The Machine-SUIF Machine Library

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

The Machine SUIF system is an extension of Stanford SUIF version 2 that supports construction of compiler back ends. The `machine` library is the core of Machine SUIF. It enables you to create machine descriptions, to construct and manipulate machine-level intermediate forms during back-end optimization, and to emit object code.

While the library itself is machine independent, it is the basis for producing other libraries that are machine specific. These machine-specific libraries supply the parameters that make machine-independent optimization passes sensitive to the peculiarities of target machines.

The optimization programming interface (OPI) used for creating Machine-SUIF passes and libraries is described elsewhere [4, 5]. This document fills in details of the OPI as implemented in Machine SUIF. It also discusses aspects of the implementation that will help you use and extend the system.

Contents

1 Introduction

The Machine SUIF system is an extension of Stanford SUIF version 2 that supports construction of compiler back ends. This document is part of a set of documents explaining how to use and how to extend Machine SUIF. It is aimed at three kinds of readers:

- Those interested in writing machine-level optimizations based on existing target machine descriptions.
- Those also interested in adding new target machine descriptions (or extending existing ones).
- Those interested in understanding enough about Machine SUIF's implementation to enable them to modify it.

We assume that you have read *An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization* and *A User's Guide to the Optimization Programming Interface*. As explained in those documents, Machine SUIF implements an interface for optimization writers, the OPI, that can be used not only for static compilation, an in Machine SUIF, but also in other settings, such as run-time code optimization. Since the OPI has a wider scope than Machine SUIF, it has been documented separately.

This document begins the description of how the OPI is implemented as an extension of SUIF version 2.[1] It also describes the mechanisms that allow extension of the system in several directions. The OPI itself can be extended by the addition of new intermediate representations. New target-architecture families can be added to the system, and existing ones can be augmented, either to reflect new implementations by vendors or to accommodate architectural experiments.

The core of the Machine SUIF implementation is the `machine` library, which is the subject of this document. It implements the most basic IR classes in the OPI and it also establishes extension machinery. Other Machine-SUIF libraries, such as the control-flow graph (`cfg`) library, continue the OPI implementation and they illustrate how Machine SUIF allows the OPI to be extended by users. The documents describing those libraries should be helpful both to users and extenders of the system.

The `machine` library, the `cfg` library, and others that develop the OPI implementation are machine-independent: they are designed to apply to all kinds of target machines. A different series of libraries supply the machine descriptions necessary to carry out machine-level optimization. We say that the OPI is *parameterized* over the features of target machines. The machine-specific libraries supply the parameter bindings.

2 Overview

The main sections of this document correspond to implementation modules. The interfaces they display are exactly the source code that is extracted and used to build the system.

As hinted above, the `machine` library plays several diferent roles.

- It builds the layer between Machine SUIF and the underlying SUIF substrate, recasting many SUIF facilities into the form used by the OPI.
- It implements the data structures for instruction-lists, instructions, operands, and annotations, including the functions that create, inspect, and manipulate these key pieces if the IR.
- It defines data structures for describing target-machine characteristics, such as register files.
- It provides the machine-independent frameworks for several kinds of machine-specific passes, such as those for finalizing optimized code and emitting it as object code in text or binary form.

[1] You will need to be familiar with the basic concepts of SUIF 2, also referred to here as base SUIF. See the SUIF 2 home page[2] for information on that system.

- It provides the basis for extending Machine SUIF in several directions: extension of the OPI itself, addition of new target machines and/or new features for existing targets, and the addition of new kinds of machine description.

We touch on each of these roles in the following subsections.

2.1 Connection to SUIF

We rely on the SUIF substrate to provide:

- Types, symbols, symbol tables, descriptions of the inital values of variables and the signatures of procedures being compiled.
- A system for generating IR classes that provides support for cloning, for intermediate-file I/O and symbolic printing of IR objects with and minimum of coding on the part of the class developer.
- A scheme for building passes that can either be applied in series to a whole intermediate file at a time or can be pipelined so that a procedure being compiled undergoes several passes at at time.

Connection to the SUIF substrate is mostly quite simple. The OPI uses a naming convention similar to that used in SUIF. Its names tend to be shorter than SUIF's, so some classes are renamed via `typedef`s. For example, SUIF's `VariableSymbol` becomes `VarSym`. The OPI uses a different style for annotating IR objects than base SUIF, so the `machine` library encapsulates SUIF annotations as OPI *notes*.

A more pervasive style difference is that base SUIF makes little use of global variables and expresses all functionality through class methods, while the OPI makes use of both global variables and overloaded global functions. Part of the `machine` library's substrate-interface role is simply to convert calls on class methods into calls on plain functions.

In Machine SUIF, when IR objects are created by OPI functions, they sometimes need to be connected to other objects in ways that are not apparent at the OPI level. The chief example is the Machine-SUIF operand, of type `Opnd`, which will be discussed in the next subsection.

2.2 Intermediate Representation

The `machine` library implements the core of the OPI's IR classes: those for instruction-lists (`InstrList`), instructions (`Instr`), operands (`Opnd`), and annotations (`Note`). It also implements functions and classes that create, inspect, and manipulate IR objects.

Optimization units. The OPI tries not to hard-wire the decision of what program units shall be optimized. While it's traditional to deal with one source procedure at a time, the OPI doesn't lock users into that model. Nevertheless, in Machine SUIF, the OPI data type `OptUnit` is exactly the SUIF `ProcedureDefinition` (which we also nickname `ProcDef`).

Instruction lists. The Machine-SUIF class `InstrList` is used to represent a sequence of machine instructions, including label "instructions" that serve as the targets of control-transfer instructions (CTI). Thus a single `InstrList` object can represent all of the control behavior of a machine-level program.

`InstrList` is implemented as a subclass of `AnyBody`, which is itself a subclass of SUIF's `ExecutionObject`. Since the the `body` field of a SUIF procedure definition has type `ExecutionObject*`, an `InstrList` can be used as the body of a SUIF procedure representation. This is one of the key points of connection between base SUIF and Machine SUIF.

Instructions. The OPI defines one instruction class, `Instr`, which it describes in four categories, two real (executable) and two "pseudo". Mirroring that breakdown, the Machine SUIF class hierarchy defines four instruction classes: `InstrAlm` (arithmetic, logical and memory), `InstrCti` (control-transfer), `InstrLabel` (label-defining), and `InstrDot` (pseudo-op). An OPI user needs to understand the categories, but shouldn't need to mention the implementation classes that correspond to them.

Instances of class `Instr` must correspond one-to-one with real machine instructions, but must be manipulated by machine-independent tools. It seems pointless to try to reflect the myriad formats of real machine instructions in the Machine-SUIF representation of instructions. For a small gain in space efficiency, we would add considerable complexity and would make it harder to tranform instructions in place, e.g., by altering opcodes and shifting operands.

Operands. The OPI defines one operand class, `Opnd`, which it breaks downs into several categories. One of these, the address expressions, is further broken down into subcategories. As with `Instr`, the implementation of `Opnd` uses a class hierarchy that closely parallels the OPI characterization.

But `Opnd` presents a special problem in the Machine SUIF implementation because the OPI describes it not as a heap-allocated object handled through pointers, but as a scalar value. The reason is that in a run-time optimization setting, operands can and should have a very lightweight implementation, requiring no storage management. It would be a mistake to require them to be treated as pointers in that setting simply to make the Machine-SUIF implementation a bit more uniform.

For this reason, the OPI leaves open the question of whether instances of non-pointer IR data types such as `Opnd` behave like references or like independent values. An implementation is free to use reference semantics, and in fact, Machine SUIF does so. For algorithms that aren't meant to be reusable in an online setting, that's all you need to know. For maximum portability of your OPI-based programs, however, you should explicitly clone mutable operands to prevent accidental side effects. (See the description of `clone` in Section 4.1.) In practice, this is seldom needed, since the most frequently-used kinds of operands are immutable.

Under the hood, a Machine-SUIF operand is really a pointer to a SUIF object. Machine SUIF hides the allocation and deallocation of these objects. The user interface to operands is through functions and interface classes defined by the OPI, not through the methods of the implementation objects.

