The C-Breeze Compiler Infrastructure

Adam Brown
Samuel Z. Guyer
Daniel A. Jiménez
Calvin Lin
Department of Computer Sciences
The University of Texas
Austin, TX 78712 USA

January 26, 2006

This document describes how to use the C-Breeze compiler infrastructure. C-Breeze accepts ANSI C as input and produces various forms of output. C-Breeze is written with C++ and makes healthy use of the Standard Template Library (STL). This document assumes a familiarity with C++, STL, compiler terminology, and the grammatical structure of the C language.

1 Introduction

C-Breeze is a compiler infrastructure, written in C++, which accepts ANSI C (the ISO9899 standard) as input and which can produce various forms of output, including C and PowerPC assembly. C-Breeze comes with a set of built-in phases that perform basic operations. For example, there are phases to parse the input, dismantle the input into a canonical form, and produce a control flow graph. C-Breeze is also intended to be extended through the addition of new phases. This document explains how to use and extend C-Breeze.

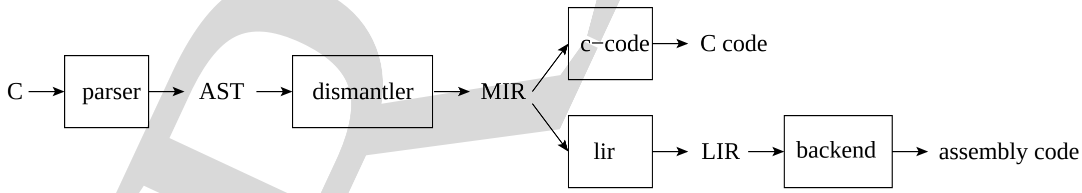

Figure 1: Overview of the basic C-Breeze compilation process.

The high level C-Breeze structure is defined by a series of built-in phases that define different forms of internal representation. As shown in Figure 1, the parser accepts C source code and produces an AST (Abstract Syntax Tree) representation. This AST is fairly high-level and bears a strong resemblance to C. The next phase in the figure, the dismantler, converts the AST into a canonical format known as MIR (Medium-level Internal Representation), which while still machine-independent, is a much simpler form with many fewer types of constructs. For example, in MIR all control flow is represented by labels and goto's, and all statements have at most a single assignment operator. The `-c-code` phase can be invoked to emit C as output. Eventually, another lowering phase will be provided to convert the MIR to a machine-specific LIR format, producing a 3-address code from which it is easier to generate assembly code. For most purposes, users will want to perform their analyses and transformations at the MIR level for simplicity reasons. Except for the parser, the invocation of the phases is under user control, so the structure shown in Figure 1 can be changed to suit other needs.

The remainder of this document is organized according to the various levels of intermediate representation. We start by describing aspects of the system that cross-cut the various levels of representation, namely, how to invoke C-Breeze and how to use the phase structure. We then describe two important IR's: the AST that is produced by the parser, and the MIR that is generated by the dismantler and its associated control flow graph We conclude by describing the C-Breeze class hierarchy.

2 The Basics of C-Breeze

Like most compilers, C-Breeze is organized as a series of phases. These can invoked programmatically from within another phase, or they can be invoked by specifying a command line flag to C-Breeze. Phases are executed in the order specified on the command line. For example, the following line will invoke the `cfg` phase to dismantle the AST and create a CFG, and then invoke the `c-code` phase to emit C code as output.

```
        cbz -cfg -c-code
```

Here and throughout this document, `cbz` refers to the name of the executable compiler. The one phase that is implicitly invoked before all other phases is the parser.

2.1 User-Defined Phases

User-defined phases can be named and invoked from the command line. To do this, define a class to be a subclass of `Phase` ane define a `run()` method that contains the code to run when the phase is invoked. For example, the trivial phase below prints `Hello, World!` to the standard output.

```
#include "c_breeze.h"

class Hello: public Phase {
        public:
        void run (void) {
                cout << "Hello, World!\n";
        }
};
```

To name this phase and register it as a flag that can be invoked from the command line, declare a global variable of type `Phases` (note the plural) constructed from the name of the phase (represented as a character string) and an instance of your subclass of `Phase`. In our Hello example, the following line registers a new phase that is invoked with the `-hello` flag.

```
Phases hello_phase ("hello", new Hello());
```

Here, the identifier `hello_phase` is a dummy variable that is never referenced; it is the invocation of the constructor that registers the phase. To define more than one phase in a file, you should obviously give them different names. It's not necessary for the names of the `Phases` variable, the subclass of `Phase`, or the identifying string to be related, but it does contribute to the readability of your code.

To create useful phases, you will want to access C-Breeze's various internal representations. These are available through the `CBZ` class, which is described in detail in Section 5. This class has several `public` and `static` members which can be accessed directly. Of particular interest is `CBZ::Program`. Each element on this STL list contains the C-Breeze representation for a translation unit (or source file). Most phases will traverse this list, applying some analyses or transformations to each unit.

For details on compiling C-Breeze, please see the release notes that accompany the source code. These notes include details on the required versions of gcc and bison.

2.2 Predefined Phases

This section briefly describes C-Breeze's predefined phases. These are listed here by their registered names, i.e., to invoke one, prepend a '-' to its name on the `cbz` command line:

2.2.1 ast

The AST phase prints a human-readable version of the abstract syntax tree to standard output. For example, given the following `foo.c`:

```
int main () {
        int     a;

        a = 1;
}
```

Entering `cbz -ast foo.c` will result in this:

```
 0.unit
 1..proc:
 2...decl: "main" proc
 3....func:
 4.....decl: (abstract) formal
 5......prim: void
 6.....prim: int
 7...block:
 8....decl: "a" block
 9.....prim: int
10....exprstmt:
11.....binary: op='='
12......const: val="1"
13.......prim: int
14......id: "a" decl=#8
```

2.2.2 c-code

The `c-code` phase produces a C code representation of the program, leaving the result in file(s) with names related to the input file(s). For example, entering `cbz -c-code foo.c bar.c` will parse, then unparse `foo.c` and `bar.c`, leaving the results in `foo.p.c` and `bar.p.c`.

2.2.3 ccobj

The `-ccobj` flag will use the CBZ::cc_cmd (usually gcc) to build an object (.o) file, which can then be linked to produce an executable. By using this flag, C-Breeze can be used as a replacement for cc or gcc in Makefiles. Note that like the `c-code` phase, this phase generates intermediate .p.c files.

2.2.4 cfg

The `cfg` phase first dismantles the C code into MIR form (see Section 4) and then creates a control flow graph using a `basicblockNode` to represent a basic block. The graph edges are referenced by the `succs()` and `preds()` STL lists in each `basicblockNode`; each `procNode` contains a body whose statement list (`stmts()`) is just a list of `basicblockNode`s that are the nodes in the CFG. The C code is also commented to show the pred/succ relationship among the blocks. For example, consider the following code:

```
int main(int argc, char ** argv) {
  int a, b, c;

  for (a = 1; a<=10; a++) {
    c = a % 2;
    if (c) {
      b = a * 2;
    } else {
      b = a;
```

```
    }
  }
}
```

After processing with the `-cfg` flag, the above code is translated to the following:

```
int main(int argc, char * * argv)
{
  int a;
  int b;
  int c;
  int __T7;
  int __T10;

  {
    /* # 1: preds( ) succs( 2 ) */
    a = 1;
    goto __T1;
  }
  {
    /* # 2: preds( 1 6 ) succs( 3 7 ) */
    __T1:
      ;
    if (a <= 10) goto __T3;
    goto __T2;
  }
  {
    /* # 3: preds( 2 ) succs( 4 5 ) */
    __T3:
      ;
    c = a % 2;
    if (c == 0) goto __T5;
    goto __T12;
  }
  {
    /* # 4: preds( 3 ) succs( 6 ) */
    __T12:
      ;
    b = a * 2;
    goto __T4;
  }
  {
    /* # 5: preds( 3 ) succs( 6 ) */
    __T5:
      ;
    b = a;
    goto __T4;
  }
  {
    /* # 6: preds( 4 5 ) succs( 2 ) */
    __T4:
      ;
    __T10 = a;
    a = a + 1;
    goto __T1;
  }
  {
    /* # 7: preds( 2 ) succs( ) */
    __T2:
```

```
      ;
    return __T7;
  }
}
```

Once the control flow graph has been generated, each function is guaranteed to have exactly one exit, and that exit is the last block in the list of statements. See `cfg.h` for information about invoking the control flow graph generator programmatically from C++ code.

