The SUIF Library

A set of core routines for manipulating SUIF data structures

Version 1.0

Stanford Compiler Group

This manual is part of the SUIF compiler documentation set.

1 An Overview of SUIF

The SUIF compiler system is a flexible framework for advanced compiler research. The primary design goal for this compiler is that it be easy to use and build upon, without sacrificing its ability to compile real-world benchmark programs. Compared to other publicly-available compilers, SUIF may be somewhat inefficient and less robust, but it is much easier to modify and extend. This makes it an ideal platform for evaluating new compiler techniques.

The SUIF system is organized as a set of compiler passes built on top of a kernel that defines the intermediate format. The passes are implemented as separate programs that link with the kernel contained in the SUIF library. Each pass typically performs a single analysis or transformation and then writes the results out to a file. This is inefficient but flexible. SUIF files always use the same output format so that passes can be reordered simply by running the programs in a different order. New passes can be freely inserted at any point in a compilation.

The SUIF library provides an object-oriented implementation of the SUIF intermediate format. It is written in C++. The library defines classes to represent all the various elements of the intermediate format and to perform some common operations on them. It also contains the code to read and write the binary files that hold the SUIF code between passes of the compiler.

This document is a reference manual for the SUIF library. This first chapter provides an overview of the SUIF intermediate representation. Like many other complex data structures, this representation is roughly organized in a hierarchy. Objects near the top of the hierarchy contain the lower-level objects. The various elements of SUIF are described here starting from the top of the hierarchy and working down.

1.1 The File Level

SUIF is designed to support interprocedural analysis and optimization with separate compilation. At the root of the hierarchy for a SUIF program is a *file set* containing a list of the files being compiled. See Section 2.1 [File Set], page 7. Each entry within the file set is a *file set entry* that contains the input and output streams for a particular file. See Section 2.2 [File Set Entries], page 8.

The file level of the SUIF hierarchy also contains the global symbol tables. The file set contains the global symbol table that is shared across all of the files. This shared symbol table is the key to supporting interprocedural analysis. References to a global symbol or type from different files can point to the same entry in the shared global symbol table, making it easy to determine that they refer to the same entity. Each file set entry also contains its own symbol table for things declared privately within that file.

Lower levels of the SUIF hierarchy can be reached through the global symbol tables. Besides the types and variables, the global symbol tables contain symbols for the procedures. The procedure bodies can be accessed through these procedure symbols. If the body of a procedure is contained in one of the input files, the corresponding procedure symbol automatically records a pointer to the input file and provides a method to read the body into memory. The procedure symbol also has other methods to write the body to an output file and to flush the body from memory. Many SUIF programs need to process all the procedures. This can be done by searching through the global symbol tables for the procedure symbols. However since this is such a common task, the file set entries include procedure iterators to step through all the procedures.

For passes that work on one file at a time, the file set is not worth much, but it is very helpful for interprocedural passes. Without it, they would have to combine all the code into one big file. While that may not be an unreasonable solution, SUIF file sets provide the benefits of separate compilation with only a moderate increase in complexity.

1.2 The Procedure Level

Because SUIF is used in a wide variety of projects, the representation of procedure bodies includes both high-level and low-level information. In the first stages of a compilation, the high-level structure is represented by a language-independent form of abstract syntax trees (ASTs). This format, which we call *high-SUIF*, is well-suited for passes such as dependence analysis and loop transformation that need the high-level structure of the code. Later in the compilation process, the ASTs are reduced to sequential lists of instructions. This form, which we call *low-SUIF*, works well for some scalar optimizations and for code generation. Note that SUIF still has only one intermediate representation; high-SUIF and low-SUIF are both implemented with the same tree data structures. See Chapter 3 [Trees], page 11. They only differ in the amount of high-level information present. As long as a SUIF pass does not depend on particular features of high-SUIF or low-SUIF, it can handle either format. High-SUIF and low-SUIF may also be mixed together within the same procedure.

The leaf nodes of an AST are *instruction nodes*. See Section 3.1.3 [Instruction Nodes], page 13. Each of these nodes contains a single instruction or expression tree. In low-SUIF code, a procedure body is reduced to a list of instruction nodes containing individual instructions. This form resembles the quadruple representation used by traditional scalar optimizers.

Block nodes represent nested scopes. See Section 3.1.7 [Block Nodes], page 19. A block node contains a symbol table and a list of the AST nodes within the block. The scope of the symbols and types defined in the symbol table is restricted to the AST nodes within the block. They cannot be referenced from outside the block. The root node of an AST is a special kind of block node called a *procedure node*. Except for some extra methods and a slightly different kind of symbol table, a procedure node is the same as a block node.

Conditional structures may be represented by *if nodes*. See Section 3.1.4 [If Nodes], page 13. An "if" node has three parts, each of which is a list of AST nodes. The *header* list contains code to evaluate the condition and either branch to the *else* list or fall through to the *then* list. Because the header can contain control flow, it is easy to implement short-circuit evaluation of conditional expressions.

SUIF has two different kinds of loops. Many optimizations only apply to certain well-behaved types of loops, so *for nodes* are provided for loops with scalar indices that vary from their initial to final values, being incremented or decremented on every iteration, and that meet various other requirements. Most Fortran `DO` loops qualify as "for" nodes. Loops that do not meet those requirements are represented by *loop nodes*, which just record the control-flow structure of the loops and are used to represent generic "do-while" loops.

A loop node contains two lists of AST nodes. See Section 3.1.5 [Loop Nodes], page 15. The *body* list comes first and holds the loop body. The *test* list contains code to evaluate the "while" expression and conditionally branch back to the beginning of the `body` list. The body may contain branches to the `continue` and `break` labels, which are implicitly located at the beginning of the `test` list and the end of the loop, respectively.

A "for" node is by far the most complicated type of AST node. Besides the loop body, it must specify the index variable and the range of values for the index. The lower bound, upper bound, and step operands are expressions that are evaluated once at the beginning of the loop. The index variable is initially assigned the value of the lower bound and then incremented by the value of the step operand on every iteration until it reaches the upper bound; the code to do this is automatically created when the "for" node is expanded to low-SUIF code. The "for" node must also specify the comparison operator used to determine when the index variable has reached the upper bound value. The optional `landing_pad` part is a list of nodes to be executed once at the beginning of the loop; this provides a place to move loop-invariant code. As with loop nodes, the `body` list may contain branches to the `continue` and `break` labels.

1.3 The Instruction Level

Each instruction node in an abstract syntax tree holds a SUIF instruction. See Chapter 4 [Instructions], page 23. Most SUIF instructions perform simple operations; the opcodes resemble those for a typical RISC processor. However, more complex instructions are used in places where it is important to retain high-level information.

SUIF supports both expression trees and flat lists of instructions. In an expression tree, the instructions for an expression are all grouped together. This works well for high-level passes. Because expression trees do not totally order the evaluation of the instructions, they do not work so well for back-end optimization and scheduling passes. Thus SUIF also provides the flat list representation where each instruction node contains a single instruction.

Most SUIF instructions use a "quadruple" format with a destination operand and two source operands; however, some instructions require more specialized formats. For example, `ldc` (load constant) instructions have an immediate value field in place of the source operands. While most SUIF instructions are very simple, it is important to retain sufficient high-level information to support detailed analysis. Thus SUIF includes several instructions with more complex behavior:

- The `cal` (call) instruction implements a machine-independent procedure call with a list of parameters. This hides the details of various linkage conventions.
- The `mbr` (multi-way branch) instruction transfers control to one of the given labels depending on the value of its source operand. This represents computed `goto` and `switch` statements.
- The `array` instruction computes the address of an element in an array given a list of index values. These instructions are eventually expanded to additions and multiplications to perform the necessary pointer arithmetic.

These instructions are much easier to analyze than the equivalent series of low-level instructions.

The following example shows the low-SUIF instructions corresponding to a simple fragment of C code:

```
x = 0;
y = *ptr;
if (y < x + z) {
    *ptr = x;
}
```

```
1: ldc (i.32) x = 0          // load integer constant 0
2: lod (i.32) y = ptr        // load from address in ptr
3: add (i.32) nd#3 = x, z    // add x and z
4: sl  (i.32) nd#4 = y, nd#3 // set if y < x + z
5: bfalse     nd#4, L:L1     // branch if false to label L1
6: str        ptr = x        // store x to address in ptr
7: lab        L:L1           // label L1
```

Most of the operands in this example are variables; however, the results of the `add` and `sl` instructions are temporary values and are not stored in variables. Such temporary values occur frequently, and rather than requiring that new variables be created to hold them, SUIF allows them to be used directly. Each temporary must have a single definition and a single use within the same basic block. In the printed code above, the temporary values are indicated by "node" numbers, using the ID numbers of the instructions that produce the values. Internally, however, the operands contain pointers between the instructions. For example, the `bfalse` source operand contains a pointer to the `sl` comparison instruction, and the `sl` destination operand contains a pointer to the branch instruction. Thus the definition and use of a temporary value are directly connected, making it easy to find one from the other.

Flat lists of instructions work well for many back-end compiler passes, but for high-level transformations expression trees are often a better representation. Thus the SUIF system supports expression trees as well as flat lists. See Section 4.3 [Expression Trees], page 28. The difference between the two representations is actually quite small. The temporary value pointers described above naturally create trees, except that with flat lists the nodes of the trees are listed in bottom-up order. When using the expression tree representation, the instructions are rearranged so that only the roots of the expression trees are included in the AST instruction nodes. The other instructions are reached through the pointers in the operands. For example, by labeling each subtree (e.g. `e1` and `e2`), the expression tree in the example could be printed as:

```
5: bfalse e1, L:L1
4:   e1: sl (i.32) y, e2
3:     e2: add (i.32) x, z
```

In this case, only the branch instruction is directly contained in an instruction node. The `sl` instruction is reached through the branch's source operand, and the `add` instruction is contained in the second source operand of the `sl` instruction.

1.4 Symbolic Information

SUIF includes detailed symbolic information. Symbols and types are defined in nested scopes corresponding to the block structure of the program. A symbol table is attached to each element of the main SUIF hierarchy that defines a new scope. Symbols record information about variables, labels, and procedures. The SUIF type system is similar to C but also has some support for Fortran and other languages.

The symbol tables (see Chapter 5 [Symbol Tables], page 37) are defined in a tree structure that forms a hierarchy parallel to the main SUIF hierarchy. Each table records a pointer to its parent and keeps a list of its children. The global symbol table at the root is attached to the file set and is shared across all the files. Its children are the file symbol tables attached to the file set entries. The procedure symbol tables for the AST procedure nodes are in the next level down, followed by the block symbol tables for block nodes within the ASTs. The block symbol tables may be nested to any level.

Each symbol table contains a list of symbols (see Chapter 6 [Symbols], page 45) that are defined within the corresponding scope. There are three different kinds of symbols: variables, labels, and procedures. Symbols are identified by name or ID number. The ID numbers are assigned automatically and should always be unique within a particular scope. Each symbol also has a set of flags to record various attributes.

Variable symbols (see Section 6.4 [Variable Symbols], page 47) may be declared in any scope. A variable symbol contains a pointer to the type for the variable. The type determines the amount of storage used to hold the variable as well as the interpretation of its contents. Some additional flags are used for variable symbols. These flags identify variables that are formal parameters of the procedure and variables that have their addresses taken. Another flag is used to identify variables that represent machine registers. For variables that are not allocated on the stack, a separate *variable definition* (see Section 6.4.3 [Variable Definitions], page 51) must be entered in the symbol table. A variable definition records the alignment restriction for the variable's storage and its initial contents.

Label symbols (see Section 6.3 [Label Symbols], page 47) can only be declared within procedures. The position of a label in the code is marked with a special instruction.

Procedure symbols (see Section 6.2 [Procedure Symbols], page 46) can only be declared in the global and file scopes. A procedure symbol contains a pointer to the AST for the body of the procedure if it exists. It also provides methods to read the body from an input file, write it to an output file, and flush it from memory. The procedure symbol also has a pointer to the type for the procedure.

The SUIF type system (see Chapter 7 [Types], page 53) is quite powerful. It can represent most, if not all, high-level types for C programs and for many other languages. The types are implemented with various kinds of type nodes. Each type node contains an operator that specifies the kind of node. Some of the type operators define base types that stand alone, while other operators refer to other types nodes. For example, a type node with the `TYPE_INT` operator defines a new integer type. A node with the `TYPE_PTR` operator can then refer to the integer type node to create a type for pointers to integers. Operators are available for integers, floating-point types, pointers, arrays, functions, structures, unions, and enumerations. Each type has a particular size. For some types, such as integers, the size can be set directly, but for others it is determined automatically.

1.5 Other Data

SUIF is designed to be extended with new kinds of analyses and optimizations. These future extensions will generally require that additional information be attached to SUIF objects and propagated between passes. For example, one pass may analyze the SUIF code and produce information about data dependences. That information must be saved in the SUIF files so that it can be used by other passes. Because many different kinds of information may need to be stored, the SUIF library cannot predefine specific fields for future extensions. Instead, it provides *annotations* which allow user-defined data structures to be attached to most SUIF objects. See Chapter 8 [Annotations], page 63. This is the primary mechanism for making SUIF easily extensible.

Several different kinds of annotations are available. They all include a name field that identifies the kind of data in the annotation. Besides the name, each annotation has some sort of data. A *structured* annotation contains data in a user-defined data structure, whereas a *flat* annotation consists of a list of immediate values. The names of structured and flat annotations are registered with the *annotation manager*. See Section 8.6 [Annotation Manager], page 68. The manager records information about each kind of annotation, such as whether they are flat or structured and whether they should be written to the output files.

Annotations can also be used without registering them with the manager. To be able to write annotations to output files, SUIF must be able to represent them as lists of immediate values[1]. Because unregistered annotations are not written to the output files, they may contain arbitrary data structures. Obviously, they can only be used internally within one pass.

[1] The manager must be provided with functions to convert structured annotations to and from these immediate values.

2 Representation of Source Files

Each SUIF program has a single internal data structure, called a *file set*, which contains the global symbol table and a list of the files being compiled. Many passes work on a single file at a time, in which case the file set contains only one file and is not very useful. Interprocedural passes, on the other hand, typically need to work on all of the files at once. By using the file set, these interprocedural passes need not resort to combining all of the code into one big source file.

A source file within a file set is represented by a *file set entry*, which contains the symbol table for the file scope and the input and output streams. File sets provide the mechanism to allow multiple files to be read and written by a SUIF pass.

The symbol tables in the file set and file set entries basically form a table of contents for the entire program. They contain entries for every symbol and type that is declared outside of procedures. The procedures and everything within them can be accessed via the procedure symbols in the global symbol tables.

The '`fileset.h`' and '`fileset.cc`' files contain the source code for the file set and file set entries.

2.1 The File Set

The file set, which contains a list of file set entries, is implemented by the `file_set` class. The list of files can be accessed directly using the `file_list` method, but most users will not need to do that because other methods are available to perform the common operations. The `add_file` method adds a new file set entry to the list. This method takes the names of the input file and output file and returns the new file set entry. Either the input file or the output file may be `NULL`. The `file_set` class also contains an iterator, as most SUIF programmers will want to iterate through the list of files. The `reset_iter` method resets the iterator to the first file set entry. The `next_file` method returns the next file set entry in the list.

Besides the list of files, the file set also contains the global symbol table (see Section 5.1.1 [Global Symbol Table], page 37) which can be accessed using the `globals` method. This symbol table contains the symbols and types that are visible in all of the source files. Before a group of files can be combined into a file set, their global symbol tables must be merged together. A separate SUIF linker is provided for this purpose. The library will complain if the files in the file set have not been properly linked.

Currently, only one file set is allowed per SUIF program. This restriction was made so that predefined types could be entered into the global symbol table at initialization time. See Section 7.9 [Predefined Types], page 60. If multiple file sets were permitted, these predefined types would be duplicated. A single global file set is created when the SUIF library is initialized; the `fileset` global variable points to this file set.

All of the low-level access to SUIF files is performed automatically by the library. The global and file symbol tables are read from the input files when they are added to the file set. Users can then read and write individual procedure bodies using methods on the procedure symbols (see Section 6.2 [Procedure Symbols], page 46). The global symbol tables are not written out until the file set is destroyed. Throughout this process, the file set keeps track of the number of open files in order to avoid exceeding the OS limit. If there are too many files open at once, the file set will automatically find one to close. This is totally invisible to users. Files that are temporarily closed are automatically reopened as needed.

As an example of using the file set, the following code reads in the files `infile1` and `infile2` and writes them out to `outfile1` and `outfile2`, respectively, after processing them with the user-defined routine `process_file`.

```
fileset->add_file("infile1", "outfile1");
fileset->add_file("infile2", "outfile2");

file_set_entry *fse;

fileset->reset_iter();
while (fse = fileset->next_file()) {
    process_file(fse);
}

delete fileset;
```

2.2 File Set Entries

File set entries are implemented by the `file_set_entry` class. The `file_set_entry_list` class is also defined to hold the file set entries in the file set. A file set entry records the input and output streams for a source file. Either the input or output streams could be empty. The `name` method returns the name of the input file, if it exists. Otherwise it returns the name of the output file. The `parent` method returns the file set which contains the file set entry.

The file set entry also contains the file symbol table. See Section 5.1.2 [File Symbol Tables], page 38. The file symbol table holds the symbols and types declared within the scope of the file. This includes variables and functions declared with static linkage and types that are only visible within the file. The file symbol table is accessed using the `symtab` method.

The file set entry class provides an iterator to visit all of the procedures that are defined in the input file. Note that this does not include procedures that are declared but whose bodies are defined elsewhere. The `reset_proc_iter` method resets the iterator to the first procedure. The `next_proc` method returns the next procedure in the input file.

The following code reads each procedure in the given file set entry, calls the function `process_proc` on the procedure, and then writes it out.

```
void process_file (file_set_entry *fse)
{
    proc_sym *psym;

    fse->reset_proc_iter();
    while (psym = fse->next_proc()) {
        psym->read_proc();

        process_procedure(psym);
```

```
            psym->write_proc(fse);
            psym->flush_proc();
        }
    }
```

3 Abstract Syntax Trees

Procedure bodies are represented using a language-independent form of abstract syntax trees (ASTs). In high-SUIF code, the trees contain nodes corresponding to the high-level control structures in the source program. When the code is expanded to the low-SUIF form, these high-level constructs are replaced and the trees degenerate to lists of instructions.

SUIF is not restricted to handling programs with structured control flow. Even in high-SUIF code, low-level branch and jump instructions may be used to represent unstructured control flow. However, since most high-level transformations are only applied to structured code, the high-level AST nodes are only used for structured code. This makes it easy to identify and optimize regions of code with structured control flow.

The files '`trees.h`' and '`trees.cc`' contain the code for tree nodes and tree node lists.

3.1 Tree Nodes

The different kinds of AST nodes are represented by objects derived from the `tree_node` class. This base class includes features, such as ID numbers, that are shared by all kinds of tree nodes.

The simplest kind of tree node is a `tree_instr`. Each of these is a leaf node that contains a single instruction or expression.

Conditional structures may be represented by `tree_if` nodes.

The ASTs include two different kinds of loop nodes. The `tree_loop` nodes just record the control-flow structure of the loops and are used to represent generic "do-while" loops. Many optimizations, however, only apply to certain well-behaved types of loops. Thus, the trees may also contain `tree_for` nodes to represent loops with scalar indices that vary from their initial to final values, being incremented or decremented on every iteration, and that meet various other requirements. Most Fortran `DO` loops qualify as `tree_for` nodes. Those loops which do not meet the requirements for `tree_for` nodes are represented by `tree_loop` nodes instead.

Nested scopes are represented by `tree_block` nodes. A `tree_block` contains a list of tree nodes and a symbol table. The symbols and types in the table may only be referenced from within the block. The node at the root of an AST is a special kind of `tree_block` called a `tree_proc`. The `tree_proc` objects use a slightly different kind of symbol table and include some extra methods but are otherwise the same as the `tree_block` objects.

3.1.1 ID Numbers for Tree Nodes and Instructions

Within each procedure, the tree nodes and instructions are assigned unique ID numbers. These numbers can be used to identify the nodes and instructions internally and in annotations that are written to the output files. The `number` method retrieves the ID numbers. It is generally best to avoid using the numbers on `tree_instr` nodes, because they are not written to the output files and the `tree_instr` nodes are automatically rearranged when converting between expression trees and flat lists of instructions. Instead, use the ID numbers for the corresponding instructions.

The `number_instrs` method in the `tree_proc` class assigns ID numbers to tree nodes and instructions that do yet have numbers (i.e. those that were created since the procedure was read in). This method is called automatically before each procedure is written out, but you may also call it whenever ID numbers are needed for new tree nodes or instructions. The next unused ID number is recorded in the procedure symbol table. In the `proc_symtab` class, the `next_instr_num` method returns the next ID number and increments the counter; the `instr_num` method returns the next number but does not increment the counter. Since you can only assign new ID numbers with the `number_instrs` method, you will rarely, if ever, need to access this counter directly.

In some circumstances, the tree node and instruction ID numbers must be reset. The most common case is when moving code from one procedure to another. Since the ID numbers are only unique within each procedure, moving things between procedures is likely to create duplicate ID numbers. To avoid this problem, the `tree_node`, `tree_node_list`, and `instruction` classes provide `clear_numbers` methods to reset the ID numbers. These methods recursively descend the ASTs and expression trees to set all of the ID numbers within the object and its children to zero.

3.1.2 Other Features Shared by All Tree Nodes

The `tree_node` class includes several methods to make it more convenient to retrieve information that is not directly stored in a node. The `scope` method returns the symbol table for the closest enclosing scope at a tree node. The `proc` method ascends to the root of the AST to find the symbol for the procedure containing the tree node. Of course, that only works if the node is properly attached to an AST that is connected to a particular procedure symbol.

Textual representations of tree nodes can easily be printed out to files using the `print` method. The optional `depth` parameter specifies the indentation level for the output. The output formats for most kinds of tree nodes are shown in subsequent sections. For instruction nodes, the SUIF instructions are printed directly.

All tree nodes, except for `tree_proc` nodes, are stored on tree node lists. Each tree node has a back-pointer to the list that contains it. The `parent` method is used to access this list pointer. The parent of a `tree_proc` is always `NULL`. The parent list pointers are set automatically when tree nodes are added to lists, so you do not have to deal with maintaining these yourself.

Like the other lists in SUIF (see Section 11.1 [Generic Lists], page 79), tree node lists are made up of list elements. Each of these list elements contains a pointer to a particular tree node. Because many of the list methods operate on the list elements rather than the tree nodes, one often needs to know the element containing a tree node. Thus the tree nodes contain back-pointers to the list elements, which can be accessed with the `list_e` method. Like the parent pointer, the list element pointer is automatically set when a tree node is entered in a list.

For example, the list elements are needed for inserting nodes before or after other nodes in a list and for removing nodes from their lists. The following code inserts the tree node `new_node` before the tree node `cur_node` and then removes `cur_node` from the list:

```
tree_node_list *tnl = cur_node->parent();
tnl->insert_before(new_node, cur_node->list_e());
tree_node_list_e *elem = tnl->remove(cur_node->list_e());
```

3.1.3 Instruction Nodes

Instruction nodes, implemented by the `tree_instr` class, are the leaves of the abstract syntax trees. Each instruction node contains a single instruction or expression tree which can be accessed using the `instr` method. The `set_instr` method attaches an instruction to a `tree_instr`, and the `remove_instr` method detaches an instruction. Note that `remove_instr` does not free the storage for the instruction, and unlike the `remove` method in the `instruction` class (see Section 4.1.5 [Source Operands], page 26), it does not remove the instruction from its expression tree. Conversely, removing an instruction from an expression tree with the `remove` method will automatically call `remove_instr` if necessary.

The SUIF library automatically maintains back-pointers from instructions to their parent `tree_instr` nodes. As a result, the `set_instr` and `remove_instr` methods must always be used to attach and detach instructions to `tree_instr` nodes. Never use the instruction in a `tree_instr` elsewhere without first calling `remove_instr`, and never use `set_instr` with a `tree_instr` that already contains an instruction.

Unlike other tree nodes, instruction nodes are considered temporary objects. They are destroyed and created automatically by the SUIF library when switching between expression trees and flat lists, and their annotations and ID numbers are not written to the output files. Consequently, annotations should generally not be attached to them but should instead be placed directly on the instructions.

The destructor for the `tree_instr` class also deletes the instruction or expression tree contained in the node. This is the desired behavior for most cases. However, if you only want to delete the `tree_instr` node, simply call `remove_instr` first and save a pointer to the instruction.

Often in SUIF one needs to perform some operation on all the instructions in a procedure. The `map` methods in the `tree_node` and `tree_node_list` classes (see Section 3.3 [Mapping Subtrees], page 21) make it easy to visit all of the instruction nodes. Once an instruction node is reached, its `instr_map` method may be used to apply a function to every instruction in the associated expression tree. Given a function of type `instr_map_f`, `instr_map` will traverse the expression tree and apply the function to every instruction. The `preorder` parameter is used to select either preorder or postorder traversals; the default is preorder. The `instr_map_f` function takes two arguments: the instruction pointer and a `void*` value that you provide to `instr_map`.

3.1.4 If Nodes

Nodes from the `tree_if` class represent "if-then-else" structures. A `tree_if` contains three tree node lists: the `header`, the `then_part`, and the `else_part`. The `header` contains code to evaluate the "if" condition and branch to the `jumpto` label, implicitly located at the beginning of the `else_part`, if the condition is false. Otherwise it falls through to the `then_part`. The `then_part` implicitly ends with a jump around the `else_part`. The `tree_if` class provides methods to access all of these parts.

The `jumpto` label must be defined in the scope containing the `tree_if` node, but the label position is implicit rather than being marked by a label instruction. When the `tree_if` is expanded to low-SUIF form, a label instruction for `jumpto` is inserted at the beginning of the `else_part`. The `jumpto` label also has the restriction that it may only be used in the header list of its tree_if; it is illegal to use this label at all in the then_part or else_part, or at all outside the tree_if.

Although the tree node lists within a tree_if can hold arbitrary ASTs, certain conventions must be followed.