Annotations. The situation is similar for Machine-SUIF annotations. Class `Note` needs to be amenable to a variety of lightweight implementations. Like `Opnd`, it is used as a scalar, not an explicit pointer. But as with `Opnd`, Machine SUIF employs a hidden pointer to a base SUIF annotation object, and it reflects reference sementics. And again, a `clone` function allows you to avoid accidental mutation.

2.3 Description of Target Machines

The `machine` library defines the classes for machine-description data structures that are instantiated by target-specific libraries. A typical example of these is the `RegInfo` class (Section 7). A `RegInfo` object describes the register architecture of a family of target machines, including the number and kinds of register files, the widths of their registers and the number in each bank, and so on. Machine descriptions of this kind are an important part of the binding that a machine-specific library establishes when it is loaded. Machine-independent optimization code queries these machine descriptions in order to cater to machine characteristics without hard-wiring in.

2.4 Frameworks for Key Back-End Passes

Most back ends constructed in Machine-SUIF start with passes for lowering and code generation and end with a finalization pass and a pass for emitting object code. Lowering is machine-independent; it translates from the base SUIF intermediate form to Machine SUIF, using the SUIFvm idealized machine as the target. The other three kinds of passes need to be specialized for the target machine, but for each kind, the pattern is much the same from target to target. The `machine` library provides base classes that are used as frameworks for such passes. For a specific target, the corresponding library derives from these base classes and fills in the target-specific particulars.

For example, a typical finalization pass does such things as allocating the stack frame for each compiled procedure, it replaces symbolic stack references by effective-address expressions with specific frame offsets, it generates register-saving and restoring prologue and epilogue code, and so on. The `machine` library defines a `CodeFin` class that provides the framework for finalization passes. To produce an actual finalization pass for the Alpha target, say, the Alpha library derives a subclass of `CodeFin` and specializes the methods representing the different finalization steps.

In addition to `CodeFin`, the `machine` library currently has such frameworks for emitting object code in two forms: assembly code (`Printer`) and C code (`CPrinter`). Eventually, there will be another one for binary object modules.

The framework for code generation is called `CodeGen`. Its input is in the form of SUIFvm code, and for that reason it's defined in the `suifvm` library, not the `machine` library.

2.5 Bases for Extension

The fact that the OPI relies heavily on global variables and functions is conducive to extensibility of the system. A problem arises, of course, if binding the OPI to a particular target machine is done simply by setting some global variables that are shared by the code for a pass and all the libraries that it links in. The issue is that occasionally passes may need to deal with more than one machine binding at a time. A binary translator, for example, needs the descriptions of both its source and target machines at the same time. If it needs flow graphs for both kinds of machines at the same time, then it wouldn't do for the CFG library to rely only on global variables when accessing machine characteristics. Elements of such a library need to be explicitly parameterized so that they can be used for more than one target within the same application.

To solve this problem, the `machine` library provides a class called `Context` (Section 15). Its main purpose is to gather in one place the target-specific bindings needed for a particular compilation context. For example, the function that takes a machine instruction and returns `true` if it reads memory is actually stored in a `Context`. That's because it is target-specific; it's definition must come from a target-specific library. Likewise, the `RegInfo` record that describes that target's register architecture is stored in a `Context`.

On the other hand, nearly every kind of pass other than a binary translator is concerned with only one target machine at a time. To accommodate that very common case, the system includes a distinguished global context, and you access its contents through simple global functions. Most pass writers can therefore ignore the context mechanism records and use these global functions.

In addition to making optimization code simpler to write, this scheme is best for porting passes into the dynamic optimization world as well. In that setting, a context record would be an unnecessary encumbrance, since the target is well known.

Libraries may of course extend the set of target-specific features of the OPI. For example, the library that supports instruction scheduling needs more detailed hardware descriptions than are defined in the `machine` library. The *Extender's Guide* describes how contexts help. This document explains the implementation.

3 Machine Instructions

The central classes in the representation of machine-level code are those for instruction lists (`InstrList`), instructions (`Instr`), and operands (`Opnd`). This section and the next introduce these classes and the OPI functions that relate to them.

3.1 Instruction constituents

Every machine instruction is an instance of `Instr`. Each has an opcode and most have operands, representing their sources ("arguments") and destinations ("results").

Opcodes. An opcode is a non-negative integer. With two exceptions, the opcode of a machine instruction is meaningful only when interpreted with respect to an architecture family. (The exceptions are `opcode_null` and `opcode_label`, which are the opcodes of any null instruction or label instruction, respectively.) The particular numbers used as opcodes are not dictated by ISAs; they are arbitrarily assigned small integers, useful for accessing instruction properties by table lookup. The same integer that represents `mov` in an $x86$ instruction may be used for `addq` in an Alpha instruction. The architecture-specific libraries assign identifiers to opcodes; for example, `MOV` stands for `mov` in the $x86$ library and `ADDQ` stands for `addq` in the Alpha library. (See Section 6 for more discussion of opcodes.)

The `get_opcode` function returns an instruction's opcode, which is always set when the instruction is created.

Operands. In general, an instruction may have source operands and destination operands; together, these are its *direct* operands. However, an operand that represents an effective address calculation may itself contain operands; so an instruction can also involve indirect operands.

Machine SUIF imposes the constraints that locations read by an instruction must be explicit in its operands (though possibly indirectly), and that locations written by an instruction must be explicit destination operands. We distinguish the *input* operands of an instruction, which are all the values read during the instruction's execution, from the *source* operands of the instruction, which simply identify its direct arguments.

As an example, consider an $x86$ instruction that adds the contents of the `%eax` register to the memory location specified by the effective address (EA) expression `40(%esp)`. Even though $x86$ has a two-address ISA, the Machine-SUIF representation of this instruction has two source operands, an address-expression operand `40(%esp)` and a register operand `%eax`, and one destination operand, an address-expression operand `40(%esp)`.[3] But this add instruction has five inputs: the three register operands (the direct source operand `%eax` and the two occurrences of `%esp` in the address expressions) and two immediate operands (also in the address expressions).

The library provides facilities for enumerating and possibly changing all the input operands of an instruction. We begin with functions that access and alter the direct operands of an instruction.

9 ⟨`Instr` *operand accessors and mutators* 9⟩≡ (16b)

```
Opnd        get_src(Instr*, int pos);
Opnd        get_src(Instr*, OpndHandle);
void        set_src(Instr*, int pos, Opnd src);
void        set_src(Instr*, OpndHandle, Opnd src);
void        prepend_src(Instr*, Opnd src);
void        append_src(Instr*, Opnd src);
void        insert_before_src(Instr*, int pos, Opnd src);
```

[3]It actually has two destination operands. The second is a register operand corresponding to the $x86$ condition-code register.

```
void        insert_before_src(Instr*, OpndHandle, Opnd src);
void        insert_after_src(Instr*, int pos, Opnd src);
void        insert_after_src(Instr*, OpndHandle, Opnd src);
Opnd        remove_src(Instr*, int pos);
Opnd        remove_src(Instr*, OpndHandle);
int         srcs_size(Instr* instr);
OpndHandle  srcs_start(Instr*);
OpndHandle  srcs_last(Instr*);
OpndHandle  srcs_end(Instr*);

Opnd        get_dst(Instr*);
Opnd        get_dst(Instr*, int pos);
Opnd        get_dst(Instr*, OpndHandle);
void        set_dst(Instr*, Opnd dst);
void        set_dst(Instr*, int pos, Opnd dst);
void        set_dst(Instr*, OpndHandle, Opnd dst);
void        prepend_dst(Instr*, Opnd dst);
void        append_dst(Instr*, Opnd dst);
void        insert_before_dst(Instr*, int pos, Opnd dst);
void        insert_before_dst(Instr*, OpndHandle, Opnd dst);
void        insert_after_dst(Instr*, int pos, Opnd dst);
void        insert_after_dst(Instr*, OpndHandle, Opnd dst);
Opnd        remove_dst(Instr*, int pos);
Opnd        remove_dst(Instr*, OpndHandle);
int         dsts_size(Instr*);
OpndHandle  dsts_start(Instr*);
OpndHandle  dsts_last(Instr*);
OpndHandle  dsts_end(Instr*);
```