2.2.5 `dismantle`

The `dismantle` phase produces the MIR intermediate format, which is described in Section 4.

2.2.6 `listphases`

The `listphases` phase prints a list of C-Breeze's pre-defined phases.

2.2.7 `vcgAST`

The `vcgAST` phase provides a way to graphically display the AST. This phase generates a specification of the AST that can be read by `xvcg`, a visualization tool that draws colored graphs. `xvcg` can display graphs on the screen and can save the graph in various formats, including PostScript. To visualize the AST for a program, `foo.c`, the `vcgAST` phase will produce the file `foo.c.vcgast`, which can then be visualized by `xvcg` as follows:

```
% xvcg foo.c.vcgast
```

If multiple input file names are given to C-Breeze, the output filename will be called `combined.vcgast`.

The output of this phase uses xvcg version 1.3 revision 3.17 running on Linux, but FreeBSD and MS Windows versions are also available. The tool has a number of useful features, including edge crossing reduction, subgraph folding, parameterized layouts, and rank assignment, many of which our phase does not exploit.

Pruning the Graph. When the source program is large, it is often desirable to exclude some nodes from the graph. For example, we might want to omit the standard C header files when displaying the AST. To exclude source files from the output graph, create a file, say `exclude.ast`, that has a list of the source files to be omitted. This exclusion file might look something like the following:

```
/usr/include/bits/types.h
/usr/include/libio.h
/usr/include/math.h
/usr/include/stdio.h
```

Note that the format of the exclusion file requires that each excluded file reside on a separate line. Once this exclusion file is created, the the `exclude flag` can be passed to the visualization phase by invoking C-Breeze as follows:

```
% cbz ... -vcgAST exclude:exclude.ast
```

2.2.8 `vcgCCG`

The `vcgCCG` phase is used to visualize the call graph of the input source programs. This phase works in the same manner as the `vcgAST` phase, except that it produces an output file with the `.callg` extension. As with the `vcgAST` phase, exclusion files can be used to limit the scope of this phase.

3 The AST Level

As its name implies, the AST representation is a tree. As mentioned in Section 1, most analyses and transformations will *not* be applied to the AST level. Nevertheless, to support the direct manipulation of the AST, C-Breeze defines two classes, walkers and changers, that simplify the traversal of the AST.

3.1 Walkers

A *walker* is a subclass of `Walker` that traverses the AST. Walkers provide control over the order in which nodes are visited (postorder, preorder or both) and the depth (top level nodes only, or visit all subtrees) of the traversal.

The `Walker` class uses the virtual dispatch of C++ to simplify this traversal: instead of defining the behavior for every possible node type, a walker need only specify actions for the node types of interest; all other nodes have a default action (i.e., nothing) performed on them. New actions can specified by by overriding the `at_`-functions of `Walker`. There is one `at_`-function for each C-Breeze node type, e.g., `at_for`, `at_decl`, etc.

3.1.1 `Walker` subclass definitions

A walker should be defined in a `.cc` file that `#includes` `"c_breeze.h"` (which will itself `#include "walker.h"`). You must define a constructor for the walker that specifies the order (`Postorder`, `Preorder` or `Both`) and the depth (`NodeOnly` or `Subtree`) of the traversal. You also provide member functions that override the default actions for the nodes you're interested in. These `void` functions accept two parameters: a pointer to the appropriate subclass of `Node`, and a parameter of type `Order` that will tell you whether the node is being visited in preorder (i.e., the parameter `== Preorder`) or postorder (i.e., `== Postorder`). You can ignore (but must still declare) this parameter if your `Walker` subclass only goes in one order.

3.1.2 A Sample Walker: `recursionWalker`

Below is a sample walker that will make this idea clear. When invoked on every `unitNode`, the `Walker` traverses the AST, listing all the recursive functions in the program.

```
class recursionWalker: public Walker {
        private:

        string current_proc_name;        // name of the current proc
        bool is_recursive;               // the current proc is recursive

        public:

        // The constructor calls the constructor for Walker, indicating that
        // we'll be descending subtrees in both preorder and postorder.

        recursionWalker (void): Walker (Both, Subtree) {};

        // What to do at a function definition

        void at_proc (procNode *p, Order ord) {

                // get the declaration for the function

                declNode *d = p->decl();

                // What order are we visiting this node from?

                if (ord == Preorder) {

                        // If we haven't visited the function body yet, simply
                        // remember the current procedure's name and initialize
                        // is_recursive to false

                        current_proc_name = d->name();
                        is_recursive = false;
                } else {

                        // After we're done with the function body, print a
```

```
                        // message if this function was determined to call
                        // itself

                        if (is_recursive)
                                cout << d->name() << " is recursive\n";
                }
        }

        // what to do at a function call (sub)expression

        void at_call (callNode *c, Order ord) {

                // If the 'name' (really the expression by which the function
                // is called) is an identifier expression whose name is the
                // same as that of the current function, the function is
                // recursive.  If we're not visiting in Preorder, then we're
                // done, so we'll avoid needlessly executing the 'if' body
                // again.

                if (ord == Preorder && c->name()->typ() == Id) {
                        idNode *i = (idNode *) c->name();
                        if (i->name() == current_proc_name)
                                is_recursive = true;
                }
        }
};
```

3.1.3 Invoking a Walker

Each AST node type has its own class which defines a `walk` member function that accepts an object of (sub)class `Walker`. By calling this member function of some node with an object of your walker class, you can start traversing the tree from that node. Often, you'll want to run your walker on the entire AST; to do this, you can use the following code segment, where `yourWalker` is the type name for your walker.

```
        yourWalker w;
        for (unit_list_p u=CBZ::Program.begin(); u!= CBZ::Program.end(); u++)
                (*u)->walk (w);
```

This code calls the `walk` method for each `unitNode` in the program using your walker.

3.2 Changers

Changers let you change the AST in C-Breeze. A *changer* is a subclass of `Changer`, which walks the AST, changing nodes as it goes. A changer is similar to a `Walker` subclass. You define your changer and override the `at_`-functions of the nodes you're interested in, but your new member functions must now return a `Node *` ; this is a pointer to the new value you want that node to assume.

Note: You need to be careful what you return from a changer's `at_`-function; it must "fit" in the AST where the old node was. For instance, if you return some sort of expression node from, say, the `at_stmt` member function, something is going to go wrong because somewhere in the tree there will be an expression node where a statement node belongs.

3.2.1 `Changer` subclass definitions

A changer is defined much like a walker. The constructor for `Changer`, which should be called by the constructor for the subclass, takes three parameters. The first two, the order and depth, are the same as the ones for `Walker`. The third is a `bool` that, if true, causes the new `Changer` subclass, when traversing the AST, to delete nodes that have been replaced with new nodes in the AST by the changer.

