"
"ld [%0],%%%c\n"
"ldub [%0],%%%c\n"
"lduh [%0],%%%c\n"
"ld [%0],%%f%c\n"       ldf(*(address__Instance *)%0, reg(%c)), %c
"ld2 [%0],%%f%c\n"      ld2(*(address__Instance *)%0, reg(%c)), %c
"st2 %%f%1,[%0]\n"      st2(reg(%1), *(address__Instance *)%0), 0
"%a:\n"                 b_def((Symbol)%a), 0
"call %0; nop\n"        _call(*(Function_Instance *)%0), nop(), 15
"call %0; st %%%1,[%%sp+64]\n"  _call(*(Function_Instance *)%0), st(reg(%1), dispa(r_sp,64)), 0
"jmp %0; nop\n"         jmp(*(address__Instance *)%0), nop(), 0
"ba %0; nop\n"          ba(msym((Symbol)%0).raddr), nop(), 0
"ldd [%0],%%f%c\n"      lddf(*(address__Instance *)%0, reg(%c)), %c
"std %%f%1,[%0]\n"      stdf(reg(%1), *(address__Instance *)%0), %0
""              0
"%"             0
```

The following functions return dynamically allocated typed constructors, which are used as operands to instructions and are returned by `emitbin` during code generation. They are allocated in the temporary arena, which is freed after code generation for each procedure. `emitbin` cannot directly return typed constructors produced by the toolkit procedures, because those constructors are implemented as structures. All these wrappers should be generated automatically.

307b ⟨*local declarations* 304c⟩+≡

```
dclproto(static address__Instance *new_absolutea,(int));
dclproto(static address__Instance *new_indirecta,(int));
dclproto(static address__Instance *new_dispa,(int, int));
dclproto(static address__Instance *new_indexa,(int, int));
dclproto(static Function_Instance *new_relocfun,(RAddr));
dclproto(static Function_Instance *new_addrfun,(address__Instance *));
```

308a ⟨*definitions* 304d⟩+≡
```
static address__Instance *new_absolutea(immed) {
    address__Instance *addr = (address__Instance *)talloc(sizeof(*addr));
    *addr = absolutea(immed);
    return addr;
}
static address__Instance *new_indirecta(reg) {
    address__Instance *addr = (address__Instance *)talloc(sizeof(*addr));
    *addr = indirecta(reg);
    return addr;
}
static address__Instance *new_dispa(reg, immed) {
    address__Instance *addr = (address__Instance *)talloc(sizeof(*addr));
    *addr = dispa(reg, immed);
    return addr;
}
static address__Instance *new_indexa(reg1, reg2) {
    address__Instance *addr = (address__Instance *)talloc(sizeof(*addr));
    *addr = indexa(reg1, reg2);
    return addr;
}
static Function_Instance *new_relocfun(reloc) RAddr reloc; {
    Function_Instance *f = (Function_Instance *)talloc(sizeof(*f));
    *f = relocfun(reloc);
    return f;
}
static Function_Instance *new_addrfun(addr) address__Instance *addr; {
    Function_Instance *f = (Function_Instance *)talloc(sizeof(*f));
    *f = addrfun(*addr);
    return f;
}
```

308b ⟨*local declarations* 304c⟩+≡
```
dclproto(static reg_or_imm_Instance *new_rmode,(int));
dclproto(static reg_or_imm_Instance *new_imode,(int));
```

308c ⟨*definitions* 304d⟩+≡
```
static reg_or_imm_Instance *new_rmode(reg) {
    reg_or_imm_Instance *op2 = (reg_or_imm_Instance *)talloc(sizeof(*op2));
    *op2 = rmode(reg);
    return op2;
}
static reg_or_imm_Instance *new_imode(immed) {
    reg_or_imm_Instance *op2 = (reg_or_imm_Instance *)talloc(sizeof(*op2));
    *op2 = imode(immed);
    return op2;
}
```

15.2 Segments

The SPARC and Intel 486 share common segment implementations, in Section 13.5.

Some constants needed by `b_dataseginit`.

308d ⟨*local declarations* 304c⟩+≡
```
#define PAGSIZ  USRTEXT
#define TXTADDR (USRTEXT + HEADERSZ)
```

15.3 Assembly and startup code

This section implements a simple assembler for the SPARC by calling toolkit-generated encoding procedures. All of this code should be generated automatically from a toolkit specification.

309a ⟨*initialize target-specific assembly code* 309a⟩≡
This code is written to file `initialize target-specific assembly code`.

`b_assemble` assembles instructions represented in structure `acode`.

309b ⟨*local declarations* 304c⟩+≡
```
dclproto(static void b_assemble,(struct asmcode, int, MTable *));
```

310 ⟨*definitions* 304d⟩+≡
```
static void b_assemble(acode, nprocs, rootfns)
    struct asmcode acode; MTable *rootfns; {
    Symbol sym, symuse;
    int i, offset;
    address__Instance Address;
    reg_or_imm_Instance Op2;
    Opfn op;
    BaseTable *bt;

    assert(sym = install_symbol(acode.symdef, &bt));
    mfunc(sym)->cgnid = nprocs+1;
    b_def(sym);
    for (i = 0; acode.insts[i].op != (Opfn)0; i++) {
        symuse = (Symbol)0;
        op = acode.insts[i].op;
        if (acode.insts[i].op == (Opfn)call__) {
            assert(symuse = install_symbol(acode.insts[i].symname, &bt));
            addtotable(Symbol *, rootfns, symuse);
            mfunc(symuse)->ncalls += 1;
        }
        else if (acode.insts[i].symname)
            assert(symuse = acode.insts[i].symptr);
        switch (acode.insts[i].optag) {
            ⟨get assembly code operands 311a⟩
        }
        if (isfuntype(op, loads))
            ⟨loadg,loada,loadf,loadc,jmpl Address,rd 311b⟩
        else if (isfuntype(op, stores))
            ⟨storeg,storea,storef,storec rd, Address 311c⟩
        else if (isfuntype(op, addrsonly))
           ⟨ldfsr,ldcsr,stfsr,stcsr,stdfq,stdcq,trap,rett,flush,call_,clr_,clrb,clrh,jmp Address ??⟩
        else if (isfuntype(op, rdonly))
            ⟨rdy,rdpsr,rdwim,rdtbr,neg,not rd 311d⟩
        else if (isfuntype(op, rs1Op2))
            ⟨wry,wrpsr,wrwim,wrtbr,cmp rs1, Op2 311e⟩
        else if (isfuntype(op, alu))
            ⟨alu rs1, Op2, rd 311f⟩
        else if (isfuntype(op, branch) || op == (Opfn)call__) {
            ⟨branch^a,call reloc 311g⟩
        } else if (isfuntype(op, float2) || op == (Opfn)mov_ || op == (Opfn)neg2)
            ⟨float2,mov_,neg2 rs2, rd 311h⟩
        else if (isfuntype(op, float3))
            ⟨float3 rs1, rs2, rd (*acode.insts[i].op) 312a⟩
        else if (isfuntype(op, fcompare))
            ⟨fcompare rs1, rs2 312b⟩
        else if (isfuntype(op, valrd))
            ⟨sethi,dec,deccc,inc,inccc,set val, rd 312c⟩
        else if (op == (Opfn)unimp)
            ⟨unimp imm22 312d⟩
        else if (isfuntype(op, emptyfuns))
            ⟨stbar, nop, ret, restore_, save_ 312e⟩
        else if (op == (Opfn)btst)
            ⟨btset Op2, rs1  312f⟩
```

```
          else if (isfuntype(op, bittest))
              ⟨bset,bclr,btog, mov Op2, rd ??⟩
          else if (op == (Opfn)rdasr || op == (Opfn)not2)
              ⟨rdasr,not2 rs1, rd 312h⟩
          else if (op == (Opfn)setr || op == (Opfn)ldr) {
              ⟨setr,ldr reloc, rd 312i⟩
          } else if (op == (Opfn)str) {
              ⟨str rd, reloc 312j⟩
          } else if (op == (Opfn)tst)
              ⟨tst rs2  312k⟩
          else if (op == (Opfn)wrasr)
              ⟨wrasr rs1, Op2, rdi 312l⟩
          else { error("Cannot assemble unknown function\n"); assert(0); }
      }
  }
```

311a ⟨*get assembly code operands* 311a⟩≡

```
  case imodeTAG:  Op2 = imode(acode.insts[i].simm13); break;
  case rmodeTAG:  Op2 = rmode(acode.insts[i].rs2); break;
  case dispATAG:  Address = dispa(acode.insts[i].rs1, acode.insts[i].simm13); break;
  case indexATAG: Address = indexa(acode.insts[i].rs1, acode.insts[i].rs2); break;
  case indirectATAG: Address = indirecta(acode.insts[i].rs1); break;
  case absoluteATAG: Address = absolutea(acode.insts[i].simm13); break;
```

This code is used in chunk 310.

311b ⟨loadg,loada,loadf,loadc,jmpl *Address,rd* 311b⟩≡

```
  ((LoadFn)*acode.insts[i].op)(Address, acode.insts[i].rd);
```

This code is used in chunk 310.

311c ⟨storeg,storea,storef,storec *rd, Address* 311c⟩≡

```
  ((StoreFn)*acode.insts[i].op)(acode.insts[i].rd, Address);
  ⟨ldfsr,ldcsr,stfsr,stcsr,stdfq,stdcq,trap,rett,flush,call_,clr_,clrb,clrh,jmp Address ??⟩=a
  ((AddrFn)*acode.insts[i].op)(Address);
```

This code is used in chunk 310.

311d ⟨rdy,rdpsr,rdwim,rdtbr,neg,not *rd* 311d⟩≡

```
  ((RdFn)*acode.insts[i].op)(acode.insts[i].rd);
```

This code is used in chunk 310.

311e ⟨wry,wrpsr,wrwim,wrtbr,cmp *rs1, Op2* 311e⟩≡

```
  ((RsOp2Fn)*acode.insts[i].op)(acode.insts[i].rs1, Op2);
```

This code is used in chunk 310.

311f ⟨alu *rs1, Op2, rd* 311f⟩≡

```
  ((AluFn)*acode.insts[i].op)(acode.insts[i].rs1, Op2, acode.insts[i].rd);
```

This code is used in chunk 310.

311g ⟨branch^a,call *reloc* 311g⟩≡

```
  assert(symuse);
  ((BranchFn)*acode.insts[i].op)(msym(symuse).raddr);
```

This code is used in chunk 310.

311h ⟨float2,mov_,neg2 *rs2, rd* 311h⟩≡

```
  ((Float2Fn)*acode.insts[i].op)(acode.insts[i].rs2, acode.insts[i].rd);
```

This code is used in chunk 310.

312a ⟨*float3 rs1, rs2, rd (*acode.insts[i].op)* 312a⟩≡
```
((Float3Fn)*acode.insts[i].op)(acode.insts[i].rs1,
                               acode.insts[i].rs2,
                               acode.insts[i].rd);
```
This code is used in chunk 310.

312b ⟨*fcompare rs1, rs2* 312b⟩≡
```
((Float2Fn)*acode.insts[i].op)(acode.insts[i].rs1, acode.insts[i].rs2);
```
This code is used in chunk 310.

312c ⟨`sethi,dec,deccc,inc,inccc,set` *val, rd* 312c⟩≡
```
((Float2Fn)*acode.insts[i].op)(acode.insts[i].val, acode.insts[i].rd);
```
This code is used in chunk 310.

312d ⟨*unimp imm22* 312d⟩≡
```
((RdFn)*acode.insts[i].op)(acode.insts[i].val);
```
This code is used in chunk 310.

312e ⟨`stbar, nop, ret, restore_, save_` 312e⟩≡
```
(*acode.insts[i].op)();
```
This code is used in chunk 310.

312f ⟨`btset` *Op2, rs1* 312f⟩≡
```
((BitFn)*acode.insts[i].op)(Op2, acode.insts[i].rs1);
```
This code is used in chunk 310.

312g ⟨`bset,bclr,btog, mov` *Op2, rd* 312g⟩≡
```
((BitFn)*acode.insts[i].op)(Op2, acode.insts[i].rd);
```
This code is written to file `[[bset,bclr,btog, mov]] Op2, rd`.

312h ⟨`rdasr,not2` *rs1, rd* 312h⟩≡
```
((Float2Fn)*acode.insts[i].op)(acode.insts[i].rs1, acode.insts[i].rd);
```
This code is used in chunk 310.

312i ⟨`setr,ldr` *reloc, rd* 312i⟩≡
```
assert(symuse && msym(symuse).raddr);
((RelocRdFn)*acode.insts[i].op)(msym(symuse).raddr, acode.insts[i].rd);
```
This code is used in chunk 310.

312j ⟨*str rd, reloc* 312j⟩≡
```
assert(symuse);
((RdRelocFn)*acode.insts[i].op)(acode.insts[i].rd, msym(symuse).raddr);
```
This code is used in chunk 310.

312k ⟨*tst rs2* 312k⟩≡
```
((RdFn)*acode.insts[i].op)(acode.insts[i].rs2);
```
This code is used in chunk 310.

312l ⟨*wrasr rs1, Op2, rdi* 312l⟩≡
```
((AluFn)*acode.insts[i].op)(acode.insts[i].rs1, Op2, acode.insts[i].rd);
```
This code is used in chunk 310.

The following arrays group related constructors.

313 ⟨*local declarations* 304c⟩+≡

```
static Opfn loads[] = {
  (Opfn)ldsb,(Opfn) ldsh,(Opfn) ldub,(Opfn) lduh,(Opfn) ld,
  (Opfn) ldd,(Opfn) ldstub,(Opfn) swap_, (Opfn)ldsba,(Opfn) ldsha,
  (Opfn) lduba,(Opfn) lduha,(Opfn) lda,(Opfn) ldda,(Opfn) ldstuba,
  (Opfn) swapa,(Opfn)ldf,(Opfn) lddf,(Opfn) ldc,(Opfn) lddc,(Opfn) 0
};
static Opfn stores[] = {
  (Opfn)stb,(Opfn) sth,(Opfn) st,(Opfn) std,(Opfn) stba,(Opfn) stha,
  (Opfn) sta, (Opfn) stda,(Opfn) stf,(Opfn) stdf,(Opfn) stc,
  (Opfn) stdc,(Opfn) 0
};
static Opfn addrsonly[] = {
  (Opfn)ldfsr,(Opfn)ldcsr,(Opfn)stfsr,(Opfn)stcsr,(Opfn)stdfq,(Opfn)stdcq,
  (Opfn)rett,(Opfn)flush,(Opfn)call_,(Opfn)clr_,(Opfn)clrb,(Opfn)clrh,
  (Opfn)jmp, (Opfn)tn,(Opfn) te,(Opfn) tle,(Opfn) tl,(Opfn)tleu,(Opfn) tcs,
  (Opfn) tneg,(Opfn) tvs, (Opfn)ta,(Opfn) tne,(Opfn) tg,(Opfn) tge,
  (Opfn) tgu,(Opfn) tcc,(Opfn) tpos,(Opfn) tvc,(Opfn)0
};
static Opfn rdonly[] = { (Opfn)clr,(Opfn)rdy,(Opfn)rdpsr,(Opfn)rdwim,
                         (Opfn)rdtbr,(Opfn)neg,(Opfn)not,(Opfn)0 };
static Opfn rs1Op2[] = { (Opfn)wry,(Opfn)wrpsr,(Opfn)wrwim,(Opfn)wrtbr,
                         (Opfn)cmp,(Opfn)0 };
static Opfn alu[] = {
  (Opfn)and,(Opfn) andcc,(Opfn) andn,(Opfn) andncc,(Opfn) or,(Opfn)orcc,
  (Opfn) orn,(Opfn) orncc,(Opfn)xor,(Opfn) xorcc,(Opfn) xnor,(Opfn) xnorcc,
  (Opfn) sll,(Opfn) srl,(Opfn) sra, (Opfn)add,(Opfn) addcc,(Opfn)addx,
  (Opfn) addxcc,(Opfn) taddcc,(Opfn) taddcctv,(Opfn)sub,(Opfn) subcc,
  (Opfn) subx,(Opfn) subxcc,(Opfn) tsubcc,(Opfn) tsubcctv,
  (Opfn)mulscc,(Opfn) umul,(Opfn) smul,(Opfn) umulcc,(Opfn) smulcc,
  (Opfn) udiv,(Opfn) sdiv,(Opfn) udivcc,(Opfn) sdivcc,
  (Opfn)save,(Opfn) restore,(Opfn) 0
};
static Opfn branch[] = {
  (Opfn)b,(Opfn) ba,(Opfn) bn,(Opfn) bnz,(Opfn) bne,(Opfn) be,
  (Opfn) bg,(Opfn) ble,(Opfn) bge,(Opfn) bl,
  (Opfn)bgu,(Opfn) bleu,(Opfn) bcc,(Opfn) blu,(Opfn) bcs,(Opfn) bgeu,
  (Opfn) bpos,(Opfn) bneg,(Opfn) bvc,(Opfn) bvs,
  (Opfn)fba,(Opfn) fbn,(Opfn) fbu,(Opfn) fbg,(Opfn) fbug,(Opfn) fbl,
  (Opfn) fbul,(Opfn) fblg,
  (Opfn)fbne,(Opfn) fbe,(Opfn) fbue,(Opfn) fbge,(Opfn) fbuge,
  (Opfn) fble,(Opfn) fbule,(Opfn) fbo,
  (Opfn)cbn,(Opfn) cb123,(Opfn) cb12,(Opfn) cb13,(Opfn) cb1,
  (Opfn) cb23,(Opfn) cb2,(Opfn) cb3,
  (Opfn)cba,(Opfn) cb0,(Opfn) cb03,(Opfn) cb02,(Opfn) cb023,
  (Opfn) cb01,(Opfn) cb013,(Opfn) cb012,
  (Opfn)b_a,(Opfn) ba_a,(Opfn) bn_a,(Opfn) bnz_a,(Opfn) bne_a,
  (Opfn) be_a,(Opfn) bg_a,(Opfn) ble_a,(Opfn) bge_a,(Opfn) bl_a,
  (Opfn)bgu_a,(Opfn) bleu_a,(Opfn) bcc_a,(Opfn) blu_a,(Opfn) bcs_a,
  (Opfn) bgeu_a,(Opfn) bpos_a,(Opfn) bneg_a,(Opfn) bvc_a,(Opfn) bvs_a,
  (Opfn)fba_a,(Opfn) fbn_a,(Opfn) fbu_a,(Opfn) fbg_a,(Opfn) fbug_a,
  (Opfn) fbl_a,(Opfn) fbul_a,(Opfn) fblg_a,
  (Opfn)fbne_a,(Opfn) fbe_a,(Opfn) fbue_a,(Opfn) fbge_a,(Opfn)fbuge_a,
```

```
    (Opfn) fble_a,(Opfn) fbule_a,(Opfn) fbo_a,
    (Opfn)cbn_a,(Opfn) cb123_a,(Opfn) cb12_a,(Opfn) cb13_a,(Opfn) cb1_a,
    (Opfn) cb23_a,(Opfn) cb2_a,(Opfn) cb3_a,
    (Opfn)cba_a,(Opfn) cb0_a,(Opfn) cb03_a,(Opfn) cb02_a,(Opfn) cb023_a,
    (Opfn) cb01_a,(Opfn) cb013_a,(Opfn) cb012_a,(Opfn) 0
  };
  static Opfn float2[] = {
    (Opfn)fitos,(Opfn)fitod,(Opfn)fitoq,(Opfn)fstoi,(Opfn)fdtoi,(Opfn)fqtoi,
    (Opfn)fstod,(Opfn)fstoq,(Opfn)fdtos,(Opfn)fdtoq,(Opfn)fqtos,(Opfn)fqtod,
    (Opfn) fmovs,(Opfn) fnegs,(Opfn) fabss,(Opfn) fsqrts,
    (Opfn) fsqrtd,(Opfn) fsqrtq,(Opfn) 0
  };
  static Opfn float3[] = {
    (Opfn)fadds,(Opfn) faddd,(Opfn) faddq,(Opfn) fsubs,
    (Opfn) fsubd,(Opfn) fsubq,
    (Opfn)fmuls,(Opfn) fmuld,(Opfn) fmulq,(Opfn) fsmuld,
    (Opfn) fdmulq,(Opfn) fdivs,(Opfn) fdivd,(Opfn) fdivq,(Opfn) 0
  };
  static Opfn fcompare[] = { (Opfn)fcmps,(Opfn) fcmpd,(Opfn) fcmpq,
    (Opfn) fcmpes,(Opfn) fcmped,(Opfn) fcmpeq,(Opfn) 0 };
  static Opfn valrd[] = { (Opfn)sethi,(Opfn)dec,(Opfn)deccc,
                          (Opfn)inc,(Opfn)inccc,(Opfn)set,(Opfn)0};
  static Opfn emptyfuns[] = { (Opfn)stbar,(Opfn) nop,(Opfn) ret,(Opfn)retl,
                              (Opfn) restore_,(Opfn) save_,(Opfn) 0};
  static Opfn bittest[] = { (Opfn)bset,(Opfn)bclr,(Opfn)btog,(Opfn) mov,(Opfn) 0};
  #define isfuntype(_f, _x) isinfunlist(_f, _x)
```

314a ⟨*local declarations* 304c⟩+≡