In the capsule summaries of the above functions, `place` represents a sequence position, which is either a zero-based integer or an `OpndHandle`:

> `get_src(instr, place)` returns the source of `instr` at `place`.
> `set_src(instr, place, src)` makes `src` the source of `instr` at `place`.
> `prepend_src(instr, src)` inserts `src` at the beginning of `instr`'s sources.
> `append_src(instr, src)` inserts `src` at the end of `instr`'s sources.
> `insert_before_src(instr, place, src)` inserts `src` before `place` in `instr`'s sources.
> `insert_after_src(instr, place, src)` inserts `src` after `place` in `instr`'s sources.
> `remove_src(instr, place)` removes and returns the operand at `place` in `instr`'s sources.
> `srcs_size(instr)` returns the number of source operands of `instr`.
> `srcs_start(instr)` returns a handle on the first source of `instr`.
> `srcs_last(instr)` returns a handle on the last source operand of `instr`.
> `srcs_end(instr)` returns the sentinel handle for `instr`'s sources.
>
> `get_dst(instr, pos)` returns the destination of `instr` at position `pos`. (`pos` defaults to 0 if omitted.)
> `set_dst(instr, pos, dst)` replaces the destination of `instr` at position `pos` by `dst`. (`pos` defaults to 0 if omitted.)
> `prepend_dst(instr, dst)` inserts `dst` at the beginning of `instr`'s destinations.
> `append_dst(instr, dst)` inserts `dst` at the end of `instr`'s destinations.
> `insert_before_dst(instr, place, dst)` inserts `dst` before `place` in `instr`'s destinations.
> `insert_after_dst(instr, place, dst)` inserts `dst` after `place` in `instr`'s destinations.
> `remove_dst(instr, place)` removes and returns the operand at `place` in `instr`'s destinations.
> `dsts_size(instr)` returns the number of destination operands of `instr`.
> `dsts_start(instr)` returns a handle on the first destination of `instr`.

`dsts_end(instr)` returns the sentinel place for `instr`'s destinations.

The `srcs_size` and `dsts_size` functions simply return the size of the underlying operand sequences. They do not necessarily represent the semantics of the instruction's opcode. If `set_src` or `set_dst` is called with an index greater than or equal to the current sequence size, the sequence is extended automatically.

Functions `get_dst` and `set_dst` each have variants in which the operand number can be omitted because the case of single-destination instructions is so common.

The indirect operands of an instruction can be accessed or changed by first fetching a direct address operand and then operating on that, using methods to be described in Section 4.

Other constituents. Some instructions have constituents that are neither sources nor destinations. These are managed by functions in the OPI that operate on instructions. Control-transfer instructions usually have target labels, for example; they are not operands, but separate attributes. These target symbols are accessed and modified by the OPI functions `get_target` and `set_target` (Section 3.4). A code label is represented by a special instruction that marks its position in the instruction sequence. It has no operands. Its constituent label is accessed and modified by the OPI functions `get_label` and `set_label`.

3.2 Creators for `Instr`

We refer to functions in the OPI that create new instances of objects as "creation functions", or simply *creators*. A machine instruction creator takes an opcode and possibly some operands, and it produces a machine instruction.

We divide instructions into broad categories and provide creators for each. Here's a summary of the breakdown and the mnemonics used for the categories.

- Active instructions (real machine operations)
  - (`alm`) Arithmetic, logical, and memory instructions
    - (`alu`) Arithmetic and logical
    - (`mem`) Load and store
  - (`cti`) Control-transfer instructions
- Inactive instructions
  - (`label`) Label instructions
  - (`dot`) Assembler pseudo-operations[4]

The purpose of defining categories is two-fold. It allows us to provide creators that are convenient to use because their argument types and numbers are appropriate for the kind of instruction being created. It also allows for an implementation in which different categories may have different representations.

Not every real machine instruction falls neatly into one of the above categories, of course. The control-transfer instructions (`cti`) are the instructions that modify the program counter non-trivially. They include conditional and unconditional branches and function calls. Each contains a symbol that represents the transfer target.

The instruction categories aren't exclusive. On some machines, a control-transfer instruction might perform arithmetic, so it might have side effects other than modifying the program counter. Furthermore, a program transformation may change an instruction from a pure ALU operation to one that both computes a value and stores it in memory. When creating an instruction, you pick the category that

[4] Many assemblers use a leading period (".") to identify non-code directives.

most closely matches the need. If an instruction needs more operands than the provided creation function accepts, you add them separately. The only constraints to be aware of are pretty obvious: an active instruction can never acquire a label, an instruction that needs a transfer target (i.e., a `target` symbol) must be created as a control-transfer instruction, and so on.

Here's are the prototypes of the instruction creators provided in the `machine` library. Their names include the category mnemonics listed above. Each returns a result of type `Instr*`.

12 ⟨`Instr` *creation functions* 12⟩≡ (16b)

```
Instr* new_instr_alm(int opcode);
Instr* new_instr_alm(int opcode, Opnd src);
Instr* new_instr_alm(int opcode, Opnd src1, Opnd src2);
Instr* new_instr_alm(Opnd dst, int opcode);
Instr* new_instr_alm(Opnd dst, int opcode, Opnd src);
Instr* new_instr_alm(Opnd dst, int opcode, Opnd src1, Opnd src2);

Instr* new_instr_cti(int opcode, Sym *target);
Instr* new_instr_cti(int opcode, Sym *target, Opnd src);
Instr* new_instr_cti(int opcode, Sym *target, Opnd src1, Opnd src2);
Instr* new_instr_cti(Opnd dst, int opcode, Sym *target);
Instr* new_instr_cti(Opnd dst, int opcode, Sym *target, Opnd src);
Instr* new_instr_cti(Opnd dst, int opcode, Sym *target,
                     Opnd src1, Opnd src2);

Instr* new_instr_label(LabelSym *label);

Instr* new_instr_dot(int opcode);
Instr* new_instr_dot(int opcode, Opnd src);
Instr* new_instr_dot(int opcode, Opnd src1, Opnd src2);
```

Note that the opcode always separates the destination from the sources, if any. The only difference between two overloadings with the same name is that they allow different numbers of operands to be put in the created instruction. Of course, the number can always be adjusted later. To obtain an instruction with more than one destination or more than two sources, you must use one of the above creators and then add operands to the instruction that it returns.

Recall our example of an $x86$ add instruction. Suppose `value` is the operand representing register `%eax` (which holds the value to be added in) and `addr` is the address operand standing for `40(%esp)`. Suppose that `body` is an `InstrList*` that we are extending as we generate code, created perhaps as follows:

```
InstrList *body = new_instr_list();
```

Then the add instruction might be created and appended like this:[5]

```
append(body, new_instr_alm(addr, ADD, addr, value));
```

(Here `ADD` is the $x86$ opcode for addition.)

On a load/store machine like the Alpha, the same operation might take three instructions and an extra register. Let `tmp` be the operand representing a scratch register. Then the sequence to accomplish the same add might be:

```
append(body, new_instr_alm(tmp,  LDQ,  addr));
append(body, new_instr_alm(tmp,  ADDQ, tmp, value));
append(body, new_instr_alm(addr, STQ,  tmp));
```

[5] Still ignoring the setting of the condition-code register.

3.3 Predicates for `Instr`

The OPI provides Boolean functions to use when your analysis or optimization pass is trying to detect a particular kind of instruction (e.g., a move instruction). Some of these have target-independent semantics.

Others make use of the prevailing target context to inspect the instruction passed to them in a machine-specific way. As an OPI user, you don't have to know which is which. As an extender, you may, and this is covered in Section 15.