3.2.2 A Sample Changer

Here is a sample changer that changes all instances of the pre-increment operator (e.g. `++a`) to the equivalent assignment:

```
#include "c_breeze.h"

class preincChanger: public Changer {

        // What to do at a unary node

        Node * at_unary (unaryNode *u, Order) {

                // Is this node doing a preinc?

                if (u->op()->id() == Operator::PREINC) {

                        // Make a node for 'expr + 1'


                        binaryNode * add = new binaryNode ('+',
                                (exprNode *) ref_clone_changer::clone(u->expr(), false),
                                new constNode(1));

                        // Make a node for 'expr = expr + 1'

                        binaryNode *assg = new binaryNode ('=', u->expr(), add);

                        // Return the node

                        return assg;
                }

                // If the node wasn't a preinc, just return it

                return u;
        }

        public:

        preincChanger (void) :
                Changer(Postorder, // visit the tree in postorder (preorder
                                   // would have worked in this case as well)

                        Subtree,   // descend subtrees

                        false) { } // don't delete the old node; in this case,
                                   // the old node remains in the tree, it's
                                   // just placed a little deeper.
};

class preincPhase: public Phase {

public:

preincChanger ic;

void run() {
```

```
        unit_list_p u;

        for (u=CBZ::Program.begin(); u!= CBZ::Program.end(); u++) {
                (*u)->change(ic);
        }
}

};

// This line allows you to use the option "-preinc" to invoke
// our phase

Phases Phase("preinc", new preincPhase());
```

3.2.3 Invoking a Changer

A changer is invoked almost the same way a walker is, e.g.

```
        yourChanger c;
        for (unit_list_p u=CBZ::Program.begin(); u!= CBZ::Program.end(); u++)
                (*u)->change (c);
```

4 The MIR Level

The goal of the MIR level is to provide a well-defined internal representation that is simpler than the AST representation while still remaining machine-independent. At the MIR level, the tree structure of the AST has been flattened so that each basic block is a list of instructions, and each instruction is considerably simpler, allowing only a single assignment operator and only goto's as control flow. In addition to the canonical MIR format, there is also an intermediate MIR format that simplifies control flow without simplifying expressions.

The *dismantler* is responsible for converting an AST into MIR format. The dismantler can be invoked from the command line by specifying the `-dismantle` flag; to dismantle control flow only, use the `-dismantle-control` flag.

The dismantler can also be invoked programmatically, which can be useful for restoring the MIR format after a transformation. Note that the dismantler has been designed to be idempotent, so that dismantling a portion of code multiple times has the same effect as dismantling it once.[1] To use the new dismantler programmatically, use one of the static methods defined on the `Dismantle` class. You must include the file `dismantle.h` in order to see this class. The method `Dismantle::dismantle_control()` will dismantle control flow only. To completely dismantle the code, use the method `Dismantle::dismantle()`. Note that although the `Dismantle` class is a `Changer`, it is *not* intended to be used in a call to `Node::change()`; using the class directly in this way can lead to subtle bugs.

Finally, as mentioned in Section 2, the `-cfg` flag instructs C-Breeze to both convert to MIR and create a CFG.

The MIR format is not well-suited for loop transformations on multi-dimensional arrays, such as loop interchange or unroll and jam. The difficulty arises because in our MIR format each `operandNode` has an optional array subscript at the end. Since it only has a single subscript, multi-dimensional arrays cannot be represented directly but instead require the use of a temporary variable.

4.1 AST input

The code that the dismantler accepts as input must be a tree, and not a more general graph. If the input code is not a tree, the output of the dismantler is not guaranteed to be correct, reasonable, or meaningful. Two practices will ensure that the code the dismantler receives is a tree. First, use the dismantler as the first phase after parsing the input file. Second, use the `ref_clone_changer::clone()` method whenever you are assigning a node subclass pointer from a field in one object to a field in another object.

[1] Within the context of C-Breeze, it is important to note that dismantling a file, outputting it to C code, and then re-dismantling the generated C file is *not* guaranteed to be idempotent. We view the dismantler as idempotent only when applied to ASTs.

The dismantler also assumes that it may destructively change any `Node` in the input AST for use in the output AST. This assumption seriously affects what assumptions code using the dismantler can make both before and after using the dismantler. For example, any code which relies on a particular instance of `Node` being the same before and after dismantling is likely to have untold problems.

4.2 Hardware platform

The MIR representation corresponds to the operations available on a modern microprocessor. There are three major assumptions underlying this statement.

First, we assume that we are generating code for a load-store RISC-type processor. This assumption guides the selection of the basic addressable memory location used in operations.

Second, we assume that the architecture supports a register indirect with offset addressing mode; if the base address of an array or structure is loaded in a register, we can load the value from the proper offset location in a single instruction.[2]

Third, we assume that the instruction set architecture (ISA) supports moving values from a condition register(s) into a general-purpose register. This assumption allows us to leave assignments where the right-hand side uses a relational operator. To determine the value of the assignment, we can use shifting, masking, and inverting to extract the relevant bit from the condition register value.

4.3 The MIR Format

This section describes the details of the MIR format. Three `Node`-subclasses can appear in at the MIR level that cannot appear at the AST level: `operandNode, conditiongotoNode,` and `threeAddrNode`. We then describe the restrictions that are placed on existing `Node` types that may appear at the MIR level.

4.3.1 `Node`-subclass prototypes

The `indexNode` class simply acts as a superclass for `constNode` and `idNode`:

```
class indexNode : public exprNode {
  indexNode(NodeType typ, typeNode * type, const Coord coord);
};

class constNode : public indexNode { ... };
class idNode : public indexNode { ... };
```

The `conditiongotoNode` class corresponds to a conditional branch in the ISA. At the MIR level, all control flow structures in the input AST are represented by combinations of `conditiongotoNodes`, `gotoNodes`, and `labelNodes`. The `Operator` in the `op` field of the `conditiongotoNode` must be a relational operator (eg: `<`, `<=`, `==`, etc.).

```
class conditiongotoNode : public gotoNode {
  conditiongotoNode(labelNode * label, indexNode * left, unsigned int op_id,
                    indexNode * right, const Coord coord = Coord::Unknown);
  void left(indexNode * left);
  indexNode * left(void);
  void right(indexNode * right);
  indexNode * right(void);
  void op(Operator * op);
  Operator * op(void);
};
```

The `operandNode` class represents a memory location or address.

[2] Actually, this is a slight simplification. For arrays, we allow the index to be either a constant, in which case the previous statement is true, or a variable, in which case we require an additional load of the variable's value and a multiply of the array element size. In this later case, we require the architecture to support a register indirect with register offset addressing mode.

```
class operandNode : public exprNode {
  operandNode(indexNode * the_var, const Coord coord = Coord::Unknown);
  operandNode(indexNode * the_var, bool star, bool addr,
              const Coord coord = Coord::Unknown);
  operandNode(indexNode * the_var, bool star, bool addr, id_list * fields,
              indexNode * array_index, const Coord coord = Coord::Unknown);

  void addr(bool addr) { _addr = addr; }
  bool addr() { return _addr; }
  void star(bool star) { _star = star; }
  bool star() { return _star; }
  void cast(typeNode * cast) { _cast = cast; }
  typeNode * cast(void) { return _cast; }
  const typeNode * cast(void) const { return _cast; }
  inline id_list & fields() { return _fields; }
  inline const id_list & fields() const { return _fields; }
  void index(indexNode * index) { _index = index; }
  indexNode * index(void) { return _index; }
  const indexNode * index(void) const { return _index; }
  void var(indexNode * var) { _var = var; }
  indexNode * var(void) { return _var; }
  const indexNode * var(void) const { return _var; }
};
```

An `operandNode` represents a simple variable or constant when only the `var` field is set. Otherwise, it represents a memory location (or the address of a memory location) that can be loaded in a single instruction. This latter case assumes that the base address of the `var` field is already loaded into a register and the offset is either constant or already computed. In a grammatical sense, the `operandNode` is defined by the following rules:

$$\begin{aligned} \mathit{operand} &\rightarrow [\mathit{typecast}][\&](*\mathit{var})\{.\mathit{field}\}^*[\texttt{[}\mathit{index}\texttt{]}] \\ &\rightarrow [\mathit{typecast}][\&]\mathit{var}\{.\mathit{field}\}^*[\texttt{[}\mathit{index}\texttt{]}] \end{aligned}$$