```
  dclproto(static int isinfunlist,(Opfn, Opfn[]));
```

314b ⟨*definitions* 304d⟩+≡

```
  static int isinfunlist(f, l) Opfn f, l[]; {
      int i;
      for (i = 0; l[i]; i++)
          if (f == l[i]) return 1;
      return 0;
  }
```

314c ⟨*local declarations* 304c⟩+≡

```
  typedef void (*LoadFn)(address__Instance, int);
  typedef void (*StoreFn)(int, address__Instance);
  typedef void (*AddrFn)(address__Instance);
  typedef void (*RdFn)(int);
  typedef void (*RsOp2Fn)(int, reg_or_imm_Instance);
  typedef void (*AluFn)(int, reg_or_imm_Instance, int);
  typedef void (*BranchFn)(RAddr);
  typedef void (*Float2Fn)(int, int);
  typedef void (*Float3Fn)(int, int, int);
  typedef void (*BitFn)(reg_or_imm_Instance, int);
  typedef void (*RelocRdFn)(RAddr, int);
  typedef void (*RdRelocFn)(int, RAddr);
```

`b_startup` emits startup code derived from /usr/lib/crt0.o.

314d ⟨*local declarations* 304c⟩+≡

```
  static Symbol endsym;
```

315 ⟨*definitions* 304d⟩+≡

```
void b_startup(nprocs, rootfns, aoutflag) MTable *rootfns; int aoutflag; {
    int i, j, elems;
    BaseTable *bt;


    static char *syscalls[] = {
        "access", "chmod", "close", "creat", "dup", "dup2", "execve",
        "fstat", "getdtablesize", "getpagesize", "getpid", "getrlimit",
        "getrusage", "gettimeofday", "kill", "link", "lseek", "lstat",
        "mprotect", "open", "read", "setrlimit",
        "sigblock", "sigsetmask", "_sigvec", "stat", "wait4", "write",
        "unlink", (char *)0 };
    static int syscallids[] = {
        SYS_access, SYS_chmod, SYS_close, SYS_creat, SYS_dup, SYS_dup2, SYS_execve,
        SYS_fstat, SYS_getdtablesize, SYS_getpagesize, SYS_getpid,
        SYS_getrlimit, SYS_getrusage, SYS_gettimeofday,
        SYS_kill, SYS_link, SYS_lseek, SYS_lstat,
        SYS_mprotect, SYS_open, SYS_read, SYS_setrlimit,
        SYS_sigblock, SYS_sigsetmask, SYS_sigvec, SYS_stat, SYS_wait4,
        SYS_write, SYS_unlink, 0 };

    b_symbol(DATA, install_symbol("environ", &bt));
    (*IR->space)(4);
    b_symbol(DATA, install_symbol("errno", &bt));
    (*IR->space)(4);
    endsym = install_symbol("_end", &bt);
    b_symbol(DATA, endsym);             /* sbrk address _end == end */
    b_def(b_newsym("end", AUTO));       /* initialize end in b_aout */
    emitu(0);

    ⟨initialize text 316a⟩
    cseg = text;
    b_newtextblock(1024, string("asmcode"));

    if (aoutflag) {
        b_seg(CODE);
        for (i = 0; Asmcode[i].symdef != 0; i++) {  /* emit assembled assembly code */
            if (strcmp(Asmcode[i].symdef, "_std_syscall") == 0)
                for (j = 0; syscalls[j]; j++) {
                    Asmcode[i].symdef = syscalls[j];
                    b_newsym(Asmcode[i].symdef, AUTO);
                    Asmcode[i].insts[0].simm13 = syscallids[j];
                    b_assemble(Asmcode[i], nprocs, rootfns);
                }
            else b_assemble(Asmcode[i], nprocs, rootfns);
        }
        ⟨initialize software-implemented functions 316b⟩
    }
}
```

316a ⟨*initialize* `text` 316a⟩≡
```
segments[text].maxcrb = nprocs + 1;
elems = segments[text].maxcrb+1;
segments[text].closures = (MTable *)alloc(elems * sizeof(MTable));
bzero((char *)segments[text].closures, elems * sizeof(MTable));
segments[text].relocblocks = (RBlock *)alloc(elems * sizeof(RBlock *));
bzero((char *)segments[text].relocblocks, elems * sizeof(RBlock *));
segments[text].names = (char **)alloc(elems * sizeof(char *));
```

This code is used in chunk 315.

316b ⟨*initialize software-implemented functions* 316b⟩≡
```
divsym = install_symbol("_div", &bt);
udivsym = install_symbol("_udiv", &bt);
remsym = install_symbol("_rem", &bt);
uremsym = install_symbol("_urem", &bt);
mulsym = install_symbol("_mul", &bt);
umulsym = install_symbol("_umul", &bt);
```
This code is used in chunk 315.

15.4 Synthesized instructions

316c ⟨*local declarations* 304c⟩+≡
```
static void ld2(address__Instance, int);
static void st2(int, address__Instance);
```

316d ⟨*definitions* 304d⟩+≡
```
static void ld2(addr, reg) address__Instance addr; int reg; {
    ldf(addr, reg);
    ⟨construct a new Address 316e⟩
    ldf(addr, reg+1);
}
static void st2(reg, addr) int reg; address__Instance addr; {
    stf(reg, addr);
    ⟨construct a new Address 316e⟩
    stf(reg+1, addr);
}
```

316e ⟨*construct a new* `Address` 316e⟩≡
```
switch(addr.tag) {
    case absolutea_TAG: addr.u.absolutea.simm13 += 4; break;
    case dispa_TAG:     addr.u.dispa.simm13 += 4; break;
    case indirecta_TAG: addr = dispa(addr.u.indirecta.rs1, 4); break;
    case indexa_TAG:
    default:              error("Malformed Address\n");
                          assert(0);
}
```
This code is used in chunk 316d.

15.5 Executable format

`b_aout` generates an executable a.out file.

316f ⟨*local macro definitions* 304a⟩+≡
```
#define FILHSZ   sizeof(struct exec)
```

317a ⟨*definitions* 304d⟩+≡

```
void b_aout(fd, aoutflag)  {
     int i, j, zerobufsz, totsz = 0;
     struct exec fh;
     char *zerobuf;
     unsigned addr;

     assert(fd);
     ⟨emit segment sizes ??⟩

     ⟨initialize file header 317b⟩

     ⟨set base addresses of data & bss segments 317c⟩
     ⟨initialize end & _end to end of bss section 318a⟩

     zerobufsz = fh.a_text - segments[text].totsize - HEADERSZ;
     zerobufsz = max(zerobufsz, fh.a_data - block_size(segments[data].relocblocks[0]));
     zerobuf = malloc(zerobufsz);
     bzero(zerobuf, zerobufsz);

     if (aoutflag) dispatch();
     write(fd, (char *)&fh, FILHSZ);

     ⟨emit text segment 318b⟩
     ⟨emit data segment 318c⟩

}
```

317b ⟨*initialize file header* 317b⟩≡

```
bzero((char *)&fh, FILHSZ);
segments[text].totsize = b_segsize(text);
fh.a_text = roundup(segments[text].totsize + HEADERSZ, PAGSIZ);
fh.a_entry = segments[text].baseaddr;
fh.a_toolversion = 1;
fh.a_machtype = M_SPARC;
fh.a_magic = ZMAGIC;
```

This code is used in chunk 317a.

317c ⟨*set base addresses of data & bss segments* 317c⟩≡

```
segments[data].baseaddr = segments[text].baseaddr - HEADERSZ + fh.a_text;
if (mapflag) fprint(2,"data %x\n", segments[data].baseaddr);
set_address(segments[data].relocblocks[0], segments[data].baseaddr);
fh.a_data = roundup(block_size(segments[data].relocblocks[0]), PAGSIZ);

segments[bss].baseaddr = segments[data].baseaddr + fh.a_data;
if (mapflag) fprint(2,"bss %x\n", segments[bss].baseaddr);
set_address(segments[bss].relocblocks[0], segments[bss].baseaddr);
fh.a_bss = roundup(segments[bss].totsize, PAGSIZ);
```

This code is used in chunk 317a.

318a ⟨*initialize end & _end to end of bss section* 318a⟩≡

```
b_seg(DATA);
org(msym(endsym).raddr->label->offset + msym(endsym).raddr->offset );
emitu(segments[bss].baseaddr + fh.a_bss);
```

This code is used in chunk 317a.

318b ⟨*emit text segment* 318b⟩≡

```
addr = segments[text].baseaddr;
for (j = 0; j <= segments[text].crb; j++) {
    assert(label_location(segments[text].relocblocks[j]->label) == addr);
    block_write(fd, segments[text].relocblocks[j], 0,
                block_size(segments[text].relocblocks[j]));
    totsz += block_size(segments[text].relocblocks[j]);
    addr = label_location(segments[text].relocblocks[j]->label) +
           block_size(segments[text].relocblocks[j]);
}
assert(totsz == segments[text].totsize);
write(fd, zerobuf, fh.a_text - segments[text].totsize - HEADERSZ);
```

This code is used in chunk 317a.

318c ⟨*emit data segment* 318c⟩≡

```
block_write(fd, segments[data].relocblocks[0], 0,
            block_size(segments[data].relocblocks[0]));
write(fd, zerobuf, fh.a_data - block_size(segments[data].relocblocks[0]));
```

This code is used in chunk 317a.

318d ⟨*sparc.binary* 318d⟩≡

```
%{
#include <c.h>
#include <mld.h>
#include <mclib.h>
#include <mldtk.h>
#include <tk.h>
#include <tags.h>
#include <sys/exec.h>
#include <machine/vmparam.h>
#include <syscall.h>
#include <sparc-config.h>
#include <sparc-asmcode.h>
#include <resextsym.h>
#include <binary.h>

⟨local declarations 304c⟩
⟨local macro definitions 304a⟩
⟨definitions 304d⟩

%}
```

This code is written to file `sparc.binary`.

16 Intel486 binary emission procedures

This section implements the target-dependent, back-end support procedures that emit binary representations of code and data on the Intel 486.

16.1 Code generation specification

319a ⟨*local declarations* 319a⟩≡

```
#define ESP 4
#define EBP 5
```

This definition is continued in chunks 320, 322–24, and 326–30.
This code is used in chunk 332d.

The format for the binary code generator is the same on all three targets and is desribed in Section 14.1.

319b ⟨*definitions* 319b⟩≡

```
⟨emitbin prologue ??⟩
%}
⟨special rules 319c⟩
⟨generic rules 322a⟩
%{
⟨emitbin epilogue ??⟩
```

This definition is continued in chunks 326–31.
This code is used in chunk 332d.

319c ⟨*special rules* 319c⟩≡

```
acon: ADDRGP "%a"        new_relocdisp(msym(%a).raddr)
acon: CNSTC "%a"         new_constdisp(use_const((unsigned )%a))
acon: CNSTI "%a"         new_constdisp(use_const((unsigned )%a))
acon: CNSTP "%a"         new_constdisp(use_const((unsigned )%a))
acon: CNSTS "%a"         new_constdisp(use_const((unsigned )%a))
acon: CNSTU "%a"         new_constdisp(use_const((unsigned )%a))
stmt: LABELV "%a:\n"     b_def((Symbol)%a), 0
addrj: ADDRGP "%a"       new_call_l(msym(%a).raddr)
addrj: reg "*%0"         new_call_rm(regis(use_reg(%0)))
addrj: mem "*%0"         new_call_rm(e(*(Mem_Instance *)%0))
cmpf: INDIRD(addr) "l\t%0"      fcompl_r64(*(Mem_Instance *)%0),0
cmpf: INDIRF(addr) "s\t%0"      fcomps_r32(*(Mem_Instance *)%0),0
cmpf: CVFD(INDIRF(addr)) "s\t%0"        fcomps_r32(*(Mem_Instance *)%0),0
memf: INDIRD(addr) "l\t%0"      new_memf_long((Mem_Instance *)%0)
memf: INDIRF(addr) "s\t%0"      new_memf_short((Mem_Instance *)%0)
memf: CVFD(INDIRF(addr)) "s\t%0"        new_memf_short((Mem_Instance *)%0)
```

This definition is continued in chunks 319–21.
This code is used in chunk 319b.

`mr` produces an `Eaddr_Instance`.

319d ⟨*special rules* 319c⟩+≡

```
mr: reg "%0"    new_regis(use_reg(%0))
mr: mem "%0"    new_e(*(Mem_Instance *)%0)
baseaddr: ADDRGP "%a"           new_relocdisp(msym(%a).raddr)
index: reg "%0"                 new_index_ss(use_reg(%0),0)
```

`addr` produces a `Mem_Instance`.

320a ⟨*special rules* 319c⟩+≡
```
addr: baseaddr "%0"     new_direct(*(Disp_Instance *)%0)
```

`mem` also produces a `Mem_Instance`.

320b ⟨*special rules* 319c⟩+≡
```
reg: mr "\tmovl\t%0,%c\n"       mov_rml_o(*(Eaddr_Instance *)%0, use_reg(%c)), %c
reg: mem "\tmovl\t%0,%c\n"      mov_rml_o(*(Eaddr_Instance *)%0, use_reg(%c)), %c
reg: con "\tmovl\t%0,%c\n"      movl(use_const(%0), use_reg(%c)), %c

rc: reg "%0"    new_mrc_mem(new_regis(use_reg(%0)))
rc: con "%0"    new_mrc_con((Symbol)%0)
mrc: mem "%0"   new_mrc_mem(new_e(*(Mem_Instance *)%0))

mrca: mem "%0"          new_mrc_mem(new_e(*(Mem_Instance *)%0))
mrca: ADDRGP "$%a"      new_mrc_mem(new_e(direct(relocdisp(msym(%a).raddr))))
rc5: CNSTI "$%a"        new_mrc_con((Symbol)%a)
rc5: reg "%%cl"         new_mrc_mem(new_regis(ECX))

"\tpushl\t%0\n"         (is_mrc_mem(%0) ? push_evl_o(mrc_mem(%0)) : push_ivl_o(mrc_con(%0))), 0

addrjmp: ADDRGP "%a"    jmp_jvl_o(msym(%a).raddr), 0
addrjmp: reg "*%0"      jmp_evl_o(regis(use_reg(%0))), 0
addrjmp: mem "*%0"      jmp_evl_o(e(*(Mem_Instance *)%0)), 0
```

The Intel has myriad overloaded assembly instructions. The following macros must distinguish between variants of `cmpl`.

320c ⟨*local declarations* 319a⟩+≡
```
#define cmpl_mrl(_mem, _mrc, _jumpi, _addr) \
    (is_mrc_mem((_mrc)) ? \
    cmp_mrl_o(mrc_reg((_mrc)), e(*(Mem_Instance *)(_mem))) : \
    cmp_il(mrc_con((_mrc)), e(*(Mem_Instance *)(_mem)))), \
    _jumpi(msym((_addr)).raddr), 0
```

320d ⟨*special rules* 319c⟩+≡
```
stmt: EQI(mem,rc) "\tcmpl\t%1,%0\n\tje\t%a\n"   cmpl_mrl(%0,%1,%a,jz)
stmt: GEI(mem,rc) "\tcmpl\t%1,%0\n\tjge\t%a\n"  cmpl_mrl(%0,%1,%a,jnl)
stmt: GTI(mem,rc) "\tcmpl\t%1,%0\n\tjg\t%a\n"   cmpl_mrl(%0,%1,%a,jnle)
stmt: LEI(mem,rc) "\tcmpl\t%1,%0\n\tjle\t%a\n"  cmpl_mrl(%0,%1,%a,jle)
stmt: LTI(mem,rc) "\tcmpl\t%1,%0\n\tjl\t%a\n"   cmpl_mrl(%0,%1,%a,jl)
stmt: NEI(mem,rc) "\tcmpl\t%1,%0\n\tjne\t%a\n"  cmpl_mrl(%0,%1,%a,jnz)
stmt: GEU(mem,rc) "\tcmpl\t%1,%0\n\tjae\t%a\n"  cmpl_mrl(%0,%1,%a,jnb)
stmt: GTU(mem,rc) "\tcmpl\t%1,%0\n\tja \t%a\n"  cmpl_mrl(%0,%1,%a,jnbe)
stmt: LEU(mem,rc) "\tcmpl\t%1,%0\n\tjbe\t%a\n"  cmpl_mrl(%0,%1,%a,jbe)
stmt: LTU(mem,rc) "\tcmpl\t%1,%0\n\tjb \t%a\n"  cmpl_mrl(%0,%1,%a,jb)
```

320e ⟨*local declarations* 319a⟩+≡
```
#define cmpl_rml(_reg, _mrc, _jumpi, _addr) \
    (is_mrc_mem((_mrc)) ? \
    cmp_rml_o(mrc_mem((_mrc)), use_reg((_reg))) : \
    cmp_il(mrc_con((_mrc)), regis(use_reg((_reg))))), \
    _jumpi(msym((_addr)).raddr), 0
```

321a ⟨*special rules* 319c⟩+≡

```
stmt: EQI(reg,mrc) "\tcmpl\t%1,%0\n\tje\t%a\n"      cmpl_rml(%0,%1,%a,jz)
stmt: GEI(reg,mrc) "\tcmpl\t%1,%0\n\tjge\t%a\n"     cmpl_rml(%0,%1,%a,jnl)
stmt: GTI(reg,mrc) "\tcmpl\t%1,%0\n\tjg\t%a\n"      cmpl_rml(%0,%1,%a,jnle)
stmt: LEI(reg,mrc) "\tcmpl\t%1,%0\n\tjle\t%a\n"     cmpl_rml(%0,%1,%a,jle)
stmt: LTI(reg,mrc) "\tcmpl\t%1,%0\n\tjl\t%a\n"      cmpl_rml(%0,%1,%a,jl)
stmt: NEI(reg,mrc) "\tcmpl\t%1,%0\n\tjne\t%a\n"     cmpl_rml(%0,%1,%a,jnz)
stmt: GEU(reg,mrc) "\tcmpl\t%1,%0\n\tjae\t%a\n"     cmpl_rml(%0,%1,%a,jnb)
stmt: GTU(reg,mrc) "\tcmpl\t%1,%0\n\tja \t%a\n"     cmpl_rml(%0,%1,%a,jnbe)
stmt: LEU(reg,mrc) "\tcmpl\t%1,%0\n\tjbe\t%a\n"     cmpl_rml(%0,%1,%a,jbe)
stmt: LTU(reg,mrc) "\tcmpl\t%1,%0\n\tjb \t%a\n"     cmpl_rml(%0,%1,%a,jb)
```

321b ⟨*special rules* 319c⟩+≡

