The Machine-SUIF Cookbook

Release version 2.02.07.15

Glenn Holloway and Michael D. Smith
{holloway,smith}@eecs.harvard.edu
Division of Engineering and Applied Sciences
Harvard University

July 15, 2002

Abstract

This cookbook contains several complete (runnable) examples of a variety of simple Machine-SUIF compiler passes. Along with the *OPI User's Guide*, these examples should help you to start writing Machine-SUIF optimization passes quickly. We also provide a target library for a very simple machine. Please feel free to use these examples as the starting point for your work.

Contents

1 Introduction

As stated in our overview document, Machine SUIF is a flexible, extensible, and easily-understood infrastructure for constructing compiler back ends. We built Machine SUIF with the philosophy that the "textbook" portions, i.e. the optimization and analysis passes, should be coded in a way that makes them as independent of the compiler environment and compilation targets as possible. To achieve this goal, we have defined an interface layer that we refer to as the *Optimization Programming Interface* (OPI). All of the examples in this cookbook adhere to this interface.

This cookbook leads you through a number of complete and working Machine-SUIF passes. We start with an extremely simple pass, the equivalent of the C "Hello World" program. This example illustrates the structure of every Machine-SUIF pass. We build from there adding a few new OPI features in every example. By the time you have completed the last example, you should have enough knowledge to understand one of the "real" analysis or optimization passes distributed with Machine SUIF. If you are looking for one of our simpler "real" passes to investigate next, we encourage you to read through the `peep` pass. It serves as an good example of how to use our control-flow-graph library and our bit-vector dataflow classes.

If you think that you have a "cookbook" example pass that illustrates a particularly tricky concept, please send it to us. We'll be happy to include it in this cookbook for the benefit of others.

2 Ex1 – A Simple Pass

The first example in our cookbook simply reads in an Machine-SUIF intermediate representation (IR) file and prints the name of each procedure in this file. The main purpose of this example is to illustrate the source file structure of a Machine-SUIF pass and the nesting of an OPI-pass object within a SUIF-pass object.

File structure. There are five source files in the directory `ex1`, not including the `Makefile`. The files `ex1.h` and `ex1.cpp` define the OPI pass. Except for the SUIF copyright include, these two files are what we call "substrate independent"; they do not make any explicit references to SUIF data structures or functions.

To build a pass that will run under SUIF, we add the three source files starting with the prefix `suif_`. The file `suif_main.cpp` is used to build the stand-alone program called `do_ex1`. To adapt this source file for your pass, use a text editor to change all instances of `ex1` to the name of your pass.

The files `suif_pass.h` and `suif_pass.cpp` define the SUIF pass built as the dynamically loadable library `libex1.so`. Adapting these files to your purpose again requires changing `ex1` to your pass name throughout. When copying the file `ex1/suif_pass.cpp`, you may also need to change the list of include files and the list of `init_<library>` calls at the end of `init_ex1`. Our `ex1` pass requires only the Machine-SUIF `machine` library. A pass like `peep` requires this library along with the Machine-SUIF `cfg` and `bvd` libraries. In summary, `suif_pass.cpp` should include and initialize the full set of libraries required by your pass. (Don't forget to list these same libraries on the `LIBS` line in the `Makefile`.)

For further information about the contents of the SUIF files, please see Appendix A in our *OPI User's Guide*.

Inputs and outputs. In addition to whatever command line options you define, the command line of a Machine-SUIF pass may end with 1 or 2 file names. The processing of a pass's inputs and outputs adheres to the following rules:

1. If you do not specify any file names on the command line, the pass assumes that this is an in-memory pass and that it will receive an already-constructed `file_set_block` containing the input files.
2. If you specify only a single file name on the command line, the pass uses the existence or absence of a `file_set_block` to determine if the file represents an input or output.
   - If the pass receives an already-constructed `file_set_block`, then the file name on the command line is used as the output file.
   - If no `file_set_block` exists when this pass starts, the file name on the command line is assumed to be an input file, and the output IR file is left in a newly-created `file_set_block`. This option is useful when your pass starts a string of in-memory optimization passes, or when the pass simply analyzes an IR file and does not change the IR in any way. This is what `do_ex1` does.
3. If you specify two file names on the command line, the pass assumes that the first file name is the input file and the second is where the optimized IR is to be written when the pass completes.