The `threeAddrNode` is the quintessential form for all arithmetic statements in the dismantler's output:

```
class threeAddrNode : public stmtNode {
 public:
  // a = b;
  threeAddrNode(operandNode * lhs, operandNode * rhs,
                const Coord coord = Coord::Unknown);
  // a = -b;
  threeAddrNode(operandNode * lhs, unsigned int op_id,
                operandNode * rhs, const Coord coord = Coord::Unknown);
  // a = sizeof(<type>)
  threeAddrNode(operandNode * lhs, typeNode * type,
                const Coord coord = Coord::Unknown);
  // a = b (op) c;
  threeAddrNode(operandNode * lhs, operandNode * rhs1, unsigned int op_id,
                operandNode * rhs2, const Coord coord = Coord::Unknown);
  // a = b(...);
  threeAddrNode(operandNode * lhs, operandNode * func,
                operand_list * arg_list, const Coord coord = Coord::Unknown);
  // a(...);
  threeAddrNode(operandNode * func, operand_list * arg_list,
                const Coord coord = Coord::Unknown);

  void lhs(operandNode * lhs);
  operandNode * lhs(void);
  void rhs1(operandNode * rhs1);
```

```
  operandNode * rhs1(void);
  void op(Operator * op);
  Operator * op(void);
  void rhs2(operandNode * rhs2);
  operandNode * rhs2(void);
  void sizeof_type(typeNode * type);
  typeNode * sizeof_type(void);
  inline operand_list & arg_list();
  inline const operand_list & arg_list() const;
};
```

Using `threeAddrNode`, we can represent assignment of variables, unary operations to operands, binary operations to operands, function calls, and determining the `sizeof` a type or operand. For function calls, the assignment is optional. The following grammar describes the forms that a `threeAddrNode` can take.

$$
\begin{array}{rcl}
threeAddrNode & \rightarrow & \texttt{lhs} = [\mathit{unaryOperator}]\ \texttt{rhs1} \\
 & \rightarrow & \texttt{lhs} = \texttt{rhs1}\ \mathit{binaryOperator}\ \texttt{rhs2} \\
 & \rightarrow & \texttt{lhs} = \texttt{sizeof}\,(\mathit{type}|\texttt{rhs1}) \\
 & \rightarrow & [\texttt{lhs} =]\ \texttt{rhs1}(arglist) \\
arglist & \rightarrow & arg\ argrest \\
argrest & \rightarrow & , arg\ argrest \\
 & \rightarrow & \epsilon \\
arg & \rightarrow & operand
\end{array}
$$

4.3.2 Details of the MIR Format

The MIR format transforms the AST in the following ways. It converts the initialization of automatic variables into assignment statements at the beginning of the relevant block. Next, all `labelNode`s are modified so that the `stmt` field contains a `blockNode` containing only an empty statement (`exprstmtNode` with a `NULL` expression). Finally, the resulting AST is flattened to remove unnecessary nesting of blocks, and all variable declarations are moved up to the `procNode` level.

The first level of dismantling transforms complex control flow constructs. During this phase, all loops (`whileNode, doNode, forNode`), if statements (`ifNode`), and switch statements (`switchNode`) are replaced with equivalent `conditiongotoNode`s, `gotoNode`s, and `labelNode`s. Additionally, the ternary operator (`ternaryNode`) and short-circuit conditional operators (`&&, ||`) are transformed into equivalent `ifNode`s, which are subsequently transformed as mentioned above.

The second dismantling level transforms expressions into a series of *simple expression* with intermediate results. Roughly speaking, a simple expression corresponds to an instruction of a modern RISC microprocessor. Thus, simple expressions contain at most one operator (excluding assignments) and one assignment; operands must be addressable by a single `load` instruction.

Fully dismantled code only contains five types of statement nodes in the body of a procedure: `labelNode, gotoNode, conditiongotoNode, threeAddrNode,` and `returnNode`. This means that for processing the code only five methods for `Walkers`, `Visitors`, and `Changers` are needed to visit these nodes: `at_label(), at_goto(), at_conditiongoto(), at_threeAddr(),` and `at_return()`. The argument lists for these methods matches the convention for the other `at_*()` methods in the relevant class.

In MIR form, the `procNode` for the procedure has two new fields. The `return_label()` methods access a field containing a pointer to the `labelNode` at the beginning of the return block. The `declNode` for the returned value is accessible using the `return_decl()` methods.

4.3.3 Tree Traversal of MIR Code

This section describes how to traverse MIR code.

Traversal of conditiongotoNode: Since conditiongotoNode inherits from gotoNode, unless a traverser (Visitor, Walker, or Changer) defines an at_conditiongoto() method, the traverser's at_goto() method will be invoked on the conditiongotoNode. In addition to traversing the conditiongotoNode, Walkers and Changers will traverse the indexNodes on the left and right of the conditional expression (the left() and right() fields).

Traversal of operandNode: Unless a traverser defines an at_operand() method, the traverser's at_expr() method will be invoked. Walkers and Changers will traverse the indexNode for the var() field, the idNodes in the id_list in the fields() field, and the indexNode in the index() field.

Traversal of threeAddrNode: By default, traversing a threeAddrNode will invoke at_stmt() if a traverser does not define an at_threeAddr() method. Traversing the threeAddrNode will also traverse the operandNodes in the lhs(), rhs1(), and rhs2() fields, assuming that the field is non-NULL, the typeNode in the sizeof_type() field, when the operator is a sizeof, and the operandNodes in the operand_list in the arg_list field, when the operator is a FUNC_CALL.

5 The C-Breeze Class Hierarchy

This section describes the major components of C-Breeze's C++ class hierarchy, which is rooted in the Node class. Not all fields or member functions are described; some fields are seldom used or are just used for internal C-Breeze bookkeeping. The complete class hierarchy is available in the on-line documentation, and the header file ast.h contains the declarations for all the node types. Nodes follow the C++ convention that all data fields are private, being accessed through member functions with related names. Before describing these major classes, we define some terminology and conventions.

Note: Some fields are not initialized in the initial AST. You may have to set them yourself, or you may have to call another function to set them for you. For example, the fields for keeping type information for each expression node are set by the semantic checker and the dismantler.

5.1 Terminology and Conventions

Often, STL lists of different types of nodes will be used in node declarations. These list types are typedefed in cbreeze.h. The naming convention is to append the string _list to the (possibly abbreviated) name of the data type to form the list type name, and to append _p to the list type name in another typedef for the iterator of that list. For example, if stmtNode is a node type, stmt_list would be a list of nodes of type stmtNode, and stmt_list_p would be the iterator type for stmt_list.

The various node types are organized in a C++ class hierarchy. For example, the loopNode class has subclasses that represent for, do and while loops. A loopNode is a subclass of a stmtNode, an even more general node type containing any kind of statement. stmtNode is a subclass of Node, the most general node type.

This class hierarchy allows the C-Breeze programmer to operate on various levels of specificity. For instance, if you want to examine all loops, but don't care what kind of loop it is, you can write code concerning only loopNode's.

5.1.1 Members of Nodes

Many nodes have fields pointing to children nodes. The fields are named according to the syntactical nature of the child. For example, a loopNode contains an exprNode* (an expression node pointer) referred to by the member function cond(). This expression node is the condition that continues the loop. loopNode also contains a blockNode* referred to by body(). This block (of statements) node contains the loop's body.

Another example is the blockNode, which contains two STL lists of declaration and statement nodes, referred to by decls() and stmts(), respectively. A block is simply whatever declarations and executable statements occur between curly braces in C. blockNode is a subclass of stmtNode (statement node), since a compound statement is a statement.

C-Breeze follows a simple naming convention to provide two kinds of access—destructive and nondestructive—to AST child nodes. In general, for an AST child field named x, there is a nondestructive x() method that returns a pointer to the field, and there is a destructive get_x() method that returns a pointer to the field and fills in the old value with NULL. This second type of accessor is useful to ensure that the AST remains a tree, as it disallows aliasing amongst AST nodes. Note, that these `get_` methods are not consistently available, and their incorrect use may unintentionally remove portions of the tree.