13 ⟨`Instr` *predicates* 13⟩≡ (16b)

```
bool is_null(Instr*);
bool is_label(Instr*);
bool is_dot(Instr*);
bool is_mbr(Instr*);
bool is_indirect(Instr*);
bool is_cti(Instr*);
bool reads_memory(Instr*);
bool writes_memory(Instr*);
bool is_builtin(Instr*);

bool is_ldc(Instr*);
bool is_move(Instr*);
bool is_cmove(Instr*);
bool is_predicated(Instr*);
bool is_line(Instr*);
bool is_ubr(Instr*);
bool is_cbr(Instr*);
bool is_call(Instr*);
bool is_return(Instr*);
bool is_binary_exp(Instr*);
bool is_unary_exp(Instr*);
bool is_commutative(Instr*);
bool is_two_opnd(Instr*);
bool is_param_init(Instr*);
```

Their capsule summaries:

> `is_null(instr)` returns true if `instr` is a null instruction.
> `is_label(instr)` returns true if `instr` is a label instruction.
> `is_dot(instr)` returns true if `instr` is a pseudo-op instruction.
> `is_mbr(instr)` returns true if `instr` is a multi-way branch instruction.
> `is_indirect(instr)` returns true if `instr` is an indirect-jump instruction.
> `is_cti(instr)` returns true if `instr` is a control-transfer instruction.
>
> `reads_memory(instr)` returns true if `instr` reads memory.
> `writes_memory(instr)` returns true if `instr` writes memory.
>
> `is_builtin(instr)` returns true if `instr` requires special implementation on each target platform.
>
> `is_ldc(instr)` returns true if `instr` is a load-constant instruction.
> `is_move(instr)` returns true if `instr` is a register-move instruction.
> `is_cmove(instr)` returns true if `instr` is a conditional move instruction.
> `is_predicated(instr)` returns true if `instr` is a predicated instruction.
> `is_line(instr)` returns true if `instr` is a source-code location instruction.
>
> `is_ubr(instr)` returns true if `instr` is an unconditional branch instruction.
> `is_cbr(instr)` returns true if `instr` is a conditional branch instruction.
> `is_call(instr)` returns true if `instr` is a call instruction.

`is_return(instr)` returns true if `instr` is a return instruction.
`is_binary_exp(instr)` returns true if `instr` is a side-effect-free binary expression.
`is_unary_exp(instr)` returns true if `instr` is a side-effect-free unary expression.
`is_commutative(instr)` returns true if `instr` has two source operands that can be exchanged without changing its meaning.
`is_two_opnd(instr)` returns true if `instr` is required to be in *two-operand* (or "two-address") form, i.e., if its first source operand must equal its (first) destination operand.
`is_param_init(instr)` returns true if `instr` transfers a procedure argument from its register or stack transmission location to the parameter variable.

A note about `is_cbr` above: a "conditional branch" instruction is one that has two targets, one of which is an implicit fall-through target. The latter characteristic distinguishes it from a multiway branch that happens to have two targets. An instruction may satisfy `is_cbr` even if its branch condition is statically known or if its fall-through target is the same as its taken target.

A typical instruction satisfying `is_builtin` is one representing an action like `va_start` in C, which is often implemented by macro expansion.

A predicated instruction (for purposes of `is_predicated`) is one that executes only if a predicate operand evaluates to true at run time. (An instruction whose predicate is statically known to be true will not satisfy the `is_predicated` test.)

The two-operand constraint is typical of arithmetic instructions on CISC machines like $x86$. While the physical instruction uses one operand field to express both a source and a destination, Machine SUIF uses distinct source and destination operand fields, but requires the operands they hold be equal.

3.4 Accessors and mutators for `Instr`

In addition to the functions of `Instr` for accessing and modifying operands, the OPI provides functions for getting and setting the fields of particular kinds of instructions:

14a ⟨`Instr` *field accessors* 14a⟩≡ (16b)

```
int get_opcode(Instr *instr);
void set_opcode(Instr *instr, int opcode);

Sym* get_target(Instr *instr);
void set_target(Instr *instr, Sym *target);

LabelSym* get_label(Instr *instr);
void set_label(Instr *instr, LabelSym *label);
```

Their summaries:

`get_opcode(instr)` returns the opcode of `instr`.
`set_opcode(instr, opcode)` replaces the opcode of `instr` with `opcode`.
`get_target(instr)` returns the target of a `CTI` instruction.
`set_target(instr, target)` replaces the target of `instr` with `target`.
`get_label(instr)` returns the label of a label instruction.
`set_label(instr, label)` replaces the label of `instr` with `label`.

3.5 Print Function for `Instr`

Printing of instructions in a form acceptable to an assembler is naturally a target-specific task. It's the subject of Section 8. Here's a function that uses the target-specific mechanism to print a single instruction, perhaps for debugging purposes.

14b ⟨Instr *print function* 14b⟩≡ (16b)
```
void fprint(FILE*, Instr*);
```

3.6 Machine-instruction lists

`InstrList` represents a simple sequence of instructions, so most of the functions related to class `InstrList` are sequence manipulators.

15a ⟨InstrList *functions* 15a⟩≡ (16b)
```
InstrList* new_instr_list();
InstrList* to_instr_list(AnyBody*);
int instrs_size(InstrList *list);
InstrHandle instrs_start(InstrList *list);
InstrHandle instrs_last(InstrList *list);
InstrHandle instrs_end(InstrList *list);
void prepend(InstrList *list, Instr *instr);
void append(InstrList *list, Instr *instr);
void replace(InstrList *list, InstrHandle handle, Instr *instr);
void insert_before(InstrList *list, InstrHandle handle, Instr *instr);
void insert_after(InstrList *list, InstrHandle handle, Instr *instr);
Instr* remove(InstrList *list, InstrHandle handle);
```

Their summaries:

> `new_instr_list()` returns a new `InstrList*` containing no instructions.
> `to_instr_list(body)` returns a new `InstrList*` after moving the contents of `body` into that new list, leaving `body` empty.
> `instrs_size(list)` returns the number of elements in `list`.
> `instrs_begin(list)` returns a handle on the first element of `list`.
> `instrs_end(list)` returns the sentinel handle for `list`.
> `prepend(list, instr)` inserts `instr` at the beginning of `list`.
> `append(list, instr)` inserts `instr` at the end of `list`.
> `replace(list, instr, handle)` replaces the element at `handle` in `list` by `instr`.
> `insert_before(list, handle, instr)` inserts `instr` before `handle` in `list`.
> `insert_after(list, handle, instr)` inserts `instr` after `handle` in `list`.
> `remove(list, handle)` removes and returns the instruction at `handle` in `list`.

Type `InstrHandle` is an abbreviation for a C++ sequence “iterator”. So it behaves like a pointer into a list of instructions.

15b ⟨InstrHandle *definition* 15b⟩≡ (16b)
```
typedef list<Instr*>::iterator InstrHandle;
```

Function `to_instr_list` serves to produce a “go-between” representation for the bodies of optimization units. To convert the body of an optimization unit, the generic type for which is `AnyBody*`, to a specific form that you need (such as `Cfg*`), you can apply `to_instr_list` and then build your specific form from the resulting linear list.

The `instrs_` functions are so named because the sequence of instruction pointers in an `InstrList` is called `instrs`. These functions are frequently used; since an `InstrList` contains only one sequence, we can provide shorter names for these handle-related functions without ambiguity.

15c ⟨InstrList *function nicknames* 15c⟩≡ (16b)
```
inline int
size(InstrList *instr_list)
{
    return instrs_size(instr_list);
}

inline InstrHandle
start(InstrList *instr_list)
{
    return instrs_start(instr_list);
}

inline InstrHandle
last(InstrList *instr_list)
{
    return instrs_last(instr_list);
}

inline InstrHandle
end(InstrList *instr_list)
{
    return instrs_end(instr_list);
}
```

To print the instructions of an `InstrList` using the conventions of the prevailing target:

16a ⟨InstrList *print function* 16a⟩≡ (16b)
```
void fprint(FILE*, InstrList*);
```

3.7 Header file `instr.h`

The header file for instructions and instruction lists has the following outline:

16b ⟨*machine/instr.h* 16b⟩≡
```
/* file "machine/instr.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_INSTR_H
#define MACHINE_INSTR_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "machine/instr.h"
#endif

#include <machine/substrate.h>
#include <machine/opnd.h>
#include <machine/machine_ir.h>

⟨Instr operand accessors and mutators 9⟩

⟨InstrHandle definition 15b⟩
```

```
⟨Instr creation functions 12⟩

⟨Instr field accessors 14a⟩

⟨Instr predicates 13⟩

⟨Instr print function 14b⟩

⟨InstrList functions 15a⟩

⟨InstrList function nicknames 15c⟩

⟨InstrList print function 16a⟩

#endif /* MACHINE_INSTR_H */
```

4 Machine Operands

4.1 `Opnd` interface

An instruction operand in Machine SUIF is an instance of class `Opnd`. Whereas you always deal with instructions using a pointer (of type `Instr*`), an operand is treated as a non-pointer value. Thus operand creation functions are not called `new_...`; their names all have the prefix `opnd_`. They return an `Opnd`, not an `Opnd*`, and you don't need to worry about `delete`-ing operands when you're finished with them.