If you want your pass to process more than one input file in a single command, you must use the scripting feature in SUIF to `load` multiple files into a single `file_set_block` and then run your pass in in-memory mode. Please note that SUIF does not allow you to specify multiple output files. When you `save` the `file_set_block` containing multiple files, you can specify only a single file name.

With respect to our example pass `ex1`, once you have compiled `ex1` to create `do_ex1`, you can run this pass on any Machine-SUIF IR file by typing, for example:

```
do_ex1 wc.svm
```

Here `wc.svm` is the output of `do_s2m` for the UNIX benchmark `wc`. Please remember to setup your environment variables as described in the Machine-SUIF overview document before compiling or running any Machine-SUIF pass.

The OPI pass. Given that `ex1` is such a simple pass, the declaration of class `Ex1` is nothing more than the OPI pass interface described in our *OPI User's Guide*.

4 ⟨*class* `Ex1` 4⟩≡

```
// This defines the substrate-independent OPI class.
class Ex1 {
  public:
    Ex1() { }

    void initialize() { printf("Running initialize()\n"); }
    void do_opt_unit(OptUnit*);
    void finalize() { printf("Running finalize()\n"); }
};
```

All of the work of the optimization pass occurs in do_opt_unit. Under Machine SUIF, an OptUnit is a SUIF ProcedureDefinition. Thus, we just grab the name of the procedure from the procedure symbol and print it.

5a ⟨Ex1::do_opt_unit 5a⟩≡
```
void
Ex1::do_opt_unit(OptUnit *unit)
{
    IdString name = get_name(get_proc_sym(unit));
    printf("Processing procedure \"%s\"\n", name.chars());
}
```

3 Ex2 – Analyzing an Instruction List

Now that you know how to build and run a Machine-SUIF pass, we will now begin to focus on the interesting stuff: what you can do in do_opt_unit. All of the files in this example are identical to those in ex1[1] except that we've added some more code to do_opt_unit.

5b ⟨Ex2::do_opt_unit 5b⟩≡
```
void
Ex2::do_opt_unit(OptUnit *unit)
{
    IdString name = get_name(get_proc_sym(unit));
    printf("Processing procedure \"%s\"\n", name.chars());

    // get the body of the OptUnit
    AnyBody *body = get_body(unit);

    // verify that it is an InstrList
    claim (is_a<InstrList>(body),
           "expected OptUnit's body in InstrList form");
    InstrList *mil = (InstrList *)body;

    printf(" ... has %d total instructions\n", size(mil));

    // loop through all instructions counting some different kinds
    int tot_cti = 0, tot_dot = 0, tot_label = 0, tot_other = 0;
    for (InstrHandle mi_h = start(mil); mi_h != end(mil); ++mi_h) {
        Instr *mi = *mi_h;

        if (is_cti(mi))          // control-transfer instructions
            tot_cti++;
        else if (is_dot(mi))     // assember directives
            tot_dot++;
        else if (is_label(mi))   // labels
            tot_label++;
        else
            tot_other++;
    }

    printf(" ...     %d control-transfer instructions\n", tot_cti);
    printf(" ...     %d assembler directives\n", tot_dot);
    printf(" ...     %d code labels\n", tot_label);
```

[1] We have, of course, changed all strings of the form ex1 to ex2.

```
    printf(" ...     %d other instructions\n", tot_other);
}
```

After printing the name of the current procedure (as was done in ex1), the code above grabs the procedure's body and casts it to an InstrList. Remember that the body of an OptUnit may be in several different forms, from a simple list of instructions to many separate instruction lists connected by explicit control-flow information (i.e., CFG form) to other forms that you may add.

We can apply several kinds of OPI functions to InstrList, and we've illustrated a few here. We use size, which indicates the number of total Instr elements in the InstrList, and start, which provides us with a handle at the beginning of the InstrList. This handle allows us to iterate through the entire Instr sequence and use some of the Instr predicates to collect information about the distribution of instruction kinds in this procedure.