The header list is only allowed limited control flow. It may contain tree_instr nodes and other tree_if nodes arbitrarily nested, but it is not allowed any tree_block, tree_for, or tree_loop nodes, even in nested tree_if nodes. Label instructions may be used within the header, but jumps or branches from outside the header into it are not allowed; such a label can only be the target of branches or jumps from elsewhere in the header. Similarly, all of the branch and jump instructions are allowed within the header, but all the target labels must be within the header list with one important exception: branch and jump targets may include the jumpto label of the original tree_if node.

The idea is that the header part contains the computation to figure out whether the then_part or else_part is executed. Often this will be a single conditional branch instruction to the jumpto label. But we want to allow more complex test computations than a single SUIF expression. In particular, we want to be able to capture all the computation in a C test expression, which may contain short-circuit evaluation of && and || operators and partial evaluation of the ?: operator. The front end puts the computation of the test expression of a C if statement in the header of the tree_if it creates. With complex C expressions, it is often useful to use the structured control flow of nested tree_if nodes within the header and that's why they are allowed. It is legal to have an empty header, but any tree_if with an empty header, or even with a header that contains no branch or jump to the jumpto label is degenerate–the then_part will always be executed instead of the else_part.

The then_part and else_part have fewer restrictions. After the front end, in standard SUIF, neither is allowed to have potential jumps or branches in or out, or between the then_part and else_part. Branches or jumps anywhere within the then_part must be to labels elsewhere in the then_part and labels in the then_part may only be used as targets by branches or jumps within the then_part; the same holds separately for the else_part. Note that the symbol table containing a label may be outside the tree_if; it is the label instruction we are concerned with here. If the label instruction is within the else_part, it may only be used as a target by instructions in other sections of the else_part. Any tree_nodes can be used within the then_part and else_part with arbitrary nesting, so long as the restriction on jumps into or out of the top level else_part or then_part lists is honored.

Either or both of the then_part and else_part may be empty lists; in fact the else_part is often empty, as in C there is often an if statements without an else. There is little point in having a tree_if if both part these parts are to be empty, but this may occur if all the code is optimized away or moved elsewhere.

Here's an example. The following C code could be translated into the SUIF code shown in a simplified form below. (The real C front-end generates a more complicated tree that involves nested tree_if structures in the header.)

```
if ((x > y) && (x > 0)) {
    y = x;
} else {
```

```
        y = 0;
    }
    IF (Jumpto=L:__L1)
    IF HEADER
        bfalse e1, L:__L1
          e1: sl y, x
        bfalse e1, L:__L1
          e1: sl e2, x
            e2: ldc 0
    IF THEN
        cpy y = x
    IF ELSE
        ldc y = 0
    IF END
```

3.1.5 Loop Nodes

A `tree_loop` node represents a "do-while" loop. It contains two tree node lists: the `body` and the `test`. The `body` list comes first and holds the loop body. The `test` list contains code to evaluate the "while" expression and conditionally branch back to the beginning of the body.

There are three labels associated with a `tree_loop` node: `toplab`, `contlab`, and `brklab`. The `toplab` label marks the beginning of the loop `body`; the `test` list typically contains a conditional branch back to `toplab`. A "continue" statement in the loop body requires a jump over the rest of the body to the beginning of the test code. The `contlab` label is positioned at the beginning of the `test` list for this purpose. Similarly, a "break" statement in the loop is translated to a jump to the `brklab` label which is located immediately after the loop. These `tree_loop` labels must be defined in the scope of the loop node, but the label positions are implicit rather than being marked by label instructions. When the `tree_loop` is expanded to low-SUIF form, the label instructions are inserted into the tree node lists.

Because the loop nodes are only intended for use with structured control flow, certain restrictions on the contents of the `tree_loop` lists are required. The `test` part has exactly the same restrictions as the `header` of a `tree_if` except that it is the `toplab` label instead of the `jumpto` label to which jumps and branches are allowed and expected. See Section 3.1.4 [If Nodes], page 13. Note in particular that use of the `brklab` and `contlab` labels is not allowed in the `test` list.

The `body` list has restrictions analogous to the restriction on `then_part` and `else_part` lists of a `tree_if`: arbitrary nesting of other `tree_node`s is allowed, but control flow into or out of the `body` is not allowed. The `body` list is allowed slightly more leeway, though: jumps or branches are allowed from anywhere in the `body` to the `brklab` or the `contlab` labels. This is the only place where the `brklab` or `contlab` labels can be used–they cannot be used outside the `tree_loop` or in the `test` list.

Note that either or both of the `test` and `body` parts may be empty lists, but if the `test` part is empty, the loop is degenerate–it will always execute exactly once.

For example, the following C code could be translated into the SUIF code shown in a simplified form below. Note that because the `test` code is at the *bottom* of the loop, the "while" loop must be converted to a "do-while" loop guarded by an "if" node. See Section 3.1.4 [If Nodes], page 13.

```
while (k > 0) {
    if (k = 1) break;
    k = k - 1;
    if (k < 10) continue;
    k = k - 10;
}
```

```
IF (Jumpto=L:__L1)
IF HEADER
    bfalse e1, L:__L1
      e1: sl e2, k
        e2: ldc 0
IF THEN
    LOOP (Top=L:__L2 Cont=L:__L3 Brk=L:__L4)
    LOOP BODY
        btrue e1, L:__L4
          e1: seq k, e2
            e2: ldc 1
        sub k = k, e1
          e1: ldc 1
        btrue e1, L:__L3
          e1: sl k, e2
            e2: ldc 10
        sub k = k, e1
          e1: ldc 10
    LOOP TEST
        btrue e1, L:__L2
          e1: sl e2, k
            e2: ldc 0
    LOOP END
IF ELSE
IF END
```

3.1.6 For Nodes

Many of our compiler passes are designed to work with Fortran `DO` loops because they are relatively easy to analyze. A `DO` loop has a single index variable that is incremented or decremented on every iteration and varies from an initial to a final value. SUIF uses `tree_for` nodes to represent well-structured `DO` loops. The exact conditions that a loop must meet to be represented as a `tree_for` in SUIF are described below. The expander's cleanup pass, which is run immediately after the front-end, converts `tree_for` nodes that violate any of these conditions into `tree_loop` nodes (see Section 3.1.5 [Loop Nodes], page 15). Even though they are primarily intended for Fortran code, `tree_for` nodes may also be used for C loops that meet the same conditions.

The index variable of a `tree_for` can be accessed using the `index` method and set using the `set_index` method. The index variable must be a scalar variable that is defined in the scope of the `tree_for` node. It may not be modified anywhere within the loop body. This applies across procedures as well. If the loop body contains a call to another procedure that modifies the index variable, then the loop cannot be represented by a `tree_for` node. Moreover, if you are using Fortran form, the index variable may not be a call-by-reference parameter. See Chapter 10 [Fortran], page 77.

The range of values for the index variable is specified by three `operand` fields. See Section 4.2 [Operands], page 27. The lower and upper bounds and the step value can be accessed by the `lb_op`, `ub_op`, and `step_op` methods. The `set_lb_op`, `set_ub_op`, and `set_step_op` methods are provided to change them. These operands are expressions that are evaluated once at the beginning of the loop. The index variable is initially assigned the lower bound value and then incremented by the step value on every iteration until it reaches the upper bound value; the code to do this is automatically created when the `tree_for` is expanded to low-SUIF code.

Most users will always use `tree_for` nodes in conjunction with expression trees. Flat lists of instructions are typically used only in the library and with back-end passes where the `tree_for` nodes have been dismantled. It is possible to use `tree_for` nodes without expression trees, but the bounds and step values cannot be treated as operands. In fact, even with expression trees those operands are actually stored on tree node lists. If necessary, these lists can be accessed directly using the `lb_list`, `ub_list`, and `step_list` methods. Each list is required to contain a single expression with a dummy copy instruction at the root. The destination of the dummy copy must be a null operand. Methods are provided in the tree node list class to extract the operands from the tree node lists (see Section 3.2 [Tree Node Lists], page 20).

The `tree_for` must also specify the comparison operator used to determine when the index variable has reached the upper bound value. The possible comparison operators are members of the `tree_for_test` enumerated type. The `test` and `set_test` methods are used to access and modify the comparison operator for a `tree_for` node. The `tree_for_test` enumeration includes quite a few comparison operators, but some of them are only used by the front-end. Both signed and unsigned versions are available for most of the comparison operators, as indicated by the letters "`S`" and "`U`" in their names.

`FOR_SLT`

`FOR_ULT` Less than. Repeat as long as the index is strictly less than the upper bound.

`FOR_SLTE`

`FOR_ULTE` Less than or equal. Repeat as long as the index is less than or equal to the upper bound.

`FOR_SGT`

`FOR_UGT` Greater than. Repeat as long as the index is strictly greater than the upper bound. Sometimes `DO` loops go backwards, using a negative step value. For those loops, the comparison operator must also be reversed.

`FOR_SGTE`

`FOR_UGTE` Greater than or equal. Repeat as long as the index is greater than or equal to the upper bound. Again, this may be used when the step value is negative.

`FOR_SGELE`

`FOR_UGELE`
: These comparisons are only used by the front-end. In FORTRAN, it may not be possible to determine the direction of a loop at compile time. For example, if the step value is not a constant, it could be either positive or negative. These comparison operators indicate that the loop test may be either "greater than or equal" or "less than or equal", depending on the sign of the step value. The expander's cleanup pass converts any `tree_for` nodes with these tests to two `tree_for` nodes and a `tree_if` node to decide between them. Thus, these comparison operators should never be encountered in most SUIF code.

`FOR_EQ`

`FOR_NEQ` Equal and not equal. These comparisons are only used by the front-end. The expander's cleanup pass dismantles `tree_for` nodes that use these comparisons.

The body of a `tree_for` loop is stored in a tree node list. The methods to get and set the loop body are `body` and `set_body`, respectively. The `body` list contains only the instructions corresponding to the body of the loop in the source program. The instructions to compare the index variable with the upper bound, increment it, and branch back to the beginning of the body are not included as part of the body; they are created when the `tree_for` is expanded to low-SUIF code.

Besides the loop body, a `tree_for` node has an additional tree node list called the `landing_pad`. The code in the landing pad is executed if and only if the loop body is executed at least one time, but the `landing_pad` is executed only once, unlike the body which is usually executed many times. The `landing_pad` is executed immediately before the first time through the loop body. The landing pad thus provides a place to move loop-invariant code.

Two labels are associated with a `tree_for`: `contlab` and `brklab`. A "continue" statement in the loop body requires a jump over the rest of the body to the code that increments the index and continues with the next iteration. This can be implemented with a jump to the `contlab` label, which is implicitly located at the end of the `body` list. Similarly, a "break" statement in the loop is translated to a jump to the `brklab` label which is located immediately after the loop. These two labels must be defined in the scope of the `tree_for` node, but the label instructions that mark their locations are not inserted into the tree node lists until the `tree_for` node is expanded into low-SUIF form.

In summary, the semantics of a `tree_for` node are as follows. The lower bound, upper bound, and step operands are evaluated once at the beginning of the loop[1]. The lower bound is compared to the upper bound. If the test fails, the loop does not execute. Otherwise, the lower bound is assigned to the index variable, and any code in the landing pad list is executed. After that, the body is executed repeatedly and the index variable is incremented by the step value on every iteration, until the index variable fails the test with the upper bound value.

As an example, the following C code could be translated into the SUIF code shown in a simplified form below:

---

[1] The code produced directly by the C front-end assumes that the upper bound and step operands are reevaluated on every iteration. The expander's cleanup pass dismantles any `tree_for` nodes for which it cannot guarantee that these semantics are equivalent.

```
for (i = 100; i >= 0; i--) {
    A[i] = i;
}
```

```
FOR (Index=i Test=SGTE Cont=L:__L1 Brk=L:__L2)
FOR LB
    ldc 100
FOR UB
    ldc 0
FOR STEP
    ldc -1
FOR LANDING PAD
FOR BODY
    str e1 = i
      e1: array e2+0, size(32), index(i), bound(e3)
        e2: ldc <A,0>
          e3: ldc 101
FOR END
```

3.1.7 Block Nodes

A `tree_block` node introduces a new scope. Nested scopes are useful for retaining scope information from source programs and for debugging purposes. They are also very useful for implementing code transformations. For example, name conflicts are easily avoided by introducing new scopes.

A `tree_block` node contains a `block_symtab` symbol table and a tree node list for the body. The symbol table is accessed with the `symtab` and `set_symtab` methods. The `body` and `set_body` methods are used to get and set the list of tree nodes inside the block.

There is a one-to-one correspondence between a `tree_block` and its symbol table, and the hierarchy of symbol tables within a procedure must parallel the hierarchy of `tree_block` nodes. When inserting a `tree_block` into an AST, the new block's symbol table must be inserted in the appropriate place in the symbol table hierarchy. When a `tree_block` is destroyed, the associated symbol table is detached from its parent table and destroyed. However, the converse is not true—when a `block_symtab` is deleted, the corresponding `tree_block` node is not deleted.

The entries in a block's symbol table may not be referenced from outside the block. There are no other restrictions on `tree_block` nodes. The bodies may be empty or contain any kinds of tree nodes. Blocks are usually entered from the beginning but that is not required; unlike other AST nodes, branches into the middle of a block are allowed.

As an example, the following code creates a new `tree_block` and its associated symbol table. It then adds the new block to the body of an existing block node:

```
block_symtab *new_symtab = new block_symtab("my_symtab");
cur_block->symtab()->add_child(new_symtab);
tree_node_list *tl = new tree_node_list;
tree_block *new_block = new tree_block(tl, new_symtab);
cur_block->body()->append(new_block);
```

3.1.8 Procedure Nodes

A special kind of block node is used as the root of each AST. These `tree_proc` nodes are derived from the `tree_block` class. They are essentially the same as other block nodes, but they include a few extra methods and have a slightly different kind of symbol table. SUIF does not support nested procedures, so `tree_proc` nodes may only be used at the roots of ASTs.

A procedure node's symbol table must be a `proc_symtab` instead of just an ordinary `block_symtab`. The `proc_syms` method is provided to cast the symbol table pointer to be a `proc_symtab`. A `tree_proc` node also has a pointer to the symbol of the corresponding procedure. This pointer is set automatically when a `tree_proc` is attached to the procedure symbol. For any tree node in an AST, the `proc` method retrieves this procedure symbol from the `tree_proc` at the root of the tree.

Because `tree_proc` nodes are always at the roots of the ASTs, their `parent` list pointers (see Section 3.1.2 [Other Node Features], page 12) are always `NULL`.

3.2 Tree Node Lists

A `tree_node_list` is a doubly-linked list of tree nodes, derived from the `dlist` class. See Section 11.4 [Doubly-Linked Lists], page 80. Most nodes in an AST have their children recorded in tree node lists. For example, the body of a block node is a tree node list. Besides the standard list features, each tree node list includes a back-pointer to its parent tree node. These pointers are maintained automatically by the SUIF library and can be accessed using the `parent` method. In addition, each node on a tree node list contains pointers back to the list and the list element. See Section 3.1.2 [Other Node Features], page 12. These pointers are set by the `set_elem` method (see Section 11.1 [Generic Lists], page 79) which is automatically called every time an element is added to the list.

In a `tree_for` node, the lower bound, upper bound, and step value are usually treated as operands; however, the operands are actually stored as tree node lists. See Section 3.1.6 [For Nodes], page 16. Each of these lists is required to contain a single expression with a dummy copy instruction at the root. The destination of the dummy copy must be a null operand. The `tree_node_list` class includes several methods to handle these special lists. The `is_op` method checks if the list consists of a single copy instruction with a null destination. The `op` method retrieves the operand stored in the list. If after converting the list to expression tree form, the `is_op` method returns `TRUE`, the `op` method returns the source operand of the dummy copy instruction; otherwise, an error occurs. The `set_op` method is used to set the contents of the tree node list to a dummy copy instruction with the specified operand as its source. The list must either be empty or already contain the dummy copy. Finally, the `op_is_intconst` method checks if the operand in the tree node list is a constant integer, and if so, returns the value.

Tree node lists can easily be printed out as text. The `print` method simply iterates through the list and calls the `print` method for each tree node. The optional `depth` parameter specifies the indentation level to be used. A separate method is used to print tree node lists that hold `tree_for` operands. If the `is_op` method for one of these lists returns `TRUE`, the `print_expr` method prints the operand directly. Otherwise, it prints the list as usual.

3.3 Mapping Functions Over Subtrees

Many SUIF programs contain code that visits all of the nodes in an AST to perform various operations. Rather than having every programmer duplicate the code for this traversal, the SUIF library includes methods to map functions over ASTs. The `tree_node` and `tree_node_list` classes both provide these `map` methods.

The function to be mapped must be of type `tree_map_f`. For a tree node, the `map` method applies the function to the tree node and all of its children. The `preorder` parameter is used to select either preorder or postorder traversals; the default is preorder. For preorder traversals, the function is applied to the tree node before it is applied to the children. For postorder, the function is first applied to the tree_node's children and descendants and then last applied to the tree node itself.

The `map` method for a tree node list calls `map` for each node in the list. The nodes are visited in order from first to last regardless of the `preorder` parameter. The `tree_node_list` version of the `map` method has an extra optional parameter that allows you to only map the function over the entries in the list without recursively visiting their children.

The `tree_map_f` functions have two parameters: a pointer to the tree node and a `void*` value passed on from the `map` method. Additional parameters can be passed by making the `void*` value a pointer to a structure containing the parameters. For example, the following code counts the number of `tree_for` and `tree_loop` nodes in a procedure:

```
struct count_result {
    int num_fors;
    int num_loops;
};

void count_loops (tree_node *t, void *x)
{
    count_result *results = (count_result *)x;

    if (t->kind() == TREE_FOR) results->num_fors++;
    else if (t->kind() == TREE_LOOP) results->num_loops++;
}

/*
 *  Return the number of tree_fors and tree_loops in the procedure.
 */

void counter (tree_proc *p, int *for_count, int *loop_count)
{
    count_result results;
    results.num_fors = 0;
    results.num_loops = 0;

    p->map(count_loops, (void *)&results);

    *for_count =  results.num_fors;
    *loop_count = results.num_loops;
}
```

4 Instructions and Expression Trees

At the leaves of the abstract syntax trees, the `tree_instr` nodes (see Section 3.1.3 [Instruction Nodes], page 13) hold SUIF instructions. Each instruction performs a single operation, specified by its opcode. With just a few exceptions, the opcodes are simple and straightforward, similar to the instruction set of a typical RISC processor. The instructions include arithmetic, data transfer, and control flow operations.

SUIF instructions may be used in two ways. Most high-level passes use expression trees, where the instructions are grouped together in trees that correspond to the expressions in the source programs. Back-end optimization passes have traditionally worked with flat lists of instructions, and SUIF supports that representation as well.

The files '`instruction.h`' and '`instruction.cc`' contain the code for the SUIF instruction classes.

4.1 Basic Instruction Features

All instructions share some basic features. This section describes the fields in the base `instruction` class. Other classes are derived from the `instruction` class to include fields that are specific to the different instruction formats.

Besides the features below, the instructions have ID numbers that can be used to identify them across passes of the compiler. This is discussed in detail elsewhere. See Section 3.1.1 [ID Numbers], page 11.

The SUIF library also declares an `instruction_list` class for working with lists of pointers to instructions. This class is not used much within the library itself because SUIF does not store instructions directly on lists. Even with the flat list form, the instructions are contained in `tree_instr` nodes, which in turn are stored in lists. However, it may sometimes be convenient to work with lists of instruction pointers even if those instructions are actually stored in `tree_instr` nodes.

4.1.1 Opcodes and Formats

The `opcode` and `set_opcode` methods may be used to access the opcode field in an instruction. These opcodes are members of the `if_ops` enumerated type. The internal names for the opcodes all begin with `io_`, but that prefix is omitted in this documentation. The `if_ops_name` function takes an opcode and returns a character string holding the opcode name (without the `io_` prefix). This is used by the library when it prints out instructions, but you may also call it directly.

Each opcode is associated with a particular instruction format. The formats are specified by members of the `inst_format` enumeration. The `format` method may be used to retrieve the format for a particular instruction. Alternatively, the `which_format` function takes an opcode and returns its format without reference to any particular instruction. Each format corresponds to a particular derived instruction class:

`inf_rrr` Three operand instructions use the `in_rrr` class.

`inf_bj` Branch and jump instructions use the `in_bj` class.

`inf_ldc` The load constant instruction uses the `in_ldc` class.

`inf_cal` The call instruction uses the `in_cal` class.

`inf_array`
The array instruction uses the `in_array` class.

`inf_mbr` The multi-way branch instruction uses the `in_mbr` class.

`inf_lab` The label instruction uses the `in_lab` class.

`inf_gen` Generic instructions use the `in_gen` class.

The `format` method is often used in `switch` statements for dealing with different kinds of instructions. For example, the following code computes the number of source operands in an instruction (instead of using the `num_srcs` method):

```
instruction *i;
switch (i->format()) {
    case inf_rrr: {
        n += 2;
        break;
    }
    case inf_bj: {
        n += 1;
        break;
    }
    case inf_cal: {
        in_cal *cali = (in_cal *)i;
        n += 1 + cali->num_args();
        break;
    }
    case inf_array: {
        in_array *arri = (in_array *)i;
        n += 2 + 2 * arri->dims();
        break;
    }
    case inf_mbr: {
        n += 1;
        break;
    }
    case inf_gen: {
        in_gen *geni = (in_gen *)i;
        n += geni->num_srcs();
        break;
    }
    default: {
        /* other formats (inf_ldc and inf_lab) have no source operands */
        break;
    }
}
```

The opcodes and instruction formats are defined in the '`opcodes.h`' and '`opcodes.cc`' files.

4.1.2 Destination Operands

The base `instruction` class includes a field for the destination operand, even though not all instructions use it. The `dst_op` method returns the value of this operand. If an instruction does not produce a result, the destination operand should always be a null operand. Moreover, the destination may be null even if an instruction does produce a result. In that case, the result value is computed and then discarded.

If the destination operand is a variable, its type must be compatible with the result type of the instruction.

If the result of an instruction is a temporary value, the destination operand is a pointer to the instruction where the value is used. Note that this is different than an instruction pointer in a source operand. The source operands of an instruction point to the children in an expression tree, while the destination operand points to the parent instruction.

The `set_dst` method may be used to set the destination operand for an instruction. However, there are some important restrictions on using this method:

- For derived classes with instructions that never produce result values, it is illegal to call `set_dst`. This allows the library to ensure that the destination will always be a null operand[1]. Trying to use `set_dst` for one of these instructions will cause an error.
- You may not set the destination operand to point to another instruction; trying to do so will cause an error. This is to prevent inconsistent pointers. The destination should only point to another instruction if that other instruction has a source operand that points back to it. Thus, the library automatically sets the destination operand when an instruction is used as a source operand.
- If the destination operand already contains a pointer to another instruction, you cannot use `set_dst` to overwrite it. This also causes a run-time error. As in the previous case, this is to prevent inconsistent pointers. Instead, you must first call the instruction's `remove` method to clear the destination operand. See Section 4.1.5 [Source Operands], page 26.

4.1.3 Result Types

Each instruction also has a field to specify the result type. The `result_type` and `set_result_type` methods may be used to access this field. If an instruction produces a result value (regardless of whether the result is actually used), the type of the result must be indicated. Otherwise, the result type should be the SUIF `void` type.

If the instruction produces a value, the result type must be an object type (i.e. not a function type) with non-zero size and that size must be known at compile time. This restriction is to exclude arrays with unknown or symbolic bounds, the only types without known sizes.

4.1.4 Parent Tree Nodes

Each instruction automatically records the `tree_instr` node to which it is attached. The `parent` method retrieves this pointer. In an expression tree, all of the instructions

[1] Note that for other classes it is your responsibility to make sure that the destination is null if the instruction does not produce a result.

share the same parent `tree_instr` node, so an instruction may not be directly connected to its parent node.

The `owner` method is identical to the `parent` method except for instructions that are used as operands of `tree_for` nodes. Those operands are actually attached to dummy copy instructions on lists attached to the `tree_for` nodes (see Section 3.1.6 [For Nodes], page 16), but many users will want to treat them as if they are directly attached to the `tree_for` nodes. Thus, for an instruction in a `tree_for` operand, the `owner` method will return a pointer to the `tree_for` node instead of returning the actual parent `tree_instr`. This only works if the `tree_for` operand list contains a single expression (not a flat list); otherwise, the actual `tree_instr` parent is returned.

4.1.5 Source Operands

Even though the base `instruction` class does not contain any source operand fields, it does provide several methods to access the source operands in derived classes. First, the `num_srcs` method returns the number of source operands in a particular instruction. Once you know the number of sources, you can access them by number. This is frequently useful when you need to visit all of the source operands without concern for how they are used in the instruction. The `src_op` and `set_src_op` methods provide this access. The operands are numbered beginning from zero (like arrays in C). Specifying an operand number that does not exist will cause an error. Do not use these methods to access a particular operand field in an instruction; the operand numbering used here is implementation-defined and subject to change.

The `src_map` method provides an alternate way to visit all of the source operands. It applies a function that you specify to all of the source operands. This can be used to implement recursive descents of expression trees by making the mapped function call `src_map` if the operand is an instruction. This is similar to what the `instr_map` method in the `tree_instr` class does. See Section 3.1.3 [Instruction Nodes], page 13. The mapped function must match the `src_map_f` type which has three parameters: a pointer to the instruction, a pointer to a copy of the operand, and a `void*` value used to pass any other information needed by the function. If the function returns `TRUE`, the copy of the operand will be assigned to the actual operand field within the instruction. If it returns `FALSE`, the source operand field is not modified.

The SUIF library requires that instruction pointers in operands be consistent. That is, a source operand may only point to an instruction if the destination of that instruction points back to where the result value is used. Because of this, you cannot simply overwrite a source operand that contains an instruction pointer. Doing so would leave the other instruction with an inconsistent destination operand. Instead, you must first use the source operand's `remove` method. This clears the instruction pointers in both the source and destination operands. If the source instruction was part of an expression tree (i.e. it didn't have its own parent `tree_instr` node), `remove` also clears its `parent` pointer by calling the parent's `remove_instr` method (see Section 3.1.3 [Instruction Nodes], page 13). If the source operand is not an instruction, the `remove` method does nothing so it is always safe to use it. The `instruction` class also has a `remove` method which does the same thing.