```
"# read register\n"     (*IR->x._kids)(p, rulenum, kids), (*IR->x.emit2)(p), 0
"# write register\n"
"# extend\n"
"# truncate\n"
"# ret\n"
"# retf\n"
"# retd\n"
"# CVFD\n"
```

322a ⟨*generic rules* 322a⟩≡
```
"(%0)"          new_disp32(constdisp(0),use_reg(%0))
"%1(%0)"        new_disp32(*(Disp_Instance *)%1, use_reg(%0))
"%a(%%ebp)"     new_disp32(relocdisp(msym(%a).raddr),EBP)
"(,%0)"         new_shortindex(constdisp(0),index_ss(%0,index),index_ss(%0,ss))
"%1(,%0)"       new_shortindex(*(Disp_Instance *)%1,index_ss(%0,index),index_ss(%0,ss))
"(%1,%0)"       new_indexr(use_reg(%1),index_ss(%0,index),index_ss(%0,ss))
"%0,%1"         new_index_ss(use_reg(%0),use_const(%1))
"\tcall\t%0\n\taddl\t$%a,%%esp\n"       callfn(*(Function_Instance *)%0), add_il(use_const((unsigned)%a), regis(ESP)), 0
"\tcall\t%0\n\tfstp\t%%st(0)\n"         callfn(*(Function_Instance *)%0), fstp_st_sti(0), 0
"\tcall\t%0\n"                          callfn(*(Function_Instance *)%0), 0
"\tcall\t%0\n\taddl\t$%a,%%esp\n\tfstp\t%%st(0)\n"      callfn(*(Function_Instance *)%0), add_il(use_const((unsigned)%a), regis(ESP)), fstp_st_sti(0), 0

"p"     fcompp(), 0
"\tfcomp%0 ; fstsw %%ax ; sahf\n\tje\t%a\n"     fstsw_ax(), sahf(), jz(msym(%a).raddr), 0
"\tfcomp%0 ; fstsw %%ax ; sahf\n\tjbe\t%a\n"    fstsw_ax(), sahf(), jbe(msym(%a).raddr), 0
"\tfcomp%0 ; fstsw %%ax ; sahf\n\tjb\t%a\n"     fstsw_ax(), sahf(), jb(msym(%a).raddr), 0
"\tfcomp%0 ; fstsw %%ax ; sahf\n\tjae\t%a\n"    fstsw_ax(), sahf(), jnb(msym(%a).raddr), 0
"\tfcomp%0 ; fstsw %%ax ; sahf\n\tja\t%a\n"     fstsw_ax(), sahf(), jnbe(msym(%a).raddr), 0
"\tfcomp%0 ; fstsw %%ax ; sahf\n\tjne\t%a\n"    fstsw_ax(), sahf(), jnz(msym(%a).raddr), 0

"\tleal\t%0,%c\n"       leal_o(*(Mem_Instance *)%0, use_reg(%c)), %c
"\tincl\t%1\n"          inc_evl_o(e(*(Mem_Instance *)%1)), 0
"\tdecl\t%1\n"          dec_evl_o(e(*(Mem_Instance *)%1)), 0
"\tnotl\t%1\n"          notl_o(e(*(Mem_Instance *)%1)), 0
"\tnegl\t%1\n"          negl_o(e(*(Mem_Instance *)%1)), 0

"1"     %a
"2"
"4"
"8"
"$%a"
"\tmovl\t%0,%c\n"       mov_rml_o(regis(use_reg(%0)), use_reg(%c)), %c
"\tmovw\t%0,%c\n"       mov_rmw_o(regis(use_reg(%0)), use_reg(%c)), %c
"\tmovb\t%0,%c\n"       movb_rm(regis(use_reg(%0)), use_reg(%c)), %c

"?\tmovl\t%0,%c\n\tnotl\t%c\n"  mov_rml_o(regis(use_reg(%0)), use_reg(%c)), notl_o(regis(use_reg(%c))), %c
"?\tmovl\t%0,%c\n\tnegl\t%c\n"  mov_rml_o(regis(use_reg(%0)), use_reg(%c)), negl_o(regis(use_reg(%c))), %c
```

This definition is continued in chunks 323–25.
This code is used in chunk 319b.

These macros implement the overloaded arithmetic assembly instructions.

322b ⟨*local declarations* 319a⟩+≡
```
#define arith_rml(_reg,_mrc,_res,_oprml,_opil) \
    mov_rml_o(regis(use_reg(_reg)), use_reg(_res)),
    (is_mrc_mem((_mrc)) ? \
        _oprml(mrc_mem((_mrc)), use_reg((_res))) : \
        _opil(mrc_con((_mrc)), regis(use_reg(_res)))), _res
```

323a ⟨*generic rules* 322a⟩+≡
```
"?\tmovl\t%0,%c\n\taddl\t%1,%c\n"       arith_rml(%0,%1,%c,add_rml_o,add_il)
"?\tmovl\t%0,%c\n\tsubl\t%1,%c\n"       arith_rml(%0,%1,%c,sub_rml_o,sub_il)
"?\tmovl\t%0,%c\n\tandl\t%1,%c\n"       arith_rml(%0,%1,%c,and_rml_o,and_il)
"?\tmovl\t%0,%c\n\torl\t%1,%c\n"        arith_rml(%0,%1,%c,or_rml_o,or_il)
"?\tmovl\t%0,%c\n\txorl\t%1,%c\n"       arith_rml(%0,%1,%c,xor_rml_o,xor_il)

"%%cl"  new_mrc_mem(new_regis(ECX))
```

These macros implement the overloaded arithmetic shift instructions.

323b ⟨*local declarations* 319a⟩+≡
```
#define shift_rml(_reg,_mrc,_res,_opreg,_opil) \
    mov_rml_o(regis(use_reg(_reg)), use_reg(_res)), \
    (is_mrc_mem(_mrc) ? \
    _opreg(regis(use_reg(_res))) : \
    _opil(mrc_con(_mrc),regis(use_reg(_res)))), _res
```

323c ⟨*generic rules* 322a⟩+≡
```
"?\tmovl\t%0,%c\n\tsall\t%1,%c\n"       shift_rml(%0,%1,%c,shl_b_ev_cll_o,shl_b_ev_ibl_o)
"?\tmovl\t%0,%c\n\tshll\t%1,%c\n"
"?\tmovl\t%0,%c\n\tsarl\t%1,%c\n"       shift_rml(%0,%1,%c,sar_b_ev_cll_o,sar_b_ev_ibl_o)
"?\tmovl\t%0,%c\n\tshrl\t%1,%c\n"       shift_rml(%0,%1,%c,shr_b_ev_cll_o,shr_b_ev_ibl_o)
```

323d ⟨*local declarations* 319a⟩+≡
```
#define shift_mrc(_mem,_reg2,_res,_opmem,_opil) \
    (is_mrc_mem(_reg2) ? \
    _opmem(e(*(Mem_Instance *) _mem)) : \
    _opil(mrc_con(_reg2), e(*(Mem_Instance *)_mem))), _res
```

323e ⟨*generic rules* 322a⟩+≡
```
"\tsall\t%2,%1\n"       shift_mrc(%1,%2,%c,shl_b_ev_cll_o,shl_b_ev_ibl_o)
"\tshll\t%2,%1\n"
"\tsarl\t%2,%1\n"       shift_mrc(%1,%2,%c,sar_b_ev_cll_o,sar_b_ev_ibl_o)
"\tshrl\t%2,%1\n"       shift_mrc(%1,%2,%c,shr_b_ev_cll_o,shr_b_ev_ibl_o)
```

323f ⟨*local declarations* 319a⟩+≡
```
#define arith_mrc(_mem,_mrc,_opmem,_opil)\
    (is_mrc_mem(_mrc) ? \
        _opmem(mrc_reg(_mrc), e(*(Mem_Instance *)_mem)) : \
        _opil(mrc_con(_mrc), e(*(Mem_Instance *)_mem))), 0
```

324a ⟨*generic rules* 322a⟩+≡
```
"\taddl\t%2,%1\n"       arith_mrc(%1,%2,add_mrl_o,add_il)
"\tsubl\t%2,%1\n"       arith_mrc(%1,%2,sub_mrl_o,sub_il)
"\tandl\t%2,%1\n"       arith_mrc(%1,%2,and_mrl_o,and_il)
"\torl\t%2,%1\n"        arith_mrc(%1,%2,or_mrl_o,or_il)
"\txorl\t%2,%1\n"       arith_mrc(%1,%2,xor_mrl_o,xor_il)

"\timul\t%0,%1,%c\n"    imul_ivl(use_const(%0), *(Eaddr_Instance *)%1, use_reg(%c)), %c
"\tmull\t%1\n"          mul_axl_o(*(Eaddr_Instance *)%1), %c
"\txorl\t%%edx,%%edx\n\tdivl\t%1\n"     xor_rml_o(regis(EDX), EDX), divl(regis(use_reg(%1))), %c

"\tcdq\n\tidivl\t%1\n"  cdq(), idivl(regis(use_reg(%1))), %c
"\tmovsbl\t%0,%c\n"     movsx_gv_ebl_o(e(*(Mem_Instance *)%0), use_reg(%c)), %c
"\tmovzbl\t%0,%c\n"     movzx_gv_ebl_o(e(*(Mem_Instance *)%0), use_reg(%c)), %c
"\tmovswl\t%0,%c\n"     movsx_gv_ew(e(*(Mem_Instance *)%0), use_reg(%c)), %c
"\tmovzwl\t%0,%c\n"     movzx_gv_ew(e(*(Mem_Instance *)%0), use_reg(%c)), %c

"\tfistpl\t%0\n"        fistpl_si32(xmemarg(*(Mem_Instance *)%0)), 0
"\tfildl\t%0\n"         fildl_si32(xmemarg(*(Mem_Instance *)%0)), 0

"\tfsubrp\t%%st,%%st(1)\n"      fsubrp_sti_st(1), 0
"\tfdivrp\t%%st,%%st(1)\n"      fdivrp_sti_st(1), 0
"\tfchs\n"                      fchs(), 0
"p\t%%st,%%st(1)"               new_memf_stack()
```

324b ⟨*local declarations* 319a⟩+≡
```
#define float_stack(_memf,_fop64,_fop32,_fstack)\
    (is_memf_long(_memf) ? \
    _fop64(memf_long(_memf)) : \
    (is_memf_short(_memf) ? _fop32(memf_short(_memf)) : _fstack(1))), 0
```

324c ⟨*generic rules* 322a⟩+≡
```
"\tfadd%1\n"            float_stack(%1,faddl_r64,fadds_r32,faddp_sti_st)
"\tfmul%1\n"            float_stack(%1,fmull_r64,fmuls_r32,fmulp_sti_st)
"\tfdiv%1\n"            float_stack(%1,fdivl_r64,fdivs_r32,fdivp_sti_st)
"\tfsub%1\n"            float_stack(%1,fsubl_r64,fsubs_r32,fsubp_sti_st)
```

324d ⟨*local declarations* 319a⟩+≡
```
#define mov_mr(_mem,_mrc,_movop,_opeb)\
    (is_mrc_mem(_mrc) ? \
    _movop(mrc_reg(_mrc), e(*(Mem_Instance *)_mem)) : \
    _opeb(mrc_con(_mrc), e(*(Mem_Instance *)_mem))), 0
```

325 ⟨*generic rules* 322a⟩+≡

```
"\tmovb\t%1,%0\n"       mov_mr(%0,%1,movb_mr,mov_eb_ibb)
"\tmovl\t%1,%0\n"       mov_mr(%0,%1,mov_mrl_o,mov_eb_ivl_o)
"\tmovw\t%1,%0\n"       mov_mr(%0,%1,mov_mrw_o,mov_eb_ivw_o)

"\tsubl\t$4,%%esp\n\tfistpl\t0(%%esp)\n\tpopl\t%c\n"    sub_il(4,regis(ESP)), fistpl_si32(indirlong(ESP)), popl_o(use_reg(%c)), %c

"\tpushl\t%0\n\tfildl\t(%%esp)\n\taddl\t$4,%%esp\n"     pushl_o(use_reg(%0)), fildl_si32(indirlong(ESP)), add_il(4, regis(ESP)), %c

"\tsub\t$4,%%esp ; fstps (%%esp) ; flds (%%esp) ; addl $4,%%esp\n"      sub_il(4,regis(ESP)), fstps_sr32(indirlong(ESP)), flds_sr32(indirlong(ESP)), add_il(4,regis(ESP)), 0

"\tfld%0\n"             (is_memf_long(%0) ? fldl_sr64(xmemarg(memf_long(%0))) :  flds_sr32(xmemarg(memf_short(%0)))), 0
"\tfstpl\t%0\n"         fstpl_sr64(xmemarg(*(Mem_Instance *)%0)), 0
"\tfstps\t%0\n"         fstps_sr32(xmemarg(*(Mem_Instance *)%0)), 0
"\tsubl\t$8,%%esp\n\tfstpl\t(%%esp)\n"  sub_il(8, regis(ESP)), fstpl_sr64(indirlong(ESP)), 0
"\tsubl\t$4,%%esp\n\tfstps\t(%%esp)\n"  sub_il(4, regis(ESP)), fstps_sr32(indirlong(ESP)), 0

"\tmovl\t$%a,%%ecx\n\trep\n\tmovsb\n"   mov_rml_o(e(direct(relocdisp(msym(%a).raddr))), ECX), rep(), movsb(), 0

"\tsubl\t$%a,%%esp\n\tmovl\t%%esp,%%edi\n\tmovl\t$%a,%%ecx\n\trep\n\tmovsb\n"   sub_il(use_const((unsigned)%a),regis(ESP)),mov_mrl_o(ESP,regis(EDI)), movl(use_const((unsigned)%a),ECX), rep(), movsb(), 0

"?\tmovl\t%0,%c\n\timull\t%1,%c\n"      mov_rml_o(regis(use_reg(%0)), use_reg(%c)), (is_mrc_mem(%1) ? imul_rml_o(mrc_mem(%1), use_reg(%c)) : imul_ivl(mrc_con(%1), regis(use_reg(%c)), use_reg(%c))), %c

"\tjmp\t%0\n"   0
""      0
"%"     0
"%0"    %0
```

Overloading assembly instructions complicates automatic translation of assembly-language templates to calls to binary-encoding procedures. Because the toolkit currently cannot distinguish between overloaded variants of opcodes, we have to do the work. The type of an operand often determines the appropriate variant before the instruction is generated. In these cases, we encapsulate the operand with a type identifier and return it from `emitbin`; when the instruction is generated, the operand is decoded and the appropriate constructor is chosen.

The `memf` rules encode floating-point instructions with short, long, or stack-located operands.

326a ⟨*local declarations* 319a⟩+≡

```
#define Memf_TAG -2
typedef struct memf {
    int tag;
    enum memftype { memf_shortv = 0, memf_longv = 1, memf_stackv = 2 } xtype;
    Mem_Instance *addr;
}*Memf;
static Memf new_memf_long(Mem_Instance *);
static Memf new_memf_short(Mem_Instance *);
static Memf new_memf_stack(void);
#define is_memf_long(X)  (assert(((Memf)X)->tag == Memf_TAG), ((Memf)X)->xtype == memf_longv)
#define is_memf_short(X) (assert(((Memf)X)->tag == Memf_TAG), ((Memf)X)->xtype == memf_shortv)
#define is_memf_stack(X) (assert(((Memf)X)->tag == Memf_TAG), ((Memf)X)->xtype == memf_stackv)
#define memf_long(X)  (*((Memf)X)->addr)
#define memf_short(X) (*((Memf)X)->addr)

static Memf new_memf(int, Mem_Instance *);
#define new_memf_long(X)    new_memf(memf_longv, X)
#define new_memf_short(X)   new_memf(memf_shortv, X)
#define new_memf_stack()    new_memf(memf_stackv, (Mem_Instance *)0)
```

326b ⟨*definitions* 319b⟩+≡

```
static Memf new_memf(xtype, addr) Mem_Instance *addr;{
    Memf memf = (Memf)talloc(sizeof(*memf));
    memf->tag = Memf_TAG;
    memf->xtype = xtype;
    memf->addr = addr;
    return memf;
}
```

The `index` and `ss` fields of an effective address are chosen before the complete effective address is constructed.

326c ⟨*local declarations* 319a⟩+≡

```
typedef struct index_ss {
    int index,  ss;
}*IndexSS;
static IndexSS new_index_ss(int, int);
#define index_ss(X, field)  ((IndexSS)X)->field
```

326d ⟨*definitions* 319b⟩+≡

```
static IndexSS new_index_ss(index, ss) {
    IndexSS is = (IndexSS)talloc(sizeof(*is));
    is->index = index;
    is->ss = ss;
    return is;
}
```

Many instructions take operands that can represent effective addresses, registers, or constants. We use the `Mrc` type to encapsulate the operands until the instruction is chosen.

327a ⟨*local declarations* 319a⟩+≡

```
#define Mrc_TAG -1
typedef struct mrc {
    int tag;
    enum mrctype { mrc_memv = 0, mrc_conv = 2 } xtype;
    union {
        Eaddr_Instance *addr;
        Symbol con;
    }u;
}*Mrc;
static Mrc new_mrc(int);
static Mrc new_mrc_mem(Eaddr_Instance *);
static Mrc new_mrc_con(Symbol);
static unsigned Mrc_con(Mrc);
static int Mrc_reg(Mrc);

#define mrc_mem(X) (assert(((Mrc)X)->tag == Mrc_TAG), *((Mrc)X)->u.addr)
#define mrc_con(X) (assert(((Mrc)X)->tag == Mrc_TAG), use_const((unsigned)((Mrc)X)->u.con))
#define mrc_reg(X) (assert(((Mrc)X)->tag == Mrc_TAG && \
                   ((Mrc)X)->u.addr->tag == regis_TAG), ((Mrc)X)->u.addr->u.regis.reg)

#define is_mrc_mem(X)  (((Mrc)X)->xtype == mrc_memv)
#define is_mrc_con(X)  (((Mrc)X)->xtype == mrc_conv)
```

327b ⟨*definitions* 319b⟩+≡

```
static Mrc new_mrc(xtype) {
    Mrc mrc = (Mrc)talloc(sizeof(*mrc));
    mrc->tag = Mrc_TAG;
    mrc->xtype = xtype;
    return mrc;
}
static Mrc new_mrc_mem(addr) Eaddr_Instance *addr; {
    Mrc mrc = new_mrc(mrc_memv);
    mrc->u.addr = addr;
    return mrc;
}
static Mrc new_mrc_con(s) Symbol s; {
    Mrc mrc = new_mrc(mrc_conv);
    mrc->u.con = s;
    return mrc;
}
```

Just as on the SPARC (Section 15.1), typed constructors must be wrapped in a dynamically allocated structure so that `emitbin` can return them during code generation.

327c ⟨*local declarations* 319a⟩+≡

```
dclproto(static Disp_Instance *new_constdisp,(int));
dclproto(static Disp_Instance *new_relocdisp,(RAddr));
```

328a ⟨*definitions* 319b⟩+≡

```
static Disp_Instance *new_constdisp(d) {
    Disp_Instance *disp = (Disp_Instance *)talloc(sizeof(*disp));
    *disp = constdisp(d);
    return disp;
}
static Disp_Instance *new_relocdisp(RAddr raddr) {
    Disp_Instance *disp = (Disp_Instance *)talloc(sizeof(*disp));
    *disp = relocdisp(raddr);
    return disp;
}
```

328b ⟨*local declarations* 319a⟩+≡

```
dclproto(static Mem_Instance *new_direct,(Disp_Instance));
dclproto(static Mem_Instance *new_indir,(int));
dclproto(static Mem_Instance *new_disp32,(Disp_Instance, int));
dclproto(static Mem_Instance *new_indexr,(int,int,int));
dclproto(static Mem_Instance *new_index32,(Disp_Instance,int,int,int));
dclproto(static Mem_Instance *new_shortindex,(Disp_Instance,int,int));
```

328c ⟨*definitions* 319b⟩+≡