4 Ex3 – Manipulating an Instruction List

You should now understand the basic operation and structure of a Machine-SUIF pass. In this and the next several examples, we will investigate a more realistic pass and show you how to modify the input Machine-SUIF IR. These examples continue to view the body of a procedure (or optimization unit) as a simple instruction list. Other later examples investigate operations on other kinds of bodies.

Description of pass function. Suppose that you want to write a pass that takes a register-allocated instruction list and rewrites that list to reserve a register. In other words, we will create a memory location that will become the "home" of the contents of the reserved register. Whenever we encounter an instruction that uses the reserved register, we will have to load it with the contents of our memory location. When an instruction writes the reserved register, we will have to store the result into our memory location. Once this has been done, we are free to use the reserved register for our own purposes anywhere between the program's uses of the reserved register. You might, for example, wish to do this to prepare an instruction list for instrumentation by a profiler.

We will assume that the pass is only run after register allocation, and we will provide the user with the option of specifying a specific register to reserve.

Additions to SUIF-specific files. The files suif_main.cpp, suif_pass.h, and suif_pass.cpp are largely unchanged from our previous examples. We added a small amount of code to the Ex3SuifPass methods initialize and parse_command_line so that we could grab the user-specified reserved register name, if one was supplied. We refer you to the SUIF documentation to learn more about the specification and processing of command line options in SUIF.

Setup. Below, we present the declaration of class Ex3. It is the same as the trivial OPI class declarations that you've seen previously, except for the fact that we've included a few instance variables.

6 ⟨*class* Ex3 6⟩≡

```
class Ex3 {
  public:
    Ex3() { }

    void initialize();
    void do_opt_unit(OptUnit*);
    void finalize();

    // set pass options
    void set_reserved_reg(IdString n)   { reserved_reg_name = n; }

  protected:
    // pass-option variables
    IdString reserved_reg_name; // name of register to reserve (optional)

    // initialization variables
    int reserved_reg;           // its abstract number
    TypeId reserved_reg_type;   // its generic type
    Opnd reserved_reg_opnd;     // its register operand

    // markers for inserted instructions
    NoteKey k_reserved_reg_load;
    NoteKey k_store_reserved_reg;
};
```

Most of the variables deal with properties or IR objects related to the reserved register. The variables `k_reserved_reg_load` and `k_store_reserved_reg` declare new `NoteKey` variables for use by this pass in its annotations.

Finding a register to reserve. The `initialize` method of the class `Ex3` identifies the register we wish to reserve, and it builds several IR components related to this choice. The abstract register number for the register that we wish to reserve is a simple `int`. If the user specified a particular register by name, we use the `reg_lookup` function to determine the abstract number corresponding to this register name. Note that the user should be aware of the target architecture because the name is target dependent.

7 ⟨*set* reserved_reg 7⟩≡

```
    // determine register to reserve
    int reserved_reg = -1;
    if (reserved_reg_name != empty_id_string) {
        // get abstract register number for user-specified reg name
        reserved_reg = reg_lookup(reserved_reg_name.chars());

    } else {
        // Since the user didn't specify a register to reserve, we use
        // the first temporary (caller-save) register.
        const NatSet *caller_saves = reg_caller_saves();
        claim(!caller_saves->is_empty(),
              "No caller-save regs and so nothing to do");
        reserved_reg = caller_saves->iter().current();
    }
```

On the other hand, if the user did not specify a register name on the command line, the pass chooses one of the caller-saved registers. The OPI function `reg_caller_saves` returns a pointer to a `NatSet` containing the abstract register numbers of all registers obeying this convention. If the set is empty, we abort the pass. Otherwise, we simply choose the first one that we find and make it the reserved register.

To create a register operand for this register number, Machine SUIF requires a `TypeId`. This `TypeId` describes the type of the data in the register. Since we're simply going to spill and reload the contents of this register as an untyped bit pattern, we could define `reserved_reg_type` as some void type. The code below is a bit more intelligent (mostly for illustrative purposes) in that it queries the width of the reserved register and then sets `reserved_reg_type` to the type for an unsigned integral value of that width.