As an example, the following code clears all of the source operands in an instruction:

```
for (int n = 0; n < i->num_srcs(); n++) {
```

```
        i->src_op(n).remove();
        i->set_src_op(n, operand());
    }
```

4.1.6 Printing Methods

Instructions can be printed out as text using the `print` method. The optional `depth` parameter specifies the indentation level for the output. The `elab` and `en` parameters are also optional and are only used when printing sub-expressions. Most users should never need to worry about these extra parameters. The `elab` parameter is the number for the label attached to the instruction, and the `en` parameter is the next number to be used for labeling subexpressions.

4.2 Operands

Most SUIF instructions have operands that are represented by objects of the `operand` class, which is implemented in the files '`operand.h`' and `operand.cc`. An operand may have three different kinds of values:

- It may be a null operand. The `is_null` method tests for this case. Null operands occur when an operand field in an instruction is unused.
- An operand may refer directly to the symbol for a variable. The `is_symbol` method tests for this case. The `symbol` and `set_symbol` methods access the symbol field. A variable in a source operand indicates that the contents of the variable are used. In a destination operand, a variable is assigned the result of the instruction.
- An operand may also point to another instruction. The `is_instr` method tests for this, and the `instr` and `set_instr` methods access the pointer field. This is used to implement expression trees and, in low-SUIF, to refer to the results of other instructions in a flat list. An instruction pointer in a source operand means that the result of that instruction is used as the operand. Conversely, an instruction pointer in a destination operand indicates that the result value is used by that instruction.

The `is_expr` method is very similar to `is_instr`. Besides checking if the operand holds an instruction pointer, it also tests if that instruction is a subexpression that is not contained in a separate `tree_instr` node (i.e. it's not in a flat list). This method should not be used for destination operands.

Before changing a source operand from an instruction pointer to some other value, the instruction must be removed. The `remove` method checks if the operand is an instruction, and if so, calls the instruction's `remove` method. See Section 4.1.5 [Source Operands], page 26.

The operand class includes several methods to simplify common operations. One frequently needs to know the type of a particular operand. This can be determined from the variable's type or the instruction's result type, but checking for these different kinds of operands and then extracting the type is cumbersome. Instead, you can use the `type` method which performs this operation for you. It returns the SUIF `void` type for null operands. Another common operation is testing an operand to see if it contains an integer constant from a load constant instruction. The `is_const_int` method checks if this is the case, and if so, it also returns the value of the constant.

Operands have two different methods for printing themselves out to a file as text. The `print` method is used by the library when printing instructions. Because source and destination operands are handled differently, `print` requires that you specify the instruction that contains the operand so that it can determine how the operand is being used. However, when debugging a program, you may not know which instruction contains an operand. If you do know that it is used as a source operand, you can use the `print_source` method to print it without specifying the instruction.

4.3 Expression Trees

SUIF supports both expression trees and flat lists of instructions. Procedures are always written out as flat lists. After a procedure body has been read from an input file, it can be converted to expression tree form. The `read_proc` method for the procedure symbol has a flag that allows you to specify this; by default it builds the expression trees. You may also do the conversion manually as described below. When the procedure body is written out, it is automatically converted back to the flat list form.

In the flat list form, each SUIF instruction has its own `tree_instr` node. With expression trees, the only change is that all of the instructions within an expression are grouped under the same `tree_instr` node. The instruction pointers in the operand fields do not change.

Up to now we have implied that high-SUIF uses expression trees and that low-SUIF uses the flat list representation, but the decision to use expression trees is actually independent from the structure of the ASTs. SUIF allows expressions trees to be used even if all of the high-level control structures have been removed, and conversely the flat list representation works with all kinds of AST nodes (although accessing the `tree_for` operands is a little awkward). In fact, the two representations can even be mixed in the same procedure!

Some caution is needed here. These two representations are not equivalent. An expression tree only specifies a partial order for the evaluation of the instructions in the tree, whereas a flat list is a total ordering. Similarly, while flat lists allow you to intermix the instructions from different expressions, with expression trees each expression must be completely evaluated before proceeding to execute the instructions for the next expression. Thus, flat lists give you precise control of the execution order for the instructions. Because of this, if you reorder the instructions in a flat list and then try to convert to expression trees, the library may complain and promote temporaries to variables in order to maintain the equivalent semantics.

Methods are provided in the `tree_node_list` class to convert back and forth between expression trees and flat lists. The `flatten` method converts all of the expression trees in the list and its child lists to flat lists of instructions. This is simply a matter of creating new `tree_instr` nodes to hold the subexpressions and inserting them into the lists.

The `cvt_to_trees` method converts a tree node list to expression tree form. This does not actually change the instructions, but rather collects the instructions within an expression under a single `tree_instr` node. Only contiguous instructions can be converted into an expression tree. If other instructions from outside the expression are intermixed, building the expression tree may change the behavior of the code. Thus, if this occurs, the library will print a warning message and split up the expression tree so that the original semantics are preserved.

4.4 Three Operand Instructions

The vast majority of SUIF instructions are represented by the `in_rrr` class which includes fields for two source operands. Not all of the operands have to be used. For example, `nop` instructions don't use any operands at all.

The `src1_op` and `src2_op` methods retrieve the source operands, and the `set_src1` and `set_src2` methods assign new operands. Besides the methods to directly access the two source operands, other methods are provided to refer to these sources according to their uses with specific opcodes:

- Shift and rotate instructions (`asr`, `lsl`, `lsr`, and `rot`) may use the `shift_cnt_op` and `set_shift_cnt_op` methods to access the operand that specifies the numbers of bits to shift or rotate. The other source operand may be accessed with the `src_op` and `set_src` methods.
- Store (`str`) and memory copy (`memcpy`) instructions may use the `dst_addr_op` and `set_dst_addr_op` methods to access the source operand that holds the destination address for the memory reference.
- Load (`lod`) and memory copy (`memcpy`) instructions may use the `src_addr_op` and `set_src_addr_op` methods to access the operand that specifies the source address for the memory reference. Note that the actual operand field that is used is different for these two opcodes.
- Instructions that only use one of the source operands may use the `src_op` and `set_src` methods without specifying whether the first or second operand field is used. In addition, a store (`str`) instruction may use these methods to refer to the operand holding the value to be stored. They are also used by shift and rotate instructions as described above.

Using these opcode-specific methods with the wrong opcodes will cause errors.

For your convenience, the `in_rrr` class provides a method, `is_unary`, to determine if an instruction produces a result value using only one of the source operands. It does not consider return (`ret`) instructions to be unary because they do not produce result values. Although load (`lod`) instructions fit the pattern, they are a special case and are not considered to be unary.

The `is_commutative` method checks the opcode of an instruction to determine if it is a commutative operation. If so, the two source operands should be interchangeable.

Some of the arithmetic instructions may generate run-time exceptions if the appropriate class of exceptions is enabled (see Section 8.5.4 [Miscellaneous Annotes], page 67). If the exceptions are not enabled, the rules for ANSI C are used to determine the result.

The following table lists all of the three operand SUIF instructions. Each entry describes the operands used with that opcode and includes any restrictions on the operand and result types.

`nop`
: Do nothing at all. All of the operands for these instructions should be null, and the result type should be the SUIF `void` type.

`lod`
: Load the value at the address contained in the `src1` operand and put it in the `dst` operand. The result type indicates the type of the value being loaded and may be any type, subject to the usual restrictions on a result type (see Section 4.1.3

[Result Types], page 25). The type of the expression in `src1` must be a pointer to the result type. The `src2` operand is not used.

`str` Store the value in the `src2` operand at the address contained in the `src1` operand. Both operands must be specified. There is no special restriction on the type of the `src2` operand, though the restrictions on instruction result types (see Section 4.1.3 [Result Types], page 25) and variables (see Section 6.4 [Variable Symbols], page 47) guarantee it will have a known, non-zero size. The `src1` operand should contain an expression that is a pointer to the type of the operand being stored. The `dst` operand is not used.

`memcpy` Memory to memory copy. Load the value from the address in the `src2` operand and store it at the address in the `src1` operand. The type of the object to be copied is subject to the same conditions as the result type of an instruction (see Section 4.1.3 [Result Types], page 25), so it must have known, non-zero size. Both of the source operands must be pointers to this object type. The `dst` operand is not used.

`cpy` Copy the `src1` operand to the `dst` operand. The `src2` operand is not used. The result type must be the same as the type of the source operand. The restrictions on instruction result types (see Section 4.1.3 [Result Types], page 25) guarantee that the object being copied has known, non-zero size.

`cvt` Convert the `src1` operand to the result type and put it in the `dst` operand. The `src2` operand is not used. Nothing can can be converted to or from a `struct`, `union`, `array`, or `void` type. Pointer types can only be converted to and from integer types and other pointer types.

`neg` Negation. Change the sign of the value in the `src1` operand and put the result in the `dst` operand. The `src2` operand is unused. The result type and the type of the operand must be the same signed integer or floating-point type.

`add` Add the values in the `src1` and `src2` operands and put the result in the `dst` operand. Except for pointer additions, the result type and the types of the operands must be the same integer or floating-point types. Pointer addition is a special case. One of the source operands may have a pointer type, as long as the other source operand has signed or unsigned integer type of any size; the result type must also be a pointer type, but need not be the same as the source pointer type.

`sub` Subtract the value in the `src2` operand from the value in the `src1` operand and put the result in the `dst` operand. Except for pointer subtractions, the result type and the types of the operands must be the same integer or floating-point types. There are two special cases for pointer subtractions. In either case, the `src1` operand must have a pointer type. First, the `src2` operand may have any integer type, in which case the result type may be any pointer type, not necessarily the same as the source pointer type. Second, the `src2` operand's type may be another pointer, in which case the result type must be type_ptr_diff.

`mul`
`div` Multiply or divide the value in the `src1` operand by the value in the `src2` operand and put the result in the `dst` operand. The result type and the types of the

operands must be the same integer or floating-point type. Integer multiplication and division are defined according to the rules for ANSI C.

`rem`
`mod` Remainder and modulus. These two instructions are very similar. Both divide the value in the `src1` operand by the value in the `src2` operand to find the remainder or modulus. The `rem` instruction is identical to the modulus operator in ANSI C. That is, if either source operand is negative, the sign of the result is undefined and depends on the semantics of integer division. The `mod` instruction is the same except that its result is always guaranteed to be positive. The result type and the types of the destination and source operands must be the same integer type.

`not` Bit-wise inversion. Compute the one's complement negation of the value in the `src1` operand and put the result in the `dst` operand. The `src2` operand is not used. The result type and the types of the operand must be the same unsigned integer type.

`and`
`ior`
`xor` Compute the bit-wise AND, inclusive OR, or exclusive OR of the values in the `src1` and `src2` operands and put the result in the `dst` operand. The result type and the type of the operands must be the same unsigned integer type.

`asr`
`lsr`
`lsl` Shift the value in the `src1` operand right or left by the amount specified in the `src2` operand. The variable in the `src2` operand must always have an unsigned integer type. The `asr` instruction performs sign extension and requires that the result type and the type `src1` operand be the same signed integer type. The `lsr` instructions does not perform sign extension and requires that the result type and type of the `src1` operand be the same unsigned integer type. Sign extension is not an issue for left shifts, so the `lsl` instruction only requires that the result type and the type of the `src1` operand be the same integer type.

`divfloor`
`divceil` Division combined with floor and ceiling operations. The `divfloor` opcode means take the rational quotient of the `src1` operand by the `src2` operand and apply the floor operation (i.e. round to the nearest integer less than or equal to the quotient). The `divceil` opcode means take the rational quotient of the `src1` operand by the `src2` operand and apply the ceiling operation (i.e. round to the nearest integer greater than or equal to the quotient). The result type and the operand types must be the same integer type.

`min`
`max` Minimum and maximum. The result value is the minimum or maximum, respectively, of the two source operands. The result type and the operand types must be the same integer or floating-point types.

`abs` Absolute value. Compute the absolute value of the `src1` operand. The `src2` operand is unused. The result type and the source operand type must be the same integer or floating-point type.

`rot` Rotate the value in the `src1` operand left or right by the amount specified in the `src2` operand. The variable in the `src2` operand must always have a signed integer type. If the shift amount is positive, the value is rotated to the left (toward the most-significant bit); if it is negative, the value is rotated to the right. The result type and the type of the `src1` operand must be the same integer type.

`seq`
`sne`
`sl`
`sle` Comparison instructions. If the `src1` operand is equal, not equal, less than, or less than or equal, respectively, to the `src2` operand, assign the integer value one to the `dst` operand. Otherwise, set the `dst` operand to zero. The result type must always be a signed integer type. The source operands must have integer, enumerated, floating point, or pointer type.

`ret` Return from a procedure. Only the `src1` operand is used and it is optional. If specified, it is the return value and may contain an operand of any type except array or function types. If the procedure's function type has void return type, the operand must be null; otherwise the operand must not be null and must have the same type as the return type of the procedure.

`mrk` This pseudo-instruction only marks a position in the program and is used to hold miscellaneous annotations such as line numbers. It is functionally equivalent to a `nop` instruction. All of the operands for these instructions should be null, and the result type should be the SUIF `void` type.

4.5 Branch and Jump Instructions

The `in_bj` class is used to hold branch and jump instructions. This class includes a field to specify the label symbol at the branch target and an optional source operand that is used for conditional branches. The `target` and `set_target` methods are used to access the label symbol for the branch target. This label must be defined in the scope where the branch occurs, and it must be within the same procedure. The `src_op` and `set_src_op` methods access the optional source operand field. The `dst` operand is unused for all branch and jump instructions and the result type should always be the SUIF `void` type. The branch and jump opcodes are described below:

`jmp` Unconditional jump. The source operand is unused. The flow of control is unconditionally transferred to the code at the target label.

`btrue` Branch if true. If the source operand, which must have an signed integer type, contains a true (non-zero) value, the flow of control is transferred to the code at the target label. Otherwise, it continues with the next instruction in sequential order.

`bfalse` Branch if false. If the source operand contains a false (zero) value, the flow of control is transferred to the code at the target label. Otherwise, it continues with the next instruction in sequential order. The source operand must have an signed integer type.

4.6 Load Constant Instructions

Rather than allowing constant values to be used directly as operands, SUIF uses separate `ldc` instructions to load constant values. The `in_ldc` class holds these instructions. Instead of the usual source operands, this class has an immediate value field (see Section 12.1 [Immeds], page 83). The `value` and `set_value` methods may be used to access this field.

Only certain kinds of immediate values are supported in an `ldc` instruction:

Symbolic addresses (See Section 12.2 [Symbolic Addresses], page 84)
: The result type of the instruction must be a pointer type.

Integers
: The result type must be an integer or pointer type. Pointer types are allowed so that the null pointer can be loaded as the integer value zero.

Floating-point values
: The result type must be a floating-point type.

Other kinds of immediate values may be stored in the `value` field of an `ldc` instruction, but most SUIF passes and certain library functions will not be able to handle them.

4.7 Call Instructions

SUIF uses a special `cal` instruction to represent procedure calls. This high-level representation hides the details of various linkage conventions. The `in_cal` class is used to represent these call instructions. A call instruction contains a source operand to hold a pointer to the procedure to be called. The `addr_op` and `set_addr_op` methods access this operand field.

The actual parameters for the procedure are stored in an array of operands. The `num_args` method returns the number of elements in this array. The size of the array can be changed at any time using the `set_num_args` method. If necessary, the array will be reallocated. Elements of the argument array may be accessed using the `argument` and `set_argument` methods. You must specify the array index. The first argument is at index zero.

Call instructions must obey some conventions on the types of the operands. The `addr` operand must hold a pointer to a function type which is compatible with the type of the procedure being called. The result type of the call instruction must match the return type of the procedure. The restrictions on instruction result types (see Section 4.1.3 [Result Types], page 25) guarantee that the return type will either be void or have known, non-zero size. If the function type specifies the number of arguments, it must match the number of actual parameters (unless the function takes a variable number of arguments). Moreover, each operand in the argument array must be compatible with the type of the corresponding formal parameter. Whether or not the function type specifies the argument types, the restrictions on instruction result types (see Section 4.1.3 [Result Types], page 25) and variables (see Section 6.4 [Variable Symbols], page 47) guarantee that all arguments will have known, non-zero size.

4.8 Array Instructions

Because many SUIF passes focus on analyzing and optimizing Fortran code, a high-level representation of array references is crucial. SUIF provides `array` instructions which retain all of the high-level information in combination with other fields needed to generate code for the address computations. The `in_array` class is used to hold these array instructions.

Array instructions include a number of fields. First, a pointer to the base of the array is specified in an operand field that can be accessed with the `base_op` and `set_base_op` methods. If the array elements are structures, a constant offset within the selected element may be included. This optional integer offset can be accessed using the `offset` and `set_offset` methods. The element size is needed to generate low-level code for the array address calculation. The `elem_size` method returns the element size in bits. The `set_elem_size` method may be used to change the element size.

Because Fortran arrays do not always begin with index zero, an optional operand, which is referenced using the `offset_op` and `set_offset_op` methods, is provided to specify an offset. Since there is a single offset operand, the offsets for all of the dimensions must be combined into a single value. The arrays are stored in row-major form, so the offset for the first dimension is multiplied by the size of the remaining dimensions, etc. If the offset operand is provided, it must have an integer type.

Array instructions can treat arrays of arrays as multidimensional arrays, even though the type system does not support that directly. Each array instruction includes a field to specify the number of dimensions in the array. This field may be accessed with the `dims` and `set_dims` methods. The indexes for the array reference are stored in an array of source operands, one for each dimension. These index operands can be accessed using the `index` and `set_index` methods. The dimensions are numbered beginning with zero. Similarly, the number of elements in each dimension are stored in another array of source operands, which can be accessed with the `bound` and `set_bound` methods.

The result type of an array instruction must be a pointer. However, it need not be a pointer to the element type. If the elements are structures, the result type may be a pointer to one of the structure fields. SUIF does not actually require that the result type match anything within the array element type, although that is highly recommended. The `elem_type` method can be used to determine the actual type of the element being addressed.

The types of the array instruction operands must follow some conventions. The index and bound operands must all have integer types. The base operand must be a pointer to an array. If the array instruction has multiple dimensions, the base must point to a nested array (an array of arrays of arrays...) with the same depth as the number of dimensions. For each dimension, if the bound operand is a constant, it must match the number of elements specified in the corresponding array type. (If the lower and upper bounds in the array type are not both constant, then the bound operand may have any value.) The bound operand for the first dimension is optional and may be null. Finally, the element size must match the size of the elements in the array type.

4.9 Multi-way Branch Instructions

Fortran computed `goto` statements and C `switch` statements are represented in SUIF by multi-way branch (`mbr`) instructions. These are easier to analyze than the equivalent series of conditional branches, and they can easily be used to generate efficient jump table code. The `in_mbr` class holds these multi-way branch instructions.

The `in_mbr` class contains a field with a pointer to an array of label symbols. The `num_labs` method returns the number of labels in the array. The size of the array can be changed at any time using the `set_num_labs` method; if necessary the array will be reallocated. A particular element within the array can be accessed using the `label` and `set_label` methods. You must specify the array index, and, as usual, the elements are numbered beginning with zero.

A multi-way branch instruction transfers control to one of the target labels depending on the value in the source operand. This operand must have an integer type. It can be accessed using the `src_op` and `set_src` methods. The value of the source operand is combined with an integer offset to determine the target label. The offset can be accessed with the `lower` and `set_lower` methods. The offset is subtracted from the value in the source operand and the result is used to index into the array of target labels. If the index is within the range of the array, the instruction branches to the label at that position in the array; otherwise, it branches to the default target label. The `default_lab` and `set_default_lab` methods access this default label field. The destination operand of a multi-way branch is unused and trying to set it will cause an error. The result type should always be the SUIF `void` type.

4.10 Label Instructions

SUIF uses special pseudo-instructions to mark the positions of labels within the lists of instructions. These label (`lab`) instructions are represented by the `in_lab` class, which contains a single field holding the symbol for a label. The `label` and `set_label` methods access this field.

No operation is performed by a label instruction. Its only purpose is to mark the location of a label symbol in an instruction list. The `label` field must be a pointer to the symbol for the label, which must be defined within the scope where the label instruction occurs. The destination operand is unused and trying to set it will cause an error. The result type should always be the SUIF `void` type.

4.11 Generic Instructions

To help support special-purpose extensions to SUIF, we have provided a generic class of instructions. This is implemented in the `in_gen` class. These generic instructions contain arbitrarily large arrays of source operands and a character string field to hold user-defined names that function as "sub-opcodes". Generic instructions are not part of standard SUIF and most SUIF passes will not handle them.

Because it is difficult to add new opcodes to SUIF at run-time, the generic instructions all share the same `gen` opcode. Instead, they are distinguished by user-defined names. The `name` and `set_name` methods may be used to access these character string fields. The set_name method automatically enters the name in the lexicon (see Section 12.3 [Lexicon], page 84).

A generic instruction contains a pointer to an array of source operands. The base class num_srcs method may be used to determine the size of this array. The size may be changed at any time using the set_num_srcs method. If necessary, the array will be reallocated. The elements of the source operand array can be accessed using the standard base class src_op and set_src_op methods. See Section 4.1.5 [Source Operands], page 26.

5 Symbol Tables

Symbol tables contain the definitions of the symbols and types used within a SUIF program. Each symbol table is associated with an object corresponding to a particular scope. For example, a procedure symbol table is attached to the abstract syntax tree representing the body of the procedure. The symbol tables can be reached through the corresponding objects and vice versa.

This section describes the symbol table hierarchy and the details of the symbol table operations, such as looking up symbol table entries and adding new entries. It also explains how the symbol tables handle the task of assigning unique ID numbers to the symbols and types. The '`symtab.h`' and '`symtab.cc`' files contain the code for symbol tables.

5.1 Symbol Table Hierarchy

The SUIF symbol tables are organized in a hierarchy of nested scopes and maintained internally within a tree structure. Every table contains a list of the symbol tables that are its children, and each table also has a pointer back to its parent in the tree (except for the global symbol table which does not have a parent). The `children` method returns a pointer to the list of children and the `parent` method gets the pointer to the parent symbol table. Thus, to search through all of the enclosing scopes, one can follow the parent pointers back to the global symbol table, visiting all of the symbol tables along the way. The `is_ancestor` method provides an easy way to check if a given symbol table is an ancestor (i.e. an enclosing scope) of the current table.

Note that the symbol table hierarchy is not independent. The primary objects in a SUIF program are the files and the abstract syntax trees for the procedures. The symbol tables are always attached to these primary objects and are generally treated as if they are parts of those objects. For example, when a block of code is deleted the associated symbol table is automatically removed from the hierarchy and deleted.

The `base_symtab` class is the base class from which the other symbol table classes are derived, but it is an abstract class and cannot be used directly. There are four different derived symbol tables classes. They have much in common, but each is used at a different level in the hierarchy and thus has slightly different features.

5.1.1 The Global Symbol Table

The global symbol table is at the top of the symbol table hierarchy and corresponds to the outermost global scope. It contains objects that are visible across source files (i.e. shared types and global symbols with external linkage). For this reason, it is associated with the `file_set` object. See Section 2.1 [File Set], page 7.

The advantage of using a shared global symbol table appears when performing interprocedural analyses and transformations. Without a common symbol table, it can be quite a burden to deal with references to symbols that are defined in some files but not in others. Even trying to determine which symbols from different files correspond to the same objects is difficult. In essence, each interprocedural pass would need to do the work of a linker! The shared global symbol table avoids all of these problems and makes interprocedural optimization relatively easy.

Along with the benefits of the global symbol table come a few difficulties. Sharing the global symbol table across files makes it difficult to support separate compilation. Each file must contain a copy of the global symbol table, and if these files are manipulated individually, their copies of the global symbol table will not be consistent. Thus, before a group of files can be combined in a SUIF file set, their global symbol tables must be "linked" together using the SUIF linker pass. Whether this is preferable to just combining all of the source files into one big SUIF file is debatable.

The `global_symtab` class is used to represent the global symbol table. It is also used as the base class for file symbol tables. Because procedure symbols may only be entered in global and file symbol tables, this class contains the methods to deal with them. The `new_proc` method creates a new procedure symbol and enters it in the table (see Section 5.4 [Creating New Entries], page 41), and the `lookup_proc` method searches for an existing procedure symbol (see Section 5.3 [Lookup Methods], page 40). The `number_globals` method in this class handles the task of assigning ID numbers to the symbols and types in global and file symbol tables (see Section 5.6 [Numbering Types and Symbols], page 42).

5.1.2 File Symbol Tables

A file symbol table corresponds to the global scope for a source file. It contains procedure symbols and global variable symbols with static linkage, as well as types that are only used within the file. Each file symbol table is associated with a particular file set entry. See Section 2.2 [File Set Entries], page 8.

The `file_symtab` class is derived from the `global_symtab` class to implement the file symbol tables. Besides the features that this class inherits from its base class, it also contains a field to record the file set entry with which it is associated. This field is set automatically when the file symbol table is created by the file set entry. The `fse` method retrieves the value of this field.

5.1.3 Procedure Symbol Tables

Procedure symbol tables represent the top-level scopes within procedures and are associated with the `tree_proc` objects at the roots of the abstract syntax trees for the procedures. See Section 3.1.8 [Procedure Nodes], page 20. Because the procedure symbol tables provide a superset of the block symbol table functions, they are implemented by deriving the `proc_symtab` class from the `block_symtab` class. Thus, all of the `block_symtab` methods can also be applied to `proc_symtab` objects.

Besides the inherited methods, the procedure symbol tables have some added features. Each procedure symbol table contains a list of the formal parameters for the procedure. The `params` method returns a pointer to this list. The entries on this list must be pointers to symbols for variables that are contained within the procedure symbol table. (Formal parameters cannot be global variables or local variables in inner scopes.) The symbols are listed in order. If the function type for the procedure specifies the parameter types, they should match the types of the variables on the parameter list.