5.1.2 The `CBZ` class

The `CBZ` class defined in `c_breeze.h` contains many public static members can be accessed directly. These include a container for the AST of the C program, flags and strings for accessing and controlling options (e.g. warning level), the current state of processing (e.g., what line number we are on) and so forth. See `c_breeze.h` for the definition of `CBZ`.

5.1.3 Units

A C program is composed of one or more translation units. You can specify more than one `.c` file on the `cbz` command line. Each of these files is parsed into its own translation unit, and the entire program is represented as an STL list of such units (each of type `unitNode*`). The list of units is available as the public static member `CBZ::Program`.

5.2 `Node`

This abstract class is the base type for all the other nodes. It has the following member functions:

Constructors:

- `Node(NodeType typ, const Coord coord, bool parenthesized = false);`

  This function makes a node of type `typ` (see below) at coordinate `coord` (see `coord.h`; the coordinate is basically the location in the C program text of this node) and with the `parenthesized` attribute, by default `false` (see below).

- `Node(const Node & other);`

  This function is a copy constructor, constructing a copy from an existing node.

Data fields (wrapped in member functions):

- `NodeType typ();`

  This method returns the type of this node. `NodeType` is an enumerated type in `c_breeze.h` with fields for all the different kinds of nodes. See `c_breeze.h` for the field names; they are pretty self-explanatory. For example, the value `While` indicates a while-loop node.

- `Coord coord();`
  `void coord(const Coord);`

  These methods return and set the coordinate of this node, i.e., location in the C program. See `coord.h` for details about coordinates.

- `bool parenthesized();`
  `void parenthesized(bool);`

  These methods return and set the “parenthesized” attribute of this node, that is, whether the node represents an expression inside parentheses. This helps when unparsing the program; rather than parenthesize aggressively, the unparser need only insert parentheses that were in the program to begin with.

Other Member Functions:

- `virtual typeNode * type();`

  This method returns `NULL` and can be overridden by a subclass to return the "type" of that node. **Note:** there are two senses in which we speak of the "type" of a node. One is the C++ sense: given a pointer to an object of an abstract class, what is the actual type of the object? The other is the parsed C sense: given a node representing an expression, what is the type of that expression (e.g., pointer, integer, etc.)? This method returns the type in the latter sense; the `typ()` field of `Node` returns the type in the former sense.

- `virtual typeNode * base_type(bool TdefIndir);`

  This method returns the "base type" of a node; how this differs from the "type" of the node depends on its parameter flag. The `TdefIndir` flag, if true, causes any `typedef` base types to be dereferenced; i.e., instead of returning a `tdefNode *`, `base_type()` will return the type referred to by the `typedef`.

- `virtual void visit(Visitor *);`

  This method can be overriden to provide a `Visitor` with something to do at that node.

- `virtual void walk(Walker &);`

  This method can be overriden to provide a `Walker` with something to do at that node.

- `virtual Node * change(Changer & , bool redispatch = false);`

  This method can be overriden to provide a `Changer` with something to do at that node.

- `virtual Node * clone() const =0;`

  This method returns a pointer to a newly allocated copy of the current node. Each subclass of `Node` simply overrides this method to return the result of calling the default copy constructor on itself.

- `virtual void output(output_context & ct, Node * par) =0;`

  Each subclass of `Node` overrides this method to send readable C code representing itself to the given output context. This method is useful for debugging. To output a node, first create an `output_context` that is initialized with the desired output stream. For example,

  ```
  output_context oc(cout); // output to cout
  ```

  Then call the node's output method as follows:

  ```
  node->output(oc, NULL);
  ```

5.3 `unitNode`

A `unitNode` corresponds to the parse tree for a single `.c` file (translation unit, in ISO9899 terms).
The `typ()` field is `Unit`.

Constructor:

- ```
  unitNode(string input_file, string output_file, string obj_file,
           const Coord coord = Coord::Unknown);
  ```

  The first parameter is the name of the input file. The second is the name of the output file, i.e., the file into which the `-c-code` phase will eventually unparse the AST. The `obj_file` parameter specifies the name of the object file, which is used by the `-ccobj` flag. The `coord` is the text coordinate of the unit, providing a context when an `#included` file generates an error message. You will not normally need to call the `unitNode` constructor because C-Breeze handles creating, parsing into and unparsing from them.
  ```

Data fields (wrapped in member functions):

- ```
  def_list & defs();
  const def_list & defs();
  ```

  These functions get and set the definition list for the `unitNode`. This is the list of global function and variable declarations and definitions.

- ```
  suespec_list & suespecs();
  ```

  Each unit has its own list of `suespecNode`s. These are the definitions to which `sueNode`s refer, i.e., this is where the actual definitions of `struct`s, `union`s and `enum`s reside, as opposed to having to duplicate their definitions for each `struct`/`union`/`enum` declaration and derived type node.

5.4 `defNode`

A `defNode` is an abstract class intended for `procNode` (procedure definition) and `declNode` (declaration) to subclass; it's basically a wrapper for these two types to allow them both to be in a `def_list`.

5.5 `declNode`

A `declNode` contains a declaration for a single object. What's an object? A variable, a `typedef`, and a function prototype are all examples of things that live in `declNode`s. `declNode` is a subclass of `defNode`, so it can live in a `def_list`, as you will find in a `unitNode`.

Typedefs:

`declNode` makes some `typedef`s:

- ```
  typedef enum {
     UNKNOWN, TOP, BLOCK, FORMAL, SU, ENUM, PROC
  } Decl_location;
  ```

  This enumerated type names the possible locations of the declaration. This information is needed for, among other things, deciding how to format the declaration for output to a `.c` file. For example, a declaration in a function prototype is followed by a comma, but a declaration in the global scope is followed by a semicolon.

- ```
  typedef enum {
     NONE, AUTO, EXTERN, REGISTER, STATIC, TYPEDEF
  } Storage_class;
  ```

  This enumerated type names the C storage classes. Note that a `typedef` is just a variable declaration with storage class `TYPEDEF`.

Constructors:

- ```
  declNode(const char * name, Storage_class sc, typeNode * the_type,
           exprNode * init, exprNode * bitsize, const Coord coord = Coord::Unknown);
  ```

  This method makes a `declNode` with name `name`, storage class `sc`, type `the_type`, initializer (possibly `NULL`) `init`, bitfield size (possible `NULL`) `bitsize`, and text coordinate `coord`.

- ```
  declNode(idNode * id, Storage_class sc, typeNode * the_type, exprNode
           * init, exprNode * bitsize);
  ```

  This method is similar to the previous one, but instead of a C character string, it uses an `idNode` to get the name for the object being declared.

- ```
  declNode(idNode * name, exprNode * value);
  ```

  This method is used when building a list of names for an enumerated type; you probably won't need to call this constructor.
  ```

- `declNode(typeNode * the_type, Storage_class sc);`

  This method is used to create a declaration from an abstract declarator, e.g., the parameter in `void foo (int[][40]);`

Data fields (wrapped in member functions):

- `typeNode * type();`
  `void type(typeNode * the_type);`

  These methods get and set the type of the declaration.

- `string & name();`
  `void name(string name);`

  These methods get and set the name of the object being declared.

- `exprNode * init();`
  `void init(exprNode * init);`

  These methods get and set the initializer for this object.

- `typeNode * no_tdef_type();`

  This method follows any `typedef`s in the type of the declaration to get the true type.

A Loop Peeling: An Example Changer

This appendix shows how to use a `Changer` to perform loop peeling. This example shows how to manipulate code at the AST level.

Loop peeling is a transformation that moves the first iteration of a loop outside the loop. Peeling a loop may expose the code to other code-improving transformations. See Muchnick, p.374, p.684 for more information.