8a ⟨*set* `reserved_reg_type` 8a⟩≡

```
// inspect reg width to get a generic type for this register
int sz = reg_width(reserved_reg);
switch (sz) {
  case  8: reserved_reg_type = type_u8;  break;
  case 16: reserved_reg_type = type_u16; break;
  case 32: reserved_reg_type = type_u32; break;
  case 64: reserved_reg_type = type_u64; break;
  default:
    claim(false, "unexpected register width = %d", sz);
}
```

Given a register number and a `TypeId`, we can now build the hard register operand `reserved_reg_opnd`.

8b ⟨*set* `reserved_reg_opnd` 8b⟩≡

```
// build a hard register operand
reserved_reg_opnd = opnd_reg(reserved_reg, reserved_reg_type);
```

Initializing the note keys. The last thing done in `Ex3::initialize` is to define the values of our pass's `NoteKeys`.

8c ⟨*set the note keys* 8c⟩≡

```
// initialize note key variables for my annotations
k_reserved_reg_load = "reserved_reg_LOAD";
k_store_reserved_reg = "STORE_reserved_reg";
```

You should consider the type `NoteKey` to be an opaque type. We usually define the value of a `NoteKey` outside the OPI pass that we're writing, since its type depends upon the substrate and may change from one substrate to another. In Machine SUIF, a note key is a string (`IdString`, in particular).

Building a stack location. The `do_opt_unit` routine begins as it did in `Ex2`. Before we begin walking the instruction list however, we build a stack location to store the current contents of the reserved register.

8d ⟨*build a stack location* 8d⟩≡

```
// build a stack location to hold the reserved register's contents
VarSym *var_reserved_reg = new_unique_var(reserved_reg_type,
                                          "_var_reserved_reg");
// build an effective-address operand for var_reserved_reg
Opnd ea_var_reserved_reg = opnd_addr_sym(var_reserved_reg);
```

The OPI function new_unique_var returns a pointer to a new variable symbol of the right type that is local to the current OptUnit. We supply a name (_var_reserved_reg) that will help us to recognize the function of this stack location.

Once we have a variable symbol, we can construct an address-symbol operand that represents the address of this variable symbol for use in load and store operations. We often refer to these kinds of operands as "effective addresses" (ea).

Manipulating the InstrList. The loop that walks the instruction list is structured the same way as the one that we used in Ex2.

9a ⟨Ex3 *walk of* InstrList 9a⟩≡

```
for (InstrHandle h = start(mil); h != end(mil); ++h) {
    Instr *mi = *h;

    if (is_label(mi) || is_dot(mi) || has_note(mi, k_store_reserved_reg))
        continue;

    bool found_reserved_reg_as_src = false;
    ⟨inspect source operands 9b⟩

    if (found_reserved_reg_as_src) {
        ⟨insert load of reserved register 10a⟩
    }

    bool found_reserved_reg_as_dst = false;
    ⟨inspect destination operands 9c⟩

    if (found_reserved_reg_as_dst) {
        ⟨insert spill of reserved register 10b⟩
    }
}
```

The current instruction mi is inspected only if it is interesting, i.e. it is an active instruction that we didn't insert. As you will see, we will be inserting instructions after as well as before the one corresponding to the handle h.

For each active instruction, we inspect its source and destination operand list looking for occurrences of the reserved register. The code to perform these inspections is nearly identical.

9b ⟨*inspect source operands* 9b⟩≡ (9a)

```
for (i = 0; (i < srcs_size(mi) && !found_reserved_reg_as_src); i++) {
    Opnd o = get_src(mi, i);
    found_reserved_reg_as_src = is_reg(o) && (o == reserved_reg_opnd);
}
```

9c ⟨*inspect destination operands* 9c⟩≡ (9a)

```
for (i = 0; (i < dsts_size(mi) && !found_reserved_reg_as_dst); i++) {
    Opnd o = get_dst(mi, i);
    found_reserved_reg_as_dst = is_reg(o) && (o == reserved_reg_opnd);
}
```

If we find an occurrence of the reserved register in the source operand list, we insert a load operation before the current instruction. We use the OPI function `opcode_load` to find the correct target-specific load opcode for the restore operation, and we mark this new instruction with one of our flag annotations.