The procedure symbol table also records the next instruction ID number for the procedure (see Section 3.1.1 [ID Numbers], page 11). The `number_locals` method handles the task of assigning ID numbers to the symbols and types in symbol tables within the procedure (see Section 5.6 [Numbering Types and Symbols], page 42).

5.1.4 Block Symbol Tables

The `block_symtab` class is used for nested block symbol tables and as the base class for procedure symbol tables. Each one is associated with a particular `tree_block` (or `tree_proc`) node in an abstract syntax tree. See Section 3.1.7 [Block Nodes], page 19. Each block symbol table contains a pointer to the corresponding `tree_block` node. The `block` method retrieves the value of this pointer. When a symbol table is connected to a `tree_block`, its `block` pointer is set automatically.

Since label symbols may not be declared in global scopes, the `block_symtab` class is the natural place to provide methods for working with labels. The `new_label` method creates a new label symbol and enters it in the table (see Section 5.4 [Creating New Entries], page 41). The `new_unique_label` does the same thing but it first makes sure that the label will have a unique name. The `lookup_label` method searches for an existing label symbol (see Section 5.3 [Lookup Methods], page 40).

Block symbol tables also provide a method to create a new child symbol table, i.e. an inner scope. The `new_unique_child` method can be used to create a new child block symtab with a unique name (see Section 5.4 [Creating New Entries], page 41). This method is not provided for global symbol tables, because their children must correspond to procedures, which already have unique names.

5.2 Basic Features

Symbol tables contain three different kinds of objects: types, symbols, and variable definitions. The entries within a symbol table may only be referenced within the corresponding scope. This includes references within registered annotations. Violating this condition may lead to strange and unexpected errors.

For simplicity, the symbol table entries are stored on lists instead of using hash tables. In theory, the actual implementation (lists or hash tables) should not be visible in the symbol table interface. Unfortunately that is not completely true for the current implementation of SUIF—the lists can be accessed directly. The `types`, `symbols`, and `var_defs` methods return pointers to the lists. However, these lists should only be accessed to examine the entries and should never be modified directly. The symbol table classes provide other methods to add and remove entries from the lists and those methods should always be used. If the list implementation becomes a performance bottleneck, we may need to switch to hash tables, and code that modifies the lists directly will be relatively hard to convert.

To distinguish the symbol tables nested within a particular scope, each table is given a name. The `name` and `set_name` methods retrieve and modify this name. If a scope in the source program has a name associated with it, that name may be used for the corresponding symbol table. For example, the name of a procedure-level symbol table should generally be the same as the name of the procedure. On the other hand, nested scopes within procedures are typically unnamed, and names must be generated for the corresponding symbol tables.

The symbol table names are used when printing a reference to a symbol or named type. Because the symbol or type name alone may not be sufficient to identify it uniquely, the `chain_name` method is used to identify the symbol table. The chain name of a symbol table includes the names of all of the symbol tables from the procedure-level downward, separated by slashes (as in a Unix path). The file-level name is not included since it should always be clear from the context. The chain name for a global or file symbol table is the empty string.

Duplicate names within a symbol table should be avoided whenever possible. Each kind of symbol has a separate name space. A variable, for example, may have the same name as a label in the same symbol table. Named types and child symbol table names are also in separate name spaces. Duplicate names may be temporarily introduced but to avoid problems they should be renamed as soon as possible. The `rename_duplicates` method is provided to check for and rename any duplicates in a symbol table. This method is automatically called before writing out each symbol table.

5.3 Lookup Methods

SUIF symbol tables provide a number of methods to search for and retrieve particular types, symbols, and variable definitions. Most of these lookup methods will optionally search all the ancestor symbol tables, making it easy to determine if an object is defined in the current scope.

The `lookup_type` method is available at all levels in the symbol table hierarchy to search for SUIF types. Given an existing type, the method searches for a type that is the same. It uses the `is_same` method from the `type_node` class to perform these comparisons. If a matching type is not found within the current symbol table, `lookup_type` will continue searching in the ancestor symbol tables by default. However, if the optional `up` parameter is set to `FALSE`, it will give up after searching the first table.

Several methods are provided to lookup symbols. Each different kind of symbol (variable, procedure, and label) has its own name space, so the `lookup_sym` method requires that you specify both the name and the kind of symbol for which to search. This method may be used with all symbol tables. For convenience, other methods are defined as wrappers around `lookup_sym`. Each of these wrappers searches for a particular kind of symbol: `lookup_var` searches for variables, `lookup_proc` searches for procedures, and `lookup_label` searches for labels. Because procedure symbols may only be defined in global symbol tables, the `lookup_proc` method is declared in the `global_symtab` class. Similarly, the `lookup_label` method is declared in the `block_symtab` class, because labels may only be defined within procedures. By default, all of these methods search the current symbol table and, if unsuccessful, proceed to search the ancestor symbol tables. The optional `up` parameters may be set to `FALSE` to turn off this default behavior and only search the current symbol table.

A symbol for a global variable is just a declaration of that variable and does not automatically have any storage allocated. Variable definitions are required to allocate storage and to specify alignment requirements and any initial data for the variable. Since the variable definitions are not directly connected to the variable symbols, the `lookup_var_def` method is provided to search a symbol table for the definition of a particular variable symbol. This method does not search the parent symbol table. In general the `definition` method in the `var_sym` class is a better way to locate a variable definition.

Symbols and types are assigned ID numbers (see Section 5.6 [Numbering Types and Symbols], page 42) that uniquely identify them within a particular context. The `lookup_type_id` method searches the types defined within a symbol table and its ancestors for a type with the specified ID number. The `lookup_sym_id` does the same thing for symbols.

Besides searching for one of the entries in a symbol table, you can also search for one of its children in the symbol table hierarchy. The `lookup_child` method searches through the list of children for a symbol table with a given name. This may not be very useful, but it is included for completeness.

5.4 Creating New Entries

To make it easier to add new entries, the symbol tables provide methods that combine the steps of creating new objects and then entering them in the tables. Some of these methods automatically make sure that the new entries have unique names and that is particularly useful.

New variables can be added to tables anywhere in the symbol table hierarchy. The `new_var` method creates a new variable with a given name and type and then enters the new variable symbol in the table. The `new_unique_var` method is similar, but it also checks that the name of the new variable is unique. If not, it appends a number to the specified name until it is unique. With this method, the base name is optional; the default value is '`suif_tmp`'.

Procedure symbols can be created in global and file symbol tables using the `new_proc` method. The name of the procedure, its type, and the source language must be specified. There is currently no method to automatically create a new procedure symbol with a unique name.

Because label symbols may only be declared within procedures, the `new_label` and `new_unique_label` methods are provided in the `block_symtab` class. The only parameter of these methods is the name of the label. The name is optional for `new_unique_label`; its default value is '`L`'. Just as with variables, unique label names are created by adding a number to the end of the base names.

Within a procedure, new inner scopes may be created to be used with new `tree_block` nodes. The `block_symtab` class provides the `new_unique_child` method to create a new symbol table, give it a unique name, and add it to the list of children. The unique name is created by appending a number to the optional base name. If the base name is not given, it defaults to '`block`'. This method is not needed at the global level, because the child symbol tables there correspond to procedures which should already have unique names.

Finally, new variable definitions can be added to any symbol table using the `define_var` method. The parameters are the variable symbol and the alignment for the storage to be defined. It returns a pointer to the new variable definition object, so that you can attach initial data annotations to it.

5.5 Adding and Removing Entries

Entries in symbol tables should always be added and removed using the methods provided by the symbol tables. Although it is possible to add and remove entries by directly manipulating the lists, that should never be done. The methods for adding and removing entries hide the underlying representation and using them will make it much easier to update your code if that representation changes. Even more importantly, most symbol table entries contain back pointers to the tables which hold them, and the adding and removing methods are responsible for maintaining those pointers and for performing a few other automatic checks and updates.

Types, symbols, and child symbol tables may be added using the `add_type`, `add_sym`, and `add_child` methods, respectively. Each of these entries contains a pointer back to the parent symbol table, and these methods automatically set those back pointers. They do not, however, perform any other sanity checks, such as checking for duplicate names. Similarly, the `remove_type`, `remove_sym`, and `remove_child` methods remove types, symbols, and child symbol table entries. These methods clear the parent pointers but do not delete the entries that are removed.

Variable definitions are treated a bit differently from other kinds of symbol table entries. They do not have parent pointers so the `add_def` and `remove_def` methods do not have to deal with that. However, adding and removing variable definitions change some attributes of the corresponding variables, and those attributes must be automatically updated. First, each variable has a flag to indicate whether a variable definition exists for it. A variable cannot have more than one definition, so the `add_def` method will fail if this flag is already set. Otherwise, it sets the flag when the new definition is added. Second, variable symbols also have a flag to indicate whether they are actual definitions or just declarations of symbols with external linkage. This `extern` flag must be set to `FALSE` when a variable definition is added for a global variable. When removing a variable definition, these flags must be reversed.

Unlike symbol nodes which always define separate symbols, multiple type nodes can represent the same type. The basic `add_type` method will add a new type even if an equivalent type was already defined in the same scope. In most cases, what is actually needed is a method to first check if an equivalent type exists and if so to throw away the duplicate and return the existing type. The `install_type` method provides this functionality. It first checks if a type has already been entered in the symbol table or one of its ancestors using the `lookup_type` method. If so, it deletes the new type and returns the existing one. If a type is not found, it is entered into the symbol table and returned. All of the components of a type are recursively installed before the type itself. This makes it easy to create new types without worrying about duplicate entries in the symbol tables.

5.6 Numbering Types and Symbols

Every symbol and type is assigned an ID number that uniquely identifies it within a particular context. These ID numbers should be used to refer to symbols and types in annotations that will be written to the output files and in other situations where pointers to the symbol and type nodes cannot be used. The `sym_id` method retrieves the ID number for a symbol, and the `type_id` method gets the number for a type.

For symbols and types within a procedure, the ID numbers are only unique within that procedure. Similarly, the ID numbers for symbols and types in a file symbol table are only unique within that file. Only in the global symbol table are the ID numbers truly unique. This is implemented by dividing the ID numbers into three ranges. Each range is reserved for a particular level in the symbol table hierarchy. To make it easier to read an ID number, the `print_id_number` function prints it as a character to identify the range (‘`g`’ for global, ‘`f`’ for file, ‘`p`’ for procedure) combined with the offset of the number within that range.

The symbol and type ID numbers cannot be assigned individually, but the symbol tables provide methods to set them. The `number_globals` method is defined in the `global_symtab` class to number the entries in global and file symbol tables, and the `number_locals` method is defined in the `proc_symtab` class to number all of the entries in the procedure symbol table and its descendents. These methods only assign ID numbers to symbols and types that do not already have numbers. These methods are called automatically before writing things out to files, but they can also be called whenever you want to assign numbers to new symbols and types.

The `clear_sym_id` symbol method and `clear_type_id` method are provided to reset the ID numbers to zero manually, but as far as the library itself is concerned, this is never necessary. The library automatically changes ID numbers when necessary, such as when moving from one symbol table to another.

6 Symbols

SUIF symbols are stored in the symbol tables (see Chapter 5 [Symbol Tables], page 37) to represent variables, labels, and procedures. The `sym_node` class is the base class for all SUIF symbols. This is an abstract class so it cannot be used directly. The library also defines the `sym_node_list` class for lists of pointers to symbols. Classes are derived from the `sym_node` class for each kind of symbol. Given an arbitrary symbol, the `kind` method identifies the kind of symbol and thus the derived class to which it belongs. This method returns a value from the `sym_kinds` enumerated type. The following values are defined in that enumeration:

`SYM_VAR` Variable symbol. The `var_sym` class represents variable symbols.

`SYM_LABEL`
: Label symbol. The `label_sym` class represents label symbols.

`SYM_PROC` Procedure symbol. The `proc_sym` class represents procedure symbols.

All symbols share some common fields including the symbol names. These are described in the first section below. Each kind of symbol also uses additional fields that are specific to that kind. For example, variable symbols specify the types of the variables. The subsequent sections describe the specific features of each kind of symbol.

The '`symbols.h`' and '`symbols.cc`' files contain the source code for SUIF symbols.

6.1 Basic Features of Symbols

The `sym_node` class defines several fields that are used by all kinds of symbols. The most obvious of these is the symbol name. Each symbol has a name that should be unique within the symbol table where it is defined. The `name` and `set_name` methods access this field. The names are automatically entered in the lexicon (see Section 12.3 [Lexicon], page 84) by `set_name`. Because the name of a symbol alone is generally insufficient to uniquely identify it, the symbols are also given ID numbers. See Section 5.6 [Numbering Types and Symbols], page 42.

When a symbol is entered in a symbol table, it automatically records a pointer to that parent table. Similarly, when the symbol is removed from the symbol table, its parent pointer is cleared. The `parent` method retrieves this parent pointer.

All symbols contain flags to specify various attributes. The `sym_node` class provides methods to access these flags. The `is_userdef` method tests a flag to see if it is a user-defined symbol (from the source code) or a new symbol introduced by the compiler. The `set_userdef` and `reset_userdef` methods change the value of this flag.

Another flag is used to mark symbols that are only declarations of external symbols, rather than actual definitions. This flag is set automatically. The `is_extern` method retrieves its value. Label symbols are never `extern`. A procedure symbol is `extern` unless the procedure body is defined in the input file(s). A global variable symbol is `extern` unless it has a separate definition (see Section 6.4.3 [Variable Definitions], page 51); no other variables are `extern`.

Since symbols may be treated differently depending on their scopes, the `sym_node` class includes methods to determine which kind of symbol table contains a symbol. The `is_global` method checks if the parent table is a global or file symbol table. This is really only useful for variable symbols, because procedures are always global and labels are never global. The is_private method checks if a symbol is global but private to one source file by checking if the parent symbol table is a file symbol table. This is obviously irrelevant for label symbols.

The add_to_table and remove_from_table methods are provided for convenience when adding or removing symbols from symbol tables. In the case of variable symbols, the entire hierarchy of sub-variables (see Section 6.4.2 [Sub-Variables], page 49) is added or removed at one time by this method.

The copy method makes a copy of a symbol. This is a virtual function so it copies the fields that are specific to each kind of symbol. However, it only copies the symbol itself: copying a procedure symbol will not copy the procedure body and copying a variable symbol will not copy the variable definition. The copy method does not copy annotations on the symbol, either. Since the copy will have the same name as the original symbol, it should generally be renamed or used in a different symbol table.

Two different methods are available for printing symbols. The print method just prints the name of the symbol. Label symbols are prefixed by 'L:' and procedure symbols by 'P:' to distinguish them from variable symbols. The print_full method is used by the library when listing the contents of symbol tables. It includes all the fields from the sym_node.

6.2 Procedure Symbols

Procedure symbols are represented by objects of the proc_sym class. SUIF does not support nested procedures, so these symbols may only be entered in global and file symbol tables. The fields in a procedure symbol hold information about the procedure, including a pointer to the body if it is in memory. The proc_sym class also provides methods to read procedure bodies from input files, write them to the output files, and flush them from memory.

Each procedure symbol contains a field to record the source language for the procedure. The src_lang and set_src_lang methods access this field, which holds a value from src_lang_type enumeration: src_unknown, src_c, src_fortran, or src_verilog. Other values may be added in the future.

A procedure symbol also has a field that specifies the type of the procedure. The type and set_type methods retrieve and change this field. The type must be a function type. See Section 7.5 [Function Types], page 57.

The body of a procedure is represented by its abstract syntax tree. See Chapter 3 [Trees], page 11. The procedure symbol contains a pointer to the root node of this tree. The block and set_block methods access this pointer. If the body is not in memory, the block pointer will be NULL; the is_in_memory method is provided to check this condition.

The proc_sym class contains the methods to read procedure bodies from binary SUIF files and to write them out again. The details of SUIF I/O are thus hidden from users; only entire procedures can be read and written. If one of the input files contains the body for a procedure, a pointer to the file set entry (see Section 2.2 [File Set Entries], page 8) is recorded in the procedure symbol. The file method retrieves this pointer for a particular proc_sym. The same procedure can be read in and flushed from memory many times, but once it has been written out it can no longer be read or written again. The procedure symbol contains a flag to indicate if it has been written out yet. The `is_written` method returns the value of this flag. The `is_readable` method checks if the procedure body exists in one of the input files and if it has not yet been written out. If this method returns `TRUE`, the `read_proc` method can be used to read the body of the procedure. By default, `read_proc` also converts the procedure to expression tree form (see Section 4.3 [Expression Trees], page 28) but it does not convert to Fortran form (see Chapter 10 [Fortran], page 77). The `exp_trees` and `use_fortran_form` parameters to `read_proc` can be used to override these defaults.

After a procedure body has been read in and possibly modified, it can be written to the output file using the procedure symbol's `write_proc` method. You must specify the file set entry to which the procedure should be written. In most cases, the input and output file set entry will be the same, and you will just use the `file` method to determine the output file set entry. As mentioned above, once a procedure has been written out it cannot be rewritten or read in again. Obviously, it should not be changed after that point because the changes could not be saved. Besides avoiding changes directly to the procedure, however, you must also be careful to avoid certain changes to the global symbol tables. The symbols and types within the procedure are written out using their ID numbers. See Section 5.6 [Numbering Types and Symbols], page 42. Thus, you must not do anything to the global symbol tables that would cause the ID numbers for those symbols and types to change. For example, moving a symbol from a file symbol table to the global symbol table would require that its ID number change. The best solution to this is to not write out the procedures until you are certain that such changes to the symbol tables will not be needed.

When a procedure body is no longer needed, typically after it has been written out, call the `flush_proc` method for the procedure symbol to deallocate the storage used by the procedure. In some cases, you may want to flush the procedure before it is written. For example, interprocedural analysis requires that all procedures be read in and analyzed together. To save space, the procedures can be summarized for the purpose of the particular analysis and then flushed. After the analysis is complete, they can be re-read and the results can be attached to the code.

6.3 Label Symbols

The `label_sym` class represents label symbols. Labels are used within procedures to specify targets of branch and jump instructions. They may not be entered in global or file symbol tables. The position of a label is usually indicated by a label instruction (see Section 4.10 [Label Instructions], page 35), but for labels associated with high-level AST nodes, the label positions may be implicit. The `label_sym` class contains no extra fields beyond those in the base `sym_node` class.

6.4 Variable Symbols

In SUIF, variable symbols represent data objects. Variable symbols are represented by objects of the `var_sym` class. This class adds a field to specify the type of the variable as well as some additional flags. Unlike procedures and labels, variables may be defined in any scope.

SUIF provides optional *sub-variables* to make it easier to deal with pieces of aggregate objects that may or may not overlap, in particular Fortran equivalences and reshaped common blocks. Instead of referring to a piece of an aggregate by an offset combined with the aggregate symbol, a sub-variable can be created to represent the data at a particular offset within the aggregate, so that it can be referenced in the same way as if it were not contained within an aggregate structure.

6.4.1 Basic Features of Variables

The type of a variable can be accessed via the `type` and `set_type` methods. The type must be visible in the scope where the variable is defined. Since the type specifies the amount of storage occupied by the variable, the size of the type must not be zero. This means that function types, void types, incomplete structure and union types, and array types with unknown bounds cannot be used for variables[1]. Pointers to those types are acceptable, of course. Sub-variables may have any type so long as the size of that type and the sub-variable's offset allow it to remain within the parent variable.

Variable symbols may also be allocated to machine registers. A flag is used to indicate variables that are contained in registers; the `is_reg` method returns the value of this flag and the `set_reg` and `reset_reg` methods change its value. The `reg_num` annotation can be attached to a register variable to specify the corresponding machine register number (see Section 8.5.4 [Miscellaneous Annotes], page 67). Variables cannot be registers if they have variable definitions. Whether a sub-variable is a register or not is slaved to this property in its parent.

Symbols for variables that are not allocated on the stack must have separate variable definitions (see Section 6.4.3 [Variable Definitions], page 51) or else they are merely declarations of external symbols. The `has_var_def` method checks if a variable symbol has a separate definition, and the `definition` method returns a pointer to the definition. It is an error to call `definition` for a variable that does not have a definition, so be sure to check first. Sub-variables are never allowed to have their own definitions.

The `var_sym` class includes several methods to check various attributes and flags for variables. The `is_auto` method checks if a variable is in the automatic storage class, i.e. if it will be allocated on the stack. Global variables and static local variables (those with variable definitions) are not automatic and `is_auto` returns `FALSE` for them; other variables are automatic. The `is_static` method is the opposite of `is_auto`, except that register variables are not considered to be static.

The formal parameters of a procedure are represented by variables that have a special flag set. The `is_param` method returns the value of this flag, and the `set_param` and `reset_param` methods change its value. Identifying the formals with flags does not allow their order to be specified, so the procedure symbol table maintains a list of the formals (see Section 5.1.3 [Procedure Symbol Tables], page 38). All of the variables for which `is_param` returns `TRUE` should be on the list of parameters, and conversely, all variables on the parameter list should have the flag set. Sub-variables are not allowed to be parameters; sub-variables of parameters are allowed, but these are not considered parameters themselves.

[1] This rule may appear to be violated when using Fortran form with call-by-reference arrays with unknown bounds, but the actual type of such a variable is a pointer to the array type (see Section 10.1 [Call-By-Ref Parameters], page 77).

Because many compiler optimizations are only applicable to variables that are not aliased, it is important to identify such variables. Each variable symbol includes a flag to indicate whether the variable has its address taken, and thus whether it could be aliased. The `is_addr_taken` method returns the value of this flag. The front-end should set these flags, but they may also need to be updated as the code is transformed. The `set_addr_taken` and `reset_addr_taken` methods can be used to change the flag for a particular variable symbol. For sub-variables, this property is slaved to this property in the parent variable.

Besides requiring that variables not be aliased, most optimizations also do not apply to aggregate data structures. The `is_scalar` method checks the type of a variable symbol and returns `TRUE` if it is not an aggregate type. Arrays, structures, and unions are considered to be aggregates.

In general, most optimizations target scalar variables that are guaranteed not to be aliased. Since it is difficult to guarantee that global variables with external linkage are not addressed, those variables may be aliased even if their flags indicate that they are not addressed. Rather than checking these conditions separately, the library provides one method to check them all. The `is_spilled` method returns `TRUE` for variables that are static, addressed, or aggregates. Variables that are not spilled can then be targeted for optimization.

6.4.2 Sub-Variables

The data objects represented by variable symbols are allowed to overlap, but only in a carefully controlled way. The mechanism that allows this is the existence of sub-variables. One variable symbol is allowed to be a sub-variable of another, occupying a portion of the parent variable symbol specified by an offset and the types of the two variables. This allows arbitrary hierarchies of variable symbols and the relations among them are known at all times.

For any hierarchy, there will always be one root ancestor in terms of which everything else can be specified. If neither of two variables is a sub-variable of anything else, the two variables are guaranteed to be completely independent objects.

Use of sub-variables is optional. Anything that can be represented with the sub-variable can be represented in terms of its parent by loading from the parent's address plus the offset and with the type of the child variable.

Sub-variable names are independent of their parent variable symbols. Whether or not they have the `userdef` flag set is also independent of the parent. There is nothing inconsistent in having a parent which is user-defined and a sub-variable which is not (if some stage of the compiler decides a particular piece of a larger user-defined object has some special property and wants a handle on it), and neither is there anything inconsistent in a sub-variable that is user-defined in a parent which is not (user-defined variables that are equivalenced cause the system to create a new aggregate object).

Among the properties general to the `sym_node` class that may be changed by the user, the only one for which sub-variables have any special restriction is the symbol table in which they reside. A sub-variable should be in the same symbol table as its parent. They may be temporarily in different symbol tables while being moved around, so the library doesn't enforce the restriction that they always be in the same symbol table at all times. The library does enforce the restrictions that they must be in the same table when a symbol is first linked in as a sub-variable by the `var_sym::add_child` method and when the owning symbol table of the parent or child variable is written out.

Among properties specific to variable symbols that the user has control over, some may be set independently in sub-variables, and some are kept consistent between sub-variables and their parents. For the properties that are kept consistent, the value of the property in the sub-variable is slaved to that in the parent. In that case, either setting or getting the property from the sub-variable has exactly the same effect as getting or setting it from the parent variable. If and when the sub-variable is removed from being a sub-variable by the `var_sym::remove_child` method, the current values of those properties in the sub-variable are set to the values of the parent at the time of separation and can be changed independently from then on.

Sub-variables are attached to parent variables by the `add_child` method. The user tells the parent which variable symbol is to be the sub-variable and at what offset it begins and the library makes it a sub-variable. If it is ever necessary to remove a sub-variable, the `remove_child` method may be used. It undoes what `add_child` did making the sub-variable into a fully independent variable. The argument of `remove_child` must already be a sub-variable of the given variable. This method should be called before a sub-variable is deleted, or else there will be dangling pointers from the parent.

Two methods, `parent_var` and `offset`, provide a way to see whether a variable is a sub-variable, and if so what variable it is a sub-variable of and where it is in that variable. If a variable is not a sub-variable, `parent_var` will return `NULL`. If it is a sub-variable, this method will return the variable symbol for the variable of which it is a sub-variable. If it is a sub-variable, the `offset` method will return the offset in bits of the beginning of the sub-variable from the beginning of the parent variable. If called on a variable that is not a sub-variable, `offset` will return zero.

There are also methods provided to access all the variables that are sub-variables of a given variable. The `num_children` method will return the number of sub-variables for a given variable symbol. Given a non-negative number less than the number of children, the `child_var` method will return one of the sub-variables. The ordering of the sub-variables is the order in which they were added, starting with zero for the oldest remaining sub-variable. For example, `child_var(2)` will return the sub-variable that was the third added of those that are still sub-variables. Hence calling this method for each integer from zero through the number of children minus one will give all the sub-variables. It is an error to call `child_var` with an argument greater than or equal to the number of sub-variables or less than zero.

Given a variable that may or may not be a sub-variable, the `root_ancestor` will return the unique ancestor of a sub-variable that is not itself a sub-variable, or the variable itself if it is not a sub-variable. The `root_offset` method will return the offset of the variable within that root ancestor, which will be the sum of all the offsets along the path up. The `overlaps` method is provided to determine whether any two variables overlap to any extent. If the result is `TRUE`, then they definitely overlap, at least in part; otherwise, the result is `FALSE`, and the variables do not overlap at all.