```
static Mem_Instance *new_direct(disp) Disp_Instance disp; {
    Mem_Instance *mem = (Mem_Instance *)talloc(sizeof(*mem));
    *mem = direct(disp);
    return mem;
}
static Mem_Instance *new_indir(reg) {
    Mem_Instance *mem = (Mem_Instance *)talloc(sizeof(*mem));
    *mem = indir(reg);
    return mem;
}
static Mem_Instance *new_disp32(disp, reg) Disp_Instance disp; {
    Mem_Instance *mem = (Mem_Instance *)talloc(sizeof(*mem));
    *mem = disp32(disp, reg);
    return mem;
}
static Mem_Instance *new_indexr(base,index,ss) {
    Mem_Instance *mem = (Mem_Instance *)talloc(sizeof(*mem));
    *mem = indexr(base,index,ss);
    return mem;
}
static Mem_Instance *new_index32(disp, base, index, ss) Disp_Instance disp;{
    Mem_Instance *mem = (Mem_Instance *)talloc(sizeof(*mem));
    *mem = index32(disp,base,index,ss);
    return mem;
}
static Mem_Instance *new_shortindex(disp,index,ss) Disp_Instance disp; {
    Mem_Instance *mem = (Mem_Instance *)talloc(sizeof(*mem));
    *mem = shortindex(disp,index,ss);
    return mem;
}
```

329a ⟨*local declarations* 319a⟩+≡
```
dclproto(static Eaddr_Instance *new_regis,(int));
dclproto(static Eaddr_Instance *new_E,(Mem_Instance));
```

329b ⟨*definitions* 319b⟩+≡
```
static Eaddr_Instance *new_regis(reg) {
    Eaddr_Instance *eaddr = (Eaddr_Instance *)talloc(sizeof(*eaddr));
    *eaddr = regis(reg);
    return eaddr;
}
static Eaddr_Instance *new_e(Mem_Instance mem) {
    Eaddr_Instance *eaddr = (Eaddr_Instance *)talloc(sizeof(*eaddr));
    *eaddr = e(mem);
    return eaddr;
}
```

329c ⟨*local declarations* 319a⟩+≡
```
dclproto(static Function_Instance *new_call_l,(RAddr));
dclproto(static Function_Instance *new_call_rm,(Eaddr_Instance));
```

329d ⟨*definitions* 319b⟩+≡
```
static Function_Instance *new_call_l(reloc) RAddr reloc;{
    Function_Instance *call = (Function_Instance *)talloc(sizeof(*call));
    *call = _call_l(reloc);
    return call;
}
static Function_Instance *new_call_rm(eaddr) Eaddr_Instance eaddr; {
    Function_Instance *call = (Function_Instance *)talloc(sizeof(*call));
    *call = _call_rm(eaddr);
    return call;
}
```

16.2 Assembly and startup code

329e ⟨*initialize target-specific assembly code* 329e⟩≡
This code is written to file `initialize target-specific assembly code`.

`b_assemble` assembles instructions represented in structure `acode`.

329f ⟨*definitions* 319b⟩+≡

329g ⟨*local declarations* 319a⟩+≡
```
dclproto(static void b_assemble,(struct asmcode, int, MTable *));
static void b_assemble(acode, nprocs, rootfns) struct asmcode acode; MTable *rootfns; {
}
```

`b_startup` emits startup code derived from /usr/lib/crt0.o.

329h ⟨*local declarations* 319a⟩+≡
```
static Symbol endsym;
```

330a ⟨*definitions* 319b⟩+≡
```
void b_startup(nprocs, rootfns, aoutflag) MTable *rootfns; int aoutflag; {
    int i, j, elems;
    BaseTable *bt;


    static char *syscalls[] = { (char *)0 };
    static int syscallids[] = { 0 };

    b_symbol(DATA, install_symbol("environ", &bt));
    (*IR->space)(4);
    b_symbol(DATA, install_symbol("errno", &bt));
    (*IR->space)(4);
    endsym = install_symbol("_end", &bt);
    b_symbol(DATA, endsym);             /* sbrk address _end == end */
    b_def(b_newsym("end", AUTO));       /* initialize end in b_aout */
    emitu(0);

    segments[text].maxcrb = nprocs + 1;
    elems = segments[text].maxcrb+1;
    segments[text].closures = (MTable *)alloc(elems * sizeof(MTable));
    bzero((char *)segments[text].closures, elems * sizeof(MTable));
    segments[text].relocblocks = (RBlock *)alloc(elems * sizeof(RBlock *));
    bzero((char *)segments[text].relocblocks, elems * sizeof(RBlock *));
    segments[text].names = (char **)alloc(elems * sizeof(char *));

    cseg = text;
    b_newtextblock(1024, string("asmcode"));
}
```

330b ⟨*assemble assembly code* 330b⟩≡
```
if (aoutflag) {
    b_seg(CODE);
    for (i = 0; Asmcode[i].symdef != 0; i++) {  /* emit assembled assembly code */
        if (strcmp(Asmcode[i].symdef, "_std_syscall") == 0)
            for (j = 0; syscalls[j]; j++) {
                Asmcode[i].symdef = syscalls[j];
                b_newsym(Asmcode[i].symdef, AUTO);
                Asmcode[i].insts[0].simm13 = syscallids[j];
                b_assemble(Asmcode[i], nprocs, rootfns);
            }
        else b_assemble(Asmcode[i], nprocs, rootfns);
    }
}
```
This code is written to file `assemble assembly code`.

Some constants needed by `b_dataseginit`.

330c ⟨*local declarations* 319a⟩+≡
```
#define USRTEXT 0
#define TXTADDR 0
```

16.3 Executable format

`b_aout` generates an executable a.out file.

331a ⟨*definitions* 319b⟩+≡

```
void b_aout(fd, aoutflag)  {
    int i, j, zerobufsz, totsz = 0;
    struct exec fh;
    char *zerobuf;
    unsigned addr;

    assert(fd);
    ⟨emit segment sizes ??⟩

    ⟨initialize file header 331b⟩

    ⟨set base addresses of data & bss segments 331c⟩
    ⟨initialize end & _end to end of bss section 332a⟩

    zerobufsz = fh.a_text - segments[text].totsize - HEADERSZ;
    zerobufsz = max(zerobufsz, fh.a_data - block_size(segments[data].relocblocks[0]));
    zerobuf = malloc(zerobufsz);
    bzero(zerobuf, zerobufsz);

    tkwarning("b_aout does not call dispatch\n");
/*    if (aoutflag) dispatch(); */
    write(fd, (char *)&fh, HEADERSZ);

    ⟨emit text segment 332b⟩
    ⟨emit data segment 332c⟩

}
```

331b ⟨*initialize file header* 331b⟩≡

```
bzero((char *)&fh, HEADERSZ);
segments[text].totsize = b_segsize(text);
if (mapflag) fprint(2,"text size %d\n", segments[text].totsize);
fh.a_text = roundup(segments[text].totsize + HEADERSZ, PAGE_SIZE);
fh.a_entry = segments[text].baseaddr;
N_SET_INFO(fh, NMAGIC, M_386, 0);
```

This code is used in chunk 331a.

331c ⟨*set base addresses of data & bss segments* 331c⟩≡

```
segments[data].baseaddr = segments[text].baseaddr - HEADERSZ + fh.a_text;
if (mapflag) fprint(2,"data %x\n", segments[data].baseaddr);
set_address(segments[data].relocblocks[0], segments[data].baseaddr);
fh.a_data = roundup(block_size(segments[data].relocblocks[0]), PAGE_SIZE);

segments[bss].baseaddr = segments[data].baseaddr + fh.a_data;
if (mapflag) fprint(2,"bss %x\n", segments[bss].baseaddr);
set_address(segments[bss].relocblocks[0], segments[bss].baseaddr);
fh.a_bss = roundup(segments[bss].totsize, PAGE_SIZE);
```

This code is used in chunk 331a.

332a ⟨*initialize end & _end to end of bss section* 332a⟩≡

```
b_seg(DATA);
org(msym(endsym).raddr->label->offset + msym(endsym).raddr->offset);
if (mapflag) fprint(2,"end %x\n", segments[bss].baseaddr + fh.a_bss);
emitu(segments[bss].baseaddr + fh.a_bss);
```

This code is used in chunk 331a.

332b ⟨*emit text segment* 332b⟩≡

```
addr = segments[text].baseaddr;
for (j = 0; j <= segments[text].crb; j++) {
    assert(label_location(segments[text].relocblocks[j]->label) == addr);
    block_write(fd, segments[text].relocblocks[j], 0, block_size(segments[text].relocblocks[j]));
    totsz += block_size(segments[text].relocblocks[j]);
    addr = label_location(segments[text].relocblocks[j]->label) + block_size(segments[text].relocblocks[j]);
}
assert(totsz == segments[text].totsize);
write(fd, zerobuf, fh.a_text - segments[text].totsize - HEADERSZ);
```

This code is used in chunk 331a.

332c ⟨*emit data segment* 332c⟩≡

```
block_write(fd, segments[data].relocblocks[0], 0, block_size(segments[data].relocblocks[0]));
write(fd, zerobuf, fh.a_data - block_size(segments[data].relocblocks[0]));
```

This code is used in chunk 331a.

332d ⟨*x86.binary* 332d⟩≡

```
%{
#include <stdio.h>
#include <c.h>
#include <mld.h>
#include <mclib.h>
#include <mldtk.h>
#include <tk.h>
#include <tags.h>
#include <x86-asmcode.h>
#include <resextsym.h>
#include <binary.h>
#include <a.out.h>
#include <syscall.h>

⟨local declarations 319a⟩
⟨definitions 319b⟩

%}
```

This code is written to file `x86.binary`.

17 File and archive management

The `file` interface provides a somewhat O/S-independent interface for managing file descriptors. The `archive` interface provides access to archives of `mill` object files. The `ranlib` interface creates table of contents for `mill` archives.

333a ⟨*external declarations* 333a⟩≡
```
dclproto(void fileClose,(BaseTable *));
dclproto(void fileInit,(void));
dclproto(int  fileOpen,(BaseTable *));
```
This definition is continued in chunks 334b and 337c.
This code is used in chunk 342c.

`fileInit` initializes the maximum file descriptor to 56. This is a horrible kludge – it happens to be the minimum number available on all the targets used by `mld`. The obvious hack of computing `fmax = FMAX - dup(0)` does not work everywhere.

333b ⟨*definitions* 333b⟩≡
```
void fileInit() {
    fmax = 56;     /* <== this is a horrible kludge */
    fprint(2, "Maximum number of open files %d\n", fmax);
}
```
This definition is continued in chunks 333–36 and 338–40.
This code is used in chunk 343.

`fileOpen` opens the given `mill` file. If the maximum number of open file descriptors is exceeded, close another file first.

333c ⟨*definitions* 333b⟩+≡
```
int fileOpen(bt) BaseTable *bt; {
    int fd;

    if (bt->fd) return 1;
    fileClose(0);
    if ((fd = open(bt->pathname, O_RDONLY)) != -1) {
        bt->fd = fd;
        filect++;
        if (Qtail) {
            Qtail->next = bt;
            Qtail = bt;
        }
        if (!Qhead) Qhead = Qtail = bt;
        return 1;
    } else {
        error("mld: Cannot open MILL file %s\n", bt->pathname);
        return 0;
    }
}
```

`fileClose` closes the given `mill` file. If the null file is provided, only close a file if the open file descriptor limit is exceeded.

334a ⟨*definitions* 333b⟩+≡
```
void fileClose(bt) BaseTable *bt; {
    if ((bt == 0 && filect < fmax) || (bt && bt->fd == 0)) return;
    if (bt == 0) {
        assert(Qhead);
        bt = Qhead;
        if (Qtail == Qhead) Qtail = 0;
        Qhead = Qhead->next;
        bt->next = 0;
    }
    if (bt && bt->fd) {
        if (dbgflag) fprint(2, "swap out %s \n", bt->pathname);
        close(bt->fd);
        bt->fd = 0;
        filect--;
    }
    if (filect >= fmax)
        fatal("archive.c", "mld: File limit exceeded.\n", 1);
}
```

`archiveOpen` opens an archive file for reading and returns its file descriptor.

334b ⟨*external declarations* 333a⟩+≡
```
dclproto(int archiveOpen,(char *));
dclproto(int archiveTOC,(int, int, MTable *));
dclproto(int archiveRead,(int, int, MTable *, int));
```

334c ⟨*definitions* 333b⟩+≡
```
int archiveOpen(filename) char *filename; {
    int fd;
    unsigned char magic[SARMAG];

    fileClose(0);
    if ((fd = open(filename, O_RDONLY)) == -1) {
        error("Cannot open archive file %s\n", filename);
        return 0;
    }
    filect++;
    read(fd, magic, SARMAG);
    if (strncmp((const char *)magic, ARMAG, SARMAG)) {
        error("%s not an archive file\n", filename);
        return 0;
    }
    return fd;
}
```

`archiveRead` reads the next archive header from `fd` and returns 1 if successful, 0 otherwise.

335a ⟨*definitions* 333b⟩+≡
```
int archiveRead(fd, idx, bmilltab, skiptoc) MTable *bmilltab; {
    struct ar_hdr ah;
    int newidx, sz, filesize = 0;
    int offset;
    char *name;
    BaseTable *bt;

    newidx = idx;
    if (!skiptoc) newidx = archiveTOC(fd, idx, bmilltab);
    if (newidx == idx) {
        offset = tell(fd);
        while((sz = read(fd, (unsigned char *)&ah, sizeof(struct ar_hdr))) != 0) {
            offset += filesize + sz;
            if (sz != sizeof(struct ar_hdr) || strncmp(ah.ar_fmag, ARFMAG, 2)) {
                error("Error while reading archive file.\n");
                return idx;
            } else if (strncmp(ah.ar_name, AR_SYMPREF, strlen(AR_SYMPREF)) == 0) {
                ⟨get archive size 336a⟩
                lseek(fd, filesize, SEEK_CUR);
            } else {
                ⟨get archive name 335c⟩
                ⟨get archive size 336a⟩
                lseek(fd, filesize, SEEK_CUR);
                ⟨update bmilltab 335b⟩
            }
        }
        newidx = idx;
    }
    return newidx;
}
```

335b ⟨*update* `bmilltab` 335b⟩≡
```
bt = newBaseTable(fd, offset, name);
updatetable(BaseTable **, bmilltab, idx, bt);
idx++;
```
This code is used in chunks 335a and 337b.

335c ⟨*get archive name* 335c⟩≡
```
{
    char *s;
    assert(s = index(ah.ar_name, ' '));
    *s = '\0';
    name = string(ah.ar_name);
}
```
This code is used in chunk 335a.

336a ⟨*get archive size* 336a⟩≡
```
{
    char *s;
    assert(s = index(ah.ar_size, ' '));
    *s = '\0';
    filesize = roundup(atoi(ah.ar_size), 2);
    *s = ' ';
}
```
This code is used in chunks 335a, 336b, and 341.

`archiveTOC` reads a table of contents from an archive and installs its entries in `mld`'s external symbol table by calling `resolvename`. It returns the index of the next available slot in `bmilltab`.

336b ⟨*definitions* 333b⟩+≡
```
int archiveTOC(fd, idx, bmilltab) MTable *bmilltab; {
    struct ar_hdr ah;
    char *s;
    int nranlibentries, nmodules, sz, filesize, offset;
    struct ranlib *ranlibbuf = 0;
    BaseTable *bt;
    int savidx = idx;

    offset = tell(fd);
    sz = read(fd, (unsigned char *)&ah, sizeof(struct ar_hdr));
    if (sz == 0) return idx;
    else if (sz != sizeof(struct ar_hdr) || strncmp(ah.ar_fmag, ARFMAG, 2)) {
        error("Error while reading archive file.\n");
        return idx;
    } else if (strncmp(ah.ar_name, AR_SYMPREF, strlen(AR_SYMPREF)) == 0) {
        ⟨get archive size 336a⟩
        read(fd, (unsigned char *)&nranlibentries, sizeof(int));
        ranlibbuf = (struct ranlib *)talloc(nranlibentries * sizeof(struct ranlib));
        read(fd, (unsigned char *)ranlibbuf, nranlibentries * sizeof(struct ranlib));
        nmodules = ranlibbuf[0].ran_off;
        ⟨read and install strings 336c⟩
        ⟨create BaseTables for modules 337b⟩
        ⟨install ranlib entries 337a⟩
        lseek(fd, filesize + SARMAG + sizeof(ah), SEEK_SET);
    } else lseek(fd, offset, SEEK_SET);
    return idx;
}
```

336c ⟨*read and install strings* 336c⟩≡
```
bt = newBaseTable(fd, 0, 0);
read(fd, (unsigned char *)&(bt->pbt.stringReg.size), sizeof(int));
bt->maxbufsz = bt->pbt.stringReg.size;
bt->pbt.stringReg.base = tell(fd);
bt->arena = newarena();
breadStrings(bt);
```
This code is used in chunk 336b.

337a ⟨*install ranlib entries* 337a⟩≡
```
{
    int i;
    char *s;
    for (i = 0; i < nranlibentries; i++) {
        s = mapstridx(bt, ranlibbuf[i].ran_un.ran_strx);
        resolvename(s, tableentry(BaseTable **, (*bmilltab), savidx+ranlibbuf[i].ran_off));
    }
}
```
This code is used in chunk 336b.

337b ⟨*create BaseTables for modules* 337b⟩≡
```
{
   BaseTable *bt;
   int i, offset;
   int *offsetmap;
   char *name = 0;

   offsetmap = (int *)talloc(nmodules*sizeof(int));
   read(fd, (unsigned char *)offsetmap, nmodules*sizeof(int));
   for (i = 0; i < nmodules; i++) {
        offset = offsetmap[i];
        ⟨update bmilltab 335b⟩
   }
}
```
This code is used in chunk 336b.

`ranlibOpen` opens a new archive into which a table of contents and the given archive will be written. Instead of overwriting the original archive, a temporary archive is created in `./_ranlib_`X`.a`.

337c ⟨*external declarations* 333a⟩+≡
```
dclproto(void ranlibOpen,(int, BaseTable *, int, int));
dclproto(void ranlibEntry,(BaseTable *, int, int));
dclproto(void ranlibStrings,(BaseTable *, Strings *));
dclproto(void ranlibClose,(int, BaseTable *, int));
```

338a ⟨*definitions* 333b⟩+≡
```
void ranlibOpen(fd, bt, nfileids, nranentries) BaseTable *bt; {
    char ranlibname[80];
    struct ar_hdr ah;
    struct ranlib r;

    sprintf(ranlibname, "./_ranlib_%d.a", fd);
    if ((bt->fd = open(ranlibname, O_WRONLY | O_CREAT)) != -1) {
        char magic[SARMAG];

        strncpy(magic, ARMAG, SARMAG);
        bufwrite(bt, magic, SARMAG);
        strcpy(ah.ar_name, AR_SYMPREF);
        ah.ar_name[AR_IHASHSEX] = (IR->little_endian ? AR_EL : AR_EB);
        ah.ar_name[AR_IUCODE] = AR_CODE;
        ah.ar_name[AR_ITARGETSEX] = (IR->little_endian ? AR_EL : AR_EB);;
        ah.ar_name[AR_IOUTOFDATE] = '_';
        ah.ar_name[sizeof(ah.ar_name)-1] = ' ';
        bufwrite(bt, (char *)&ah, sizeof(ah));
        nranentries += 1;
        bufwrite(bt, (char *)&nranentries, sizeof(int));
        /* first ranlib entry contains number of files in archive */
        ranlibEntry(bt, 0, nfileids);
        bufflush(bt);
        chmod(ranlibname, 0744);
    } else {
        error("Cannot open %s\n", ranlibname);
        exit(1);
    }
}
```

ranlibEntry writes a ranlib entry that contains the string index of the entry and the file index of the module that defines the symbol.

338b ⟨*definitions* 333b⟩+≡
```
void ranlibEntry(bt, stridx, fileid) BaseTable *bt; {
    struct ranlib r;
    r.ran_un.ran_strx = stridx;
    r.ran_off = fileid;
    bufwrite(bt, (char *)&r, sizeof(r));
}
```

ranlibStrings emits the string region for a complete archive.

339 ⟨*definitions* 333b⟩+≡

```
void ranlibStrings(bt, lstrs) BaseTable *bt; Strings *lstrs; {
    int strsize = 0;
    int offset;
    bufflush(bt);
    offset = tell(bt->fd);
    fprint(2,"string offset %d\n", offset);
     /* skip space for size of string region */
    bufwrite(bt, (char *)&strsize, sizeof(int));
    bemitLStrings(bt, lstrs);
    lseek(bt->fd, offset, SEEK_SET);
    write(bt->fd, (char *)&bt->pbt.stringReg.size, sizeof(int));
    lseek(bt->fd, 0, SEEK_END);
    fprint(2,"cur offset %d\n", tell(bt->fd));
}
```

`ranlibClose` emits the appropriate header data and closes the archive.

340 ⟨*definitions* 333b⟩+≡