10a ⟨*insert load of reserved register* 10a⟩≡ (9a 10c)
```
Instr *ld_tmp0;
ld_tmp0 = new_instr_alm(reserved_reg_opnd,
                        opcode_load(reserved_reg_type),
                        clone(ea_var_reserved_reg));
set_note(ld_tmp0, k_reserved_reg_load, note_flag());
insert_before(mil, h, ld_tmp0);
```

For occurrences of the reserved register in the destination operand list, we insert a store operation after the current instruction. This instruction is built and marked in an analogous manner.

10b ⟨*insert spill of reserved register* 10b⟩≡ (9a 10c)
```
Instr *st_tmp0;
st_tmp0 = new_instr_alm(clone(ea_var_reserved_reg),
                        opcode_store(reserved_reg_type),
                        reserved_reg_opnd);
set_note(st_tmp0, k_store_reserved_reg, note_flag());
insert_after(mil, h, st_tmp0);
```

Finally, please note that we clone the address-symbol operand `ea_var_reserved_reg` before inserting it into either the load or store instruction. This ensures that each instruction has a unique copy of the address operand. In this way, if later pass modifies a component of one of these address operands, we will not inadvertently affect the others. This is necessary because some implementations of the OPI (e.g., Machine SUIF) define the type `Opnd` to have reference semantics.

5 Ex4 – Operand Mapping

The code in Example 3 walked an instruction's operands looking for a register operand that matched the one for our reserved register. An astute reader might have noticed a problem with the output of `do_ex3`: it did not identify occurrences of the reserved register in address expressions. This is because address expressions are considered operands, and an address-expression operand is never equal to a register operand. To find all occurrences of our reserved register in an instruction, we must search not only the instruction's operands but also the suboperands of all encountered address expressions. We will accomplish this feat with the use of the OPI function `map_opnds` and the OPI class `OpndFilter`.

In particular, this example starts with the code from Example 3 and makes a just few small modifications to class `Ex3` and its implementations. We begin by listing the loop that walks over the `InstrList`.

10c ⟨`Ex4` *walk of* `InstrList` 10c⟩≡
```
for (InstrHandle h = start(mil); h != end(mil); ++h) {
    Instr *mi = *h;
    MyOpndFilter filter(reserved_reg_opnd);

    if (is_label(mi) || is_dot(mi) || has_note(mi, k_store_reserved_reg))
        continue;

    map_opnds(mi, filter);

    if (filter.found_reserved_reg_as_input()) {
```

```
            ⟨insert load of reserved register 10a⟩
        }

        if (filter.found_reserved_reg_as_output()) {
            ⟨insert spill of reserved register 10b⟩
        }
    }
```

This loop is nearly identical to the ⟨`Ex3` *walk of* `InstrList` 9a⟩. The one important difference is that we have replaced the two code snippets ⟨*inspect source operands* 9b⟩ and ⟨*inspect destination operands* 9c⟩ with a call to `map_opnds`. Also, the insertion of our load and store instructions is now protected by Boolean values set in `map_opnds`. Notice that the code to ⟨*insert load of reserved register* 10a⟩ and ⟨*insert spill of reserved register* 10b⟩ is the same as that which we used in `Ex3`.

The work to inspect the inputs and outputs of an instruction is done by `filter`. This object is an instance of ⟨*class* `MyOpndFilter` 11a⟩, which is a derived class of `OpndFilter`. The class defines the two flags we need: one indicating that the reserved register was seen as an input to this instruction and the other indicating that the register was an instruction output. Instances of this class are also initialized with a value indicating the register operand to match against.

11a ⟨*class* `MyOpndFilter` 11a⟩≡

```
class MyOpndFilter : public OpndFilter {
  public:
    MyOpndFilter(Opnd r) : reserved_reg_opnd(r),
        in_flag(false), out_flag(false) { }

    Opnd operator()(Opnd, InOrOut);

    bool found_reserved_reg_as_input() { return in_flag; }
    bool found_reserved_reg_as_output() { return out_flag; }

  protected:
    Opnd reserved_reg_opnd;
    bool in_flag, out_flag;
};

⟨MyOpndFilter::operator() 11b⟩
```

The OPI function `map_opnds` invokes the method ⟨`MyOpndFilter::operator()` 11b⟩ on each operand and suboperand in the instruction. It indicates whether the operand is an input or an output via the formal parameter `in_or_out`.