It is often useful to know whether a given portion of a variable treated as a given type has a corresponding sub-variable. The `find_child` method will answer that question. If such a sub-variable exists, it will return a pointer to it, otherwise it will return NULL.

As an alternative to attaching an existing variable symbol as a sub-variable to another variable, the `build_child` method is provided. Given a location and type within a variable and a name, it creates an sub-variable, installs it in the proper symbol table if the parent variable was in a symbol table, and adds it as a sub-variable, all at once.

6.4.3 Variable Definitions

Variable definitions are used to allocate storage for variables that are not on the stack. These variable definitions are implemented by the `var_def` class, which is defined in the '`vardef.h`' and '`vardef.cc`' files. Each variable definition contains a field with a pointer to the corresponding variable. The `variable` and `set_variable` methods access this field.

The effects of a variable definition depend on the kind of variable. Without separate variable definitions, the symbols for global variables are merely declarations of external symbols. Adding a definition for a global variable changes it from a declaration to an actual definition. The local variables in a procedure or nested scope are allocated on the stack by default. Adding a definition for a local variable makes it static. Similarly, if you remove a definition of a local variable, it reverts to being allocated on the stack. Because of their special status, register variables cannot have definitions. Sub-variables are also not allowed to have their own definitions.

The location of a `var_def` identifies the place where the variable is defined. For most variables, the `var_def` will be in the same symbol table as the variable. However, variables in the global symbol table must be defined in exactly one of the source files. Thus, the `var_def` for a variable in the global symbol table must be in one of the file symbol tables.

Besides identifying the location of the definition, a `var_def` also specifies the alignment restriction for the variable's storage. The `alignment` method retrieves the size of the alignment unit in bits, and the `set_alignment` method sets the alignment unit size. The alignment unit should be at least as big as the smallest addressable unit for the target architecture.

The initial data for a static variable can be specified by attaching annotations to the variable definition. These annotations are optional. If omitted, the initial data is assumed to be all zeros. The initial data annotations are described elsewhere in this manual. See Section 8.5.1 [Initial Data Annotes], page 65.

Variable definitions are printed out as part of each symbol table. The `print` method is used for this. You may also use this method for debugging. The optional `depth` parameter specifies the indentation level for the output.

7 Types

SUIF types are used to describe the results of instructions and the contents of variables. Each symbol table (see Chapter 5 [Symbol Tables], page 37) contains a list of types that are visible within the scope of that table. All SUIF types are derived from the `type_node` base class, which is defined in the files '`types.h`' and '`types.cc`'. The `type_node_list` class is provided for lists of pointers to types.

Different kinds of type nodes are distinguished by their type operators. The `op` method returns the operator for a particular `type_node`. Some type operators define complete types by themselves while others build upon existing types. The `type_ops` enumeration lists the type operators:

`TYPE_VOID`
`TYPE_INT`
`TYPE_FLOAT`
: These operators, along with `TYPE_ENUM` below, define the base types. They use the `base_type` derived class.

`TYPE_PTR`
: A pointer operator defines a type for pointers to an existing type. The `ptr_type` derived class holds this kind of type node.

`TYPE_ARRAY`
: The array operator uses the `array_type` class and defines a type that is an array with elements of another type.

`TYPE_FUNC`
: The function operator defines a type for a SUIF procedure in terms of the return type and the types of the arguments. The `func_type` derived class is used with this operator.

`TYPE_STRUCT`
`TYPE_UNION`
: These operators are for named structure and union types and both use the `struct_type` derived class. Both combine fields of different types into a single record.

`TYPE_ENUM`
: The enumerated type operator defines an integer type where each value may have a name. Like the ordinary integer types, enumerated types are also base types. Thus, the `enum_type` class is derived from the `base_type` class.

`TYPE_CONST`
`TYPE_VOLATILE`
`TYPE_CALL_BY_REF`
`TYPE_NULL`
: These operators are modifiers for other types. They use the `modifier_type` derived class.

The integer, floating-point, pointer, and enumerated types are all classified as *scalar* types, and the `is_scalar` method checks if a type is one of these. The array, structure, and union types are all *aggregate* types.

The type_node base class defines several methods that are applicable to all types, and those are described in the first section below along with some other comments about types in general. The bulk of this chapter is devoted to the various kinds of types and the corresponding derived classes. The last section describes the common types that are predefined in the global symbol table.

7.1 Features Shared By All Types

The base type_node class defines some common features of all types. This section describes the methods from this base class along with some comments about conventions for using types. Besides the things covered here, each type_node contains an ID number that uniquely identifies it within a particular context. See Section 5.6 [Numbering Types and Symbols], page 42.

New types can be created as necessary, but once a type has been entered in a symbol table it should generally not be changed. This helps to avoid duplicate types and to prevent unintended side effects when a type is used in multiple places. Instead, the copy method can be used to get a new type that is the same as the existing type, except that it does not copy annotations. The copy can then be freely modified and installed in a symbol table. If necessary, annotations on the type can be copied separately using the copy_annotes method. See Section 8.7 [SUIF Objects], page 70.

When a type is entered in a symbol table, it automatically records a pointer to that parent table. Similarly, when the type is removed from the symbol table, its parent pointer is cleared. The parent method retrieves this parent pointer.

Each type has a size method that returns the number of bits used to store values of that type. (That is not very meaningful for function types, so they always have a fixed size of zero.) The size for a type may or may not be set directly, depending on the type operator. The sizes for type nodes from each of the derived classes are described in the appropriate sections below.

Structures, unions, and enumerated types are all given names. The is_named method checks to see if a type_node is one of these types. The name of a type should be unique within the symbol table where it is defined, but the type name space is separate from the names for symbols. Because the type names are automatically entered in the lexicon (see Section 12.3 [Lexicon], page 84) when the types are created, they can be compared as pointers without doing string comparisons.

More than one type_node can represent the same type. That is, types may be equivalent even if they are represented by different type nodes. Given that, the only reason to reuse existing types is to keep the symbol tables from getting too big. The is_same method is provided to check if two types are equivalent, so that the symbol tables can be kept to a reasonable size. However, is_same is only intended to help get rid of duplicates. Because it assumes that duplicate types are still perfectly legal, it may potentially return false negatives to avoid the expense of comparing annotations on type nodes[1]. For named types, the is_same method assumes that all type nodes are unique; it does not check for structural equivalence. In most cases, is_same need not be called directly. Instead, the symbol table

[1] Even if the current implementation compares annotations, future versions of the SUIF library may not, so do not depend on that behavior.

`install_type` method (see Section 5.5 [Adding and Removing Entries], page 41), which uses `is_same` to detect and avoid duplicate types, is the recommended way to add new types to a symbol table.

SUIF has its own definition of type compatibility. Two types do not need to be strictly equivalent to be compatible. Besides the rules for type equivalence, the following conditions define which types are compatible:

- Enumerated types and integers of the same size are compatible if they are both signed or both unsigned.
- Any two pointer types are compatible.
- Type modifiers and annotations do not affect compatibility.

The `compatible` method is included in the `type_node` class to determine if two types are compatible according to these rules.

There are three different methods for printing SUIF types. The `print_abbrev` method is used when printing the result types of instructions. It prints the type ID number along with a single character to identify the type operator followed by a period and the size (e.g. '`i.32`' for a 32-bit integer). The `print` method shows the ID number prefixed with '`t:`' to identify it as a type. The `print_full` method prints all of the type information and is used when listing symbol tables. It's optional `depth` parameter can be used to specify the indentation level.

Many kinds of type nodes contain fields that refer to other types. Moreover, each type may contain annotations that include references to other types. With one exception, however, recursive type references are not allowed. In other words, if the type nodes are viewed as a directed graph with the references between them forming the edges, there can be no cycles. To support recursive data structures, we allow an exception to this rule: types within a field of a structure or union may refer back to the structure or union type.

7.2 Base Types

The `TYPE_VOID`, `TYPE_INT`, `TYPE_FLOAT`, and `TYPE_ENUM` operators define the base types. Whereas other kinds of types are defined in terms of existing types, the base types stand alone. The `base_type` class is derived from `type_node` to hold the base types. It has only two fields: the size and a flag to indicate whether integer and enumerated types are signed or unsigned.

The `set_size` method sets the size of a base type in bits. Types with a `TYPE_VOID` operator should always have a size of zero. Integer and floating-point types can have any sizes, although normally the sizes are restricted by the target machine parameters (see Section 12.4 [Target Machine Parameters], page 85). Other sizes may cause the code generator to fail. For integer and enumerated types, the `is_signed` method is used to check the flag that indicates whether they are signed or unsigned. The `set_signed` method changes the value of the flag.

Besides the features that they share with the other base types, enumerated types have other fields and methods used to record the names and values of their members. See Section 7.7 [Enumerated Types], page 59.

7.3 Pointer Types

Pointer types are defined using the TYPE_PTR operator and the ptr_type derived class. The ref_type method returns the type that is pointed to by a particular ptr_type node. The set_ref_type method can be used to change this referent type. The size of a pointer type is determined by the target machine (see Section 12.4 [Target Machine Parameters], page 85); it cannot be changed for individual pointer types.

7.4 Array Types

A SUIF array type defines a new type that is a one-dimensional vector with elements of another type. Multi-dimensional arrays are handled as nested arrays (i.e. arrays of arrays) as in C. The size of an array is specified by the upper and lower bounds. Each bound is either an integer expression or unknown. If the bound is an integer expression, it can either be an integer known at compile time, or a SUIF variable symbol representing an integer value to be computed at run time.

The upper and lower bounds of an array type are stored in objects of the array_bound class. An array bound can be one of three things, an integer constant, a SUIF variable symbol, or entirely unknown. The is_constant, is_variable, and is_unknown methods check for these three conditions. If the bound is a constant, the constant method returns the integer value of the bound. Likewise, if it is a variable, the variable method returns a pointer to the variable symbol. The print method is also available to print out the value of an array bound. The global variable unknown_bound is a predefined array bound that can be used when a bound is unknown.

A type node with a TYPE_ARRAY operator is stored in an array_type object. The array_type class contains three fields: the lower and upper bounds and the element type. The lower_bound, upper_bound, and elem_type methods retrieve the contents of these fields, and the set_lower_bound, set_upper_bound, and set_elem_type methods change their values. The SUIF definition of arrays says that the elements will be stored adjacently. If this would violate alignment restrictions, the element types must be padded with extra space. For example, if some machine has 24 bit integers with a 32 bit alignment requirement, the 24 bit integers may not be used directly as array elements; instead, one would have to create a structure of size 32 bits that contains a 24 bit integer at offset zero and no other fields, and then use arrays of this structure type.

The bounds of an array type tell two things: the number of elements and the how the index of an array access instruction will be interpreted. The number of elements determines the size of the array – the size will be the number of elements times the size of each element. Any array type with an upper or lower bound that is anything but an integer constant will not have a size that is known at compile time, so such a type cannot be used anywhere a known size is required, such as for a variable type, an expression type, or the type of an object copied by a memcpy instruction.

The other time when array bounds are meaningful is in an array access (a SUIF in_array instruction). There each index is required to be between the given bounds or the result is undefined. The location referred to is determined by subtracting the lower bound from the index and then multiplying by the size of the next lower dimension, or element size if there are no more dimensions in the array access instruction. The size in this case is not necessarily known at compile time and is considered to be the difference between the upper and lower bounds plus one (the number of elements) times the size of the next lower dimension, or the element size if this is the smallest dimension. The bounds are evaluated after all sub-expressions of the array access instruction have been evaluated and before the result of the array access instruction itself. Any bounds needed in this calculation must evaluate to integers when this array access instruction is executed, so they must be integer constants or variable symbols, and if variable symbols they must have defined values at this point. Note that this implies that the array type for an array access may not have unknown lower bounds and may have only one unknown upper bound, that of the outermost dimension. The `are_bounds_unknown` method checks if the upper bound of an array type is unknown.

Note also that all of this information from the array type is also available in a different form from the `in_array` class, between the `bound` expressions and `offset_op` expression. The results are required to be the same at run time whether the array type bound information is used or whether the `bound` and `offset_op` expressions are used, or else the resulting behavior is undefined. This redundancy allows more flexibility in analyzing array access patterns. Code transformation passes can treat the `in_array` sub-expressions like any other expressions and propagate additional information in, and dependence analysis on individual accesses can look at these general expressions. Code dealing with objects and types only has to deal with the symbolic placeholders.

Array bounds are subject to certain constraints. The upper bound may never be less than the lower bound. If both are constants, this constraint applies to those constants. If one or both are symbolic bounds, this constraint must hold for every array access for which this is the array type. Symbolic bounds are typically local variables that are assigned once each at the very beginning of a procedure and used only as bounds, so their values have the same scope as the types they are used in. This corresponds directly to the declarations in Fortran code of array types with non-constant bounds, but this particular form is not required by SUIF, and heavily transformed code may be somewhat different. What is required is that symbolic bounds not be sub-variables or call-by-reference parameters, since these are really just short-hand for other expressions (see Chapter 10 [Fortran], page 77).

7.5 Function Types

The `TYPE_FUNC` operator is used with the `func_type` derived type class to describe the types of SUIF procedures. A function type includes the type of the return value and optionally the types of the arguments as well. The `return_type` and `set_return_type` methods access the return type field. The type system itself places no restrictions on the return types, but the SUIF call and return instructions do. See Section 4.7 [Call Instructions], page 33

Most of the complexity of function types involves the arguments. Pointers to the argument types are stored in a dynamically-allocated array. The number of arguments, and thus the size of the array, is stored in a field that can be accessed with the `num_args` and `set_num_args` methods. The number of arguments can be changed at any time; if necessary the argument array will be reallocated. The type of a particular argument can be retrieved with the `arg_type` method and set with the `set_arg_type` method. The arguments are numbered beginning with zero.

Argument types are under the same restrictions as result types of instructions (see Section 4.1.3 [Result Types], page 25) – they must be object types with known, non-zero size.

Since functions can be declared in C without complete prototypes, a function type may not be able to include any information about the arguments. This is indicated in a SUIF function type by the result of the `args_known` method. This flag can be set with the `set_args_known` and `set_args_unknown` methods. Setting `args_known` FALSE sets the argument count to zero and the `has_varargs` flag to FALSE. Conversely, setting the argument count to a non-zero value or the `has_varargs` flag to TRUE sets the `args_known` flag to TRUE.

Some functions in C and other languages have a variable number of arguments. The SUIF function types support this by keeping a flag to indicate if there may be additional arguments besides the ones that are explicitly listed. The `has_varargs` and `set_varargs` methods access this flag. Note that the `has_varargs` flag should not be used if nothing else is known about the arguments.

7.6 Structure and Union Types

Structure and union types are defined with the `TYPE_STRUCT` and `TYPE_UNION` operators and use the `struct_type` derived type class. Both of these types include a name and a list of fields. The difference between them is that the fields of a union are all stored at the same offset so that only one field may be used at a time, while the fields of a structure are not allowed to overlap. The `name` method returns the name of a structure or union type, which should be distinct from the names of other types within the symbol table where it is defined. The name is automatically entered in the lexicon (see Section 12.3 [Lexicon], page 84) when it is set with the `set_name` method.

The `struct_type` class contains a field to record the total size of the type in bits. The `set_size` method is used to assign to this field. Because structure and union types may be used as array elements, they must be able to tile an array without violating any alignment restrictions. This means that extra padding may need to be added at the end of a structure or union. Consequently, the total size of the type may be greater than the sizes of the fields.

The names, types, and offsets of the fields are stored in dynamically-allocated arrays within a `struct_type` node. The `set_num_fields` method determines the number of fields and thus the size of these arrays. The number of fields may be changed at any time. If necessary, additional space will be allocated. The `num_fields` method returns the current number of fields. The `field_name`, `field_type`, and `offset` methods retrieve the field names, types, and offsets, and the `set_field_name`, `set_field_type`, and `set_offset` methods change their values. The fields are numbered beginning with zero. For structures the fields must be in order of increasing offsets. The field offsets for union types should all be zero. The field names are automatically entered in the lexicon (see Section 12.3 [Lexicon], page 84).

The `find_field_by_name` method can be used to search for a field with a particular name. If successful, it returns the index of the field; otherwise, it returns a value of `-1`. The `find_field_by_offset` method can also be used for structure types to find the field at a certain offset. If the specified offset is not exactly at the beginning of a field, `find_field_by_offset` returns the index of the field containing it and saves the offset within that field in the `left` parameter.

7.7 Enumerated Types

Enumerated types are much like integer types, except that each member of an enumeration is given a name. The TYPE_ENUM operator is classified as one of the base type operators, and the enum_type class which it uses is derived from the base_type class. See Section 7.2 [Base Types], page 55. Since an enumerated type behaves like an integer type, the same base_type methods can be used to specify the size and whether or not it is signed.

Like structures and unions, enumerated types are given names. The name method returns the name for an enum_type, which should be different from the names of other types in the symbol table where it is defined. The set_name method automatically enters the name in the lexicon (see Section 12.3 [Lexicon], page 84).

Each enumerated type contains arrays to hold the names and values of its members. The num_values method returns the number of members. The set_num_values method may be called at any time to change the number of members. It will allocate more space for the arrays if necessary. The member method returns the name of a particular member, and the value method returns its value. The members are numbered beginning with zero. The set_member method changes the name of a member and enters the new name in the lexicon. Similarly, the set_value method changes the value for a method. All of the values must fit within the size of the type, and negative values are not allowed with unsigned enumerations.

The find_member_by_name method searches through the members of an enumerated type for a particular name. If successful, it returns the index of the member; otherwise, it returns a value of -1. The find_member_by_value method returns the index of the first member that it finds with a certain value. If the value is not found, it returns a value of -1, just like find_member_by_name.

7.8 Modifier Types

Modifier types are used to add various attributes to other types. Each attribute is represented by a different type operator:

TYPE_CONST
: This modifier signals that the type is for constant values.

TYPE_VOLATILE
: This is used for the types of variables whose contents may change at any time.

TYPE_CALL_BY_REF
: This is used in the Fortran mode (see Chapter 10 [Fortran], page 77) to identify call-by-reference parameters.

TYPE_NULL
: This modifier has no effect on the type but simply provides a place to attach annotations. This is needed for named types because copying them would create completely new types.

All of these modifier types use the modifier_type derived class. The base method returns a pointer to the type that is modified, and the set_base method changes that field. More than one modifier can be applied to the same type. The order of the modifiers does not matter, but no modifier should be used more than once with the same type (for example, a constant constant integer is illegal, but a constant pointer to a constant integer is OK).

The base `type_node` class includes several methods to deal with modifiers. The `is_const`, `is_volatile`, and `is_call_by_ref` methods check for the corresponding modifiers on a type. The `unqual` method skips over any modifier types and returns the unqualified type. This is used frequently when checking for particular kinds of types. Finally, the `find_modifier` method checks if a type is modified by a particular modifier type operator. If so, it returns a pointer to that modifier type; otherwise it returns `NULL`.

7.9 Predefined Types

The global symbol table (see Section 5.1.1 [Global Symbol Table], page 37) is initialized with some predefined types. The SUIF library also defines a number of variables containing pointers to these predefined types, so that you don't need to search the symbol tables for most commonly-used types. The first set of predefined types are fixed-size scalar types:

`type_v0` This is a `void` type with size zero.

`type_s8`
`type_s16`
`type_s32`
`type_s64` These are signed integer types of 8, 16, 32, and 64 bits.

`type_u8`
`type_u16`
`type_u32`
`type_u64` These are unsigned integer types of 8, 16, 32, and 64 bits.

`type_f32`
`type_f64`
`type_f128`
These are floating-point types of 32, 64, and 128 bits.

`type_error`
This variable is just set to `NULL`. It may be used to represent various error conditions.

Because the fixed-size types do not necessarily match the target machine parameters (see Section 12.4 [Target Machine Parameters], page 85), they should rarely be used. Instead the SUIF library also predefines pointers to types that correspond to the most common C types. These "portable" types are automatically initialized using the target machine parameters.

`type_void`
Since `void` types always have a size of zero, this type is equivalent to `type_v0`, but it is included for completeness.

`type_ptr` This type is a pointer to `type_void`. Like all other pointer types, it automatically has the size of pointers on the target machine.

`type_char`
This is an integer type with the size specified for `C_char` types in the target machine parameters. The `char_is_signed` machine parameter determines whether it is signed or unsigned.

`type_signed_char`
`type_unsigned_char`
: These are signed and unsigned integer types with the size specified for `C_char` types in the target machine parameters.

`type_signed_short`
`type_unsigned_short`
: These are signed and unsigned integer types with the size specified for `C_short` types in the target machine parameters.

`type_signed`
`type_unsigned`
: These are signed and unsigned integer types with the size specified for `C_int` types in the target machine parameters.

`type_signed_long`
`type_unsigned_long`
: These are signed and unsigned integer types with the size specified for `C_long` types in the target machine parameters.

`type_signed_longlong`
`type_unsigned_longlong`
: These are signed and unsigned integer types with the size specified for `C_longlong` types in the target machine parameters.

`type_float`
`type_double`
`type_longdouble`
: These are floating-point types with the sizes specified for `C_float`, `C_double`, and `C_longdouble` types in the target machine parameters.

8 Annotations

Because it is a research compiler, SUIF is designed to be both flexible and extensible. *Annotations*, which may contain arbitrary data structures, are the primary means of providing this extensibility. Objects that are derived from the `suif_object` class have lists of annotations associated with them. This includes most of the major SUIF objects such as instructions, tree nodes, symbols, types, symbol tables, and file set entries. New annotations can be declared by any program and used to record whatever information is needed within that program. They can also be written to the SUIF output files so that other programs can use them. The *annotation manager* records the annotation names and the format of the data associated with each kind of annotation.

8.1 Overview

Annotations are represented by `annote` objects which are implemented in the files '`annote.h`' and '`annote.cc`'. Each annotation contains two fields: a name and a data pointer. The name is a character string that must be entered in the `lexicon`. See Section 12.3 [Lexicon], page 84. The name serves to identify the purpose of the annotation, and with the help of the annotation manager, it specifies the format of the associated data. The `set_name` method can be used to change the name of an annotation, but in most cases it's easier to just create a new annotation.

As a convention, each annotation name should have a global variable that is a pointer to its entry in the `lexicon`. The names of these variables should be `k_` followed by the annotation name. For example, `k_repeat_init` is a global variable defined in the library to point to the `"repeat_init"` annotation name. Because the annotation names are always in the `lexicon`, you can just compare the names with the global variables instead of performing string comparisons. For example:

```
annote *an;
if (an->name() == k_repeat_init) {
    an->print();
}
```

The data field in an annotation is a `void*` pointer so that, with the appropriate type cast, it can refer to any data structure. The `data` and `set_data` methods may be used to directly access an annotation's data field. The `immeds` and `set_immeds` methods also provide access to the data in an annotation, but only if it can be represented by a list of `immed` values; this is described in more detail below for each kind of annotation.

The `print` method prints an annotation to the specified file. The format depends on the kind of annotation. Flat annotations are printed as lists of `immed` values. For unregistered annotation, the contents of the data field are printed directly as a pointer value (a hexidecimal representation on most systems). Structured annotations may have user-defined printing functions registered in the annotation manager; if such a function is defined, the library uses it. If no printing function is defined but a function to convert to a flat `immed` list is defined, that is used to temporarily convert to flat lists and print as such. If neither a printing nor a conversion function is provided, the printing is done as with unregistered annotations.

8.2 Flat Annotations

The simplest type of annotation is a *flat annotation*. The data in a flat annotation is a pointer to a list of `immed` values. See Section 12.1 [Immeds], page 83. By default, annotations read from an input file are flat annotations. That is, if a SUIF input file contains an annotation with a name that has not yet been registered, the SUIF library will automatically register it as a flat annotation.

When creating new flat annotations, you may specify both the annotation name and a pointer to an `immed` list. In many cases, however, it is more convenient to only provide the annotation name and let the annotation constructor automatically fill the data field with a pointer to a new list.

For flat annotations, the `immeds` method merely provides a shortcut for accessing the data by casting it to an `immed_list` pointer. Similarly, the `set_immeds` method is functionally equivalent to `set_data` in this case.

8.3 Structured Annotations

In many cases, flat annotations are cumbersome to use because the fields in the `immed` value list are unlabeled and difficult to access. With *structured annotations*, the data is held in a user-defined structure. You can access this structured data by using the `data` method and casting the result to the desired type. It may be helpful to define macros to perform the type casts.

An important restriction on structured annotations is that if they are to be stored in SUIF files, they are stored as `immed` lists. Therefore, if you wish to write such an annotation to a file, you must provide the annotation manager with functions to convert back and forth between the `immed` lists and the structured data. When a structured annotation is first read from an input file, the annotation manager applies the function to convert the `immed` list to your data structure. Similarly, when the annotation is written out, the manager applies the function to convert it back to a list of `immed` values.

The data in a structured annotation that defines a conversion function must always be maintained in a form that can be converted to an `immed` list. This is necessary because the library occasionally needs to flatten annotations so that it can update the contents. For example, when an annotation containing references to local symbols is moved to another scope, the symbols may need to be updated. When this occurs, the library also converts the data back to its structured form after it has been updated.

Just as the library needs to temporarily flatten structured annotations, you may need to do the same. The `immeds` and `set_immeds` methods make this task much easier. The `immeds` method returns the annotation data represented as an `immed` list. For a structured annotation, it does this by using the conversion function registered with the annotation manager to convert the structure to a flat list. Note however that the result is a pointer to a new list, whereas for flat annotations, it is a pointer to the actual annotation data. Similarly, the `set_immeds` method replaces the annotation data with the data in an `immed` list, but for structured annotations, it first converts the data to the structured format. It does *not* try to deallocate the previous contents of the data field, so you are responsible for handling that. In summary, the `immeds` and `set_immeds` methods allow you to access both flat and structured annotation data as `immed` lists. The only difference is that for structured annotations the `immed` lists are not the actual annotation data, and consequently you may need to deallocate the duplicate data structures.