```
void ranlibClose(arfd, bt, nfileids) BaseTable *bt; {
    struct ar_hdr ah;
    int arsize, filesize, mapoffset;
    char magic[SARMAG];
    int *offsetmap;

    bufflush(bt);
    mapoffset = tell(bt->fd);
    filesize = roundup(mapoffset + nfileids * sizeof(int), 2);
    offsetmap = (int *)malloc(nfileids * sizeof(int));
    /* leave space for offset map */
    filesize = filesize - (SARMAG + sizeof(ah));
    fprint(2,"file size %d string size %d\n", filesize, bt->pbt.stringReg.size);

    /* update file size in archive header */
    lseek(bt->fd, 0, SEEK_SET);
    fprint(2,"offset %d\n", tell(bt->fd));
        strncpy(magic, ARMAG, SARMAG);
    write(bt->fd, magic, SARMAG);
        strcpy(ah.ar_name, AR_SYMPREF);
        ah.ar_name[AR_IHASHSEX] = AR_EL;
        ah.ar_name[AR_IUCODE] = AR_CODE;
        ah.ar_name[AR_ITARGETSEX] = AR_EL;
        ah.ar_name[AR_IOUTOFDATE] = '_';
        ah.ar_name[sizeof(ah.ar_name)-1] = ' ';
        strncpy(ah.ar_date, "10102000    ", 12);
        strncpy(ah.ar_uid,"0     ", 6);
        strncpy(ah.ar_gid,"0     ", 6);
        strncpy(ah.ar_mode,"0       ", 8);
    sprintf(ah.ar_size,"%d ", filesize);
    strncpy(ah.ar_fmag, ARFMAG, 2);
    write(bt->fd, (char *)&ah, sizeof(ah));
    lseek(bt->fd, filesize, SEEK_CUR);
    ⟨emit archive 341⟩
    close(bt->fd);
}
```

341 ⟨*emit archive* 341⟩≡
```
{
    struct ar_hdr ah;
    int sz, filesize;
    int pos, bufsize = 0, n = 0;
    char *s, *buf = 0;

    assert((pos = lseek(arfd, SARMAG, SEEK_SET)) == SARMAG);
    while((sz = read(arfd, (unsigned char *)&ah, sizeof(struct ar_hdr))) != 0) {
        if (sz != sizeof(struct ar_hdr) || strncmp(ah.ar_fmag, ARFMAG, 2)) {
            error("Error while emitting archive file.\n");
            exit(1);
        } else if (strncmp(ah.ar_name, AR_SYMPREF, strlen(AR_SYMPREF)) == 0) {
            /* skip current ranlib */
            ⟨get archive size 336a⟩
            lseek(arfd, filesize, SEEK_CUR);
        } else {
            ⟨get archive size 336a⟩
            write(bt->fd, (char *)&ah, sizeof(ah));
            if (filesize > bufsize) {
                bufsize = roundup(filesize, 4096);
                if (buf) free(buf);
                buf = (char *)malloc(bufsize);
            }
            read(arfd, (unsigned char *)buf, filesize);
            offsetmap[n++] = tell(bt->fd);
            write(bt->fd, buf, filesize);
        }
    }
    assert (n == nfileids);
    lseek(bt->fd, mapoffset, SEEK_SET);
    write(bt->fd, (char *)offsetmap, nfileids * sizeof(int));
}
```
This code is used in chunk 340.

342a ⟨*sanity check* 342a⟩≡

```
{
    struct ar_hdr ah;
    int arsize, filesize, nranentries;
    unsigned char magic[SARMAG];
    char *s;

    lseek(bt->fd, 0, SEEK_SET);
    read(bt->fd, magic, SARMAG);
    fprint(2,"magic %s\n", magic);
    read(bt->fd, (unsigned char *)&ah, sizeof(ah));
    assert(s = index(ah.ar_name, ' '));
    *s = '\0';
    fprint(2,"name %s\n", s);
    assert(s = index(ah.ar_size, ' '));
    *s = '\0';
    fprint(2,"size %s\n", ah.ar_size);
    read(bt->fd, (unsigned char *)&nranentries, sizeof(int));
    fprint(2,"entries %d\n", nranentries);
    lseek(bt->fd, nranentries*sizeof(struct ranlib), SEEK_CUR);
    read(bt->fd, (unsigned char *)&nranentries, sizeof(int));
    fprint(2,"string size %d\n", nranentries);
    lseek(bt->fd, nranentries, SEEK_CUR);
    /* emit the archive */
}
```

This code is written to file `sanity check`.

342b ⟨*local declarations* 342b⟩≡

This code is used in chunk 343.

342c ⟨*archive.h* 342c⟩≡

```
#ifndef INCLUDE_ARCHIVEH
#define INCLUDE_ARCHIVEH
#include <string.h>
⟨external declarations 333a⟩
#endif
```

This code is written to file `archive.h`.

343 ⟨*archive.c* 343⟩≡

```
#include <sys/file.h>
#include <stdio.h>
#include <ar.h>
#include <ranlib.h>
#include <c.h>
#include <mld.h>
#include <archive.h>
#include <resextsym.h>

#ifndef AR_SYMPREF
#define AR_SYMPREF "__.SYMDEF"
#define AR_SYMPREFLEN   11                      /* length of prefix */
#define AR_IHASHSEX     11                      /* index for hash sex char */
#define AR_IUCODE       12                      /* index of ucode char */
#define AR_ITARGETSEX   13                      /* index for target sex char */
#define AR_IOUTOFDATE   14                      /* index for out of date char */
#define AR_OUTOFDATE    'X'                     /* out of date char */
#define AR_EL           'L'                     /* EL char */
#define AR_EB           'B'                     /* EB char */
#define AR_UCODE        'U'                     /* UCODE char */
#define AR_CODE         'E'                     /* regulart code char */
#endif

dclproto(extern char *index,(char *, char));

static int fmax = 0;
static int filect = 0;
static BaseTable *Qhead = 0, *Qtail = 0;

⟨local declarations 342b⟩
⟨definitions 333b⟩
```

This code is written to file `archive.c`.

17.1 Buffer management

This module provides file buffer support for `mlcc` and `mld`. `mlcc` writes and `mld` reads large `mill` object files. Before a `mill` object file is converted from its on-disk into its in-memory representation, it is read into a file buffer one region at a time. Each module attempts to allocate a file buffer large enough for its largest region. File buffers are shared among multiple modules, and a maximum pool of 100K bytes are allocated to the modules.

If modules were each allocated their own file buffer, many mega-bytes of memory would be earmarked for buffering files. Files are read sequentially, so it's not difficult to share the file buffers among multiple modules. Nonetheless, I can't remember the original justification for this code so some profiling is necessary to determine if it's really needed.

344a ⟨*external declarations* 344a⟩≡

```
typedef struct filebuf {
    char *ptr, *pos, *limit;
    int sz;
    struct btable *user;
    struct filebuf *next;
}*Filebuf;
```

This definition is continued in chunks 344 and 346.
This code is used in chunk 347a.

344b ⟨*external declarations* 344a⟩+≡

```
#define bufincr(_bt, _sz) (_bt)->buf->pos += _sz
#define bufptr(_bt) (_bt)->buf->pos
#define bufread(_bt, _v, _sz) \
        { bcopy((_bt)->buf->pos,(char *)_v,_sz); (_bt)->buf->pos += _sz; }
#define bufsize(_bt) (_bt)->buf->sz
```

344c ⟨*local declarations* 344c⟩≡

```
#define POOLLIMIT  0x100000
#define MBUFSIZE   8192
dclproto(static Filebuf newbuf,(BaseTablePtr, int));
dclproto(static void bufrealloc,(Filebuf, int));
```

This definition is continued in chunk 346c.
This code is used in chunk 347b.

344d ⟨*data definitions* 344d⟩≡

```
static int poollimit = 0;
static struct filebuf heads = { 0, 0, 0, 0, 0, &heads};
static Filebuf head = &heads;
```

This code is used in chunk 347b.

`bufalloc` allocates a buffer for the `mill` file represented by the `BaseTable bt`. If the pool limit has not been exceeded, the file gets its own buffer. Otherwise, a first-fit heuristic is used to deallocate a buffer in use and reallocate to the given file. If no such buffer is found, a new buffer is allocated anyway and a warning is issued.

344e ⟨*external declarations* 344a⟩+≡

```
typedef struct btable *BaseTablePtr;
dclproto(Filebuf bufalloc,(BaseTablePtr));
```

345a ⟨*definitions* 345a⟩≡
```
Filebuf bufalloc(bt) BaseTablePtr bt; {
    int sz = (bt->maxbufsz > 0 ? bt->maxbufsz : MBUFSIZE);
    Filebuf buf, end;

    if (bt->buf) {
        bt->buf->pos = bt->buf->ptr;
        return bt->buf;
    }
    /* if pool limit not exceeded, alloc a buffer */
    if (poollimit + sz < POOLLIMIT)
        return bt->buf = newbuf(bt, sz);
    else { /* find buffer that fits and deallocate buffer */
        end = head;
        head = head->next;
        while (head != end) {
            if (head->sz > sz) {
                head->user->buf = (Filebuf)0;
                head->user = bt;
                head->pos = head->ptr;
                return bt->buf = head;
            }
            head = head->next;
        }
    }
    ⟨pool limit warning 345b⟩
    return bt->buf = newbuf(bt, sz);
}
```
This definition is continued in chunks 345 and 346.
This code is used in chunk 347b.

345b ⟨*pool limit warning* 345b⟩≡
```
if (poollimit > POOLLIMIT)
    warning("Pool limit (%d bytes) exceeded (%d bytes). Trying to allocate %d.\n",
            POOLLIMIT, poollimit, sz);
```
This code is used in chunks 345a and 346a.

345c ⟨*definitions* 345a⟩+≡
```
static Filebuf newbuf(user, sz)  BaseTablePtr user; {
    Filebuf buf;

    buf = malloc(sizeof(*buf));
    assert(buf->ptr = malloc(sz));
    buf->pos = buf->ptr;
    buf->sz = sz;
    buf->limit = buf->ptr + sz;
    buf->user = user;
    buf->next = head->next;
    head->next = buf;
    poollimit += sz;
    return buf;
}
```

`bufrealloc` increases the size of a current buffer.

346a ⟨*definitions* 345a⟩+≡
```
static void bufrealloc(buf, sz) Filebuf buf; {
    if (sz <= buf->sz) return;
    assert(buf->ptr == buf->pos);
    assert(buf->ptr = realloc(buf->ptr, sz));
    poollimit += (sz - buf->sz);
    ⟨pool limit warning 345b⟩
    buf->sz = sz;
    buf->limit = buf->ptr + sz;
    buf->pos = buf->ptr;
}
```

`bufflush` flushes the current buffer to the file represented by the file descriptor `bt->fd`.

346b ⟨*external declarations* 344a⟩+≡
```
dclproto(void bufflush,(BaseTablePtr));
extern int totwrites;
```

346c ⟨*local declarations* 344c⟩+≡
```
int totwrites = 0;
```

346d ⟨*definitions* 345a⟩+≡
```
void bufflush(bt) BaseTablePtr bt; {
    Filebuf buf = bt->buf;
    if (buf->pos == buf->ptr) return;
    if (buf->pos < buf->ptr) assert(0);
    write(bt->fd, buf->ptr, buf->pos - buf->ptr);
    totwrites++;
    buf->pos = buf->ptr;
}
```

`bufwrite` writes `sz` bytes from buffer `v` into the file `bt`'s write buffer.

346e ⟨*external declarations* 344a⟩+≡
```
dclproto(void bufwrite,(BaseTablePtr , char *, int));
```

346f ⟨*definitions* 345a⟩+≡
```
void bufwrite(bt, v, sz) BaseTablePtr bt; char *v; {

    if (!bt->buf) {
        if (bt->maxbufsz < sz) bt->maxbufsz = sz;
        bt->buf = bufalloc(bt);
     } else if (sz > bt->buf->sz) {
        bufflush(bt);
        bufrealloc(bt->buf, sz*2);
    } else if (bt->buf->pos + sz > bt->buf->limit)  /* dump current buffer */
        bufflush(bt);
    if (sz > bt->buf->sz) bufrealloc(bt->buf, sz*2);
    bcopy(v, bt->buf->pos, sz);
    bt->buf->pos += sz;
}
```

347a ⟨*buf.h* 347a⟩≡
```
#ifndef INCLUDE_BUFH
#define INCLUDE_BUFH
⟨external declarations 344a⟩
#endif
```

This code is written to file `buf.h`.

347b ⟨*buf.c* 347b⟩≡
```
#include <c.h>
#include <mld.h>
#include <buf.h>

⟨local declarations 344c⟩
⟨data definitions 344d⟩
⟨definitions 345a⟩
```
This code is written to file `buf.c`.

18 Internal data structures and support procedures

18.1 Memory management

`mlcc` and `mld` use an early version of `lcc`'s arena-based memory allocator. This module is derived from that source. We need the older version, because `mld` allocates arenas on a per-module basis, which permits deallocation of the arena once a module is no longer used. This is important when reading large modules from libraries. `lcc`'s current version does not allow creation of an arbitrary number of arenas.

This implementation provides two default arenas: `permanent` and `temporary`. The `permanent` arena is never deallocated; `temporary` is deallocated after code is generated for each procedure.

348 ⟨*local declarations* 348⟩≡
```
static struct arena first[] = {
    0, 0, 0, &first[0], 0,
    0, 0, 0, &first[1], 0,
};
Arena permanent = &first[0];    /* permanent storage */
Arena transient = &first[1];    /* transient storage; released at function end */
static Arena freearenas;    /* list of free arenas */
```
This code is used in chunk 350d.

`mlcc_allocate` allocates `n` bytes in arena `*p`, adding a new arena if necessary. `mlcc_deallocate` releases all the space in arena `*p`, except the first arena. `arenasize` returns the size of the given arena in bytes. `tfree` releases the space in all the `transient` arenas.

349a ⟨*definitions* 349a⟩≡
```
char *mlcc_allocate(n, p) Arena *p; {
    Arena ap = *p;

    while (ap->avail + n > ap->limit)
        if (ap->next) {        /* move to next arena */
            ap = ap->next;
            ap->avail = (char *)ap + roundup(sizeof *ap, sizeof (double));
        } else if (ap->next = freearenas) {
            freearenas = freearenas->next;
            ap = ap->next;
            ap->avail = (char *)ap + roundup(sizeof *ap, sizeof (double));
            ap->first = (*p)->first;
            ap->next = 0;
        } else {        /* allocate a new arena */
            int m = n + MEMINCR*1024 + roundup(sizeof *ap, sizeof (double));
            ap->next = (Arena) malloc(m);
            if (!(ap->next && (int)ap->next >= 0)) memory_usage();
            assert(ap->next && (int)ap->next >= 0);
            if ((char *)ap->next == ap->limit) /* extend previous arena? */
                ap->limit = (char *)ap->next + m;
            else {            /* link to a new arena */
                ap = ap->next;
                ap->limit = (char *)ap + m;
                ap->avail = (char *)ap + roundup(sizeof *ap, sizeof (double));
            }
            ap->first = (*p)->first;
            ap->next = 0;
        }
    *p = ap;
    ap->avail += n;
    return ap->avail - n;
}
```
This definition is continued in chunks 349 and 350.
This code is used in chunk 350d.

349b ⟨*definitions* 349a⟩+≡
```
void mlcc_deallocate(p) Arena *p; {
    Arena first = (*p)->first;

    (*p)->next = freearenas;
    freearenas = first->next;
    first->next = 0;
    *p = first;
    assert(arenasize(*p) == 0);
}
```

newarena returns a new arena.

350a ⟨*definitions* 349a⟩+≡
```
struct arena *newarena(void) {
    Arena arena;
    arena = (Arena)alloc(sizeof(struct arena));
    BZERO(arena, struct arena);
    arena->first = arena;
    return arena;
}
```

350b ⟨*definitions* 349a⟩+≡
```
int arenasize(p) Arena p; {
    int size = 0, bufsz;
    Arena first = p->first->next, ap;

    for (ap = first; ap; ap = ap->next) {
        bufsz = ap->limit - ((char *)ap + roundup(sizeof *ap, sizeof (double)));
        assert(bufsz > 0);
        size += bufsz;
    }
    return size;
}
```

350c ⟨*definitions* 349a⟩+≡
```
void tfree() {
    if (!xref)
        mlcc_deallocate(&transient);
}
```

350d ⟨*mlccalloc.c* 350d⟩≡
```
#include "c.h"
```
⟨*local declarations* 348⟩
⟨*definitions* 349a⟩

This code is written to file `mlccalloc.c`.

18.2 mill Tables (MTables)

`mill` tables (`MTable`s) are dynamically allocated arrays and are the primary data structure used to represent tables and graphs in `mlcc` and `mld`. An `MTable` contains a pointer to the table's memory, the address of the arena from which the table is allocated, the table's size, its element size, and the current maximum element.

351a ⟨*external declarations* 351a⟩≡
```
typedef struct mtable {
        Generic *table;
        struct arena **arena;
        int     tablesz;
        int     elemsz;
        int     maxelem;
}MTable;
#define MTABLESZ        sizeof(struct mtable)
typedef struct range {
        int low,hi;
}Range;
#define RANGESZ sizeof(Range)
```

This definition is continued in chunks 351–54.
This code is used in chunk 354e.

Most of the interfaces for accessing tables are (horrid) macros. Because C doesn't support generic or template types, type casts are used to access the elements of `MTable`s.

`idxtable` asserts that index `_idx` is in the range of table `_t` and returns the `_idx`-th element in `_t`. `tableentry` is reckless and just returns the element, without a range check.

351b ⟨*external declarations* 351a⟩+≡
```
#define idxtable(_type,_t,_idx) (assert(_idx <= (_t).maxelem), ((_type)((_t).table))[_idx])
#define tableentry(_type,_t,_idx) ((_type)((_t).table))[_idx]
```

`updatetable` sets `_t[_idx]` to `_val`. `addtotable` inserts `_val` as the new last element of `_t`. `popfromtable` returns the last element in `_t` and decrements its size by 1.

351c ⟨*external declarations* 351a⟩+≡
```
#define updatetable(_type,_t,_idx,_val) { \
        if (_idx >= (_t)->tablesz) expandtable(_t,_idx); \
        ((_type)((_t)->table))[_idx] = _val;  \
        if ((_t)->maxelem < _idx) (_t)->maxelem = _idx; \
}
#define addtotable(_type,_t,_val) { \
        if ((_t)->maxelem+1 >= (_t)->tablesz) expandtable(_t,(_t)->maxelem+1); \
        ((_type)((_t)->table))[(_t)->maxelem+1] = _val;  \
        (_t)->maxelem += 1; \
}
#define popfromtable(_type,_t) (assert((_t)->maxelem >= 0), \
        (_t)->maxelem -= 1,((_type)((_t)->table))[(_t)->maxelem+1])
```

`isemptytable` is true if `_t` has never been initialized. `emptytable` is true if `_t` is uninitialized or it contains no elements. `cleartable` reuses `_t` by clearing the table of all elements and re-initializing its arena; `reusetable` just clears the table.

352a ⟨*external declarations* 351a⟩+≡
```
#define isemptytable(_t)        ((_t).tablesz == 0)
#define emptytable(_t)          ((_t).tablesz == 0 || (_t).maxelem == -1)
#define cleartable(_t, _arena)  \
        { (_t).tablesz = 0; (_t).maxelem = -1; if (!(_t).arena) (_t).arena = _arena; }
#define reusetable(_t)  { (_t).maxelem = -1; }