11b ⟨`MyOpndFilter::operator()` 11b⟩≡ (11a)

```
Opnd
MyOpndFilter::operator()(Opnd opnd, InOrOut in_or_out)
{
    if (is_reg(opnd) && (opnd == reserved_reg_opnd)) {
        // found a reference to the reserved reg; update flags
        in_flag  |= (in_or_out == IN);
        out_flag |= (in_or_out == OUT);
    }

    return opnd;
}
```

The method returns an `Opnd` so that you can walk over an instruction's operands and conditionally replace some of them. Whatever you return replaces the current input operand `opnd` in the instruction. For this example, we simply return what we were passed since we don't want to change any of the instruction's operands.

Finally, you should be aware that address-expression operands can appear in either the source or destination operand list. An address-expression operand in the destination list represents a memory location where the instruction is going to write a result. The suboperands of an address-expression operand are always considered instruction inputs. It would be incorrect to use the function `map_dst_opnds` and consider an occurrence of the reserved register in the destination operand list as unconditionally requiring the insertion of a store operation.

6 Ex5 – Removing Instructions from an Instruction List

To this point, our examples have only inserted instructions. This example demonstrates how to remove safely one or more instructions from an instruction list. In particular, since removing anything in C/C++ is always exciting, we'll show you how to do this without invalidating your `InstrHandle`.

The `do_opt_unit` method of `Ex5` searches the `InstrList` of an `OptUnit` for instructions annotated with either our `k_reserved_reg_load` or `k_store_reserved_reg` flag notes. When it finds such an instruction, it calls the OPI function `remove` and deletes the returned instruction.

12 ⟨Ex5 *walk of* `InstrList` 12⟩≡

```
for (InstrHandle h = start(mil); h != end(mil); ) {
    InstrHandle cur_h = h++;        // advance before possible instr removal
    Instr *mi = *cur_h;

    if (has_note(mi, k_reserved_reg_load) ||
        has_note(mi, k_store_reserved_reg))
        delete remove(mil, cur_h);
}
```

The `for` loop is carefully constructed to increment the instruction handle `h` (the loop's induction variable) before calling `remove`. Because of the implementation of `InstrHandle` in Machine SUIF, we are not guaranteed to be able to increment `h` correctly if we have removed the instruction associated with it.

Removing Operands from an Instruction Unlike an `InstrList`, you should view the source and destination operand dequences as implemented as arrays. Even though the OPI provides you with equivalents of `remove`, called `remove_src` and `remove_dst`, that are able to remove an operand from an operand sequence given its handle, we typically do not use these functions. They are relatively expensive and are not generally needed.

As an example of where you might use this functionality, consider code generation. During code generation, we change one instruction of target X into one or more instructions of target Y. For translations that map one X instruction to one Y instruction, you might be tempted to reuse the storage container for the first instruction. For instance, translating a SUIFvm integer AND instruction into an Alpha AND instruction requires only that we change the opcode from `AND` to `AND`. Both take the same number of source and destination operands, and Machine SUIF uses the same data structure for both target architectures.

Suppose however that the SUIFvm AND instruction specified three source operands while the Alpha

AND required only two of these operands.[2] During code generation, you would want to remove one of the SUIFvm AND source operands in addition to changing the opcode. Instead of this approach, we recommend that you construct the Alpha AND instruction from the needed components of the SUIFvm AND using the appropriate instruction creator function.

7 Ex6 – AnyBody Converters

This section describes the contents of the two converters associated with class `Cfg`, a derived class of `AnyBody`. If you create a new kind of `OptUnit` body, you should provide two converters like these.

For each converter, we present only the OPI pass method `do_opt_unit`. The rest of the files in the source directories are, except for the pass name strings and the parsing of some command line arguments, identical to those explained in `Ex1`.

`Il2Cfg::do_opt_unit`. To convert the body of an `OptUnit` from an `InstrList` to a `Cfg`, we invoke the OPI function `new_cfg`. This function modifies the input `InstrList` during the `Cfg` creation. You can think of this as "recycling" whatever `InstrList` contents the new `Cfg` object can use.