Besides the conversion functions, the annotation manager records functions to deallocate and print the structured data. Both of these functions are optional. If the data fields for a particular kind of structured annotation contain pointers and you do not supply a function to deallocate the storage that they reference, it is your responsibility to ensure that that storage is deallocated elsewhere in your code. (For example, you may want to have the pointers all refer to entries in a table that is deallocated separately.)

8.4 Unregistered Annotations

If an annotation name is not registered with the manager, the format of the annotation data is unknown. Such annotations cannot be written to the SUIF output file, and when they are printed the data is just shown as a hexadecimal value. If the data needs to be deallocated, you must do so explicitly before the annotations are deleted. The library cannot automatically update the data in unregistered annotations, and the cloning functions do not copy them. The `immeds` method returns `NULL` for unregistered annotations and `set_immeds` will not work at all for them.

Due to these limitations, annotations should generally be registered with the manager. There are, however, many situations where unregistered annotations are useful. Because they are not written out, the data in these annotations does not have to be converted to `immed` lists. This means that they may contain totally arbitrary data structures. The drawback is that it is entirely up to you to ensure that the annotation data is maintained properly.

8.5 Predefined Annotations

The SUIF library automatically defines a number of annotations for its own use. Most of these are only used internally to handle reading and writing SUIF files. A few of them are visible to users. Unless specified otherwise, they are all flat annotations.

8.5.1 Initial Data Annotations

Variables that are not in the automatic storage class can be initialized by attaching annotations to their definitions (see Section 6.4.3 [Variable Definitions], page 51). These annotations are very low-level. Each one specifies the initial value for a certain number of bits. Multiple annotations may be used, in which case their order is significant.

`k_fill`
: This annotation indicates that the variable should be initialized with `N` bits filled with bit pattern `V`, where `N` is the first entry on the list and `V` is the second. The size must be a multiple of the byte size and the bit pattern must be one byte long. Currently, zero is the only bit pattern that is supported.

`k_repeat_init`
: This is a generalized version of the `fill` annotation. The variable is initialized to `R` copies of `N` bits holding the value `V`, where `R`, `N`, and `V` are the first, second, and third entries on the `immed` list. The values that are currently supported are integers, floating point values, and symbolic addresses.

`k_multi_init`
: This is another initial value annotation that is used to initialize character strings. The first entry on the list is the size in bits of each character. The rest of the entries on the `immed` list are integers that specify the characters in the string.

Despite the low-level nature of the initial data annotations, SUIF still requires that they correspond to the types of the variables. For example, character strings should only be used to initialize character arrays. Each value specified in one of these annotations must fit exactly into the corresponding variable, field, or array element. The only exception to this is that multiple fields or array elements may be initialized to zero with a single `fill` annotation.

For example, this structure

```
struct {
    int x[2];
    float fp;
    char string[7];
} sample;
```

might be initialized with the following annotations:

```
["fill": 64 0]
["repeat_init": 1 32 1.2000e+00]
["multi_init": 8 102 97 116 112 105 103 0]
```

8.5.2 Call-By-Reference Annotations

Fortran and other languages pass procedure parameters by reference but SUIF uses call-by-value. To implement call-by-reference parameters, SUIF passes pointers to the actual parameters and then dereferences those pointers when the formal parameters are used. While this is perfectly adequate for many compiler passes, the pointer dereferences are difficult to handle in some high-level Fortran passes. The SUIF Fortran mode automatically converts the code to make it appear that the parameters are passed by reference. The library uses two predefined annotations to implement this.

`k_call_by_ref`
: The Fortran front-end puts this annotation on pointer types that are used to implement call-by-reference parameter passing. The library uses this to detect parameters that should be modified when converting to Fortran mode. This annotation has no data on its `immed` list.

`k_orig_type`
: While in Fortran mode, the types of the call-by-reference parameters are temporarily replaced. The `orig_type` annotations are attached to the parameter variables to record their original types. The `immed` list contains a single entry, the original pointer type of the parameter. These annotations are never written to the output files.

8.5.3 Common Block Annotations

Fortran `common` blocks are represented in SUIF by global group types. These can be accessed just like any other structures. In addition, and the Fortran front-end will create sub-variables to represent the fields in the `common` block, so most of the time only the sub-variables themselves will be seen. If you want to know whether a particular global group represents a `common` block, you can check for the `k_common_block` annotation, which is put only on variables representing `common` blocks (the blocks, not the fields).

8.5.4 Miscellaneous Annotations

The SUIF library defines a few other annotations to record various attributes of SUIF programs. These are straightforward except for the `fields` annotation which is described here in more detail.

`k_line` This annotation records line numbers from the source code to be used when debugging the object code. It is usually attached to a `mrk` instruction. The first list entry is the integer line number and the second is the character string for the file name.

`k_history`
: As each SUIF pass runs, the library automatically records the command-line for the pass as an annotation on the file set entries. These `history` annotations allow you to see how a particular SUIF file was generated.

`k_enable_exceptions`
: This annotation is attached to a `proc_symtab` to indicate which run-time exceptions should be detected within that procedure. The entries on the `immed` list are character strings. The system currently recognizes `"int_divide-by-0"` and `"int_overflow"`. IEEE floating point exceptions need to be added.

`k_reg_num`
: This annotation is attached to variables that represent machine registers to record the register numbers. It has one immediate value which is the integer register number. The meaning of the register numbers are machine-dependent.

`k_fields` A field within a structure or union variable is specified in SUIF by a constant offset and a base address. That is all that is needed to generate code. However, if the offset is zero or if the variable is a union, that information is not sufficient to determine the field. The `fields` annotation may be used to provide the field names that were used in the original source code. The `immed` list for one of these annotations contains the names of the fields being accessed. (There may be multiple field names because of nested structures or unions.)

The `fields` annotation is somewhat more complicated than the others. The difficulty is in defining where this annotation should be used. Intuitively, the `fields` annotation should be placed on the first instruction that produces a pointer to the field. There are basically three different situations to consider.

In the simplest case, a field can be addressed directly with an `ldc` instruction (see Section 4.6 [Load Constant Instructions], page 33). Since that produces a pointer to the field, a `fields` annotation may be placed on the `ldc` instruction. Of course, the entries on the annotation must be valid field names for the variable that is addressed.

To access a field of a structure or union variable that is referenced by a pointer, the constant offset must be included with an explicit `add` instruction. Since the result of the addition is a pointer to the field, the `fields` annotation should be placed on the `add` instruction. Note that if the field offset is zero, the `add` instruction may be replaced by a `cvt` instruction. The same rule applies: the `cvt` instruction gets the `fields` annotation because it produces a pointer to the field.

For arrays of structures or unions, an `array` instruction (see Section 4.8 [Array Instructions], page 34) may include a constant offset for a field within the array element being addressed, so this is another type of instruction that may include a `fields` annotation. The entries on the `fields` annotation must be valid fields within the array element type.

8.6 Annotation Manager

The annotation manager keeps track of the annotations that have been registered. For each annotation name, it records whether the annotation is flat or structured and whether it should be written to the output file. Additional information is stored for structured annotations.

The annotation manager is implemented in the files '`aman.h`' and '`aman.cc`'. The `annote_def` class is used to hold the annotation information. Each `annote_def` object records the information for annotations with a particular name (which can be accessed with the `name` method). As with the annotations themselves, the name in the `annote_def` is entered in the `lexicon` (see Section 12.3 [Lexicon], page 84), except that in this case the `annote_def` constructor automatically makes sure that the name is in the `lexicon`.

The manager is implemented as a list of `annote_def` objects. This list is initialized by the `init_aman` function, which is automatically called when the library is initialized. The `register_annote` function adds an `annote_def` to the list. An error occurs if the name in the `annote_def` is already used in another registered `annote_def`. Rather than calling the `register_annote` function directly, use the `ANNOTE` and `STRUCT_ANNOTE` macros described below to enter new annotations.

The information in the manager can be retrieved using the `lookup_annote` function. Given an annotation name that is entered in the `lexicon`, `lookup_annote` searches the list of registered annotations and returns the `annote_def` corresponding to that name. If the name is not found, it returns `NULL`.

8.6.1 Flat Annotation Definitions

Flat annotation definitions (see Section 8.2 [Flat Annotes], page 64) are represented by base `annote_def` objects, for which the `is_structured` method returns `FALSE`. Other than the annotation name, these entries contain a flag that indicates whether the annotations should be written to the SUIF output file. The `output` method is used to retrieve this flag and the `set_output` method to change its value.

The `ANNOTE` macro provides a convenient way to register flat annotations. This macro enters the annotation name in the `lexicon`, creates an `annote_def` object, and registers it with the manager. It takes three arguments: the variable that will hold the pointer to the annotation name in the lexicon, the annotation name, and the value of its output flag. For example:

```
char *k_my_annote;
ANNOTE(k_my_annote, "my_annote", TRUE);
```

This enters the name "my_annote" in the lexicon and sets k_my_annote to point to that string. It also registers that name as a flat annotation that will be written to the SUIF output file. Typically, k_my_annote will be a global variable that is used throughout the program to refer to annotations of this type.

8.6.2 Structured Annotation Definitions

The struct_annote_def class, which is derived from the annote_def class, is used to record the definitions of structured annotations. Objects of this class behave just like base annote_def objects, except the is_structured method returns TRUE. In addition, these objects contain pointers to four functions:

from
: The from function converts the annotation data from a list of immed values to the user's data structure. This function is required.

to
: The to function converts the user's data structure to a list of immed values. This function is required.

free
: The free function is optional, but it should be provided if the annotation data needs to be deallocated when the annotation is deleted.

print
: The print function is also optional, but without it the data in a structured annotation is simply printed as a hexadecimal value. This function allows you to print the annotations in more meaningful formats. As much as possible, the annotations should be printed in the same style as flat annotations.

Structured annotations must be registered with the manager before the SUIF input file is read. Otherwise, the annotations will remain as immed value lists, instead of being converted to the appropriate data structures. The STRUCT_ANNOTE macro should be used to register structured annotations. This is just like the ANNOTE macro except that it takes four more arguments for the struct_annote_def functions. For example:

```
char *k_struct_annote;
void *ann_from(char *name, immed_list *il, suif_object *obj);
immed_list *ann_to(char *name, void *data);
void ann_free(void *data);

STRUCT_ANNOTE(k_struct_annote, "struct_annote", TRUE,
              ann_from, ann_to, ann_free, NULL);
```

This registers the annotation "struct_annote" as a structured annotation. The ann_from function will be used to convert the annotation data from a list of immed values; the ann_to function will be used to convert the data back to an immed_list; and the ann_free function will be used to deallocate the annotation data. No function is provided to print the annotation data.

8.6.3 Unregistered Annotations in the Input File

When an unregistered annotation is read from a SUIF input file, it is automatically registered as a flat annotation. This is done so that the manager knows to write the annotation back in the output.

Annotations are often added to a SUIF file in one pass to record some sort of information, perhaps the results of a data flow analysis. Those annotations may then used in later passes to guide various transformations. Even if intervening passes do not recognize the annotations, they will still be propagated to the output. (It is up to the user, of course, to make sure that the intervening passes do not invalidate the information in the annotations without updating them.)

8.7 SUIF Objects

Most of the significant classes in the SUIF library are derived from a common base class. This `suif_object` class includes a field with a pointer to a list of annotations. Thus, annotations can be attached to most objects in SUIF, and all of these objects share the same interface for accessing the annotations. The files '`suifobj.h`' and '`suifobj.cc`' contain the code for the `suif_object` class.

Besides the annotation list, each `suif_object` has a field to identify the kind of the object. The `object_kind` method retrieves the value from this field. The `object_kinds` enumerated type defines the possible values:

`FILE_OBJ` File set entry. See Section 2.2 [File Set Entries], page 8.

`TREE_OBJ` Abstract syntax tree node. See Section 3.1 [Tree Nodes], page 11.

`INSTR_OBJ`
SUIF instruction. See Chapter 4 [Instructions], page 23.

`SYMTAB_OBJ`
Symbol table. See Chapter 5 [Symbol Tables], page 37.

`SYM_OBJ` Symbol node. See Chapter 6 [Symbols], page 45.

`DEF_OBJ` Variable definition. See Section 6.4.3 [Variable Definitions], page 51.

`TYPE_OBJ` Type node. See Chapter 7 [Types], page 53.

The rest of the methods in the `suif_object` class are related to handling annotations. First of all, the `are_annotations` method checks to see if an object has any annotations attached to it. If so, the `annotes` method can be used to retrieve a pointer to the list of annotations. Do not just grab the list of annotations and check to see if it is empty, but use `are_annotations` instead. Since many objects don't have any annotations, the library doesn't allocate annotation lists unless they are needed. Calling `annotes` will create a new list if one did not already exist, so to avoid creating a lot of empty lists use `are_annotations` first.

The `prepend_annote` and `append_annote` methods are a convenient way to add new annotations to an object. They simply create a new annotation using the specified name and data and then add it to the beginning or end, respectively, of the annotation list. If the object is only supposed to have one annotation with a particular name, the `set_annote` method works well for assigning new values for that annotation. If the object already has an annotation with the specified name, `set_annote` will replace the existing one (or the first one that it finds); otherwise, it just adds the new annotation at the end of the list.

There are two different methods to retrieve data from annotations attached to an object. The first, `peek_annote`, simply returns the data field from the first annotation that it finds with the specified name. If it doesn't find any annotations with that name, it returns a `NULL` pointer. The `get_annote` method does the same thing except that it is destructive. Besides returning the data from the annotation, it also removes the annotation from the list and destroys it. Note that with these methods it is impossible to distinguish the case where an annotation exists but has a `NULL` data field from the case where the annotation does not exist. If you want to retrieve the actual annotation objects instead of just their data fields, the annotation list (see Section 8.8 [Annote Lists], page 71) provides `peek_annote` and `get_annote` methods to do that.

When some kinds of SUIF objects (e.g. symbols and types) are copied, the annotations are omitted. If you want to copy the annotations as well as the base objects, the `copy_annotes` method must be called separately. This method works for any two SUIF objects; they do not have to be the same kind. If the target object already has some annotations, the new ones are appended to the end of the list. Unregistered annotations are not copied.

The `suif_object` class provides a `print_annotes` method for printing the annotation list to a text file. This is used by the `print` methods for the derived classes, but it could also be used directly by users. The optional `depth` parameter specifies the indentation level. Beware that a new-line character is printed *before* each annotation and thus you probably want to print a new-line after calling this method.

The `num_output_annotes` method checks to see if an object has any annotations, and if so, it counts the ones that will be written to the output file. This is primarily used by the SUIF library and most users will not need it.

8.8 Annotation Lists

Annotations are stored on lists attached to SUIF objects. The library defines an `annote_list` class for this purpose. Besides the standard list functions (see Section 11.1 [Generic Lists], page 79), the `annote_list` class provides two additional methods. The `peek_annote` method searches through the list for an annotation with the specified name. It returns a pointer to the first such annotation or `NULL` if the search was unsuccessful. The `get_annote` method does the same thing, but it removes the annotation from the list.

9 Replicating Trees and Instructions

Many SUIF programs need to duplicate parts of abstract syntax trees or expression trees. For example, loop transformations such as unrolling and peeling require copies of the loop bodies, and entire procedures must be copied to implement procedure inlining. Because code replication can be quite complicated in SUIF, it is handled in the library by the *cloning* methods.

Several different kinds of SUIF objects can be cloned. The `tree_node`, `tree_node_list`, `instruction`, and `operand` classes have `clone` methods. The various derived `tree_node` classes also have wrapper methods to cast the result of the base class `clone` method to the appropriate pointer types. When an object is cloned, everything within it is recursively cloned. In addition to the child nodes and instructions, this includes symbol tables and the types and symbols within them. All of the known references to cloned objects (including those in registered annotations) are updated.

The complexity of cloning is primarily due to SUIF's scope rules. The code being copied may reference symbols and types that are not visible in the scope where the copy will be used. Thus the cloning process is divided into three steps:

1. The code to be copied is scanned for references to symbols and types that are not visible in the specified destination scope. These are referred to as *exposed references*.
2. The exposed references are resolved in the symbol table for the destination scope. New variables are created to replace the exposed variables. Types are usually moved up in the symbol table hierarchy so that they are visible in both the source and destination scopes.
3. The code is actually copied. Symbol tables nested within the code are copied and references to them and their contents are updated to use the copies. Similarly, the exposed references are replaced with references to any new symbols or types that were created in the previous step.

Each of these steps is described in more detail below. The methods that perform the individual steps are available to users who need more control over the cloning process. For example, you may want to insert code to initialize new variables that are created in the destination scope due to exposed references. The cloning methods can also be used to replace references to particular symbols or types without actually copying anything.

The `replacements` structure is used throughout the cloning process to keep track of exposed references and objects that have been replaced. This structure includes lists of symbols, types, variable definitions, symbol tables, and instructions. For each kind of object, there are two lists. Except in intermediate steps, these lists should be the same length. An entry on the first list refers to an object in the original code and the corresponding entry on the second list is a new object that has been created to replace the original.

9.1 Finding Exposed References

The first step in the cloning process is to scan through the code being copied to search for references to objects that are not visible in the destination scope. The `tree_node_list`, `tree_node`, `operand`, `instruction`, and `block_symtab` classes have `find_exposed_refs` methods to perform this operation. The results are returned in a `replacements` structure by putting the exposed references in the lists of original objects. The lists of new objects are left empty.

Besides checking for exposed references in the objects themselves, the `find_exposed_refs` methods also check for references contained in annotations attached to the objects. Both flat and structured annotations are checked, but unregistered annotations are ignored since they are not copied. The `find_annote_refs` method is used to check a `suif_object` for exposed references.

Label symbols are a special case. Even if a label symbol is visible in the destination scope, it must be replaced if the label instruction is to be cloned. Otherwise, there would be two label instructions sharing the same label symbol! Thus, the `find_exposed_refs` method adds the symbol in a label instruction to the list of exposed references. The exception to this is that label symbols that are defined within cloned symbol tables will automatically be replaced and are not included in the exposed references.

9.2 Resolving Exposed References

After the exposed references have been found, the destination scope must be updated to include definitions for the objects in the exposed references. The `base_symtab` class provides the `resolve_exposed_refs` method to accomplish this. This method takes the lists of original objects in the `replacements` structure produced by the `find_exposed_refs` method (see Section 9.1 [Finding Exposed References], page 73) and adds corresponding entries to the lists of new objects. New symbols are created and added to the symbol table to replace the originals. Variable definitions are also created as needed.

Because some types use name equivalence, they cannot be copied without creating different types. Thus, whenever possible, types are moved up in the symbol table hierarchy so that they are visible in both the source and destination scopes. If a type is successfully moved, the old and new entries for the `replacements` lists are the same. Some types contain references to variables (e.g. as array bounds), and if those variables are not already visible in the destination scope, the types cannot be moved. When that happens, `resolve_exposed_refs` resorts to making a new type where the variable references are updated to use the cloned variables.

9.3 Copying the Objects

Once the exposed references have been resolved, the code can be copied. The `tree_node_list`, `tree_node`, `operand`, `instruction`, `block_symtab`, and `proc_symtab` classes have `clone_helper` methods to perform the actual copying. These methods change references to objects in the original objects lists of the `replacements` structure to references to the newly created replacements.

Annotations on the objects are also cloned. The `clone_annotes` method is used to copy the annotations from one `suif_object` to another. Unregistered annotations cannot be copied because the structure of their data fields is unknown. The `clone_helper` methods call `clone_annotes` for each object that is cloned; you generally do not need to call it directly.

The `clone_helper` and `clone_annotes` methods can also be used to perform another related function. It is occasionally necessary to replace references to one set of objects with references to another set of objects. Since the cloning methods do that anyway, it is easy to have them update the references without actually making copies. Setting the `no_copy` parameter causes them to update the references as specified in a `replacements` structure without copying the objects.

10 Features for Compiling Fortran

SUIF is roughly based on C semantics and does not directly support some Fortran features. In particular, call-by-reference parameters occur frequently in Fortran programs and must be implemented in terms of other SUIF features. This introduces extra complexity that makes it harder to analyze and optimize the procedures. Fortunately this complexity can be hidden somewhat by converting the procedures to the SUIF Fortran form. The SUIF library provides functions to translate a procedure to this Fortran form after it is read from a file and to translate it back to the original SUIF form before it is written out. Common blocks and equivalences are generally handled using sub-variables, so the library's Fortran form does not need to do anything special to deal with them. Other more obscure Fortran features are also not handled in the Fortran form because they are less common and hard to deal with however they are represented.

10.1 Call-By-Reference Parameters

Procedure parameters in Fortran are passed by reference. SUIF uses call-by-value parameters and implements call-by-reference parameters by passing pointers to the actual arguments. When the formal parameters are used, the pointers must be dereferenced. However, one of the primary advantages of compiling Fortran programs is that there are no pointers in the source code. If not for the call-by-reference parameter pointers, most compiler passes that only deal with Fortran can be simplified by not having to deal with pointers. Thus the SUIF Fortran form automatically converts the code to make it appear that the parameters are passed by reference. The parameters are changed to call-by-reference types instead of pointers and the pointer dereferences are temporarily removed.

The `make_ref_params` function is used to convert call-by-reference parameters in a procedure to the Fortran form. It must be called before converting the procedure body to expression trees. This is done automatically by the `read_proc` method for a `proc_sym` if the `use_fortran_form` flag is set. Before writing a procedure to an output file, the call-by-reference parameters must be converted back to pointers using the `undo_ref_params` function. Since it is illegal to write out a procedure in the Fortran form, the `write_proc` method for the `proc_sym` always calls `undo_ref_params`; you do not need to call it directly. Both `make_ref_params` and `undo_ref_params` are defined in the file '`callbyref.cc`'.

The Fortran front-end must identify call-by-reference formal parameters by putting `call_by_ref` annotations (see Section 8.5.2 [Call-By-Ref Annotes], page 66) on the pointer types used by those parameters. Before actually changing the type of a call-by-reference parameter and removing its dereferences, `make_ref_params` checks that the pointer parameter is neither addressed nor assigned within the procedure. If this check fails, the pointer is not a valid call-by-reference parameter, so `make_ref_params` prints a warning message and does not convert that parameter. Otherwise, it goes ahead and changes the type of the parameter from a pointer to a call-by-reference type (see Section 7.8 [Modifier Types], page 59). Any dereferences of that pointer within the procedure are removed, and other references to the pointer are changed to `ldc` (load constant) instructions that take the address of the parameter. The end result of all this is that it appears that the parameters are passed by reference. Note that the call sites are not changed; the actual arguments for call-by-reference parameters are still passed as pointers.

The `undo_ref_params` function does as its name suggests and undoes the transformations applied by `make_ref_params`. The only potential complication is that while in the Fortran form the user may have used the call-by-reference parameters in ways that cannot be expressed outside of the Fortran form. Specifically, the call-by-reference parameters cannot be used as index variables of `for` loops (see Section 3.1.6 [For Nodes], page 16), and they cannot be used as bounds in array types (see Section 7.4 [Array Types], page 56). Both of those uses require direct references to variable symbols, and there is no place to insert the pointer dereferences required outside of Fortran form.

11 Generic Data Structures

The SUIF library includes a number of generic data structure classes. Some of these are used extensively by the library itself, while others are only provided for your convenience. Since most of these data structures are quite easy to understand, this section only gives a brief description of each one and points out the unusual and potentially confusing features.

11.1 Generic Lists

While linked lists are not very efficient data structures, they are flexible and work well for a research compiler like SUIF. Various kinds of lists are used throughout the SUIF library and most SUIF programs. Almost all of these lists are derived from the `glist` base class. The files '`glist.h`' and '`glist.cc`' contain the declaration and implementation of this class.

The individual elements of these lists are derived from the `glist_e` class. The base class only contains a pointer to the next element. To make a useful list, a derived class must add data fields to the list elements. For doubly-linked lists, the list elements are further extended to include backwards pointers (see Section 11.4 [Doubly-Linked Lists], page 80).

The `glist` class includes pointers to the head and tail elements. This makes it possible to efficiently add elements at either end of the list. A variety of methods are provided to insert and remove individual elements, as well as to combine entire lists in different ways. All of these operations are straightforward and are clearly documented in the code.

The SUIF library also provides iterators for lists. This allows you to write code at a higher level of abstraction. Instead of rewriting the same code everywhere that your program traverses lists, you can just use the built-in iterators. The base iterator class is `glist_iter`. An iterator is initialized with a pointer to a particular list. It then returns successive elements from the list (via the `step` method) until the `is_empty` method returns `TRUE`. Other methods are available to access the current and next list elements without advancing the iterator.

To make a list with a particular type of elements, one must derive new list classes that inherit from the base classes described above. While this is not difficult, it is cumbersome and time consuming. Instead of explicitly declaring these derived classes, the `DECLARE_LIST_CLASS` macro may be used to declare them automatically. This macro takes two arguments: the name of the new list class and the type for the data in the list elements. It then creates classes derived from `glist`, `glist_e`, and `glist_iter` that use the specified type for the element data[1]. For example, `DECLARE_LIST_CLASS(string_list, char*)` generates classes named `string_list`, `string_list_e`, and `string_list_iter`. The `string_list_e` class includes a `contents` field with type `char*`. The methods in all of these classes are changed to use the `char*` type where appropriate and some new methods are added. (Once the element data type is known, it is possible to provide methods, such as `copy`, that must access the data fields.)

The `DECLARE_LIST_CLASS` macro is implemented by other macros, one for each kind of class. Using these other macros directly gives you more control over the names of the

[1] C++ templates would be a better solution, but at the time when this was developed, we did not have a C++ compiler with a robust implementation of templates.

new classes and allows you to add more methods in the derived classes. There are several examples of this in the library code. Another advanced feature of the DECLARE_LIST_CLASS macro is the virtual set_elem method which is included in the generated list class. This method is called for every element that is added to a list. The base set_elem method does nothing, but a derived class can change this method to automatically update information in list elements when they are added to a list.