That's not to say that operands cannot have reference behavior. As mentioned earlier, class `Opnd` encapsulates a pointer to a SUIF object. If you insert a mutable operand into multiple instructions without cloning it, then a side effect on one occurrence will affect the others. You must avoid this if you want to reuse OPI code in settings other than Machine SUIF.

Common functions on operands. There are several kinds of operands, and most of the operand-related functions have to do with one particular kind or another. Before going through the kinds, we describe some functions that can be applied to any operand.

To ask what kind of operand is represented by a particular `Opnd` value (e.g., register operand versus address-symbol operand), you use the `get_kind` function, which returns an integer indicator. As we go through the operand kinds in the rest of this section, we'll give the symbolic names for the various operand-kind indicators.

17a ⟨`Opnd` *common functions* 17a⟩≡ (30) 17b ▷

```
int get_kind(Opnd);
```

There are also predicates for testing whether an operand is of a particular kind. Although they are applicable to any operand, we'll also list these predicates as we go through the kinds.

Every operand has a type, i.e., a `TypeId` that gives the type of the value that the operand represents. You use the `get_type` function to obtain this type.

17b ⟨`Opnd` *common functions* 17a⟩+≡ (30) ◁ 17a 18a ▷

```
TypeId get_type(Opnd);
```

The other function that all operands have in common is a replication utility.

18a ⟨Opnd *common functions* 17a⟩+≡ (30) ◁17b 18b▷
```
Opnd clone(Opnd);
```

`clone` returns its argument unchanged unless it is of a kind that has mutable fields. (Currently, only the address-expression operands have that property.) In this case, it returns a copy of the argument operand, one whose fields can be changed without affecting the original.

Two operands are equal under operator `==` only if their kind and type attributes are equal and they satisfy additional conditions. These conditions are spelled out below. The disequality operator (`!=`) just inverts the equality result.

Because you sometimes need to map from operands to other types of values, we define a function that extracts a hash code from an operand.

18b ⟨Opnd *common functions* 17a⟩+≡ (30) ◁18a 18c▷
```
size_t hash(Opnd);
```

For debugging purposes, there is a function for printing operands in a machine-independent manner. This is not the way operands are printed in machine-specific assembly output, of course. See Section 8 to understand more about of how that works.

18c ⟨Opnd *common functions* 17a⟩+≡ (30) ◁18b
```
void fprint(FILE*, Opnd);
```

Now let's go through the different kinds of operands and the functions that relate to each kind.

Null operands. These are simply placeholders. They always have `void` type. Any null operand is equal to another one. There are two OPI functions for null operands:

18d ⟨*null-operand functions* 18d⟩≡ (30)
```
Opnd opnd_null();
bool is_null(Opnd);
```

Synopses:

> `opnd_null()` returns a null operand.
> `is_null(opnd)` returns `true` if `opnd` is null.

In addition, the default constructor for class `Opnd` is publicly accessible and yields a null operand.

The kind identifier for a null operand is `opnd::NONE`.

Variable-symbol operands. These are occurrences of source-language variables or compiler-generated temporary variables. (Not to be confused with virtual registers, which do not involve a symbol.)

The OPI has the following functions for variable-symbol operands:

18e ⟨*variable-symbol-operand functions* 18e⟩≡ (30)
```
Opnd opnd_var(VarSym* var);
bool is_var(Opnd);
VarSym* get_var(Opnd);
```

Synopses:

> `opnd_var(var)` returns a variable-symbol operand referring to `var`.
> `is_var(opnd)` returns `true` if `opnd` is a variable-symbol operand.
> `get_var(opnd)` returns the variable symbol embedded in `opnd`.

The kind identifier for a variable-symbol operand is `opnd::VAR`. Two variable-symbol operands are equal if and only if their embedded symbols are equal. Since the type of this kind of operand is taken from the type of the embedded variable-symbol, this definition of equality implicitly requires equal types.

Register operands. These represent machine registers, both the real architectural registers of the target machine and any virtual registers (sometimes called pseudo-registers) created by the compiler prior to register allocation. Each contains a non-negative register number and an explicit type. The real hardware registers have numbers that can be interpreted via the register description of the target machine (see Section 7).

The OPI provides the following functions for register operands:

19a ⟨*register-operand functions* 19a⟩≡ (30)

```
Opnd opnd_reg(int reg, TypeId, bool is_virtual = false);
Opnd opnd_reg(TypeId);
bool is_reg(Opnd);
bool is_hard_reg(Opnd);
bool is_virtual_reg(Opnd);
int get_reg(Opnd);
```

Synopses:

> `opnd_reg(reg, type, is_virtual)` returns an operand for register `reg`.
> `opnd_reg(type)` returns a new virtual register operand.
> `is_reg(opnd)` returns `true` if `opnd` is a register.
> `is_hard_reg(opnd)` returns `true` if `opnd` is a hardware register.
> `is_virtual_reg(opnd)` returns `true` if `opnd` is a virtual register.
> `get_reg(opnd)` returns the register number of `opnd`.

The usual way to create a virtual-register operand is simply to provide its type; in this case, an unused virtual number is allocated and incorporated in the new operand. To create a hard-register operand, you supply the register number and the type. Occasionally, you need to create a virtual-register operand with a specific number. In that case, you provide the number, the type, and the Boolean `true` when calling `opnd_reg`.

The kind identifier for hardware-register operands is `opnd::REG_HARD`. The kind identifier for virtual-register operands is `opnd::REG_VIRTUAL`. Two register operands are equal if and only if they are of the same kind, and their `reg` and `type` attributes are are pairwise equal.

Immediate operands. There are two kinds of immediate operands: one holds integers and the other strings. Integer immediates can represent either signed or unsigned integer immediate values of any magnitude. The accompanying type gives the intended precision. For floating-point immediate operands, we use strings describing the numeric value, again paired with a type. We also use untyped immediate string operands, though they don't appear in operate instructions. They allow the pseudo-instruction (`dot`) class, which sometimes requires string literals, to use the same operand interface as the other instruction classes.

The OPI functions for immediate operands are as follows:

19b ⟨*immediate-operand functions* 19b⟩≡ (30)
```
Opnd opnd_immed(int, TypeId);
Opnd opnd_immed(Integer, TypeId);
Opnd opnd_immed(IdString, TypeId);
Opnd opnd_immed(IdString);

bool is_immed(Opnd);
bool is_immed_integer(Opnd);
bool is_immed_string(Opnd);

int get_immed_int(Opnd);
Integer get_immed_integer(Opnd);
IdString get_immed_string(Opnd);
```

> `opnd_immed(integer, type)` returns an integer immediate operand with type `type`.
> `opnd_immed(string, type)` returns a floating immediate operand with type `type`.
> `opnd_immed(string)` returns a string immediate operand, with no type.
> `is_immed(opnd)` returns `true` if `opnd` is an immediate.
> `is_immed_integer(opnd)` returns `true` if `opnd` is an integer immediate.
> `is_immed_string(opnd)` returns `true` if `opnd` is a string immediate.
> `get_immed_int(opnd)` returns the value of an integer immediate operand as a C `int`, or raises an exception if that's impossible.
> `get_immed_integer(opnd)` returns the value of an integer immediate operand.
> `get_immed_string(opnd)` returns the value of a string immediate operand.