A.0.1 `while`

For a `while` loop, we'll transform the following

```
while (cond) body;
```

into the following:

```
if (cond) {
        body;
        while (cond) body;
}
```

The original meaning is preserved; nothing is executed if the conditional is false, and the inner `while` loop is still guarded by the same conditional.

A.0.2 `do`

For a `do` loop, we'll convert the following

```
do {
        body;
} while (cond);
```

into the following:

```
        body;
        if (cond) {
                do {
                        body;
                } while (cond);
        }
```

Again, the semantics are preserved; the body of a `do` must be executed at least once; after that, if the conditional, `cond`, is still true, we continue with the `do` loop.

A.0.3 `for`

For `for` loops we'll cheat a little. We'll dismantle the following

```
        for (init; cond; next) body;
```

into the following equivalent:

```
        init;
        while (cond) {
                body;
                next;
        }
```

and then do the same transformation that we did for the `while` loop. Our transformation is incorrect if there is a `continue` in the body of the `for` loop, since `continue` doesn't mean the same thing for `while` as it does for the `for` statement.

(Note: C9X, the new ANSI C standard, will most likely allow `init` to be a variable definition with an initializer and scope limited to the `for` body, just like in C++; in this case, we will also need to enclose the new code in curly braces to limit the scope of `init`.)

A.1 A Loop Peeling Changer

A `Changer` can change every loop in a program, but this might not be desirable. We can use heuristics to decide when to peel, but we don't want to encode these into our `Changer` since we'd like to keep things simple and let some other optimization pass decide when to peel. So we'd like to structure our `Changer` as a module that can be used by another `Changer`.

The `Changer` can be invoked by passing an instance of it to a unit's `change` method, or by another optimization pass that subclasses from it.

A.1.1 Class Declaration: `looppeel.h`

Here is our `Changer`'s class declaration:

```
//
// looppeel.h
//

class LoopPeelingChanger: public Changer {

        // This function will peel a While loop

        Node * peel_while_loop (whileNode *);

        // This function will peel a Do loop

        Node * peel_do_loop (doNode *);
```

```
        // This function will peel a For loop

        Node * peel_for_loop (forNode *);

        // This method will peel all loops in the program

        virtual Node * at_loop (loopNode *, Order);

        public:

        // The constructor for LoopPeelingChanger will call the constructor
        // for Changer such that LoopPeelingChanger will visit tree in
        // Postorder.

        LoopPeelingChanger (void) :
                Changer(Postorder, Subtree, false) { }

        // This function peels one loop, calling one of the private functions
        // above.  peel_loop() can be used from a subclass of
        // LoopPeelingChanger to peel just one loop.

        Node * peel_loop (loopNode *);
};
```

Most of the member functions of a Walker or Changer accept and return a (subclass of) `Node *`. The parameter is the old loop, before peeling. The return value contains the newly constructed result of peeling the loop.

Note that `loopNode` is an abstract class, whose subclasses are `whileNode`, `doNode` and `forNode`. This gives rise to the `at_loop()` virtual function of the `Changer` (and `Walker`); from that point, we are guaranteed to be working with a loop, and we can find out (using the `typ()` of the `loopNode *`) what kind of loop it is, i.e. `typ()` will be `For`, `While` or `Do`.

A.2 Class Definition

Here are the member functions of `LoopPeelingChanger`, in order of decreasing generality. The code is explained in the comments.

A.2.1 `Node * at_loop (loopNode *, Order);`

```
#include "c_breeze.h"
#include "ast.h"
#include "ref_clone_changer.h"
#include "looppeel.h"

// Peel each loop we encounter.  This may not be a good criterion for loop
// peeling, but it demonstrates the code.  Subclass off of LoopPeelingChanger
// to implement your own loop peeling policy.  (Note: this depends on the
// LoopPeelingChanger only visiting the parse tree in postorder; preorder
// would cause it to peel loops that had just been peeled, ad infinitum.)

Node * LoopPeelingChanger::at_loop (loopNode *p, Order) {
        return peel_loop (p);
}
```

A.2.2 `Node * peel_loop (loopNode *);`

```
// A loop may be either a for, while, or do.

Node * LoopPeelingChanger::peel_loop (loopNode *p) {
```

```
        // call the appropriate peeling functions depending on the type of
        // this node

        switch (p->typ()) {
                case While:
                        return peel_while_loop ((whileNode *) p);
                case Do:
                        return peel_do_loop ((doNode *) p);
                case For:
                        return peel_for_loop ((forNode *) p);
                default:
                        fprintf (stderr, "unknown loop type!\n");
                        exit (0);
        }
}
```

A.2.3 `Node * peel_while_loop (whileNode *);`

```
// Loop peeling transformation for while:
//
// while (expr) stmt;
// =>
// if (expr) {
//      stmt;
//      while (expr) stmt;
// }
//
// Returns a node that is the result of transforming the while node into the
// if/while node described above.

Node * LoopPeelingChanger::peel_while_loop (whileNode *p) {

        // Build a new block to contain the If stmt body:
        //      { }

        blockNode *newblock = new blockNode (NULL, NULL);

        // Put the copy of the body of the While loop into the
        // if stmt body:
        //      { stmt; }

        newblock->stmts().push_back (
                (stmtNode *) ref_clone_changer::clone(p->body(), false));

        // Make a new copy of the While loop.  This is important because the
        // changer is going to change 'this' While loop with our new peeled
        // loop.

        whileNode *newwhile = new whileNode (
                (exprNode *) ref_clone_changer::clone(p->cond(), false),
                (stmtNode *) ref_clone_changer::clone(p->body(), false));

        // Put the copy of the While loop after the While loop body in the If
        // body:
        //      { stmt; while (expr) stmt; }

        newblock->stmts().push_back ((stmtNode*) newwhile);
```

```
        // Wrap everything up in an If stmt with a copy of the While condition
        // and return it:
        //      if (expr) { stmt; while (expr) stmt; }

        ifNode *newif = new ifNode (
                (exprNode *) ref_clone_changer::clone(p->cond(), false), newblock, NULL);
        return newif;
}
```

A.2.4 `Node * peel_do_loop (doNode *);`

```
// Loop peeling transformation for Do:
//
// do stmt while (expr);
// =>
// stmt; if (expr) { do stmt while (expr); }
//
// Returns a node that is the result of the above transformation.

Node * LoopPeelingChanger::peel_do_loop (doNode *p) {

        // We'll return a new block containing both the stmt and the If:
        //      { }

        blockNode *newblock = new blockNode (NULL, NULL);

        // Insert a copy of the Do loop body:
        //      { stmt; }

        newblock->stmts().push_back (
                (stmtNode *) ref_clone_changer::clone(p->body(), false));

        // Make a copy of the entire Do loop:
        //      do stmt while (expr);

        doNode *newdo = new doNode (
                (stmtNode *) ref_clone_changer::clone(p->body(), false),
                (exprNode *) ref_clone_changer::clone(p->cond(), false));

        // Prepend an If to execute the Do if the first iteration hasn't
        // changed the Do condition:
        //      if (expr) do stmt while (expr);

        ifNode *newif = new ifNode (
                (exprNode *) ref_clone_changer::clone(p->cond(), false), newdo, NULL);

        // Now put everything together in the first block:
        //      { stmt; if (expr) do stmt while (expr); }

        newblock->stmts().push_back ((stmtNode *) newif);
        return newblock;
}
```

A.2.5 `Node * peel_for_loop (forNode *);`

```
// `For' is a little trickier since it has so many elements.  We'll first
// transform it to a While stmt, i.e.:
//
// for (init; cond; next) stmt;
```

```
// =>
// init;
// while (cond) {
//      stmt;
//      next;
// }
//
// and then apply the loop peeling code for While.

Node * LoopPeelingChanger::peel_for_loop (forNode *p) {

        // Transform the For loop into a While loop.

        // Everything goes in a new block:
        //      { }

        blockNode *newblock = new blockNode (NULL, NULL);

        // First, insert the initialization of the For:
        //      { init; }

        newblock->stmts().push_back (new exprstmtNode (
                (exprNode *) ref_clone_changer::clone(p->init(), false)));

        // Now build the body of a new While loop...

        blockNode *whilebody = new blockNode (NULL, NULL);

        // ...from the body of the For:
        //      { stmt; }

        whilebody->stmts().push_back (
                (stmtNode *) ref_clone_changer::clone(p->body(), false));

        // ...and the next from the For:
        //      { stmt; next; }

        whilebody->stmts().push_back (new exprstmtNode (
                (exprNode *) ref_clone_changer::clone(p->next(), false)));

        // Now we wrap this body up in a While using the For condition:
        //      while (expr) { stmt; next; }

        whileNode *newwhile = new whileNode (
                (exprNode *) ref_clone_changer::clone(p->cond(), false),
                whilebody);

        // Do loop peeling on the while loop, inserting the result after the
        // init in the block we made before.

        newblock->stmts().push_back (
                (stmtNode *) peel_while_loop (newwhile));
        return newblock;
}
```

A.3 Discussion

There are some details in the code that aren’t immediately obvious.