11.2 Move-to-front Lists

The move-to-front list class mtflist is just a minor variation of the glist class. See Section 11.1 [Generic Lists], page 79. The user must provide a function that compares a list element with some other unspecified object. The mtflist class then provides a lookup method that uses the comparison function to search for a particular element. After every successful lookup, the returned element is moved to the front of the list. This works well for applications with good locality. The DECLARE_MTFLIST_CLASS macro works just like the DECLARE_LIST_CLASS macro, but it generates move-to-front list classes. The code for the mtflist class is in the files 'mtflist.h' and 'mtflist.cc'.

11.3 Association Lists

An association list element contains both a key and a data pointer. The data associated with a particular key can be retrieved with a simple lookup method. The key and data fields are both void* pointers. Association lists are implemented by the alist and alist_e classes in the files 'alist.h' and 'alist.cc'. These classes are derived from the generic list classes (see Section 11.1 [Generic Lists], page 79) and behave similarly.

The amtflist class has the same interface as the alist class and provides the same functionality. The only difference is that it is based on the move-to-front list class. See Section 11.2 [Move-to-front Lists], page 80. You may want to use this if you expect your application to access list elements with a high degree of locality.

Like all of the other SUIF lists, the association lists have iterators to make it easy for you to traverse them. The alist_iter class uses the same interface as the generic list iterator and works for both alist and amtflist objects.

11.4 Doubly-Linked Lists

For some applications, especially those where it is necessary to traverse lists in both directions, doubly-linked lists work much better than singly-linked lists. The dlist class provides doubly-linked lists with the same interface as the generic lists. See Section 11.1 [Generic Lists], page 79. Thus you can interchange these data structures without having to rewrite your code. The dlist_e elements are derived from the generic list element class but they also include the backwards link fields. The dlist_iter iterator works like the standard glist_iter iterator, and the DECLARE_DLIST_CLASS macro is the same as the DECLARE_LIST_CLASS macro except that it produces doubly-linked lists. All of these things are implemented in the 'dlist.h' and 'dlist.cc' files.

11.5 Bit Vectors

Bit vectors are frequently used in compilers to represent sets of integers, particularly for data flow analysis. The SUIF library includes a `bit_set` class with an extensive collection of methods. This class is implemented in the '`bitset.h`' and '`bitset.cc`' files. When you create a new `bit_set`, you must specify the range of integers which it may contain.

The `bit_set_iter` class provides an easy and efficient way to iterate through the entries in a `bit_set`. This iterator is slightly different than the SUIF list iterators. You must call the `is_empty` method once before each call to the `step` method. Other than that, it is straightforward.

11.6 Hash Tables

SUIF hash tables are implemented by the `hash_table` class in the files '`hash.h`' and '`hash.cc`'. Each hash table is a fixed-size array of `hash_chain` buckets. A `hash_chain` is just a move-to-front list. See Section 11.2 [Move-to-front Lists], page 80. The entries in a hash table are derived from the `hash_e` class, which contains a single unsigned value used as the signature.

If you wish to store pointers in a hash table, you can use the `hash_e` class directly (assuming that the pointers can fit into the unsigned signature fields). Otherwise, you need to create a derived `hash_e` class. The only other thing needed is a function to compare two `hash_e` entries to determine if they are equal. You can then create a new `hash_table` object. Even if you are using a derived `hash_e` class, it may be easier to just type cast the pointers than to derive a new `hash_table` class that includes the correct type casts.

The hash table `enter` method tries to add a new entry. If the entry was already in the table it returns `TRUE` and does not enter a duplicate; otherwise, it adds the new entry and returns `FALSE`. The `lookup` method checks to see if a particular entry is in the table. If so, it returns the pointer to the entry.

11.7 Extensible Arrays

Arrays are very efficient data structures but they can only be used when their size is known in advance. For situations where the number of elements is not even known at run-time, most programmers resort to using linked lists. SUIF provides a compromise. Extensible arrays provide almost the same efficiency as arrays without requiring a bound on the number of elements. They are not as flexible as linked lists, however, because it is still not possible to insert new elements in the middle of an extensible array.

Extensible arrays are implemented by the `x_array` class in the files '`xarray.h`' and '`xarray.cc`'. When an `x_array` is first created, you must specify the number of elements to be allocated in the first chunk of memory. If more elements are needed later, additional chunks of the same size will be allocated automatically. The `ub` method returns the number of elements that are currently in an `x_array`.

The elements of an `x_array` cannot be referenced until they are added to the array. New elements can only be added at the end of an array by using the `extend` method. In this way an `x_array` behaves like a linked list. Elements must be appended (with `extend`) to the end of an array before they can be used.

The `x_array` class also provides an `operator[]` method so that once an element has been entered in an `x_array` it can be referenced using the standard array reference syntax. This works regardless of which chunk of memory contains the element.

Each element of an `x_array` is a `void*` pointer. You can cast these pointers to any other type that fits in the same amount of storage. A better solution is to use the `DECLARE_X_ARRAY` macro to derive a subclass of `x_array` with elements that are pointers to the correct type. This macro takes three arguments: the name of the derived class, the type to which the elements should point, and the default chunk size. As with the macros for creating new list classes, this is just a cheap substitute for templates.

12 Other Features of the SUIF Library

This chapter describes various features of the SUIF library that are too small to warrant separate chapters. These include immediate values and symbolic addresses, which are used in annotations and `ldc` instructions, the `lexicon`, in which almost all character strings are entered, the `machine_params` structure, which records a variety of parameters for the target architecture, and a variety of functions for initializing the library, handling errors, and iterating through procedures.

12.1 Immediate Values

An immediate value may be an integer constant, string constant, floating-point constant, SUIF type, symbolic address, expression operand, or instruction, or it can be undefined. It can be used as the source of an `ldc` (load constant) instruction (see Section 4.6 [Load Constant Instructions], page 33) or as an element of an annotation (see Chapter 8 [Annotations], page 63). Only integers, floating-point values, and symbolic addresses may be used in `ldc` instructions, while any kind of immediate value may appear in an annotation. Immediate values are implemented by the `immed` class defined in the files '`immed.h`' and '`immed.cc`'.

The kind of value in an `immed` is identified by the `kind` method. This returns a member of the `immed_kinds` enumerated type. The members of this enumeration are:

`im_int`
: Integer. The `is_integer` method tests for this kind of immediate and the `integer` method returns the integer value.

`im_string`
: Character string. The string is automatically entered in the lexicon (see Section 12.3 [Lexicon], page 84) when the `immed` is created. The `is_string` method checks if an `immed` is a string, and the `string` method returns a pointer to the string.

`im_float`
: Floating-point value. The SUIF library stores all floating-point values using the `double` type, so the precision is limited by the precision for the `double` type. The `is_flt` method checks for this kind of immediate, and the `flt` method returns the value.

`im_symbol`
: Symbolic address. See Section 12.2 [Symbolic Addresses], page 84. The `is_symbol` method checks if an `immed` is a symbolic address. The `addr` method returns the entire `sym_addr` object, but the `symbol` and `offset` methods are also available to retrieve the symbol pointer and integer offset separately.

`im_type`
: SUIF type. The `is_type` method checks for this kind of immediate, and the `type` method returns a pointer to the SUIF type node.

`im_op`
: Expression operand. The immed is considered to "own" the entire expression tree when it's in an annotation, so the same expression shouldn't be used in another annotation or as the source for an instruction; instead a copy should be made with the `clone` method. The `is_op` method checks for this kind of immediate, and the `op` method returns the expression.

`im_instr` SUIF instruction. The immed is considered to "own" the instruction and its expression trees when it's in an annotation, so the same instruction shouldn't be used in another annotation or in a `tree_node` or expression; instead a copy should be made with the `clone` method. The `is_instr` method checks for this kind of immediate, and the `instr` method returns the instruction.

`im_undef` Undefined value. This is included to signal errors by marking an immediate as undefined. The `is_error` method also checks for this condition.

Since an immediate value is small, there are no methods to change the value of an `immed`. Instead a variety of constructors are provided to make it easy to create new immediate values. That is, you can initialize different kinds of immediates by providing arguments of various types to constructors.

The `immed` class provides two print methods, `print` and `rawprint`. The difference between them is that the latter doesn't escape '`"`' and '\' when printing strings.

12.2 Symbolic Addresses

Symbolic addresses are currently only used in immediate values (see Section 12.1 [Immeds], page 83). A symbolic address has two parts, a symbol and an integer offset. In many cases, the offset is set to zero, but for an aggregate data structure the offset may specify the number of bits from the beginning of the data structure to a particular field. The `sym_addr` class defined in the '`symaddr.h`' and '`symaddr.cc`' files implements these symbolic addresses.

The `symbol` method returns a pointer to the symbol in a symbolic address, and the `offset` method returns the integer offset. As with most other offsets in SUIF, the offset is in *bits*. Be careful to avoid treating this as a byte or word offset. As with immediate values, there are no methods to change the contents of a symbolic address. Because these are small structures, it is just as easy to create new `sym_addr` objects whenever they are needed. To print a symbolic address, use the `print` method.

12.3 Lexicon

Almost all character strings in SUIF are entered in a hash table called the `lexicon`. This table removes duplicate strings so that string comparisons can be reduced to simple pointer comparisons. The `lexicon` is an object of the `string_table` class defined in the '`stringtable.h`' and '`stringtable.cc`' files.

A `string_table` is an open hash table; each bucket is a linked list of string entries represented by objects of the `string_e` class. The only public method in the `string_table` class is called `enter`. This method searches the table for a specified string. If the string is found, `enter` returns a pointer to the existing `string_e` entry. Otherwise, it creates a new entry and adds it to the table. In either case, the actual character string pointer can be retrieved directly from the `sp` field of the resulting `string_e` entry.

None of the strings in the `lexicon` are deallocated until a SUIF program terminates. Consequently, when adding a new string to the table, it is copied to the heap in case the original string is stored on the stack. If the string is known to be allocated on the heap and will never be modified or deallocated, this string copy can be avoided by setting the optional `fixed` flag when calling `enter`.

12.4 Target Machine Parameters

The SUIF library includes a structure to record various parameters of the target machine. The back-end of the compiler may be either a code generator for a specific processor or the system's C compiler, using the SUIF-to-C translator. Thus, the target machine parameters include fields that deal both with hardware requirements and with details of the back-end C compiler. These parameters are stored in the `target machine_params` structure. The front-end is responsible for setting the target parameters, but they are then saved in the SUIF files and passed along to all subsequent passes of the compiler.

The first fields in the `machine_params` structure deal with the addressing in the underlying hardware. The `is_big_endian` field specifies the byte order, and the `addressable_size` field specifies the size in bits of the smallest addressable unit. Since most machines are byte-addressable, the `addressable_size` is usually set to eight.

Next, the sizes and alignments for various C types are specified. The possible C types are listed in the `C_types` enumeration:

`C_char`

`C_short`

`C_int`

`C_long`

`C_longlong`

`C_float`

`C_double`

`C_longdouble`

`C_ptr`

The `size` and `align` fields of the `machine_params` structure are arrays containing the size and alignment for each C type. Both are specified in bits (not bytes). Some C compilers require additional alignment restrictions for arrays and structures. The `array_align` and `struct_align` fields contain these alignment requirements in bits. Note that these do not replace the alignment restrictions for the components of an array or structure; the most restrictive alignments must always be maintained.

The remaining fields are only applicable to the back-end C compiler. The `char_is_signed` field specifies if the default `char` type is signed. This is just to allow the SUIF-to-C translator to avoid cluttering up the C code by explicitly putting `signed` or `unsigned` in every declaration of a `char` type. When two pointers are subtracted in C, the result type is implementation defined. The `ptr_diff_type` field in `machine_params` specifies the type produced by such pointer subtractions for use in SUIF-to-C.

12.5 Initialization and Finalization

Before using any features of the SUIF library, a program must first initialize it. The `init_suif` function, defined in the '`initsuif.h`' and '`initsuif.cc`' files, performs this initialization. The `main` function of a SUIF program should call `init_suif` first. The `argc` and `argv` parameters are passed to `init_suif` so that it can check for the following standard options:

`-version` Print the version numbers and compilation information for the program and for any libraries with which it is linked. The `prog_ver_string` and `prog_who_string` strings contain the version number and compilation information for the program. They are usually set automatically by the standard SUIF makefiles. Similarly, the `libsuif1_ver_string` and `libsuif1_who_string` strings are used for the SUIF library. Information for other libraries is recorded when they are registered (as described below).

`-print-flat`
: Do not print structured annotations (see Section 8.3 [Structured Annotes], page 64) using the user-defined printing functions. If all annotations are printed as flat lists of immediate values, the output could be parsed and converted back to a binary SUIF file.

`-no-types`
: Do not print the result types for instructions. This is just to help make the output more readable in situations where the result types are not of interest.

`-no-symtabs`
: Do not print out symbol tables. If the symbolic information is not needed, this makes the output shorter and easier to read.

These options are potentially applicable to all SUIF programs, so they are included in the library.

At the end of a SUIF program, the data structures in the SUIF library may be deallocated. Although this is not absolutely necessary, it may simplify debugging by making it easier to read the output from Purify and other tools. Besides deallocating the data structures, it may someday be useful to perform other actions at the end of a SUIF program. The `exit_suif` function is provided to perform this finalization. If used, it should be called at the very end of the program.

Besides the base SUIF library, many SUIF programs are linked with other libraries that also need to be initialized. Rather than just initializing them directly, they are registered with the SUIF library along with initialization and finalization functions. The initialization functions are then called automatically by `init_suif` and the finalization functions by `exit_suif`. This allows the SUIF library to record and print version numbers and other information for all of the libraries linked with a program. The `register_library` function records the name, version, and compilation information for a library along with pointers to the functions to initialize and finalize that library. The initialization function must match the `lib_init_f` type, which takes the `argc` and `argv` parameters for the command line and returns `void`. Similarly, the finalization function must match the `lib_exit_f` type, which takes no arguments and returns `void`. The parameters for the name, version, and compilation information are required, but the initialization and finalization functions are optional. The libraries must be registered, in the order in which they should be initialized, before calling `init_suif`. The finalization functions are called in the reverse order in which the libraries were registered.

The situation is much simpler if you use the standard SUIF makefiles. These makefiles automatically generate strings holding the version and compilation information. The

LIBRARY macro may be used to register a library using these version strings. You only need to specify the base name of the library and the names of the initialization and finalization functions[1]. To make things even easier, the SUIF makefiles also generate a new function called start_suif. This function automatically registers all of the libraries linked with the program and then calls init_suif. So to make a long story short, if a SUIF program begins with a call to start_suif all of the libraries will be automatically registered and initialized.

12.6 Command-Line Parser

The SUIF library provides a generic interface for a command line parser. The parser is defined and implemented in the files 'cmdparse.h' and 'cmdparse.cc'. The SUIF program can provide a structure that contains a list of command line options, the type of arguments these options can handle, and pointers to data where the argument values will be stored. The data locations will be initialized to default values specified in the options table before parsing begins. The parse_cmd_line function is called with argc, argv, the options table, and the number of options in the table. If the parser finds one of the options on the command line, the option (and its argument, if any) are removed from argv and argc is appropriately adjusted. Thus when parsing is finished, argv contains only unrecognized options (such as file names).

The following types of arguments are supported:

CLO_NOARG
: No argument is expected. The default value is 0. If the argument is present, the value becomes 1.

CLO_INT
: A single integer is expected.

CLO_STRING
: A single string is expected.

CLO_MULTI_STRING
: A single string is expected for each occurrence of this option, but the option can be repeated with different arguments.

For example:

```
static boolean quiet;
static int size;
static cmd_line_option my_options[] = {
    { CLO_NOARG, "-quiet", "", &quiet },
    { CLO_INT, "-size", "8", &size }
};

parse_cmd_line(argc, argv, my_options,
               sizeof(my_options)/sizeof(cmd_line_option));
```

[1] Unlike when calling register_library directly, the function arguments are required when using the LIBRARY macro.

12.7 Error Handling

The SUIF library includes a number of functions for error handling. These functions handle three kinds of problems: errors, warnings, and assertions. Assertions are tests for things that should never occur in a correct program. Any problem that could potentially occur in a correct program due to bad input or other external conditions should be treated as an error rather than an assertion failure. Warnings are used to report potentially troublesome conditions that are not serious enough to cause a fatal error.

Two macros are available to implement assertions. The `assert` macro takes a single expression as an argument. If the expression evaluates to zero, a message is printed showing the current line number, file name, and the expression that failed, and then the `abort` function is called to terminate the program. The `assert_msg` macro is similar except that it also prints a user-defined message if the assertion fails. Besides the expression to be tested, `assert_msg` takes an argument that contains the message to be printed. This second argument must hold the `printf` format string and the arguments for the call to `printf`, separated by commas. To keep the commas from splitting up the second argument, it must be surrounded by parentheses. For example:

```
assert_msg(sym->parent(), ("no parent for %s", sym->name()));
```

Errors may be reported using the `error_line` function. The `return_code` parameter specifies the method of terminating the program. If the return code is negative, the `abort` function is called. If it is greater than zero, the `exit` function is called with the specified return code. If the return code is zero, the error message will be printed but the program will not terminate. The `the_node` parameter is optional and may be set to `NULL`. If used, it points to the AST node closest to the point where the error was detected, and the library tries to use it to find and print the corresponding source line number. Finally, the `fmt` parameter specifies the `printf` format string used to print the error message and any additional parameters are passed on to `printf`. The `verror_line` function is identical to `error_line` except that it passes the variable arguments for `printf` using a `varargs`-style list. This is just like the standard `vprintf` function.

The `warning_line` and `vwarning_line` functions are just like the corresponding error functions, except that they print warning messages and do not terminate the program. Consequently, they do not have `return_code` parameters.

12.8 Procedure Iterators

Most SUIF programs work on one procedure at a time. These programs require basically the same code to iterate through the file set entries and procedures. The `suif_proc_iter` routine is provided to handle this common case.

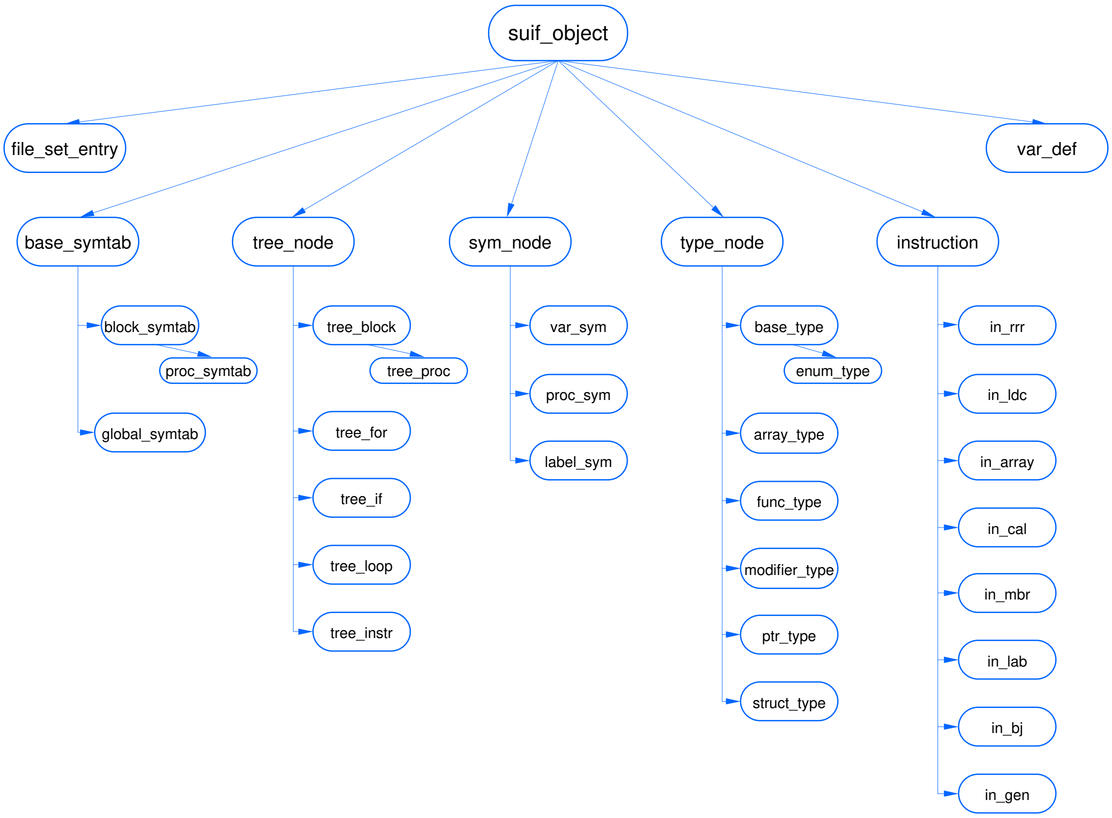
Before calling the procedure iterator, you must initialize the SUIF library (see Section 12.5 [Initialization], page 85), which also parses and removes some standard options from the command-line arguments. Either the SUIF library's command-line parser (see Section 12.6 [Command-Line Parser], page 87) or the `getopt` function (from the standard C library) may be called before the procedure iterator to parse command-line options not removed by the library initialization. The `suif_proc_iter` function then reads the input and output file names from the command-line. The first file name on the command line is treated as an input file. If the `writeback` flag is set, the second file is treated as an output file and any additional file names on the command line are alternately treated as input and output files. If the writeback flag is not set, all the file names are treated as input files.

The procedure bodies are then read one at a time and processed by a user-specified function. This function must be of the type prociter_f, which takes a single tree_proc argument and has a void return type. When a procedure is read, the exp_trees and use_fortran_form parameters are passed on to the read_proc method for the proc_sym. The exp_trees argument specifies whether the procedure bodies are converted to expression trees (see Section 4.3 [Expression Trees], page 28), and the use_fortran_form argument determines whether Fortran form is used (see Chapter 10 [Fortran], page 77). The defaults are to create expression trees but not to use Fortran form. If the writeback flag is set, the procedure bodies are written to the output files after they have been processed.

Appendix

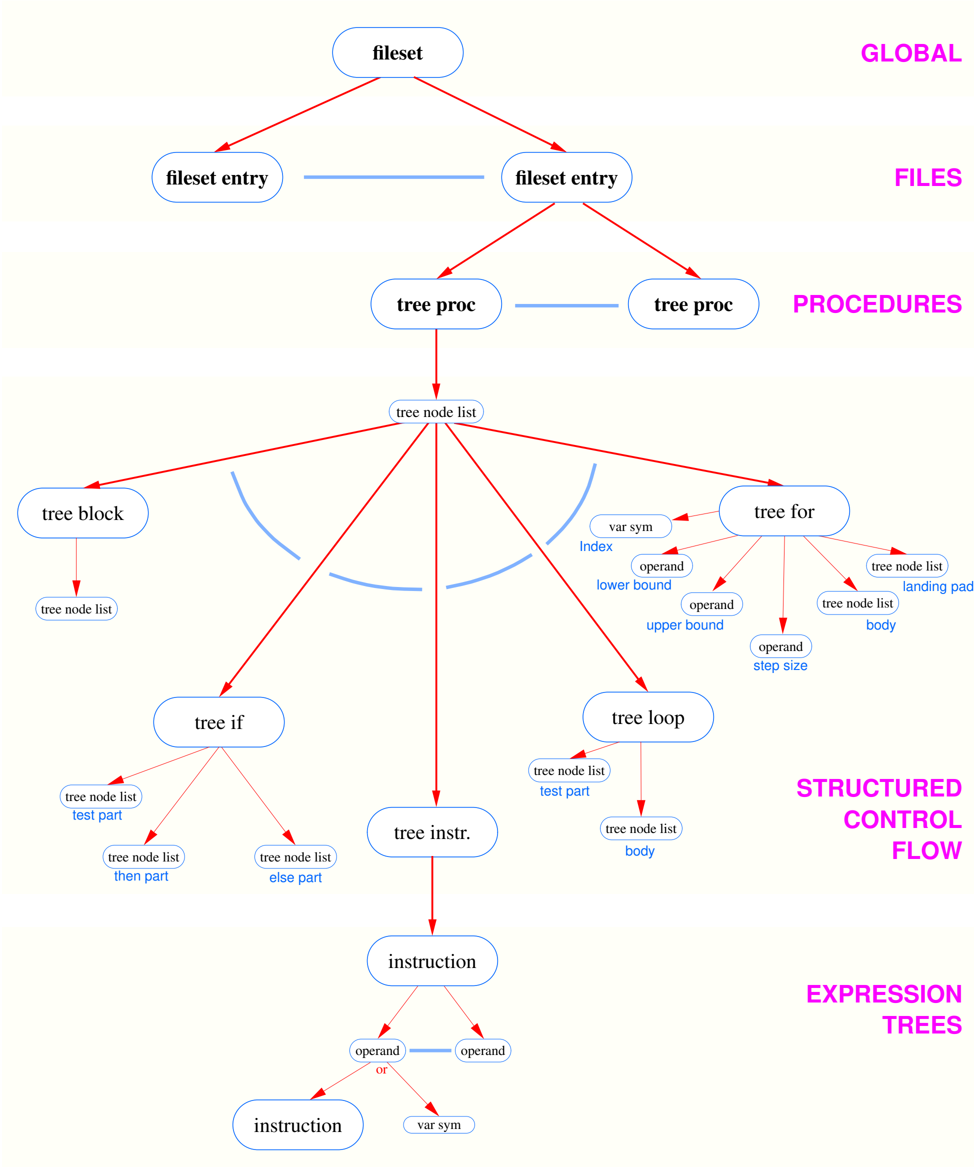
Class Inheritance of Major SUIF Structures

All major SUIF data structures are derived from the `suif_object` base class. See Section 8.7 [SUIF Objects], page 70. The class `file_set_entry` is defined in Section 2.2 [File Set Entries], page 8, the class `tree_node` and its derived classes are explained in Section 3.1 [Tree Nodes], page 11, the class `sym_node` and its derived classes are described in Chapter 6 [Symbols], page 45, the class `type_node` and its derived classes are defined in Chapter 7 [Types], page 53, the class `instruction` and its derived classes are defined in Chapter 4 [Instructions], page 23 and the class `var_def` is defined in Section 6.4 [Variable Symbols], page 47.