Note that, when operand `opnd` satisfies `is_immed_string(opnd)`, you can tell whether it's numeric or not by checking its type: `get_type(opnd)` returns `0` if `opnd` represents a simple string literal. (In practice, the nature of the immediate operand is nearly always apparent from the context.)

The kind identifier for an integer-immediate operand is `opnd::IMMED_INTEGER`; that for a string-immediate operand is `opnd::IMMED_STRING`. Two immediate operands are equal if and only if they are of the same kind and their `type` and `value` attributes are pairwise equal.

Address operands. The remaining kinds of builtin operands are address symbols and address expressions, which together we call *address operands*. An address operand represents the effective-address (EA) calculation performed during a machine instruction to generate a memory address. The type of an address operand is always a pointer type. Such an operand is created with the pointer-to-`void` type appropriate for the compilation target. (This is the same `TypeId` that results from evaluating `type_ptr`. See Section 5.)

It is often useful to be able to determine the *referent* type of an address operand, i.e., the type of the value obtained by dereferencing the address that the operand represents. Obviously, this information could be incorporated in the address-operand type if the OPI required precise pointer-type constructors. But that would entail a good deal of pointer type composition and decomposition at optimization time. So instead, address operands carry an additional type attribute called `deref_type`. You access and update it it using:

20 ⟨*address-operand functions* 20⟩≡ (30) 21a ▷
```
TypeId get_deref_type(Opnd);
void set_deref_type(Opnd, TypeId);
```

If non-null, this attribute gives the type of the dereferenced value. (We explain below when and why the referent type attribute may be null.)

There is a predicate satisfied only by an address operand:

21a ⟨*address-operand functions* 20⟩+≡ (30) ◁20
```
bool is_addr(Opnd);
```

Address symbols. The simplest kind of address operand is the address symbol, which represents the address of a variable symbol or a code label symbol.

The OPI functions for address symbols are:

21b ⟨*address-symbol-operand functions* 21b⟩≡ (30)
```
Opnd opnd_addr_sym(Sym* sym);
bool is_addr_sym(Opnd);
Sym* get_sym(Opnd addr_sym);
```

Synopses:

> `opnd_addr_sym(sym)` returns an address symbol based on `sym`.
> `is_addr_sym(opnd)` returns `true` if `opnd` is an address symbol.
> `get_sym(opnd)` returns the symbol underlying `opnd`.

The kind identifier for an address-symbol operand is `opnd::ADDR_SYM`. Two address-symbol operands are equal if and only if they have equal `sym` values. The `deref_type` of an address symbol is always derived from the underlying symbol. If it's a variable or procedure symbol, then the `deref_type` is the type of the variable or procedure. If it's a label symbol, then `get_deref_type` returns `NULL`.

Recall our earlier example of an $x86$ add-to-memory instruction, with `addr` representing the address of the memory location that is incremented by `value`. If the memory location is that of variable `foo`, then operand `addr` would be an address symbol constructed from the `VarSym*` for `foo` (which we might call `var_foo`). The code might look like this:

```
Opnd addr = opnd_addr_sym(var_foo);

append(body, new_instr_alm(addr, ADD, addr, value));
```

Address expressions. There are several varieties of address expression, corresponding to the addressing modes used in hardware. We use address operands to represent whole EA calculations. Like address symbols, all address expression operands are created with a type attribute equal to `type_ptr`.

The unique characteristics of address expressions are that they embed other operands and they are mutable: you can replace the operands that make up an address expression. For example, a "base-plus-displacement" operand represents the address that results at run time from adding a fixed displacement to the contents of a base register. Earlier, for instance, we used the example of an $x86$ stack cell located at `$esp + 40`. The corresponding address-expression operand would contain the register operand for `$esp` as its base part and an immediate operand for the number `40` as its displacement part.

A typical program transformation that could change the constituent operands of an address expression is register allocation. A base-plus-displacement operand with a virtual register as its base would be modified when the virtual register is allocated to a physical one.

Because address expressions are mutable, you should be aware that the implementation allows the mutable parts to be shared between occurrences. Machine SUIF doesn't force you to maintain separate copies, since this would often be semantically pointless. In our earlier example of the $x86$ add-to-memory instruction that updates stack location 40(%esp), we didn't bother to clone the address expression before using it a second time:

```
append(body, new_instr_alm(addr, ADD, addr, value));
```

Neither its base register %esp nor its displacement 40 will be affected by a register allocator, and in any case, the hardware sees addr as a single operand, not as two separate once. Be warned, though, that Machine SUIF doesn't guarantee to preserve sharing patterns that you may establish in this way. For example, when an instruction is *cloned*, i.e., replicated, its address expressions are replicated individually.

To produce the variable of type Opnd called addr in the above example, we might write:

```
Opnd esp  = opnd_reg(REG_esp, type_ptr);
Opnd i40  = opnd_immed(40, type_s32);
Opnd addr = BaseDispOpnd(esp, i40, type_s32);
```

Here REG_esp is an integer variable previously set to the register number for %esp (see Section 7). It is incorporated into a register operand esp with the generic pointer type that is appropriate for the current target; the OPI calls this type_ptr. Next, the displacement operand i40 is created and given a signed 32-bit integral type. Finally, the address expression addr is constructed from these base and displacement suboperands. The third argument to the constructor BaseDispOpnd is the deref_type, i.e., the type of the value in the address that addr refers to.

Sometimes an address expression is used purely for address computation; the referent type is unknown and unneeded. For those cases, the OPI allows you to omit it, which leaves it NULL. For example, in Alpha code, a stack frame is allocated by subtracting a constant from the stack-pointer register. This is usually expressed using a load-address instruction:

```
Opnd sp = opnd_reg(REG_sp, type_ptr);   // stack pointer reg ($sp)
Opnd sz = opnd_immed(-80,  type_s32);   // -(frame size)

append(body, new_instr_alm(sp, LDA, BaseDispOpnd(sp, sz)));
```

where LDA is the Alpha load-address opcode.

Machine SUIF includes these functions for testing and creating address expressions:

22a ⟨*address-expression-operand functions* 22a⟩≡ (30)
```
bool is_addr_exp(Opnd);
Opnd opnd_addr_exp(int kind);
Opnd opnd_addr_exp(int kind, TypeId deref_type);
```

is_addr_exp(opnd) returns true when opnd is an address expression.
opnd_addr_exp(kind, deref_type) returns an address expression operand of kind kind referring to a value of type deref_type.

The kind argument to opnd_addr_exp must be one associated with a specific address-expression kind, e.g., one of those in Table 1. There is no way to create a "generic" address expression.

The deref_type argument to opnd_addr_exp may be omitted, in which case the resulting operand's referent type is unspecified (and it equals zero when tested).

Programming with address expressions. The function opnd_addr_exp described above is *not* part of the OPI. It's meant for extenders of the system. In optimization passes or libraries, you should use class AddrExpOpnd and its subclasses as the interface to address expressions.

22b ⟨*class* AddrExpOpnd 22b⟩≡ (30)

```
class AddrExpOpnd : public Opnd {
  public:
    AddrExpOpnd() { }
    AddrExpOpnd(const Opnd&);
    AddrExpOpnd(Opnd&);
    AddrExpOpnd(int kind, TypeId deref_type = 0)
        : Opnd(opnd_addr_exp(kind, deref_type)) { }

    TypeId get_deref_type() const;

    int srcs_size() const;
    OpndHandle srcs_start()const ;
    OpndHandle srcs_end() const;
    Opnd get_src(int pos) const;
    Opnd get_src(OpndHandle handle) const;
    void set_src(int pos, Opnd src);
    void set_src(OpndHandle handle, Opnd src);
};
```

Here

> `AddrExpOpnd(kind, deref_type)` produces a new address expression with the given kind and referent type. (The latter is optional.)
> `srcs_size()` returns the size of the address expression's `srcs` sequence.
> `srcs_start()` gives a handle on the first element of `srcs`, if any.
> `srcs_end()` gives the end-sentinel of `srcs`.
> `get_src(p)` returns the source of the address expression at position `p`, which may be a zero-based integer or a handle.
> `set_src(p, src)` substitutes `src` for the source of the address expression at position `p` (which may be a zero-based integer or a handle).