We must have the Changer traverse the tree in postorder. We are changing loops into blocks of statements that contain loops. When we return a block in the place of a loop, a preorder traversal, having not visited the "next" statement in that block (indeed, it didn't exist before we created it) would march down to the loop inside the block we just created. Since our policy is to unpeel all loops (or let someone subclass off of us and make up their own policy), this would cause an infinite loop in which we keep peeling the same loops over and over again.

Note that we often make copies of Nodes. This is usually because we want the AST to remain a tree. If we use the same loop body (i.e., the same Node*) outside and inside a loop, then we are asking for trouble; later on, someone may decide to change something in the loop body, and that change will unintentionally occur outside the loop, too.

The following line

```
whilebody->stmts().push_back (new exprstmtNode (
        (exprNode *) ref_clone_changer::clone(p->next(), false)));
```

occurs in the code for for. Here, we are making a copy of the "next" expression of the for, and then turning it into a statement by constructing an exprstmtNode from it. An exprstmtNode is required because exprNode is not a subclass of stmtNode. When the C-Breeze AST expects a stmtNode, the retuned node *must* be a subclass of stmtNode. An expression can be a statement in C, but can also occur in contexts where a statement isn't allowed (such as a condition in a while loop). So we keep expressions that are statements in special exprstmtNodes. Note that if you don't convert the expression into a statement (it's easy to forget), C-Breeze might not immediately complain, but you will be introducing a bug into your program that will eventually catch up to you later on.