13 ⟨Il2cfg::do_opt_unit 13⟩≡

```
void
Il2cfg::do_opt_unit(OptUnit *unit)
{
    IdString name = get_name(get_proc_sym(unit));
    debug(1, "Processing procedure \"%s\"", name.chars());

    // get the body of the OptUnit
    AnyBody *orig_body = get_body(unit);

    if (is_kind_of<Cfg>(orig_body)) {
        // just make sure that the options are correct
        canonicalize((Cfg*)orig_body, keep_layout,
                     break_at_call, break_at_instr);
        return;
    }

    claim(is_kind_of<InstrList>(orig_body),
          "expected OptUnit body in InstrList form");

    // convert input body to a Cfg
    Cfg *cfg = new_cfg((InstrList*)orig_body, keep_layout,
                       break_at_call, break_at_instr);

    // replace original body
    copy_notes(orig_body, cfg);
    set_body(unit, cfg);

    if_debug(5)
        fprint(stderr, cfg, false, true);              // no layout, just code

    // clean up
    delete orig_body;
}
```

[2] This example is a bit hokey, but we did warn you that `remove_src` was generally not needed.

We explicitly copy the annotations attached to the InstrList to the Cfg since we may (in the future) not wish to copy all of the annotations on the original AnyBody to the Cfg. Finally, we replace the original body with the new Cfg and delete the original body.

You can also use this function to ensure that a body in Cfg form has the specific form you want. For bodies already in Cfg form, we invoke the OPI function canonicalize with the user-specified Cfg options and then return.

Cfg2il::do_opt_unit. To convert the body of an OptUnit from a Cfg to an InstrList, we invoke the OPI function to_instr_list. As with new_cfg, this function modifies the input Cfg.

14 ⟨Cfg2il::do_opt_unit 14⟩≡

```
void
Cfg2il::do_opt_unit(OptUnit *unit)
{
    IdString name = get_name(get_proc_sym(unit));
    debug(1, "Processing procedure \"%s\"", name.chars());

    // get the body of the OptUnit
    AnyBody *orig_body = get_body(unit);

    if (is_kind_of<InstrList>(orig_body))
        return;         // nothing to do

    claim(is_kind_of<Cfg>(orig_body),
          "expected OptUnit body in Cfg form");

    // print the CFG if debugging verbosely
    if_debug(5)
        fprint(stderr, static_cast<Cfg*>(orig_body), false, true);

    // convert input body to an InstrList
    InstrList *instr_list = to_instr_list(orig_body);

    // replace original body
    copy_notes(orig_body, instr_list);
    set_body(unit, instr_list);

    // clean up
    delete orig_body;
}
```

This converter ends in the same way as above. For convenience, it quietly leaves an OptUnit body alone if the body is already in InstrList form.

AnyBody code in a pass. As we mentione in *The Extender's Guide*, it is the responsibility of the person stringing together optimization passes to ensure that the input needs of a pass are satisfied. In the do_opt_unit method of an OPI optimization pass, we include only the following code for casting the OptUnit's body to the appropriate type. Note that this example assumes that the pass wants the body as a Cfg.

```
// get the body of the OptUnit
AnyBody *body = get_body(unit);
```

```
// verify that it is a Cfg
claim (is_a<Cfg>(body),
       "expected OptUnit's body in Cfg form");
Cfg *cfg = (Cfg *)body;
```

When done, this pass writes the `OptUnit` body as a `Cfg`. It is not expected that an individual pass would convert the body back to a “standard” form.

8 Acknowledgments

This work was supported in part by an DARPA/NSF infrastructure grant (NDA904-97-C-0225), a NSF Young Investigator award (CCR-9457779), and a NSF Research Infrastructure award (CDA-9401024). We also gratefully acknowledge the generous support of this research by Advanced Micro Devices, Compaq, Digital Equipment, Hewlett-Packard, International Business Machines, Intel, and Microsoft.

A Copyright

All of our code is protected by the following copyright notice.

15 ⟨*Machine-SUIF copyright* 15⟩≡

```
/*
    Copyright (c) 2000 The President and Fellows of Harvard College

    All rights reserved.

    This software is provided under the terms described in
    the "machine/copyright.h" include file.
*/
```