Both `get_src` and `set_src` extend the `srcs` sequence if necessary.

Just as an instruction contains a sequence of direct source operands that we call its `srcs` field, every address expression has a `srcs` sequence, and the same functions for scanning, accessing, and changing the elements of the sequence apply. For each kind of address expression, there is a subclass of Opnd that allows referring to source operands by more descriptive field names, such as `base` or `disp`, but these fields are just elements of the overall source sequence.

Although address expressions describe how to compute the *value* of an effective address, they don't encode side effects that might be associated with a particular addressing mode, such as post-increment.

For two address expression operands to be equal under the `==` operator, they must be of the same kind and have the same `deref_type`, and their source operands must be equal in number and pairwise equal under `==`.

Specific address-expression kinds. The `AddrExpOpnd` interface is convenient for operations that are indifferent to the specific kind of an address expression. A register allocator, for instance, may make substitutions among the components of an address expression without needing to know the particular kind. For other purposes, though, a specific interface is needed for each kind of address expression that the target machine supports.

The specific kinds of address expression that the OPI defines are summarized in Table 1, which matches each kind indicator with the corresponding form of address expression.

`opnd::SYM_DISP`			*address-symbol*	+	*displacement*
`opnd::INDEX_SYM_DISP`	*index-register*	+	*address-symbol*	+	*displacement*
`opnd::BASE_DISP`	*base*			+	*displacement*
`opnd::BASE_INDEX`	*base*	+	*index*		
`opnd::BASE_INDEX_DISP`	*base*	+	*index*	+	*displacement*
`opnd::INDEX_SCALE_DISP`			*index* × *scale*	+	*displacement*
`opnd::BASE_INDEX_SCALE_DISP`	*base*	+	*index* × *scale*	+	*displacement*

where

index-register	is a register operand
address-symbol	is an address-symbol operand
base and *index*	are either register or variable-symbol operands; the latter stand for variables to be assigned to registers
scale	is an immediate operand containing an unsigned integer (typically a small power of 2)
displacement	is an immediate operand containing a signed integer

Table 1: Address expression categories.

Each of these address-expression kinds has an explicit name for each element of its `srcs` sequence, so that it can be fetched and replaced using mnemonically-named methods. Here are the field names:

`addr_sym`	an address symbol
`disp`	an integer immediate
`index`	a register operand
`base`	a register or variable-symbol operand
`scale`	an integer immediate

For each kind of address expression, we now describe the subclass of `Opnd` that gives access to the applicable fields through its methods. For each of these kinds, there is an operand-kind identifier, a creation function (prefix `opnd_`) and a predicate (prefix `is_`).

The *address-symbol*+ *displacement* kind. This address form represents a fixed displacement (`disp`) from the memory address of a symbol (`addr_sym`). Its `Opnd` subclass is `SymDispOpnd`. Its kind identifier is `opnd::SYM_DISP`.

Class `SymDispOpnd` gives the methods for composing, inspecting and changing symbol-plus-displacement operands:

24 ⟨*class* SymDispOpnd 24⟩≡ (30)

```
class SymDispOpnd : public AddrExpOpnd {
  public:
    SymDispOpnd(Opnd addr_sym, Opnd disp, TypeId deref_type = 0);
    SymDispOpnd(const Opnd&);
    SymDispOpnd(Opnd&);

    Opnd get_addr_sym() const;
    void set_addr_sym(Opnd);
    Opnd get_disp() const;
    void set_disp(Opnd);
};

bool is_sym_disp(Opnd);
```

The *index-register* + *address-symbol* + *displacement* kind. This address form combines the run-time value of an index register with a fixed displacement and the memory address of a symbol. Its kind identifier is `opnd::INDEX_SYM_DISP`.

Class `IndexSymDispOpnd` gives the methods for composing, inspecting and changing index-plus-symbol-plus-displacement operands:

25a ⟨*class* `IndexSymDispOpnd` 25a⟩≡ (30)

```
class IndexSymDispOpnd : public AddrExpOpnd {
  public:
    IndexSymDispOpnd(Opnd index, Opnd addr_sym, Opnd disp,
                     TypeId deref_type = 0);
    IndexSymDispOpnd(const Opnd&);
    IndexSymDispOpnd(Opnd&);

    Opnd get_index() const;
    void set_index(Opnd);
    Opnd get_addr_sym() const;
    void set_addr_sym(Opnd);
    Opnd get_disp() const;
    void set_disp(Opnd);
};

bool is_index_sym_disp(Opnd);
```

The *base* + *displacement* kind. This address form combines the value of a base register with a fixed displacement. Its `Opnd` subclass is `BaseDispOpnd`. Its kind identifier is `opnd::BASE_DISP`. The base operand may either be a register operand or a variable symbol that represents a register candidate.

Class `BaseDispOpnd` gives the methods for composing, inspecting and changing base-plus-displacement operands:

25b ⟨*class* `BaseDispOpnd` 25b⟩≡ (30)

```
class BaseDispOpnd : public AddrExpOpnd {
  public:
    BaseDispOpnd(Opnd base, Opnd disp, TypeId deref_type = 0);
    BaseDispOpnd(const Opnd&);
    BaseDispOpnd(Opnd&);

    Opnd get_base() const;
    void set_base(Opnd);
    Opnd get_disp() const;
    void set_disp(Opnd);
};

bool is_base_disp(Opnd);
```

The *base* + *index* kind. This address form combines the values two registers, base and index. Its `Opnd` subclass is `BaseIndexOpnd`. Its kind identifier is `opnd::BASE_INDEX`.

Class `BaseIndexOpnd` gives the methods for composing, inspecting and changing base-plus-index operands:

25c ⟨*class* BaseIndexOpnd 25c⟩≡ (30)
```
class BaseIndexOpnd : public AddrExpOpnd {
  public:
    BaseIndexOpnd(Opnd base, Opnd index, TypeId deref_type = 0);
    BaseIndexOpnd(const Opnd&);
    BaseIndexOpnd(Opnd&);

    Opnd get_base() const;
    void set_base(Opnd);
    Opnd get_index() const;
    void set_index(Opnd);
};

bool is_base_index(Opnd);
```

The *base* + *index* + *displacement* kind. This address form combines two registers (base and index) with a fixed displacement. Its `Opnd` subclass is `BaseIndexDispOpnd`. Its kind identifier is `opnd::BASE_INDEX_DISP`.

Class `BaseIndexDispOpnd` gives the methods for composing, inspecting and changing base-plus-index-plus-displacement operands:

26a ⟨*class* BaseIndexDispOpnd 26a⟩≡ (30)
```
class BaseIndexDispOpnd : public AddrExpOpnd {
  public:
    BaseIndexDispOpnd(Opnd base, Opnd index, Opnd disp, TypeId deref_type = 0);
    BaseIndexDispOpnd(const Opnd&);
    BaseIndexDispOpnd(Opnd&);

    Opnd get_base() const;
    void set_base(Opnd);
    Opnd get_index() const;
    void set_index(Opnd);
    Opnd get_disp() const;
    void set_disp(Opnd);
};

bool is_base_index_disp(Opnd);
```

The *index* × *scale* + *displacement* kind. This address form multiplies an index register's value by a fixed scale factor and adds a fixed displacement. Its `Opnd` subclass is `IndexScaleDispOpnd`. Its kind identifier is `opnd::INDEX_SCALE_DISP`.

26b ⟨*class* IndexScaleDispOpnd 26b⟩≡ (30)
```
class IndexScaleDispOpnd : public AddrExpOpnd {
  public:
    IndexScaleDispOpnd(Opnd index, Opnd scale, Opnd disp,
                       TypeId deref_type = 0);
    IndexScaleDispOpnd(const Opnd&);
    IndexScaleDispOpnd(Opnd&);

    Opnd get_index() const;
    void set_index(Opnd);
    Opnd get_scale() const;
```

```
    void set_scale(Opnd);
    Opnd get_disp() const;
    void set_disp(Opnd);
};

bool is_index_scale_disp(Opnd);
```

The *base* + *index* × *scale* + *displacement* kind. This address form combines the value of a base register with the result of scaling an index register and adding a fixed displacement. Its `Opnd` subclass is `BaseIndexScaleDispOpnd`. Its kind identifier is `opnd::BASE_INDEX_SCALE_DISP`.