#define gentable(_t,_i,_s)      for(_i = _s; (_t).table && _i <= (_t).maxelem; _i++)
#define intablerange(_t,_i)     (_i <= (_t).maxelem)
#define tablesize(_t)   ((_t).maxelem+1)
#define tablebytesize(_t) (tablesize(_t)*(_t).elemsz)
```

`newtable` returns a new table with `nelem` elements with size `elemsz` initialized to zero. The table is allocated in the given memory `arena`.

352b ⟨*external declarations* 351a⟩+≡
```
dclproto(void newtable,(MTable *t,int sz, int elemsz, struct arena **));
```

352c ⟨*definitions* 352c⟩≡
```
void newtable(t, nelem, elemsz, arena) MTable *t; struct arena **arena; {
    assert(t);

    count++;
    t->elemsz = elemsz;
    t->tablesz = nelem;
    t->maxelem = -1;
    t->arena = arena;
    if (nelem && elemsz) {
        assert(arena && *arena);
        t->table = (Generic *)yyalloc(nelem * elemsz, (*arena));
        bzero((char *)t->table,nelem * t->elemnelem);
    }
    else t->table = (Generic *)0;
}
```

This definition is continued in chunks 353 and 354.
This code is used in chunk 354d.

`expandtable` expands the table `t` to size `sz` if the new size is greater than the current table size. `expandtable` preserve's the table's contents and initializes the expanded area of the table to zero.

352d ⟨*external declarations* 351a⟩+≡
```
dclproto(void expandtable,(MTable *t, int sz));
```

353a ⟨*definitions* 352c⟩+≡
```
static int expansions = 0;
static int count = 0;
void expandtable(t, tsz) MTable *t; {
    char *newt;
    Arena arena;
    assert(t && t->elemsz);
    if (tsz >= t->tablesz) {
        int sz;

        assert(t->arena);
        sz = max(tsz << 1,DEFAULTTABSZ);
        if (t->table) {
            assert(t->arena && *(t->arena));
            newt = yyalloc(sz*t->elemsz, (*(t->arena)));
            bcopy((char *)t->table, newt, t->tablesz*t->elemsz);
            t->table = (Generic *)newt;
            bzero((char *)t->table+(t->tablesz*t->elemsz),(sz-t->tablesz)*t->elemsz);
            expansions++;
        }
        else {
            t->table = (Generic *)yyalloc(sz * t->elemsz, (*(t->arena)));
            bzero((char *)t->table,sz * t->elemsz);
        }
        t->tablesz = sz;
        assert(tsz < t->tablesz);
    }
}
```

`copyrange` copies a range of elements in the table into the end of the table, expanding the table if necessary.

353b ⟨*external declarations* 351a⟩+≡
```
dclproto(void copyrange,(MTable *, Range r));
```

353c ⟨*definitions* 352c⟩+≡
```
void copyrange(t, r) MTable *t; Range r; {
    if ((t->maxelem + r.hi - r.low + 1) > t->tablesz)
        expandtable(t, t->tablesz + r.hi - r.low + 1);
    bcopy((char *)t->table + (r.low * t->elemsz),
        (char *)t->table + (tablesize(*t) * t->elemsz),
        (r.hi - r.low + 1)*t->elemsz);
    t->maxelem = t->maxelem + (r.hi - r.low + 1);
}
```

`copytable` copies the table `oldt` into the table `newt`, allocating space for `newt` in `arena`, if necessary.

353d ⟨*external declarations* 351a⟩+≡
```
dclproto(void copytable,(MTable *, MTable *, struct arena **));
```

354a ⟨*definitions* 352c⟩+≡

```
void copytable(oldt, newt, arena) MTable *oldt, *newt; struct arena **arena; {
    assert(oldt && newt);
    *newt = *oldt;
    newt->arena = arena;
    if (newt->elemsz && newt->tablesz) {
        assert(arena && *arena);
        newt->table = (Generic *)yyalloc(newt->elemsz * newt->tablesz, (*arena));
        assert(oldt->table);
        bcopy((char *)oldt->table, (char *)newt->table, newt->elemsz * newt->tablesz);
    }
    else newt->table = (Generic *)0;
}
```

`tabledata` reports the number of tables created and the number of table expansions.

354b ⟨*external declarations* 351a⟩+≡

```
dclproto(void tabledata,(void));
```

354c ⟨*definitions* 352c⟩+≡

```
void tabledata() {
    fprint(2,"total tables %d expansions %d\n", count, expansions);
}
```

354d ⟨*mtable.c* 354d⟩≡

```
#include "c.h"
#define DEFAULTTABSZ    8
⟨definitions 352c⟩
```

This code is written to file `mtable.c`.

354e ⟨*mtable.h* 354e⟩≡

```
#ifndef INCLUDE_MTABLEH
#define INCLUDE_MTABLEH

⟨external declarations 351a⟩
#endif
```

This code is written to file `mtable.h`.

18.3 Sets

A `Set` is represented by a *unique* vector of bits. `initset` initializes a `Set` with a maximum of `size` members. `xset`, `uset`, and `diffset` combine two sets by applying the set operators intersection, union, and difference respectively. The first set is destructively modified. `member` returns true if `member` is in the given `Set`. `insert` inserts `member` into the given `Set`; `delete` deletes `member` from the given `Set`. Storing unique sets conserves memory and makes set equality trivial because we simply compare set pointers.

355a ⟨*external declarations* 355a⟩≡
```
typedef struct set {
    int  charsize;
    unsigned char *vals;
}* Set;
#define SETSZ     sizeof(struct set)

#define ORSET  0
#define ANDSET 1
#define DIFFSET 2
#define uset(v1, v2)      combineset(v1, v2, ORSET)
#define xset(v1, v2)      combineset(v1, v2, ANDSET)
#define diffset(v1, v2)   combineset(v1, v2, DIFFSET)
#define eqsets(s1, s2)    ((s1)->vals == (s2)->vals)
#define copyset(dest, src) (*(dest) = *(src))
```

This definition is continued in chunks 355b and 361d.
This code is used in chunk 362b.

355b ⟨*external declarations* 355a⟩+≡
```
void combineset(Set, Set, int);
void delete(Set, int);
void insert(Set, int);
void initset(Set, int);
int  member(Set, int);
void freesets(void);
```

`tempset` holds a temporary set large enough to hold the largest set created.

355c ⟨*definitions* 355c⟩≡
```
void initset(v, size) Set v;{
    int charsize = CHARSIZE(size);

    v->charsize = 0;
    v->vals = (unsigned char *)0;
    if (tempset.charsize < charsize) {
        tempset.charsize = charsize;
        tempset.vals = (unsigned char *)yyalloc(charsize, setarena);
        bzero((char *)tempset.vals, charsize);
        maxsize = max(maxsize, size);
    }
}
```
This definition is continued in chunks 357, 358b, and 360–62.
This code is used in chunk 362c.

356 ⟨*local declarations* 356⟩≡

```
#define emptyset(s) (s->vals == 0)
```

This definition is continued in chunks 358–60.
This code is used in chunk 362c.

`combineset` is the work horse: it applies `op` to `v1` and `v2`, destructively modifying `v1`.

357 ⟨*definitions* 355c⟩+≡

```
void combineset(v1, v2, op) Set v1, v2; {
    int minsize, maxsize, i, changed = 0;
    Set v;
    struct set tmps;

    switch (op) {
    case ORSET:
        if (emptyset(v2) || v1->vals == v2->vals) return;
        else if (emptyset(v1)) {
            v1->charsize = v2->charsize;
            v1->vals = v2->vals;
            return;
        }
        break;
    case ANDSET:
        if (emptyset(v1) || emptyset(v2)) {
            v1->vals = 0;
            return;
        } else if (v1->vals == v2->vals) return;
        break;
    case DIFFSET:
        if (emptyset(v1) || emptyset(v2)) return;
        else if (v1->vals == v2->vals) {
            v1->vals = 0;
            return;
        }
        break;
    default: assert(0);
    }

   v = (v1->charsize > v2->charsize) ? v1 : v2;
   if ((tmps.vals = lookupopc(v1->vals, v2->vals, op)) == 0) {
        minsize = (v1->charsize < v2->charsize) ? v1->charsize : v2->charsize;
        assert(tempset.charsize >= minsize);
        ⟨clear tempset 359b⟩
        for (i = 0; i < minsize; i++) {
            switch (op) {
            case ORSET:   tempset.vals[i] = v1->vals[i] | v2->vals[i];    break;
            case ANDSET:  tempset.vals[i] = v1->vals[i] & v2->vals[i];    break;
            case DIFFSET: tempset.vals[i] = v1->vals[i] & ~(v2->vals[i]); break;
            }
            changed |= ((tempset.vals[i] ^ v1->vals[i]) != 0);
        }
        maxsize = (v1->charsize > v2->charsize) ? v1->charsize : v2->charsize;
        assert(tempset.charsize >= maxsize);
        if (minsize != maxsize && op != ANDSET) {
            changed = 1;
            if (op == ORSET || v == v1)
                for (i = minsize; i < maxsize; i++)
                    tempset.vals[i] = v->vals[i];
        }
        if (changed) {
```

```
            tmps.vals = tempset.vals;
            tmps.charsize = v->charsize;
            installset(&tmps);
        } else tmps.vals = v1->vals;
        installopc(v1->vals, v2->vals, op, tmps.vals);
    }
    v1->vals = tmps.vals;
    v1->charsize = v->charsize;
}
```

We use a set expansion heuristic of 20% of the current maximum size of the set.

358a ⟨*local declarations* 356⟩+≡

```
#define EXPANSION_FACTOR(k) ((k) + (int)((k)/5))
```

358b ⟨*definitions* 355c⟩+≡

```
void insert(v, mem) Set v; {
    int word;

    if (!member(v, mem)) {
        ⟨copy v into tempset 359a⟩
        if (mem/BITSPERCHAR >= v->charsize)
            v->charsize = min(tempset.charsize, CHARSIZE(EXPANSION_FACTOR(mem)));
        assert(v->charsize > 0 && mem/BITSPERCHAR < v->charsize);
        tempset.vals[mem/BITSPERCHAR] |= (0x01 << (mem % BITSPERCHAR));
        v->vals = tempset.vals;
        installset(v);
    }
    maxmem = max(maxmem, mem);
}
void delete(v, mem) Set v; {
    int word;

    if (member(v, mem)) {
        ⟨copy v into tempset 359a⟩
        tempset.vals[mem/BITSPERCHAR] &= ~(0x01 << (mem % BITSPERCHAR));
        v->vals = tempset.vals;
        installset(v);
    }
}

int member(v, bit) Set v; {
    unsigned char c;

    if (bit/BITSPERCHAR >= v->charsize || v->vals == 0) return 0;
    c = v->vals[bit/BITSPERCHAR];
    return (c >> (bit % BITSPERCHAR)) & 0x01;
}
```

359a ⟨*copy* `v` *into* `tempset` 359a⟩≡
```
{
    int i;
    ⟨clear tempset 359b⟩
    assert(tempset.charsize >= v->charsize);
    for (i = 0; i < v->charsize; i++)
        tempset.vals[i] = v->vals[i];
}
```
This code is used in chunk 358b.

359b ⟨*clear* `tempset` 359b⟩≡
```
bzero((char *)tempset.vals, tempset.charsize);
```

This code is used in chunks 357 and 359a.

18.3.1 Caching results of set operations

It is often useful to cache the result of a set operation because during iterative data flow analysis, the same operation is often applied to the same two sets multiple times. Caching has proven useful in `mld`: the set cache has a hit ratio of greater than 30% for most programs.
Unique and temporary sets are allocated in the `setarena`.

359c ⟨*local declarations* 356⟩+≡
```
static struct arena first = {
        0, 0, 0, &first, 0
};
static Arena setarena = &first; /* set storage */
struct set tempset = { 0, 0 };

static Strings lsets = { 0, 0, &setarena, 0, 0 };
#define BITSPERCHAR 8
```

The set-operation cache is an extensible hash table whose keys are `l, op, r` triples and whose contents are the value `x = l op r`.

359d ⟨*local declarations* 356⟩+≡
```
#define OPCTABLESIZE 1024
#define OPLEN sizeof(char *)*2
typedef struct opcache {
    unsigned char *l, *r, *x;
    int op;
    struct opcache *next;
}*OpCache;

OpCache setopcache[OPCTABLESIZE];
#define CHARSIZE(size) (((size)+1)/BITSPERCHAR + (((size)+1) % BITSPERCHAR > 0 ? 1 : 0))
#define min(x,y) ((x) < (y) ? (x) : (y))
```

`hashstr` returns a hash key for the pair of null-terminated character strings by applying a has function to their *addresses*, not their contents. `lookupopc` searches for the triple `l op r` in the set operation cache and returns the result set of the operation, if it is cached. It returns `0`, otherwise. `installopc` sets the cache element for the triple `l op r` to `x`. `emptycache` re-initializes the set operation cache. `freesets` empties the set operation cache and deallocates the set arena. This is usually called after a procedure has been analyzed and emitted.

360a ⟨*local declarations* 356⟩+≡

```
dclproto(static int hashstr,(unsigned char *, unsigned char *));
dclproto(static unsigned char *lookupopc,(unsigned char *, unsigned char *, int));
dclproto(static void installopc,(unsigned char *, unsigned char *, int, unsigned char *));
dclproto(static void emptycache,(void));
static int hits = 0, misses = 0, maxsize = 0, maxmem = 0, installed = 0;
```

360b ⟨*definitions* 355c⟩+≡

```
static int hashstr(l, r) unsigned char *l, *r; {
    unsigned h;
    int i;
    unsigned char *end, *str[2];
    extern int scatter[];

    str[0] = l;
    str[1] = r;
    for (h = 0, i = OPLEN, end = (unsigned char *)str; i > 0; i--)
        h = (h<<1) + scatter[*end++];
    h &= OPCTABLESIZE-1;
    return h;
}
```

360c ⟨*definitions* 355c⟩+≡

```
static unsigned char *lookupopc(l, r, op) unsigned char *l, *r; {
    OpCache opc;
    int h = hashstr(l,r);

    for (opc = setopcache[h]; opc; opc = opc->next)
        if (opc->l == l && opc->r == r && opc->op == op) {
            hits++;
            return opc->x;
        }
    return 0;
}
```

361a ⟨*definitions* 355c⟩+≡
```
static void installopc(l, r, op, x) unsigned char *l, *r, *x; {
    int h;
    OpCache opc = (OpCache)yyalloc(sizeof(*opc), setarena), tmpc;

    misses++;
    h = hashstr(l,r);
    opc->l = l;
    opc->r = r;
    opc->op = op;
    opc->x = x;
    tmpc = setopcache[h];
    opc->next = tmpc;
    setopcache[h] = opc;
}
```

361b ⟨*definitions* 355c⟩+≡
```
static void emptycache() {
    int i;
    for (i = 0; i < OPCTABLESIZE; i++)
        setopcache[i] = 0;
}
```

361c ⟨*definitions* 355c⟩+≡
```
void freesets() {
    fprint(2, "hits %d misses %d maxsize %d maxmem %d installed %d\n", hits, misses, maxsize, maxmem, installed);
    empty_strings(&lsets);
    emptycache();
    hits = misses = maxsize = maxmem = installed = 0;
    tempset.charsize = 0;
    tempset.vals = 0;
    mlcc_deallocate(&setarena);
}
```

18.3.2 Temporary sets

All the set operators defined above return a unique set. Often during data flow analysis, a set is initialized by inserting one element at a time. Using the default set operators, a unique set would be created after each element is inserted. Temporary sets are provided to enable creation of a set without extraneous intermediary sets.

`inittempset` creates a new temporary set. `tinsert` inserts an element into a temporary set. `installset` installs a temporary set into the set cache and returns its unique set representation.

361d ⟨*external declarations* 355a⟩+≡
```
void inittempset(Set, int);
void tinsert(Set, int);
void installset(Set);
```

362a ⟨*definitions* 355c⟩+≡

```
void inittempset(v, size) Set v;{
    initset(v, size);
}
void tinsert(v, mem) Set v; {
    int newsize;
    unsigned char *tmpv;

    if (!member(v, mem)) {
        if (mem/BITSPERCHAR >= v->charsize) {
            newsize = min(tempset.charsize, CHARSIZE(EXPANSION_FACTOR(mem)));
            assert(newsize > 0);
            tmpv = (unsigned char *)yyalloc(newsize, setarena);
            bzero((char *)tmpv, newsize);
            if (v->vals) bcopy((char *)v->vals, (char *)tmpv, v->charsize);
            v->charsize = newsize;
            v->vals = tmpv;
        }
        assert (mem/BITSPERCHAR < v->charsize && v->vals);
        v->vals[mem/BITSPERCHAR] |= (0x01 << (mem % BITSPERCHAR));
    }
    maxmem = max(maxmem, mem);
}
void installset(v) Set v; {
    v->vals = (unsigned char *)ustringn((char *)v->vals, v->charsize, &lsets);
    installed++;
}
```

18.4 Boilerplate

362b ⟨*set.h* 362b⟩≡

```
#ifndef SETH_INCLUDE
#define SETH_INCLUDE
⟨external declarations 355a⟩
#endif
```

This code is written to file set.h.

362c ⟨*set.c* 362c⟩≡

```
#include <c.h>
#include <string.h>
#include <set.h>

⟨local declarations 356⟩
⟨definitions 355c⟩
```

This code is written to file set.c.

18.5 String management

The source in this file is derived from `lcc`'s string module. It has been modified to support unique strings in `mill` code.

Strings are allocated from a string region of type `Strings`. The region includes the arena from which strings are allocated and a hash table for maintaining the unique strings.

363a ⟨*external declarations* 363a⟩≡

```
#define TABLESIZE 4096

typedef struct string {   /* strings: */
    char *str;            /* pointer to the string */
    int len;              /* its length */
    unsigned short index;         /* index in string region */
    struct string *link;  /* next one on this chain */
} *Buckets[TABLESIZE];    /* the string table */

typedef struct strregion {
    char *next;        /* slot for next allocated string */
    char *strlimit;    /* 1 past end of current string region */
    struct arena **arena;       /* arena from which strings are alloc'd */
    List strtbl;        /* list of strings in order written into string region */
    short sindex;       /* index of last string in strtbl */
    Buckets buckets;
}Strings;
```

This definition is continued in chunks 363c, 364d, and 368–70.
This code is used in chunk 371b.

The generic string interface uses `lstrings` for its string region.

363b ⟨*local declarations* 363b⟩≡

```
static Strings lstrings = { 0, 0, 0, 0, 0 };
```

This definition is continued in chunks 365–68.
This code is used in chunk 371c.

Unique strings are created by `string`, `stringd`, and `stringn`. `string` creates a copy of the null-terminated string. `stringd` converts an integer into a string. `stringn` saves a copy of a string with length `n`. `ustringn` is equivalent to `stringn` but it accepts a `Strings` region instead of using the default `lstrings`. This permits other modules to have their own regions for storing unique strings.

363c ⟨*external declarations* 363a⟩+≡

```
dclproto(char *ustringn,(char *, int, Strings *));
```

363d ⟨*definitions* 363d⟩≡

```
char *string(str) char *str; {
    char *s;

    for (s = str; *s; s++)
        ;
    return stringn(str, s - str);
}
```

This definition is continued in chunks 364–66 and 368–71.
This code is used in chunk 371c.

364a ⟨*definitions* 363d⟩+≡

```
char *stringd(n) {
    char str[30], *s = &str[30];
    unsigned m;

    if (n >= 0 && n < sizeof nums/sizeof nums[0])
        return nums[n];
    if (n == INT_MIN)
        m = INT_MAX + 1;
    else if (n < 0)
        m = -n;
    else
        m = n;
    do
        *--s = m%10 + '0';
    while (m /= 10);
    if (n < 0)
        *--s = '-';
    return stringn(s, &str[30] - s);
}
```

364b ⟨*definitions* 363d⟩+≡

```
char *stringn(str, n) char *str; {
    int i;
    struct string *p;

    assert(str);
    if (n > 0 && str[0] >= '0' && str[0] <= '9')
        if (n == 1)
            return nums[str[0]-'0'];
        else if (str[1] >= '0' && str[1] <= '9' && n == 2)
            return nums[10*(str[0]-'0') + str[1]-'0'];
    p = stringptr(str, n, &lstrings);
    return p->str;
}
```

364c ⟨*definitions* 363d⟩+≡

```
char *ustringn(str, n, strings) char *str; Strings *strings; {
    struct string *p;

    p = stringptr(str, n, strings);
    return (p->str);
}
```

`empty_strings` empties the given `Strings` region.

364d ⟨*external declarations* 363a⟩+≡