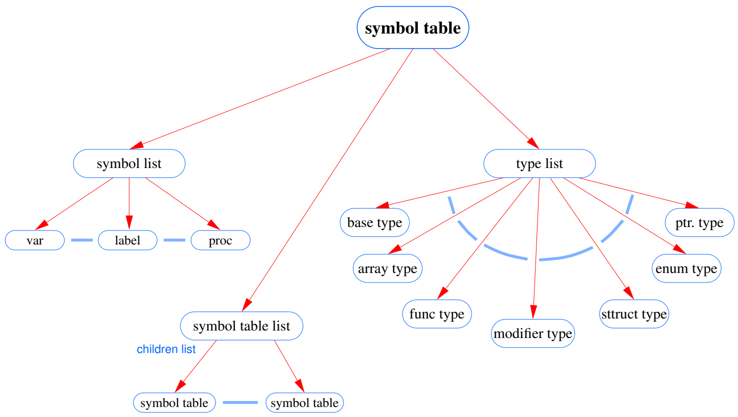
Overview of the SUIF Hierarchy

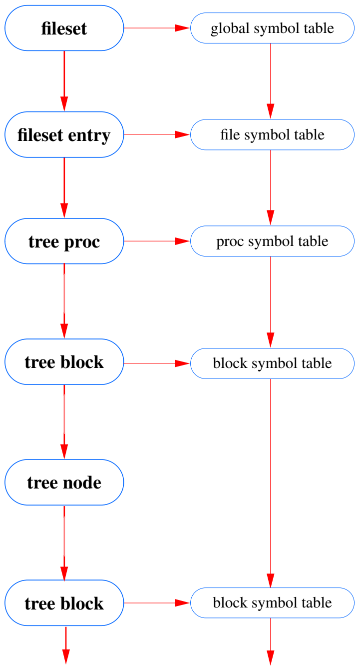
This diagram represents the overall hierarchical structure of the SUIF intermediate representation. At the top level, the global symbol information is kept. See Section 5.1.1 [Global Symbol Table], page 37. In the next level, information regarding each input and output file is kept. See Chapter 2 [File Representation], page 7. The third level is the abstract syntax tree representation. See Chapter 3 [Trees], page 11. The bottom level contains the expression trees. See Chapter 4 [Instructions], page 23.



The Symbol Table Representation

The first diagram below illustrates the symbol table structure. The second shows the hierarchy of symbol tables. See Section 5.1 [Symbol Table Hierarchy], page 37 for more details.





Function, Variable, and Type Index

A

alist ... 80
alist_e ... 80
alist_iter ... 80
amtflist ... 80
annote ... 63
ANNOTE ... 68
annote, data ... 63
annote, immeds ... 63
annote, name ... 63
annote, print ... 63
annote, set_data ... 63
annote, set_immeds ... 63
annote, set_name ... 63
annote_def ... 68
annote_def, is_structured ... 68
annote_def, name ... 68
annote_def, output ... 68
annote_def, set_output ... 68
annote_list ... 71
annote_list, get_annote ... 71
annote_list, peek_annote ... 71
array_bound ... 56
array_bound, constant ... 56
array_bound, is_constant ... 56
array_bound, is_unknown ... 56
array_bound, is_variable ... 56
array_bound, print ... 56
array_bound, variable ... 56
array_type ... 56
array_type, are_bounds_unknown ... 56
array_type, elem_type ... 56
array_type, lower_bound ... 56
array_type, set_elem_type ... 56
array_type, set_lower_bound ... 56
array_type, set_upper_bound ... 56
array_type, upper_bound ... 56
assert ... 88
assert_msg ... 88

B

base_symtab ... 37
base_symtab, add_child ... 42
base_symtab, add_def ... 42
base_symtab, add_sym ... 42
base_symtab, add_type ... 42
base_symtab, chain_name ... 39
base_symtab, children ... 37
base_symtab, define_var ... 41
base_symtab, install_type ... 42
base_symtab, is_ancestor ... 37
base_symtab, lookup_child ... 41
base_symtab, lookup_sym ... 40
base_symtab, lookup_sym_id ... 40
base_symtab, lookup_type ... 40
base_symtab, lookup_type_id ... 40
base_symtab, lookup_var ... 40
base_symtab, lookup_var_def ... 40
base_symtab, name ... 39
base_symtab, new_unique_var ... 41
base_symtab, new_var ... 41
base_symtab, parent ... 37
base_symtab, remove_child ... 42
base_symtab, remove_def ... 42
base_symtab, remove_sym ... 42
base_symtab, remove_type ... 42
base_symtab, rename_duplicates ... 40
base_symtab, resolve_exposed_refs ... 74
base_symtab, set_name ... 39
base_symtab, symbols ... 39
base_symtab, types ... 39
base_symtab, var_defs ... 39
base_type ... 55
base_type, is_signed ... 55
base_type, set_signed ... 55
base_type, set_size ... 55
bit_set ... 81
bit_set_iter ... 81
block_symtab ... 39
block_symtab, block ... 39
block_symtab, clone_helper ... 74
block_symtab, find_exposed_refs ... 73
block_symtab, lookup_label ... 40
block_symtab, new_label ... 41
block_symtab, new_unique_child ... 41
block_symtab, new_unique_label ... 41

C

C_types ... 85
cmd_line_option ... 87
cmd_line_option_kind ... 87

D

DECLARE_DLIST_CLASS 80
DECLARE_LIST_CLASS 79
DECLARE_MTFLIST_CLASS 80
DECLARE_X_ARRAY 82
dlist 80
dlist_e 80
dlist_iter 80

E

enum_type 59
enum_type, find_member_by_name 59
enum_type, find_member_by_value 59
enum_type, member 59
enum_type, name 59
enum_type, num_values 59
enum_type, set_member 59
enum_type, set_name 59
enum_type, set_num_values 59
enum_type, set_value 59
enum_type, value 59
error_line 88
exit_suif 86

F

file_set 7
file_set, add_file 7
file_set, file_list 7
file_set, globals 7
file_set, next_file 7
file_set, reset_iter 7
file_set_entry 8
file_set_entry, name 8
file_set_entry, next_proc 8
file_set_entry, parent 8
file_set_entry, reset_proc_iter 8
file_set_entry, symtab 8
file_set_entry_list 8
file_symtab 38
file_symtab, fse 38
file_symtab, number_globals 43
fileset 7
func_type 57
func_type, arg_type 57
func_type, args_known 58
func_type, has_varargs 58
func_type, num_args 57
func_type, return_type 57
func_type, set_arg_type 57
func_type, set_args_known 58
func_type, set_args_unknown 58
func_type, set_num_args 57
func_type, set_return_type 57
func_type, set_varargs 58

G

glist 79
glist_e 79
glist_iter 79
global_symtab 38
global_symtab, lookup_proc 40
global_symtab, new_proc 41

H

hash_chain 81
hash_e 81
hash_table 81
hash_table, enter 81
hash_table, lookup 81

I

if_ops 23
if_ops_name 23
immed 83
immed, addr 83
immed, flt 83
immed, instr 84
immed, integer 83
immed, is_error 84
immed, is_flt 83
immed, is_instr 84
immed, is_integer 83
immed, is_op 83
immed, is_string 83
immed, is_symbol 83
immed, is_type 83
immed, offset 83
immed, op 83
immed, print 84
immed, rawprint 84

immed, string 83
immed, symbol 83
immed, type 83
immed_kinds 83
in_array 34
in_array, base_op 34
in_array, bound 34
in_array, dims 34
in_array, elem_size 34
in_array, elem_type 34
in_array, index 34
in_array, offset 34
in_array, offset_op 34
in_array, set_base_op 34
in_array, set_bound 34
in_array, set_dims 34
in_array, set_elem_size 34
in_array, set_index 34
in_array, set_offset 34
in_array, set_offset_op 34
in_bj 32
in_bj, set_src_op 32
in_bj, set_target 32
in_bj, src_op 32
in_bj, target 32
in_cal 33
in_cal, addr_op 33
in_cal, argument 33
in_cal, num_args 33
in_cal, set_addr_op 33
in_cal, set_argument 33
in_cal, set_num_args 33
in_gen 35
in_gen, name 35
in_gen, set_name 35
in_gen, set_num_srcs 36
in_lab 35
in_lab, label 35
in_lab, set_label 35
in_ldc 33
in_ldc, set_value 33
in_ldc, value 33
in_mbr 35
in_mbr, default_lab 35
in_mbr, label 35
in_mbr, lower 35
in_mbr, num_labs 35
in_mbr, set_default_lab 35
in_mbr, set_label 35
in_mbr, set_lower 35
in_mbr, set_num_labs 35
in_mbr, set_src 35
in_mbr, src_op 35
in_rrr 29
in_rrr, dst_addr_op 29
in_rrr, is_commutative 29
in_rrr, is_unary 29
in_rrr, set_dst_addr_op 29
in_rrr, set_shift_cnt_op 29
in_rrr, set_src 29
in_rrr, set_src_addr_op 29
in_rrr, set_src1 29
in_rrr, set_src2 29
in_rrr, shift_cnt_op 29
in_rrr, src_addr_op 29
in_rrr, src_op 29
in_rrr, src1_op 29
in_rrr, src2_op 29
init_aman 68
init_suif 85
inst_format 23
instr_map_f 13
instruction 23
instruction, clear_numbers 12
instruction, clone 73
instruction, clone_helper 74
instruction, dst_op 25
instruction, find_exposed_refs 73
instruction, format 23
instruction, num_srcs 26
instruction, opcode 23
instruction, owner 26
instruction, parent 25
instruction, print 27
instruction, remove 26
instruction, result_type 25
instruction, set_dst 25
instruction, set_opcode 23
instruction, set_result_type 25
instruction, set_src_op 26
instruction, src_map 26
instruction, src_op 26
instruction_list 23

K

k_call_by_ref 66
k_common_block 67
k_enable_exceptions 67
k_fields 67
k_fill 65
k_history 67
k_line 67
k_multi_init 66
k_orig_type 66
k_reg_num 67
k_repeat_init 65

L

label_sym 47
lexicon 84
lib_exit_f 86
lib_init_f 86
LIBRARY 86
lookup_annote 68

M

machine_params 85
make_ref_params 77
modifier_type 59
modifier_type, base 59
modifier_type, set_base 59
mtflist 80

O

object_kinds 70
operand 27
operand, clone 73
operand, clone_helper 74
operand, find_exposed_refs 73
operand, instr 27
operand, is_const_int 27
operand, is_expr 27
operand, is_instr 27
operand, is_null 27
operand, is_symbol 27
operand, print 27
operand, print_source 27
operand, remove 26
operand, set_instr 27
operand, set_symbol 27
operand, symbol 27
operand, type 27

P

parse_cmd_line 87
print_id_number 42
proc_sym 46
proc_sym, block 46
proc_sym, file 46
proc_sym, flush_proc 47
proc_sym, is_in_memory 46
proc_sym, is_readable 46
proc_sym, is_written 46
proc_sym, read_proc 46
proc_sym, set_block 46
proc_sym, set_src_lang 46
proc_sym, set_type 46
proc_sym, src_lang 46
proc_sym, type 46
proc_sym, write_proc 47
proc_symtab 38
proc_symtab, clone_helper 74
proc_symtab, instr_num 11
proc_symtab, next_instr_num 11
proc_symtab, number_locals 43
proc_symtab, params 38
prociter_f 89
ptr_type 56
ptr_type, ref_type 56
ptr_type, set_ref_type 56

R

register_annote 68
register_library 86
replacements 73

S

src_lang_type 46
src_map_f 26
start_suif 86
string_e 84
string_table 84
string_table, enter 84
STRUCT_ANNOTE 69
struct_annote_def 69
struct_annote_def, free 69
struct_annote_def, from 69
struct_annote_def, print 69
struct_annote_def, to 69
struct_type 58
struct_type, field_name 58
struct_type, field_type 58
struct_type, find_field_by_name 58
struct_type, find_field_by_offset 58
struct_type, name 58
struct_type, num_fields 58
struct_type, offset 58
struct_type, set_field_name 58
struct_type, set_field_type 58
struct_type, set_name 58
struct_type, set_num_fields 58
struct_type, set_offset 58
struct_type, set_size 58
suif_object 70
suif_object, annotes 70
suif_object, append_annote 70
suif_object, are_annotations 70
suif_object, clone_annotes 74
suif_object, copy_annotes 71
suif_object, find_annote_refs 74
suif_object, get_annote 71
suif_object, num_output_annotes 71
suif_object, object_kind 70
suif_object, peek_annote 71
suif_object, prepend_annote 70
suif_object, print_annotes 71
suif_object, set_annote 70
suif_proc_iter 88
sym_addr 84
sym_addr, offset 84
sym_addr, print 84
sym_addr, symbol 84
sym_kinds 45
sym_node 45
sym_node, add_to_table 46
sym_node, clear_sym_id 43
sym_node, copy 46
sym_node, is_extern 45
sym_node, is_global 45
sym_node, is_private 45
sym_node, is_userdef 45
sym_node, kind 45
sym_node, name 45
sym_node, parent 45
sym_node, print 46
sym_node, print_full 46
sym_node, remove_from_table 46
sym_node, reset_userdef 45
sym_node, set_name 45
sym_node, set_userdef 45
sym_node, sym_id 42
sym_node_list 45

T

target 85
tree_block 19
tree_block, body 19
tree_block, clone 73
tree_block, set_body 19
tree_block, set_symtab 19
tree_block, symtab 19
tree_for 16
tree_for, body 18
tree_for, brklab 18
tree_for, clone 73
tree_for, contlab 18
tree_for, index 16
tree_for, landing_pad 18
tree_for, lb_list 17
tree_for, lb_op 17
tree_for, set_body 18
tree_for, set_brklab 18
tree_for, set_contlab 18
tree_for, set_index 16
tree_for, set_landing_pad 18
tree_for, set_lb_list 17
tree_for, set_lb_op 17
tree_for, set_step_list 17
tree_for, set_step_op 17
tree_for, set_test 17

tree_for, set_ub_list 17
tree_for, set_ub_op 17
tree_for, step_list 17
tree_for, step_op 17
tree_for, test 17
tree_for, ub_list 17
tree_for, ub_op 17
tree_for_test 17
tree_if 13
tree_if, clone 73
tree_if, else_part 13
tree_if, header 13
tree_if, jumpto 13
tree_if, set_else_part 13
tree_if, set_header 13
tree_if, set_jumpto 13
tree_if, set_then_part 13
tree_if, then_part 13
tree_instr 13
tree_instr, clone 73
tree_instr, instr 13
tree_instr, instr_map 13
tree_instr, remove_instr 13
tree_instr, set_instr 13
tree_loop 15
tree_loop, body 15
tree_loop, brklab 15
tree_loop, clone 73
tree_loop, contlab 15
tree_loop, set_body 15
tree_loop, set_brklab 15
tree_loop, set_contlab 15
tree_loop, set_test 15
tree_loop, set_toplab 15
tree_loop, test 15
tree_loop, toplab 15
tree_node 11
tree_node, clear_numbers 12
tree_node, clone 73
tree_node, clone_helper 74
tree_node, find_exposed_refs 73
tree_node, list_e 12
tree_node, map 21
tree_node, number 11
tree_node, parent 12
tree_node, print 12
tree_node, proc 12
tree_node, scope 12
tree_node_list 20
tree_node_list, clear_numbers 12
tree_node_list, clone 73
tree_node_list, clone_helper 74
tree_node_list, cvt_to_trees 28
tree_node_list, find_exposed_refs 73
tree_node_list, flatten 28
tree_node_list, is_op 20
tree_node_list, map 21
tree_node_list, op 20
tree_node_list, op_is_intconst 20
tree_node_list, parent 20
tree_node_list, print 20
tree_node_list, print_expr 20
tree_node_list, set_elem 20
tree_node_list, set_op 20
tree_proc 20
tree_proc, clone 73
tree_proc, number_instrs 11
tree_proc, proc_syms 20
type_char 61
type_double 61
type_error 60
type_f128 60
type_f32 60
type_f64 60
type_float 61
type_longdouble 61
type_node 53
type_node, clear_type_id 43
type_node, compatible 55
type_node, copy 54
type_node, find_modifier 60
type_node, is_call_by_ref 60
type_node, is_const 60
type_node, is_named 54
type_node, is_same 54
type_node, is_scalar 53
type_node, is_volatile 60
type_node, op 53
type_node, parent 54
type_node, print 55
type_node, print_abbrev 55
type_node, print_full 55
type_node, size 54
type_node, type_id 42
type_node, unqual 60
type_node_list 53

type_ops 53
type_ptr 61
type_s16 60
type_s32 60
type_s64 60
type_s8 60
type_signed 61
type_signed_char 61
type_signed_long 61
type_signed_longlong 61
type_signed_short 61
type_u16 60
type_u32 60
type_u64 60
type_u8 60
type_unsigned 61
type_unsigned_char 61
type_unsigned_long 61
type_unsigned_longlong 61
type_unsigned_short 61
type_v0 60
type_void 60

U

undo_ref_params 77
unknown_bound 56

V

var_def 51
var_def, alignment 51
var_def, print 51
var_def, set_alignment 51
var_def, set_variable 51
var_def, variable 51
var_sym 47
var_sym, add_child 50
var_sym, build_child 50
var_sym, child_var 50
var_sym, definition 48
var_sym, find_child 50
var_sym, has_var_def 48
var_sym, is_addr_taken 48
var_sym, is_auto 48
var_sym, is_param 48
var_sym, is_reg 48
var_sym, is_scalar 49
var_sym, is_spilled 49
var_sym, is_static 48
var_sym, num_children 50
var_sym, offset 50
var_sym, overlaps 50
var_sym, parent_var 50
var_sym, remove_child 50
var_sym, reset_addr_taken 48
var_sym, reset_param 48
var_sym, reset_reg 48
var_sym, root_ancestor 50
var_sym, root_offset 50
var_sym, set_addr_taken 48
var_sym, set_param 48
var_sym, set_reg 48
var_sym, set_type 48
var_sym, type 48
verror_line 88
vwarning_line 88

W

warning_line 88
which_format 23

X

x_array 81
x_array, extend 81
x_array, operator[] 81
x_array, ub 81

Concept Index

A

abstract syntax trees 11
addresses, symbolic 84
annotation lists 71
annotation manager 68
annotations 63
annotations, call-by-reference 66
annotations, common block 67
annotations, flat 64
annotations, initial data 65
annotations, predefined 65
annotations, structured 64
annotations, unregistered 65
annotes 63
arguments, command-line 87
array instructions 34
array types 56
assertions 88
association lists 80
AST nodes 11
ASTs 11

B

base types 55
bit sets 81
bit vectors 81
block nodes 19
block symbol tables 39
branch and jump instructions 32

C

call instructions 33
call-by-reference annotations 66
call-by-reference parameters 77
cloning 73
command-line arguments 87
common block annotations 67

D

data structures, generic 79
destination operands 25
doubly-linked lists 80
duplication 73

E

enumerated types 59
errors 88
exception handling 67
expression trees 28
extensible arrays 81

F

fields annotations 67
file set 7
file set entries 8
file symbol tables 38
flat annotations 64
for nodes 16
formal parameters 38
Fortran form 77
function types 57

G

generic data structures 79
generic instructions 35
generic lists 79
global symbol table 37

H

hash tables 81
high-SUIF 2

I

ID numbers 11
ID numbers, instruction 11
ID numbers, symbol 42
ID numbers, tree node 11
ID numbers, type 42
if nodes 13
immediate values 83
initial data annotations 65
initialization 85
instruction formats 23
instruction ID numbers 11
instruction nodes 13
instructions 23
instructions, array 34

instructions, basic features 23
instructions, branch and jump 32
instructions, call 33
instructions, generic 35
instructions, label 35
instructions, load constant 33
instructions, multi-way branch 35
instructions, printing 27
instructions, three operand 29
introduction 1
iterators, bit set 81
iterators, list 79
iterators, procedure 88

L

label instructions 35
label symbols 47
landing pads 18
lexicon 84
line numbers 67
linked lists 79
lists, association 80
lists, doubly-linked 80
lists, generic 79
lists, move-to-front 80
load constant instructions 33
loop nodes 15
low-SUIF 2

M

machine registers 48
manager, annotations 68
mapping instructions 13
mapping subtrees 21
modifier types 59
move-to-front lists 80
multi-way branch instructions 35

N

nested scopes 19

O

objects, SUIF 70
opcodes 23
operands 27
operands, destination 25
operands, source 26
options, standard 85
overview 1

P

parameters, formal 38
pointer types 56
predefined annotations 65
predefined types 60
procedure iterators 88
procedure nodes 20
procedure symbol tables 38
procedure symbols 46

R

register numbers 67
registers, machine 48

S

slaving sub-variable properties 50
source operands 26
standard options 85
string table 84
structure types 58
structured annotations 64
structured annotations, conversion 69
structured annotations, deallocation 69
structured annotations, printing 69
sub-variables 49
SUIF objects 70
symbol ID numbers 42
symbol table chain name 39
symbol table name 39
symbol table, global 37
symbol tables 37
symbol tables, adding entries 41
symbol tables, basic features 39
symbol tables, block 39
symbol tables, file 38

symbol tables, lookup methods . . . 40
symbol tables, procedure . . . 38
symbol tables, removing entries . . . 41
symbolic addresses . . . 84
symbols . . . 45
symbols, label . . . 47
symbols, procedure . . . 46
symbols, variable . . . 47

T

target machine parameters . . . 85
three operand instructions . . . 29
tree blocks . . . 19
tree fors . . . 16
tree ifs . . . 13
tree instructions . . . 13
tree loops . . . 15
tree node ID numbers . . . 11
tree node lists . . . 20
tree nodes . . . 11
tree procedures . . . 20
trees . . . 11
type ID numbers . . . 42
types . . . 53
types, array . . . 56
types, base . . . 55
types, enumerated . . . 59
types, function . . . 57
types, modifiers . . . 59
types, pointer . . . 56
types, predefined . . . 60
types, structure . . . 58
types, union . . . 58

U

union types . . . 58
unregistered annotations . . . 65

V

variable definitions . . . 51
variable symbols . . . 47
variables, initialization . . . 65

W

warnings . . . 88

Table of Contents

1 An Overview of SUIF 1

1.1 The File Level 1
1.2 The Procedure Level 2
1.3 The Instruction Level 3
1.4 Symbolic Information 4
1.5 Other Data 5

2 Representation of Source Files 7

2.1 The File Set 7
2.2 File Set Entries 8

3 Abstract Syntax Trees 11

3.1 Tree Nodes 11
3.1.1 ID Numbers for Tree Nodes and Instructions 11
3.1.2 Other Features Shared by All Tree Nodes 12
3.1.3 Instruction Nodes 13
3.1.4 If Nodes 13
3.1.5 Loop Nodes 15
3.1.6 For Nodes 16
3.1.7 Block Nodes 19
3.1.8 Procedure Nodes 20
3.2 Tree Node Lists 20
3.3 Mapping Functions Over Subtrees 21

4 Instructions and Expression Trees 23

4.1 Basic Instruction Features 23
4.1.1 Opcodes and Formats 23
4.1.2 Destination Operands 25
4.1.3 Result Types 25
4.1.4 Parent Tree Nodes 25
4.1.5 Source Operands 26
4.1.6 Printing Methods 27
4.2 Operands 27
4.3 Expression Trees 28
4.4 Three Operand Instructions 29
4.5 Branch and Jump Instructions 32
4.6 Load Constant Instructions 33
4.7 Call Instructions 33
4.8 Array Instructions 34
4.9 Multi-way Branch Instructions 35
4.10 Label Instructions 35
4.11 Generic Instructions 35

5 Symbol Tables 37

5.1 Symbol Table Hierarchy 37

5.1.1 The Global Symbol Table 37

5.1.2 File Symbol Tables 38

5.1.3 Procedure Symbol Tables 38

5.1.4 Block Symbol Tables 39

5.2 Basic Features 39

5.3 Lookup Methods 40

5.4 Creating New Entries 41

5.5 Adding and Removing Entries 41

5.6 Numbering Types and Symbols 42

6 Symbols 45

6.1 Basic Features of Symbols 45

6.2 Procedure Symbols 46

6.3 Label Symbols 47

6.4 Variable Symbols 47

6.4.1 Basic Features of Variables 48

6.4.2 Sub-Variables 49

6.4.3 Variable Definitions 51

7 Types 53

7.1 Features Shared By All Types 54

7.2 Base Types 55

7.3 Pointer Types 56

7.4 Array Types 56

7.5 Function Types 57

7.6 Structure and Union Types 58

7.7 Enumerated Types 59

7.8 Modifier Types 59

7.9 Predefined Types 60

8 Annotations 63

8.1 Overview 63

8.2 Flat Annotations 64

8.3 Structured Annotations 64

8.4 Unregistered Annotations 65

8.5 Predefined Annotations 65

8.5.1 Initial Data Annotations 65

8.5.2 Call-By-Reference Annotations 66

8.5.3 Common Block Annotations 67

8.5.4 Miscellaneous Annotations 67

8.6 Annotation Manager 68

8.6.1 Flat Annotation Definitions 68

8.6.2 Structured Annotation Definitions 69

8.6.3 Unregistered Annotations in the Input File 70

8.7 SUIF Objects 70

8.8 Annotation Lists 71

9 Replicating Trees and Instructions 73

9.1 Finding Exposed References 73
9.2 Resolving Exposed References 74
9.3 Copying the Objects 74

10 Features for Compiling Fortran 77

10.1 Call-By-Reference Parameters 77

11 Generic Data Structures 79

11.1 Generic Lists 79
11.2 Move-to-front Lists 80
11.3 Association Lists 80
11.4 Doubly-Linked Lists 80
11.5 Bit Vectors 81
11.6 Hash Tables 81
11.7 Extensible Arrays 81

12 Other Features of the SUIF Library 83

12.1 Immediate Values 83
12.2 Symbolic Addresses 84
12.3 Lexicon 84
12.4 Target Machine Parameters 85
12.5 Initialization and Finalization 85
12.6 Command-Line Parser 87
12.7 Error Handling 88
12.8 Procedure Iterators 88

Appendix 91

Class Inheritance of Major SUIF Structures 91
Overview of the SUIF Hierarchy 92
The Symbol Table Representation 93

Function, Variable, and Type Index 95

Concept Index 103