B Loop Peeling: A MIR Example

This section provides an example of a loop peeler that operates at the MIR level.

```
/*
Loop Peeler, for dismantled code
11/30/03 Walter Chang <walter@cs.utexas.edu>

This demonstrates loop peeling at the MIR level, using cfg, dominators,
loopTree, and related goodies.

Expects: Your stuff in CFG form.  Unfortunately, you must pass it an entire
         unit at a time, because that's what Dismantle::dismantle wants, and
         we need to re-dismantle to fix up labels.

Gives:   Dismantled code with loops peeled.
*/

#include <algorithm>
#include "c_breeze.h"
#include "looppeeldis.h"
#include "cfg.h"

LoopPeelingChangerDismantled::LoopPeelingChangerDismantled(void) :
  Changer (Postorder, Subtree, false) {}

/*
  at_proc(the_proc, ord)

  Runs dominators and loopTree on the_proc, then walks the loop
  tree for the_proc
*/
Node * LoopPeelingChangerDismantled::at_proc(procNode *the_proc, Order ord) {

  myProc = the_proc;
```

```
  /* let Dominators decorate things */
  Dominators d(the_proc, true);

  /* figure out the loop tree */
  loopTree l(the_proc);

  /* walk that tree */
  loopTreeWalk(l.topLoop());

  return the_proc;
}

/*
  at_unit(the_unit, ord)

  If we've finished everything for the unit (coming up in postorder),
  re-dismantle the unit to fixup label names.
  Changers should be postorder anyway.
*/
Node * LoopPeelingChangerDismantled::at_unit(unitNode *the_unit, Order ord) {
  cfg_changer::generate_cfg(the_unit);
  return the_unit;
}

/*
  change(u)
  Loop-peels the unit u.
*/
void LoopPeelingChangerDismantled::change(unitNode *the_unit) {
  if(!the_unit) return;
  LoopPeelingChangerDismantled lpcd;
  the_unit->change(lpcd);
}

/*
  change()
  Loop-peels everything.
*/
void LoopPeelingChangerDismantled::change() {
    unit_list_p u;
    for (u=CBZ::Program.begin(); u != CBZ::Program.end(); u++) {
      change( *u );  // change each unit
    }
}

/*
  loopTreeWalk(l)
  Walks loop tree in postorder.  Peel every natural (SingleEntry) loop.
*/
void LoopPeelingChangerDismantled::loopTreeWalk(loopTreeNode * l) {
  loop_set children = l->nestedLoops();
  loop_set::iterator child;

  /* walk the nested loops */
  for(child = children.begin(); child != children.end(); child++) {
    loopTreeWalk( (*child) );
  }

  /* peel this loop, if it is single-entry */
```

```
  if(l->kind() == loopTreeNode::SingleEntry) {
    peel(l);
  }
  return;
}

/*
  peel(loopTreeNode * l)

  Peels the single-entry loop indicated by l.

  This should leave the original entry blocks to the loop going to the new
  copy, and all backedges in the loop copy going to the loop top pointing to
  the old loop top.  The body of the loop is not otherwise tinkered with.
*/
void LoopPeelingChangerDismantled::peel(loopTreeNode * l) {

  /* make list of all basic blocks in l's blockset */

  /*
    NOTE:
    We implicitly assume that when we iterate over the block list in a loop,
    that the FIRST block is the loop header.  This assumption is empirically
    true as of 11/30/03.  Should this no longer hold, we must find and move
    the header to the front BEFORE ref-cloning the thing.
  */

  stmt_list loopBlocks = bbset2list(all_blocks(l));

  /* put all basic blocks into one blockNode */
  blockNode loopBodyBlock(NULL,
                          &loopBlocks,
                          Coord::Unknown,
                          Coord::Unknown);
  /*
    ref-clone the block with all the basic blocks in the loop.
    ref-clone should not change the order of the basic blocks.
  */
  blockNode * loopBodyCopy =
    (blockNode *) (ref_clone_changer::clone(&loopBodyBlock, false));

  stmt_list::iterator lp;
  basicblock_list::iterator lpp;
  stmt_list loop_preds;

  /*
    Look for the entries into the loop, ie, predecessors of
    the header that are not themselves part of the loop
  */
  basicblockNode * loop_header = l->header();
  basicblockNode * loop_newheader =
    (basicblockNode *) loopBodyCopy->stmts().front();

  for(lpp = loop_header->preds().begin();
      lpp != loop_header->preds().end();
      lpp++) {
    /* If this pred is NOT in the loop's blockset */
    if( find(l->blocks().begin(), l->blocks().end(),
             (*lpp) )
```

```
      == l->blocks().end()) {
    /* add this pred to the list of loop entries */
    loop_preds.push_front( (*lpp) );
  }
}

string lnhName = ((labelNode *) loop_newheader->stmts().front())->name();

/*
  A basic block always ends in a goto.  It is possible for the next-
  to-last to be a conditionGoto (but not a goto).  These are the only
  possible jump statements you will find in a basic block.
*/

/* For every (external) entry point to this loop... */
for(lp = loop_preds.begin(); lp != loop_preds.end(); lp++) {

  /* get the last statement in the block*/
  stmt_list::iterator lastStmtIter =
    ((basicblockNode *) (*lp))->stmts().end(); lastStmtIter--;

  /* If it comes in via the fall-through... */
  if( ((gotoNode *) (*lastStmtIter))->label()->name() == lnhName) {

    /* ...join that block to the NEW header */
    bbjoin( (basicblockNode *) (*lp), loop_newheader);
  }

  /* if there is a next-to-last... */
  if(lastStmtIter != ((basicblockNode *) (*lp))->stmts().begin()) {
    lastStmtIter--;
    /* ... and it is a conditionGoto... */
    if( (*lastStmtIter)->typ() == Condition ) {
      /* ... and it enters the loop... */
      if( ((conditiongotoNode *) (*lastStmtIter))->label()->name() ==
          lnhName) {
        /* ...join that conditionGoto to the NEW header*/
        bbjoincond( (basicblockNode *) (*lp), loop_newheader);
      }
    }
  }
}

/*
  Now we go through all the blocks in the loop, and for any block that goes
  to the top of the loop, make sure it goes to the OLD top of the loop, and
  not the top of the cloned body.  Ordinarily there should be just one such
  block.  [I'm not sure about continue...]
*/

/* The first thing in a basic block is ALWAYS a label */
labelNode * loopTop = (labelNode *) loop_header->stmts().front();

/* for all blocks b that head to the loop top */
for(lp = loopBodyCopy->stmts().begin();
    lp != loopBodyCopy->stmts().end();
    lp++) {
  /* The last statement in a basic block is ALWAYS a goto */
  gotoNode * endingGoto = (gotoNode *)
```

```
      ((basicblockNode *) (*lp))->stmts().back();

    /* If we're going to some loop top... */
    if(endingGoto->label()->name() == loopTop->name()) {
      /* ... make sure it goes to the OLD loop top */
      bbjoin( (basicblockNode *) (*lp), loop_header);
    }

    /* Now we fix up any conditional jumps to the top */

    stmt_list::iterator lastStmtIter =
      ((basicblockNode *) (*lp))->stmts().end(); lastStmtIter--;

    /* if there is a next-to-last... */
    if(lastStmtIter != ((basicblockNode *) (*lp))->stmts().begin()) {
      lastStmtIter--;
      /* ... and it is a conditionGoto... */
      if( (*lastStmtIter)->typ() == Condition ) {
        /* ... and it goes to some loop top... */
        if( ((conditiongotoNode *) (*lastStmtIter))->label()->name() ==
            loopTop->name()) {
          /* ...join that conditionGoto to the OLD loop top */
          bbjoincond( (basicblockNode *) (*lp), loop_header);
        }
      }
    }
  }

  /*
    At this point, all the labels and gotos for the blocks should be pointing
    at the correct places, and all that remains is to put the blocks into the
    procedure proper.
  */

  /* We'll try to add the new blocks immediately before the original */
  stmt_list::iterator orig_start = find(myProc->body()->stmts().begin(),
                                        myProc->body()->stmts().end(),
                                        l->header());
  for(lp = loopBodyCopy->stmts().begin();
      lp != loopBodyCopy->stmts().end();
      lp++) {
    /* Add block to main body */
    myProc->body()->stmts().insert(orig_start, (*lp) );

    /* Add to our parent loop's blockset as well
       Note that we'll never peel the root "loop" which has no parent */
    l->parentLoop()->addBlock( (basicblockNode *) (*lp) );
  }
}

/*
  bbjoin(b1, b2)

  Joins basic blocks b1 and b2, that is, b1's ending goto points to b2's
  starting label.  Use bbjoincond if you want the conditional to be changed.

  Note that bbjoin does NOT make sure that preds and succs are kept sane.
  This shouldn't be a problem in practice because we'll only be joining "on
  the inside" and re-dismantling will fix it anyway.
```

```
*/
void LoopPeelingChangerDismantled::bbjoin(basicblockNode * b1,
                                          basicblockNode * b2) {
  gotoNode * endingGoto = (gotoNode *) b1->stmts().back();
  labelNode * startLabel = (labelNode *) b2->stmts().front();

  /* remove old label's reference */
  endingGoto->label()->references().remove(endingGoto);
  /* hook up to new label */
  endingGoto->label(startLabel);
}

/*
  bbjoincond(b1, b2)

  Joins basic blocks b1 and b2 via b1's conditionGoto.  The conditionGoto in
  b1 will go to b2, and the fall-through remains unchanged.  Use bbjoin if
  you want to join via the fall-through.

  Note that bbjoincond does NOT make sure that preds and succs are kept
  sane.  This shouldn't be a problem in practice because we'll only be
  joining "on the inside" and re-dismantling will fix it anyway.

*/
void LoopPeelingChangerDismantled::bbjoincond(basicblockNode * b1,
                                              basicblockNode * b2) {
  labelNode * startLabel = (labelNode *) b2->stmts().front();

  /* Find the next-to-last node, if it is there... */
  stmt_list::iterator i = (b1->stmts()).end(); i--;
  if (i != (b1->stmts()).begin()) {
    i--;
    stmtNode * nextToLast = (*i);
    /* if it is a conditionGoto, fix the target to go to b2. */
    if (nextToLast->typ() == Condition) {
      ((conditiongotoNode *) nextToLast)->
        label()->references().remove((conditiongotoNode *) nextToLast);
      ((conditiongotoNode *) nextToLast)->label(startLabel);
    }
  }
}

/*
  bbset2list(bbset)

  Converts a basicblock_set to a basicblock_list
*/
stmt_list LoopPeelingChangerDismantled::bbset2list(basicblock_set &
                                                   bbset) {
  stmt_list bbl;
  basicblock_set::iterator bbsi;
  for(bbsi = bbset.begin(); bbsi != bbset.end(); bbsi++) {
    bbl.push_back( (*bbsi));
  }
  return stmt_list(bbl);
}

/*
```

```
  all_blocks(l)

  Returns all the blocks the loop l includes, including nested
  loops.  Changes loopTreeNodes so that blocks include nested loops.
*/

basicblock_set & LoopPeelingChangerDismantled::all_blocks(loopTreeNode * l) {

  loop_set children = l->nestedLoops();
  loop_set::iterator i;
  basicblock_set::iterator j;
  basicblock_set & b = l->blocks();
  basicblock_set tmp;

  for(i = children.begin(); i != children.end(); i++) {
    tmp = all_blocks( (*i) );
    for(j = tmp.begin(); j != tmp.end(); j++) {
      b.insert( (*j) );
    }
  }
  return b;
}
```

C Known Bugs

C-Breeze has a number of known bugs. This section describes a few of these, which remain because their causes are hard to find, or the cause is known but the solution is sufficiently difficult to implement that it has not yet been worth the effort.

1. *C-Breeze is a work in progress, and as such some of the members of the various classes may be defunct or uninitialized, but still included in the header files. Also, some members are not intended to be initialized unless the user (you) specifically requests it with a call to some method; for instance, a lot of type information is this way. Don't assume something does what its name implies it does; make sure to test it first.*

2. C-Breeze is not GCC. Although not exactly a bug, this can be annoying when trying to use `glibc` standard header files. In order to get certain functionality that many programs needs, C-Breeze can define the macro `__GNUC__`, that tells the header files that they are being read by GCC. Sometimes the header files think that this means it's OK to use GCC extensions not handled by C-Breeze, such as inline assembly language. C-Breeze can handle certain GCC extensions (as long as they are in header files, so that they'll be unparsed back to #include directives), but some others are not handles so easily. The workaround is to use or not use the `-gcc` flag on the C-Breeze command line. The presence or absence of this flag toggles certain preprocessor directives that try to accommodate weird GCCisms. If C-Breeze fails to compile a program, try using the `-gcc` flag and see if that helps. In particular, when using the `stat(2)` system call, the `struct stat` structure contains a field that is a scalar (a `long long`) when `__GNUC__` is defined, but an array (of two `int`s) when it isn't defined. When this field is treated as a scalar by user code, you get problems.

3. Error messages produced by the C-Breeze parser are not always useful in tracking down invalid C syntax.