```
dclproto(void empty_strings,(Strings *));
```

365a ⟨*definitions* 363d⟩+≡

```
void empty_strings(strings) Strings *strings; {
    int i;
    strings->next = 0;
    strings->strlimit = 0;
    for (i = 0; i < TABLESIZE; i++)
        strings->buckets[i] = 0;
    strings->strtbl = 0;
}
```

`stringptr` returns the unique string `s` of length `n` in the `Strings` region `strings`. `hashstr` returns the hash key of `str`; `findstr` uses the hash key to find `str`. `newstr` creates a new unique string when it does not exist in the given `Strings` region.

365b ⟨*local declarations* 363b⟩+≡

```
dclproto(static struct string *stringptr,(char *, int, Strings *));
dclproto(static int hashstr,(char *, int));
dclproto(static struct string *findstr,(char *, int, unsigned, Strings *));
dclproto(static struct string *newstr,(char *, int, unsigned, Strings *, int));
```

365c ⟨*definitions* 363d⟩+≡

```
static struct string *stringptr(str, n, strings) char *str; Strings *strings; {
    unsigned h;
    struct string *p;

    h = hashstr(str, n);
    p = findstr(str, n, h, strings);
    if (!p) p = newstr(str, n, h, strings, 1);
    return p;
}
```

365d ⟨*definitions* 363d⟩+≡

```
static int hashstr(str, n) char *str; {
    unsigned h;
    int i;
    char *end;

    for (h = 0, i = n, end = str; i > 0; i--)
        h = (h<<1) + scatter[*end++];
    h &= TABLESIZE-1;
    return h;
}
```

365e ⟨*definitions* 363d⟩+≡

```
static struct string *findstr(str, n, h, strings)
         char *str; unsigned h; Strings *strings;{
    register char *s1, *s2;
    struct string *p;
    register int i;

    for (p = strings->buckets[h]; p; p = p->link)
        if (n == p->len)
            for (i = n, s1 = str, s2 = p->str; i > 0 && *s1++ == *s2++; i-- )
                if (i == 1)
                    return p;
    return (struct string *)0;
}
```

366a ⟨*definitions* 363d⟩+≡

```
static struct string *newstr(str, n, h, strings, nullterm)
    char *str; unsigned h; Strings *strings; {
    struct string *p;
    int i;

    if (strings->arena == (Arena *)0) strings->arena = &permanent;
    if (strings->next + n + nullterm >= strings->strlimit) {
        int m = roundup(n, 8) + BUFSIZE;
        strings->next = yyalloc(m, (*strings->arena));
        strings->strlimit = strings->next + m;
    }
    p = (struct string *) yyalloc(sizeof *p, (*strings->arena));
    p->len = n;
    p->index = 0;
    for (i = n, p->str = strings->next; i > 0 ; i--)
        *(strings->next)++ = *str++;
    if (nullterm) *(strings->next)++ = 0;
    p->link = strings->buckets[h];
    strings->buckets[h] = p;
    return p;
}
```

366b ⟨*local declarations* 363b⟩+≡

```
static char *nums[] = { /* strings for common numbers */
     "0",  "1",  "2",  "3",  "4",  "5",  "6",  "7",  "8",  "9",
    "10", "11", "12", "13", "14", "15", "16", "17", "18", "19",
    "20", "21", "22", "23", "24", "25", "26", "27", "28", "29",
    "30", "31", "32", "33", "34", "35", "36", "37", "38", "39",
    "40", "41", "42", "43", "44", "45", "46", "47", "48", "49",
    "50", "51", "52", "53", "54", "55", "56", "57", "58", "59",
    "60", "61", "62", "63", "64", "65", "66", "67", "68", "69",
    "70", "71", "72", "73", "74", "75", "76", "77", "78", "79",
    "80", "81", "82", "83", "84", "85", "86", "87", "88", "89",
    "90", "91", "92", "93", "94", "95", "96", "97", "98", "99"
};
```

367 ⟨*local declarations* 363b⟩+≡

```
int scatter[] = {       /* map characters to random values */
        2078917053, 143302914, 1027100827, 1953210302, 755253631,
        2002600785, 1405390230, 45248011, 1099951567, 433832350,
        2018585307, 438263339, 813528929, 1703199216, 618906479,
        573714703, 766270699, 275680090, 1510320440, 1583583926,
        1723401032, 1965443329, 1098183682, 1636505764, 980071615,
        1011597961, 643279273, 1315461275, 157584038, 1069844923,
        471560540, 89017443, 1213147837, 1498661368, 2042227746,
        1968401469, 1353778505, 1300134328, 2013649480, 306246424,
        1733966678, 1884751139, 744509763, 400011959, 1440466707,
        1363416242, 973726663, 59253759, 1639096332, 336563455,
        1642837685, 1215013716, 154523136, 593537720, 704035832,
        1134594751, 1605135681, 1347315106, 302572379, 1762719719,
        269676381, 774132919, 1851737163, 1482824219, 125310639,
        1746481261, 1303742040, 1479089144, 899131941, 1169907872,
        1785335569, 485614972, 907175364, 382361684, 885626931,
        200158423, 1745777927, 1859353594, 259412182, 1237390611,
        48433401, 1902249868, 304920680, 202956538, 348303940,
        1008956512, 1337551289, 1953439621, 208787970, 1640123668,
        1568675693, 478464352, 266772940, 1272929208, 1961288571,
        392083579, 871926821, 1117546963, 1871172724, 1771058762,
        139971187, 1509024645, 109190086, 1047146551, 1891386329,
        994817018, 1247304975, 1489680608, 706686964, 1506717157,
        579587572, 755120366, 1261483377, 884508252, 958076904,
        1609787317, 1893464764, 148144545, 1415743291, 2102252735,
        1788268214, 836935336, 433233439, 2055041154, 2109864544,
        247038362, 299641085, 834307717, 1364585325, 23330161,
        457882831, 1504556512, 1532354806, 567072918, 404219416,
        1276257488, 1561889936, 1651524391, 618454448, 121093252,
        1010757900, 1198042020, 876213618, 124757630, 2082550272,
        1834290522, 1734544947, 1828531389, 1982435068, 1002804590,
        1783300476, 1623219634, 1839739926, 69050267, 1530777140,
        1802120822, 316088629, 1830418225, 488944891, 1680673954,
        1853748387, 946827723, 1037746818, 1238619545, 1513900641,
        1441966234, 367393385, 928306929, 946006977, 985847834,
        1049400181, 1956764878, 36406206, 1925613800, 2081522508,
        2118956479, 1612420674, 1668583807, 1800004220, 1447372094,
        523904750, 1435821048, 923108080, 216161028, 1504871315,
        306401572, 2018281851, 1820959944, 2136819798, 359743094,
        1354150250, 1843084537, 1306570817, 244413420, 934220434,
        672987810, 1686379655, 1301613820, 1601294739, 484902984,
        139978006, 503211273, 294184214, 176384212, 281341425,
        228223074, 147857043, 1893762099, 1896806882, 1947861263,
        1193650546, 273227984, 1236198663, 2116758626, 489389012,
        593586330, 275676551, 360187215, 267062626, 265012701,
        719930310, 1621212876, 2108097238, 2026501127, 1865626297,
        894834024, 552005290, 1404522304, 48964196, 5816381,
        1889425288, 188942202, 509027654, 36125855, 365326415,
        790369079, 264348929, 513183458, 536647531, 13672163,
        313561074, 1730298077, 286900147, 1549759737, 1699573055,
        776289160, 2143346068, 1975249606, 1136476375, 262925046,
        92778659, 1856406685, 1884137923, 53392249, 1735424165,
        1602280572 };
```

18.5.1 Emitting strings in `mill` code

Unique strings are "pickled" in `mill` code. A string region in `mill` code contains the number of strings followed by `length, string` pairs. The index of the string in the string region is used in other "pickled" `mill` code to refer to the string.

`stridx` returns the index of string `str` in a the default `mill` code string region; `ustridx` the index in a specific region.

368a ⟨*local declarations* 363b⟩+≡
```
#define SINDEX 101
dclproto(static void emitString,(BaseTable *, List));
dclproto(extern int tell,(int));
```

368b ⟨*external declarations* 363a⟩+≡
```
dclproto(unsigned short stridx,(char *, int));
```

368c ⟨*definitions* 363d⟩+≡
```
unsigned short stridx(str, n) char *str; {
    return ustridx(str, n, &lstrings);
}
```

368d ⟨*external declarations* 363a⟩+≡
```
dclproto(unsigned short ustridx,(char *, int, Strings *));
```

368e ⟨*definitions* 363d⟩+≡
```
unsigned short ustridx(str, n, strings) char *str; Strings *strings; {
    struct string *p;

    assert(str);
    if (n > 0 && str[0] >= '0' && str[0] <= '9')
        if (n == 1) return ((str[0]-'0') + 1);
        else if (str[1] >= '0' && str[1] <= '9' && n == 2)
            return (10*(str[0]-'0') + (str[1]-'0') + 1);
    p = stringptr(str, n, strings);
    if (!p->index) {
        if (strings->sindex == 0) strings->sindex = SINDEX;
        p->index = strings->sindex++;
        strings->strtbl = push((Generic)p, strings->strtbl);
    }
    return p->index;
}
```

`mapidx2len` returns a string's length given its index. `mapstridx` returns a string given its index.

368f ⟨*external declarations* 363a⟩+≡
```
dclproto(int mapidx2len,(BaseTable *, unsigned short));
```

368g ⟨*definitions* 363d⟩+≡
```
int mapidx2len(BaseTable *bt, unsigned short idx) {
    if (idx > 0 && idx < SINDEX) return strlen(nums[(int)idx-1]);
    else if (intablerange(bt->stringmap, idx-SINDEX))
        return tableentry(struct string **, bt->stringmap, idx-SINDEX)->len;
    else error("mapidx2len: invalid string index %d\n", idx);
    return 0;
}
```

369a ⟨*external declarations* 363a⟩+≡
```
dclproto(char *mapstridx,(BaseTable *, unsigned short));
```

369b ⟨*definitions* 363d⟩+≡
```
char *mapstridx(BaseTable *bt, unsigned short idx) {
    if (idx == 0)
        return 0;
    if (idx > 0 && idx < SINDEX) return nums[(int)idx-1];
    else if (intablerange(bt->stringmap, idx-SINDEX))
        return tableentry(struct string **, bt->stringmap, idx-SINDEX)->str;
    else error("mapstridx: invalid string index %d \n", idx);
    return 0;
}
```

`breadStrings` reads the string region from the `mill` code file represented by `bt`.

369c ⟨*definitions* 363d⟩+≡
```
#ifdef MLD
void breadStrings(bt) BaseTable *bt; {
    char *s;
    short len, nstrings;
    int i;
    struct string **stringmap, *sptr;

    breadRegion(bt, bt->pbt.stringReg);
    bufread(bt, (char *)&nstrings, SHORTSZ);
    newtable(&bt->stringmap, nstrings, sizeof (struct string *), &(bt->arena));
    for (i = 0; i < nstrings; i++) {
        bufread(bt, (char *)&len, SHORTSZ);
        s = bufptr(bt);
        sptr = stringptr(s, len, &lstrings);
        updatetable(struct string **, (&bt->stringmap), i, sptr);
        bufincr(bt, len);
    }
}
```

369d ⟨*external declarations* 363a⟩+≡
```
dclproto(int strtablesize,(void));
```

370a ⟨*definitions* 363d⟩+≡
```
int strtablesize() {
    int i,j, n, max, min;
    struct string *sptr;
    for (sptr = lstrings.buckets[0], j = 0; sptr; j++, sptr=sptr->link) ;
    max = min = n = j;
    for (i = 1; i < TABLESIZE; i++) {
        for (sptr = lstrings.buckets[i], j = 0; sptr; j++, sptr=sptr->link) ;
        max = (j > max) ? j : max;
        min = (j < min) ? j : min;
        n += j;
    }
    fprint(2,"#strings %d %d %d %d; total %d\n", n, n/i, max, min, n * sizeof(struct string));
    return n;
}
#endif
void bemitStrings(bt) BaseTable *bt; {
    bemitLStrings(bt, &lstrings);
}
```

370b ⟨*external declarations* 363a⟩+≡
```
dclproto(void bemitLStrings,(BaseTable *bt, Strings *strings));
```

370c ⟨*definitions* 363d⟩+≡
```
void bemitLStrings(bt, strings) BaseTable *bt; Strings *strings;{
    unsigned short nstrings;

    bufflush(bt);
    bt->pbt.stringReg.base = tell(bt->fd);
    nstrings = strings->sindex - SINDEX;
    bufwrite(bt, (char *)(&nstrings), SHORTSZ);    /* Emit number of strings */
    if (strings->strtbl) emitString(bt, strings->strtbl);
    bufflush(bt);
    bt->pbt.stringReg.size = tell(bt->fd) - bt->pbt.stringReg.base;
}
```

370d ⟨*definitions* 363d⟩+≡
```
static void emitString(bt, s) BaseTable *bt; List s; {
    struct string *p;
    short len;

    if (s->link)
        emitString(bt, s->link);
    p = (struct string *)s->x;
    len = p->len;
    bufwrite(bt, (char *)&len, SHORTSZ);
    bufwrite(bt, p->str, p->len);
}
```

18.6 List management

370e ⟨*external declarations* 363a⟩+≡
```
dclproto(List push,(Generic, List));
```

371a ⟨*definitions* 363d⟩+≡

```
List push(x, list) Generic x; List list; {
    List new;
    new = (List)alloc(sizeof *new);
    new->link = list;
    new->x = x;
    return new;
}
```

371b ⟨*string.h* 371b⟩≡

```
#ifndef INCLUDE_STRINGH
#define INCLUDE_STRINGH
⟨external declarations 363a⟩
#endif
```

This code is written to file `string.h`.

371c ⟨*string.c* 371c⟩≡

```
#include <c.h>
#include <string.h>
#include <buf.h>
#include <aemit.h>
#include <bemit.h>

⟨local declarations 363b⟩
⟨definitions 363d⟩
```

This code is written to file `string.c`.

List of chunks

⟨ *add* `f` *to procedure index* 10b⟩
⟨*add new label following CALL* 14c⟩
⟨*add* `p` *to forest of nodes* 169b⟩
⟨*add* `p`*'s name to string table* 22c⟩
⟨*add* `p`*'s type to type table* 22d⟩
⟨*aemit.c* 66b⟩
⟨*aemit.h* 66a⟩
⟨*alias sanity check* 217a⟩
⟨*allocate a new arena for* `fn` 188d⟩
⟨*allocate default methods* 261c⟩
⟨`alu` *rs1, Op2, rd* 311f⟩
⟨*ambiguous definition* 219g⟩
⟨*ambiguous type definition* 219d⟩
⟨*ambiguous type use* 222c⟩
⟨*ambiguous var definition* 219c⟩
⟨*analysis.c* 253d⟩
⟨*analysis.h* 253b⟩
⟨ *announce a double* 126b⟩
⟨ *announce a symbol's address* 127a⟩
⟨*announce defs that reach this use* 270e⟩
⟨*announce temporary symbols* 168b⟩
⟨*append all* `s: x = c` *in* `fn` *to* `Q` 247c⟩
⟨*append all* `s: x = c` *in* `fn` *to* `Q` (never defined)⟩
⟨*append* `s: x = c` *to* `Q` 248a⟩
⟨*append* `s: x = c` *to* `Q` (never defined)⟩
⟨ *apply inlining* 119c⟩
⟨*archive.c* 343⟩
⟨*archive.h* 342c⟩
⟨*assemble assembly code* 330b⟩
⟨*assign* `$gp` *to sdata/sbss boundary address* 300d⟩
⟨*assign return value to appropriate temporary* 202a⟩
⟨*assign the remaining REF typecodes* 259b⟩
⟨*assignment to replacement symbol* 219a⟩
⟨*associate* `p`*'s dataval with use* 243a⟩
⟨*bemit.c* 66d⟩
⟨*bemit.h* 66c⟩
⟨*binary definitions* 43b⟩
⟨*binary external declarations* 43a⟩
⟨*binary local declarations* 46a⟩
⟨*binary.h* 285b⟩
⟨*binding example* 171b⟩
⟨*binding grammar* 171a⟩
⟨`branch^a,call` *reloc* 311g⟩
⟨*bread.c* 91⟩
⟨`bset,bclr,btog, mov` *Op2, rd* 312g⟩
⟨`bset,bclr,btog, mov` *Op2, rd* (never defined)⟩
⟨`btset` *Op2, rs1* 312f⟩
⟨*buf.c* 347b⟩
⟨*buf.h* 347a⟩
⟨`callN` *info* 37c⟩
⟨*cfold-node.spec* 97b⟩
⟨*cfold-tree.spec* 98a⟩

⟨*change call site in* `caller` 209d⟩
⟨*change dereference of formal to constant ref, if possible* 201b⟩
⟨*change formal reference to local, if necessary* 201c⟩
⟨*change Return node to Jump node* 202b⟩
⟨*change s':u = w o x to s': u = w o c* 246a⟩
⟨*check for argument count mismatch* 199b⟩
⟨*check if* `callee` *is converted call site* 164b⟩
⟨*check if* `fn`*'s callee selected for optimization* 164a⟩
⟨*check if method is overridden* 263b⟩
⟨*check if* `outB` *changed* 230e⟩
⟨*check if* `p` *is excluded from reachability search* 20c⟩
⟨*check that inlining doesn't cause cycle* 186b⟩
⟨*clear* `tempset` 359b⟩
⟨ *close open files* 119f⟩
⟨*collapse symbols and update any computed symbols* 201a⟩
⟨*commute* `dataval` *for each child of* `ADDP` *expression* 95a⟩
⟨*compute* `gksize`*, size of* `gen/kill` *sets* 226b⟩
⟨*compute* `ivmap_size` 213c⟩
⟨*compute mean size of procedures and methods* 141a⟩
⟨*compute* `sdefs`*, set of all* `(root,v)` 227b⟩
⟨*compute size of* `t` 260e⟩
⟨ *compute sizes of abstract data segments and initialize real segments* 119a⟩
⟨*compute sizes of children* 261b⟩
⟨*compute the link-time address of the segment* 282a⟩
⟨*compute total size of data segments* 300b⟩
⟨*compute* `t`*'s offset in supertype* `st`*'s method image* 263c⟩
⟨*compute* `vdefs`*, set of all* `(s,v)` 227c⟩
⟨*constant rules* 99a⟩
⟨*construct a new* `Address` 316e⟩
⟨`conststmt` = *true if* ∀ `def(x,s'')` ∈ `ud(x,s')`, `s'': x = c` 244c⟩
⟨*convert method to direct call* 275a⟩
⟨*copy parent's default methods* 262b⟩
⟨*copy* `v` *into* `tempset` 359a⟩
⟨*count defs in* `du(s',x)` 244a⟩
⟨*count defs in* `ud(x,s')` 244b⟩
⟨*count globals which may be used in optimization* 122d⟩
⟨*count leaf and non-leaf types* 275c⟩
⟨*create a CallN (procedure call) flow graph node* 13b⟩
⟨*create a JumpN (unconditional branch) flow graph node* 13d⟩
⟨*create a new* `RETI` *node* 154b⟩
⟨*create a RetN (procedure return) flow graph node* 14a⟩
⟨*create a StmtsN (straight-line) flow graph node* 14b⟩
⟨*create a TestN (conditional branch) flow graph node* 13c⟩
⟨*create an ArgN flow graph node* 13a⟩
⟨*create BaseTables for modules* 337b⟩
⟨*create* `du(sid,v)` *chain from* `sid` *to* `isid` 233f⟩
⟨*create* `lvmap/vmap/ivmap` *for locals and globals* 213d⟩
⟨*create* `smap/ismap` *for statements* 212c⟩
⟨*create table of contents* 142a⟩
⟨*create* `ud(v, root)` 233d⟩
⟨`D = D - ( (s,v)` | *root is an unambiguous def of v* `)` 233b⟩
⟨`D = D` *U* `( (root, v)` | *root is a definition of v*⟩ 233c⟩
⟨*data definitions* 124a⟩
⟨*data definitions* 344d⟩

⟨data structures 34a⟩
⟨dataimage.c 182c⟩
⟨dataimage.h 182b⟩
⟨deallocate s 190a⟩
⟨deallocate s if not inlined 189c⟩
⟨decl declarations 159a⟩
⟨decl definitions 160a⟩
⟨decl.c 170⟩
⟨decl.h 169d⟩
⟨default rules 104f⟩
⟨default tree rules 106a⟩
⟨definitions 6⟩
⟨definitions 45f⟩
⟨definitions 67b⟩
⟨definitions 79a⟩
⟨definitions 95f⟩
⟨definitions 121c⟩
⟨definitions 147d⟩
⟨definitions 152b⟩
⟨definitions 174a⟩
⟨definitions 184a⟩
⟨definitions 212a⟩
⟨definitions 255a⟩
⟨definitions 279a⟩
⟨definitions 286c⟩
⟨definitions 288a⟩
⟨definitions 304d⟩
⟨definitions 319b⟩
⟨definitions 333b⟩
⟨definitions 345a⟩
⟨definitions 349a⟩
⟨definitions 352c⟩
⟨definitions 355c⟩
⟨definitions 363d⟩
⟨def/use sanity check 215d⟩
⟨delete context 238b⟩
⟨delete def or use of s 253a⟩
⟨determine if p returns a function 69a⟩
⟨determine if p takes address of a function 69b⟩
⟨determine if return type of fn is known 273a⟩
⟨dominator sanity check 237b⟩
⟨dom(n) $= n \bigcup (\bigcap ($p $\in$ pred(n)$)$ dom(p)) 236d⟩
⟨dump dataimage 175b⟩
⟨du(s'',x) = du(s'',x) - (use(x,s')) 245a⟩
⟨eliminate all flowgraph nodes dominated by other 157a⟩
⟨eliminate CALL node 196b⟩
⟨eliminate jump 240b⟩
⟨eliminate overridden methods from type's default method image 264b⟩
⟨eliminate unnecessary jumps 240a⟩
⟨eliminate unused modules 120b⟩
⟨eliminate unused statement s 245c⟩
⟨ emit abstract data segments 45a⟩
⟨emit archive 341⟩
⟨emit data segment 318c⟩

⟨*emit data segment* 332c⟩
⟨*emit default method image* 266b⟩
⟨ *emit executable program* 119e⟩
⟨ *emit global symbol table* 44c⟩
⟨ *emit module's call graph* 44e⟩
⟨ *emit procedure index* 44b⟩
⟨*emit relatives* 266a⟩
⟨*emit segment sizes* 282d⟩
⟨*emit segment sizes* (never defined)⟩
⟨*emit segment sizes* (never defined)⟩
⟨*emit segment sizes* (never defined)⟩
⟨ *emit strings* 45b⟩
⟨*emit text segment* 318b⟩
⟨*emit text segment* 332b⟩
⟨ *emit type table* 44d⟩
⟨*emitbin epilogue* 278b⟩
⟨*emitbin epilogue* (never defined)⟩
⟨*emitbin epilogue* (never defined)⟩
⟨*emitbin epilogue* (never defined)⟩
⟨*emitbin prologue* 278a⟩
⟨*emitbin prologue* (never defined)⟩
⟨*emitbin prologue* (never defined)⟩
⟨*emitbin prologue* (never defined)⟩
⟨*evaluate individual sites* 185a⟩
⟨ *expect a floating point value* 176c⟩
⟨ *expect a memory image* 177d⟩
⟨ *expect a string* 177a⟩
⟨ *expect a structure or union* 177c⟩
⟨ *expect a symbol* 177b⟩
⟨ *expect an integer* 176b⟩
⟨*exported data declarations* 173a⟩
⟨*exported macro definitions* 173b⟩
⟨*exported procedure declarations* 173c⟩
⟨*external declarations* 40c⟩
⟨*external declarations* 45e⟩
⟨*external declarations* 95d⟩
⟨*external declarations* 121d⟩
⟨*external declarations* 148a⟩
⟨*external declarations* 152a⟩
⟨*external declarations* 161c⟩
⟨*external declarations* 183a⟩
⟨*external declarations* 211a⟩
⟨*external declarations* 254d⟩
⟨*external declarations* 333a⟩
⟨*external declarations* 344a⟩
⟨*external declarations* 351a⟩
⟨*external declarations* 355a⟩
⟨*external declarations* 363a⟩
⟨*external definitions* 32b⟩
⟨*external definitions* 127b⟩
⟨*fcompare rs1, rs2* 312b⟩
⟨*find base symbol of computed symbol* `s` 96a⟩
⟨*find binding values for symbol* `s` 95c⟩
⟨*find* `callee`*'s context* 239b⟩

⟨*find* `du(s,x)` 247a⟩
⟨*find* `du(s,x)` (never defined)⟩
⟨*find enclosing* `Entry` *node of* `Call` *node* 197b⟩
⟨*find leaf node of* `p` 95b⟩
⟨*find* `p`*'s flowgraph node* 246b⟩
⟨*find* `p`*'s flowgraph node* (never defined)⟩
⟨*find* `sid` *in* `v's` *du chain* 234a⟩
⟨*find* `s`*'s auxilliary type* 27e⟩
⟨*find temporary variables' uses that cross codelist boundary* 197c⟩
⟨*find type of object* 270d⟩
⟨*find* `ud(x,s')` 243e⟩
⟨*find* `ud(x,s')` (never defined)⟩
⟨ *flatten* `enum` 24c⟩
⟨ *flatten fields of* `struct`*s and* `union`*s* 24b⟩
⟨*float3 rs1, rs2, rd (*acode.insts[i].op)* 312a⟩
⟨`float2,mov_,neg2` *rs2, rd* 311h⟩
⟨`fn` *not yet emitted* 130b⟩
⟨`fn` *selected for optimization* 164d⟩
⟨*for each global, find global's auxilliary type* 27c⟩
⟨*for each local, find local's auxilliary type* 27d⟩
⟨*for each* `root` *in* `fgnode` 233a⟩
⟨*for each* `use(x,s')` ∈ `du(s,x)` 243c⟩
⟨*for each* `use(x,s')` ∈ `du(s,x)` (never defined)⟩
⟨*free all sets* 243b⟩
⟨*gen = gen* - (*(s,v)* | *root is an unambiguous def of v*) 227d⟩
⟨*gen = gen U* (*root* | *root is a definition of v*) 227e⟩
⟨*gen.c* 33d⟩
⟨*generate and emit code if called elsewhere* 192a⟩
⟨ *generate code* 119d⟩
⟨*generate code for inlined call site* 169a⟩
⟨*generate code for nodes rooted in* `fgnode` 168c⟩
⟨*generate dags for procedure prologue* 167a⟩
⟨*generate* `mill` *code for* `p` 11c⟩
⟨*generate nested descendents* 131d⟩
⟨*generate nested descendents* 191a⟩
⟨*generate new callee* 209c⟩
⟨*generic rules* 289⟩
⟨*generic rules* 305a⟩
⟨*generic rules* 322a⟩
⟨ `gen/kill` *sanity check* 228c⟩
⟨*get actual's value* 195c⟩
⟨*get archive name* 335c⟩
⟨*get archive size* 336a⟩
⟨*get assembly code operands* 311a⟩
⟨*get def/use info for* `root` 226d⟩
⟨*get formal's variable and type* 195a⟩
⟨*give multiply defined* `p` *a unique name* 150c⟩
⟨*give static* `p` *a unique name* 150b⟩
⟨*goal rules* 98d⟩
⟨*header files* 146⟩
⟨*identify reachable modules* 120a⟩
⟨*if dag is indirect CALL node, attempt to convert to direct call* 155b⟩
⟨`inB` = (*U (*`p` ∈ `pred(fgnode)`*)* `out(p)`) 230b⟩
⟨*increment current data segment size* 20b⟩

⟨*initialize* bss 295a⟩
⟨*initialize* data 294c⟩
⟨*initialize dominator sets* 236c⟩
⟨*initialize end & _end to end of bss section* 318a⟩
⟨*initialize end & _end to end of bss section* 332a⟩
⟨*initialize file header* 302a⟩
⟨*initialize file header* 317b⟩
⟨*initialize file header* 331b⟩
⟨*initialize file order* 131b⟩
⟨ *initialize* f *'s tables* 10a⟩
⟨*initialize* gen *and* kill *sets* 226c⟩
⟨*initialize* in *and* out *sets* 229c⟩
⟨*initialize* inB *to* out *of first predecessor* 230a⟩
⟨*initialize method images of subtypes* 264a⟩
⟨*initialize module order* 131a⟩
⟨*initialize* rdata 294b⟩
⟨*initialize* sbss 294e⟩
⟨*initialize* sdata 294d⟩
⟨*initialize* selfLink*s for root objects* 255b⟩
⟨*initialize sets for statement* 213b⟩
⟨*initialize sets for variable* 214b⟩
⟨*initialize software-implemented functions* 316b⟩
⟨*initialize symbol and node tables* 153a⟩
⟨*initialize target-specific assembly code* (never defined)⟩
⟨*initialize target-specific assembly code* 297e⟩
⟨*initialize target-specific assembly code* 309a⟩
⟨*initialize target-specific assembly code* 329e⟩
⟨*initialize* text 300a⟩
⟨*initialize* text 316a⟩
⟨*initialize universe* allN 236b⟩
⟨*initialize* uses*, table of all typedef uses* 27b⟩
⟨*inline inlineable call sites in this function* 189b⟩
⟨*inline nested descendents first* 189a⟩
⟨in/out *sanity check* 231a⟩
⟨*insert call in function's call graph* 17b⟩
⟨*insert* isid *in* du(sid, v) 234b⟩
⟨*insert statement in aliased variables'* adefs 220a⟩
⟨*insert statement in aliased variables'* auses 222d⟩
⟨*insert statement in variable's* adefs 220b⟩
⟨*insert statement in variable's* auses 222e⟩
⟨*insert statement in variable's* udefs 219b⟩
⟨*insert statement in variable's* uuses 222b⟩
⟨ *install basic types, symbols and constants* 7a⟩
⟨*install constant-literal global* 124b⟩
⟨*install ranlib entries* 337a⟩
⟨*install roots in* ismap 213a⟩
⟨*interface definitions* 29⟩
⟨ivmap *sanity check* 277a⟩
⟨*kill = kill -* (*root* | *root is a definition of v*) 228b⟩
⟨*kill = kill U* (*(s,v)* | *root is an unambiguous def of v*) 228a⟩
⟨*label a* CNSTI *integer constant* 94c⟩
⟨*label a* CNSTU *unsigned constant* 94d⟩
⟨ *label a floating-point constant* 94b⟩
⟨ *label kids first* 94a⟩

⟨*layout data* 123a⟩
⟨ldfsr,ldcsr,stfsr,stcsr,stdfq,stdcq,trap,rett,flush,call_,clr_,clrb,clrh,jmp *Address* (never defined)⟩
⟨*link* fgnode*'s roots into a forest* 169c⟩
⟨loadg,loada,loadf,loadc,jmpl *Address,rd* 311b⟩
⟨*local declarations* 5a⟩
⟨*local declarations* 45d⟩
⟨*local declarations* 67a⟩
⟨*local declarations* 80b⟩
⟨*local declarations* 92⟩
⟨*local declarations* 118a⟩
⟨*local declarations* 147a⟩
⟨*local declarations* 153e⟩
⟨*local declarations* 173d⟩
⟨*local declarations* 183b⟩
⟨*local declarations* 211c⟩
⟨*local declarations* 254a⟩
⟨*local declarations* 278c⟩
⟨*local declarations* 286b⟩
⟨*local declarations* 288b⟩
⟨*local declarations* 304c⟩
⟨*local declarations* 319a⟩
⟨*local declarations* 342b⟩
⟨*local declarations* 344c⟩
⟨*local declarations* 348⟩
⟨*local declarations* 356⟩
⟨*local declarations* 363b⟩
⟨*local macro definitions* 106b⟩
⟨*local macro definitions* 280f⟩
⟨*local macro definitions* 286a⟩
⟨*local macro definitions* 292a⟩
⟨*local macro definitions* 304a⟩
⟨*lookup supertypes that type t overrides* 263a⟩
⟨*lttype.c* 267d⟩
⟨*lttype.h* 267c⟩
⟨*m3 definitions* 268b⟩
⟨*m3 external declarations* 268a⟩
⟨*m3 local declarations* 268c⟩
⟨*main program* 118b⟩
⟨*make a* clone *of* callee 209a⟩
⟨*map actual to formal* 196a⟩
⟨*m3.c* 277c⟩
⟨*memory instrumentation* 138b⟩
⟨*m3.h* 277b⟩
⟨*mill.c* 78⟩
⟨*mill.h* 42b⟩
⟨*mips.binary* 303b⟩
⟨*mlccalloc.c* 350d⟩
⟨*mld definitions* 128b⟩
⟨*mld local declarations* 128a⟩
⟨*mld.c* 144c⟩
⟨*mld.h* 144b⟩
⟨*mldpp.c* 145a⟩
⟨*mranlib program* 141b⟩

⟨mranlib stabinit definitions 143⟩
⟨mranlib.c 145b⟩
⟨mtable.c 354d⟩
⟨mtable.h 354e⟩
⟨multitestN info 38a⟩
⟨new fn selected for optimization 164c⟩
⟨node constant rules 100a⟩
⟨node definitions 93⟩
⟨node rules 104e⟩
⟨node-simp.c 115a⟩
⟨ open and parse binding file 119b⟩
⟨open and read modules; install global symbols 122a⟩
⟨optmill.c 210c⟩
⟨optmill.h 210b⟩
⟨outB = genB U (inB - killB) 230d⟩
⟨p belongs in global symbol table 22b⟩
⟨p is s: x = c 247d⟩
⟨pad 165b⟩
⟨patch predecessors 237a⟩
⟨pool limit warning 345b⟩
⟨postprocess codelist 11b⟩
⟨pp program 140⟩
⟨prepend context to mfn 163⟩
⟨prepend context to mfn (never defined)⟩
⟨preprocess codelist 11a⟩
⟨ print contents of node 53b⟩
⟨print debug info 275b⟩
⟨print statement header 216a⟩
⟨print statement's adefs 216c⟩
⟨print statement's auses 216e⟩
⟨print statement's udefs 216b⟩
⟨print statement's uuses 216d⟩
⟨ print type of node 53a⟩
⟨print var's adef 224c⟩
⟨print var's ause 225a⟩
⟨print var's type 225b⟩
⟨print var's udef 224b⟩
⟨print var's uuse 224d⟩
⟨process command-line arguments 7b⟩
⟨promote callee's return value if it is a temporary 198c⟩
⟨promote temporaries that span Call node boundary 198b⟩
⟨propagate constant-valued pointer 242b⟩
⟨propagate simple constant 242a⟩
⟨propagate type 270c⟩
⟨Q sanity check 247b⟩
⟨Q sanity check (never defined)⟩
⟨ranlib symbol 142b⟩
⟨rdasr,not2 rs1, rd 312h⟩
⟨rdy,rdpsr,rdwim,rdtbr,neg,not rd 311d⟩
⟨read and generate code for procedure fn 132a⟩
⟨read and install strings 336c⟩
⟨read clean copy of callee's text 209b⟩
⟨read & inline inlineable call sites in this function 191b⟩
⟨read inlining file 128c⟩

⟨*read optimization file* 129a⟩
⟨*report argument information* 208⟩
⟨*report call site information* 165a⟩
⟨*resextsym.c* 151c⟩
⟨*resextsym.h* 151b⟩
⟨*resolve external references to* `p` 150d⟩
⟨`returnN` *info* 38b⟩
⟨*reveal all opaque types explicitly* 266c⟩
⟨*rules* 100b⟩
⟨`s` = *pop(*`Q`*)* 246c⟩
⟨`s` = *pop(*`Q`*)* (never defined)⟩
⟨*sanity check* 342a⟩
⟨*save current* `outB` 230c⟩
⟨*search backward from Call node for temporaries' definitions* 197d⟩
⟨*search forward from Call node for temporaries' uses* 198a⟩
⟨*set base address and size of* `data` *segments* 302c⟩
⟨*set base address and size of* `text` *segment* 302b⟩
⟨*set base addresses of data & bss segments* 317c⟩
⟨*set base addresses of data & bss segments* 331c⟩
⟨*set* `sbrk` *address _end == end* 300c⟩
⟨*set symbol id of callee in the caller's symbol table* 155c⟩
⟨*set.c* 362c⟩
⟨*set.h* 362b⟩
⟨`sethi,dec,deccc,inc,inccc,set` *val, rd* 312c⟩
⟨`setr,ldr` *reloc, rd* 312i⟩
⟨*setup context for inlined* `callee` 239a⟩
⟨*shuffle the typecells into their correct slots* 260a⟩
⟨*simp.h* 112b⟩
⟨ *simplify* `dagn` 153c⟩
⟨*sparc.binary* 318d⟩
⟨*special rules* 288c⟩
⟨*special rules* 304e⟩
⟨*special rules* 319c⟩
⟨*stabinit definitions* 135⟩
⟨*standard code-generator interfaces* 32a⟩
⟨`stbar, nop, ret, restore_, save_` 312e⟩
⟨`storeg,storea,storef,storec` *rd, Address* 311c⟩
⟨*str rd, reloc* 312j⟩
⟨*string.c* 371c⟩
⟨*string.h* 371b⟩
⟨*tag globals used in this function* 214e⟩
⟨*tag* `ReturnN` *node with type value* 272c⟩
⟨*tree definitions* 96b⟩
⟨*tree rules* 101d⟩
⟨*tree-simp.c* 113⟩
⟨*try to simplify expression, when type is known* 272d⟩
⟨*tst rs2* 312k⟩
⟨*TYPE definition* 172b⟩
⟨*types* 98b⟩
⟨`ud/du-chain` *sanity check* 234c⟩
⟨`ud(v,root)` = (`(s,v)` | `(s,v)` $\in D$) 233e⟩
⟨`ud(x,s')` = $\emptyset$ 245b⟩
⟨*unambiguous array definition* 219e⟩
⟨*unambiguous field definition* 219f⟩

⟨*unambiguous var definition* 218b⟩
⟨*unambiguous var use* 222a⟩
⟨ *unconditional jump to FALSE branch* 156d⟩
⟨ *unconditional jump to TRUE branch* 156c⟩
⟨*unimp imm22* 312d⟩
⟨ *unmill a* `Multitest` *flow-graph node* 153b⟩
⟨*unmill.c* 158c⟩
⟨*unmill.h* 158b⟩
⟨*unused definitions* 252c⟩
⟨*unused external declarations* 252b⟩
⟨*update* `bmilltab` 335b⟩
⟨*update flow graph* 12⟩
⟨*update* `lvmap` *and* `ivmap` *with locals* 214a⟩
⟨ *update region offsets* 45c⟩
⟨*update* `vmap` *and* `ivmap` *with globals* 214c⟩
⟨*use is assignment* `y = x` 270a⟩
⟨*use is return value* 270b⟩
⟨`use(x,s')` *is unambiguous* 243d⟩
⟨`use(x,s')` *is unambiguous* (never defined)⟩
⟨*visit children* 184b⟩
⟨*walk call graph* 121a⟩
⟨*wrasr rs1, Op2, rdi* 312l⟩
⟨*write data segment to* `fd` 303a⟩
⟨*write text segment to* `fd` 302d⟩
⟨`wry,wrpsr,wrwim,wrtbr,cmp` *rs1, Op2* 311e⟩
⟨*x86.binary* 332d⟩

References

Aho, Alfred, Ravi Sethi, and Jeffrey Ullman. 1986.
Compilers – Principles, Techniques and Tools.
Addison Wesley.

Fernández, Mary. 1995a (October).
A Retargetable, Optimizing Linker.
PhD thesis, Princeton University, Department of Computer Science.

———. 1995b (June).
Simple and effective link-time optimization of Modula-3 programs.
In *SIGPLAN Conference on Programming Language Design and Implementation.*

Fraser, Christopher W. and David R. Hanson. 1995.
A Retargetable C Compiler: Design and Implementation.
Redwood City, CA: Benjamin/Cummings.
ISBN 0-8053-1670-1.

Holler, Anne M. 1991 (March).
A Study of the Effects of Subprogram Inlining.
PhD thesis, Department of Computer Science, University of Virginia.

Ramsey, Norman. 1994 (September).
Literate programming simplified.
IEEE Software, pages 97–105.