27a ⟨*class* `BaseIndexScaleDispOpnd` 27a⟩≡ (30)

```
class BaseIndexScaleDispOpnd : public AddrExpOpnd {
  public:
    BaseIndexScaleDispOpnd(Opnd base, Opnd index, Opnd scale, Opnd disp,
                           TypeId deref_type = 0);
    BaseIndexScaleDispOpnd(const Opnd&);
    BaseIndexScaleDispOpnd(Opnd&);

    Opnd get_base() const;
    void set_base(Opnd);
    Opnd get_index() const;
    void set_index(Opnd);
    Opnd get_scale() const;
    void set_scale(Opnd);
    Opnd get_disp() const;
    void set_disp(Opnd);
};

bool is_base_index_scale_disp(Opnd);
```

4.2 Class `OpndCatalog`

It is often useful in program analysis and optimization to map operands to a dense range of natural numbers. Class `OpndCatalog` is an abstract interface for such a map. For different problems, it is realized by different concrete subclasses. In bit-vector data-flow problems, the numbers assigned to operands are often called *slots*, meaning positions in the bit vectors. Here we'll call them *indices*.

Every `OpndCatalog` maps operands to integer indices, but not every one has to store the inverse map, which gives the operand corresponding to an index. The decision whether to record the inverse map is made when the catalog object is constructed. One benefit of keeping the inverses is that they allow the catalog's contents to be printed. So even when a particular optimization algorithm doesn't need the inverse map, its implementation may save it anyway during debugging, in order to activate the catalog's `print` method.

27b ⟨*class* `OpndCatalog` 27b⟩≡ (30)

```
class OpndCatalog {
  public:
    virtual ~OpndCatalog();

    virtual int size() const = 0;
            int num_slots() const { return size(); }    // deprecated

    virtual bool enroll(Opnd, int *index = NULL) = 0;
```

```
    virtual bool lookup(Opnd, int *index = NULL) const = 0;

    virtual void print(FILE* = stdout) const;
    virtual Opnd inverse(int index) const;

  ⟨OpndCatalog protected parts 29d⟩
  };
```

Here's what the above methods do.

`size()`	Returns the total number of indices allocated so far for all enrolled operands.
`num_slots()`	A deprecated synonym for `size()`.
`enroll(`o`, `i`)`	Tries to put operand o under management. Returns `true` exactly when o has not been enrolled before and is entered successfully (not filtered out). In addition, the optional index pointer i serves as an output variable. If i is non-null, the index for operand o (if it has one) is stored in the location that i points to. This return of o's index via i occurs whether o is newly enrolled or was in the catalog previously.
`lookup(`o`, `i`)`	Returns `true` if operand o already in the catalog. In that case, and if the optional index pointer i is non-null, the index for o is stored into the location i points to. Never makes a new entry in the catalog.
`inverse(`i`)`	If the catalog holds the optional inverse map of indices to operands, `inverse` returns the operand associated with index i, which must be less than the number of operands enrolled. Otherwise, it returns a null operand.
`print(`s`)`	If the catalog records the inverse map of operands enrolled, `print` lists each member in order of entry on the output stream s, preceded by the index assigned to it.

4.3 `Opnd` implementation

28a ⟨`Opnd` *definition* 28a⟩≡ (30)

```
  class Opnd {
    public:
      Opnd();
      Opnd(const Opnd &other) { o = other.o; }
      Opnd(IrOpnd *other_o) { o = other_o; }
      Opnd & operator=(const Opnd &other) { o = other.o; return *this; }
      ~Opnd() { }

      bool operator==(const Opnd &other) const;
      bool operator!=(const Opnd &other) const { return !(*this == other); }

      operator IrOpnd*() { return o; }
      operator bool();

    protected:
      IrOpnd *o;
  };
```

Type `OpndHandle` is used for scanning the `srcs` field of an address expression. The implementation of this field has type `suif_vector<IrOpnd>`. Therefore, `OpndHandle` is an alias for the iterator type associated with this sequence type.

28b ⟨OpndHandle *definition* 28b⟩≡ (30)
```
typedef suif_vector<IrOpnd*>::iterator OpndHandle;
```

Current optimization unit. To be able to record operands in the appropriate symbol table and to manage virtual register numbers, we need to focus on the current optimization unit when its processing starts and to defocus when it ends.

29a ⟨*focus functions* 29a⟩≡ (30)
```
void focus(FileBlock*);
void focus(OptUnit*);

void defocus(OptUnit*);
```

29b ⟨*scope management* 29b⟩≡ (30)
```
extern FileBlock *the_file_block;
extern ScopeTable *the_file_scope;

extern OptUnit *the_local_unit;
extern ScopeTable *the_local_scope;
```

29c ⟨*virtual-register management* 29c⟩≡ (30)
```
extern int the_vr_count;

int next_vr_number();
```

Partial implementation of `OpndCatalog`. Since `OpndCatalog` is an interface, meant to be subclassed, it has no public constructor. Its protected constructor takes an option Boolean argument `record` that, when explicitly set to `true`, causes the catalog to remember the inverse map from indices to operands. This map is represented as a pointer to a vector. The pointer is `NULL` when the inverse map is not saved. A second optional argument is just for efficiency when the map is recorded. It specifies the initial capacity of the vector (i.e., space for elements that will be appended to it later).

The protected `enroll_inverse` method is invoked by the implementation of `enroll` when a new association is being added to the catalog. It takes care of recording the inverse map, if one is being kept.

29d ⟨OpndCatalog *protected parts* 29d⟩≡ (27b)
```
    protected:
      OpndCatalog(bool record = false, unsigned roll_reserve = 100);

      void enroll_inverse(unsigned index, Opnd);

    private:
      Vector<Opnd> *roll;
```

Assess storage consumption by operand objects.

29e ⟨*storage measurement* 29e⟩≡ (30)
```
void audit_opnds(FILE*, const char *heading);
```

4.4 Header file `opnd.h`

The header file for operands has the following outline:

30 ⟨*machine/opnd.h* 30⟩≡

```
/* file "machine/opnd.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_OPND_H
#define MACHINE_OPND_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "machine/opnd.h"
#endif

#include <machine/substrate.h>

class IrOpnd;

namespace opnd {

enum { NONE,
       REG_HARD,
       REG_VIRTUAL,
       VAR,
       IMMED_INTEGER,
       IMMED_STRING,
       ADDR_SYM,
       SYM_DISP,
       INDEX_SYM_DISP,
       BASE_DISP,
       BASE_INDEX,
       BASE_INDEX_DISP,
       INDEX_SCALE_DISP,
       BASE_INDEX_SCALE_DISP,
};

} // namespace opnd

#define LAST_OPND_KIND opnd::BASE_INDEX_SCALE_DISP

⟨OpndHandle definition 28b⟩

⟨Opnd definition 28a⟩

⟨Opnd common functions 17a⟩

⟨null-operand functions 18d⟩

⟨variable-symbol-operand functions 18e⟩

⟨register-operand functions 19a⟩
```

⟨*immediate-operand functions* 19b⟩

⟨*address-operand functions* 20⟩

⟨*address-symbol-operand functions* 21b⟩

⟨*address-expression-operand functions* 22a⟩

⟨*class* `AddrExpOpnd` 22b⟩

⟨*class* `SymDispOpnd` 24⟩

⟨*class* `IndexSymDispOpnd` 25a⟩

⟨*class* `BaseDispOpnd` 25b⟩

⟨*class* `BaseIndexOpnd` 25c⟩

⟨*class* `BaseIndexDispOpnd` 26a⟩

⟨*class* `IndexScaleDispOpnd` 26b⟩

⟨*class* `BaseIndexScaleDispOpnd` 27a⟩

⟨*focus functions* 29a⟩

⟨*scope management* 29b⟩

⟨*storage measurement* 29e⟩

⟨*virtual-register management* 29c⟩

⟨*class* `OpndCatalog` 27b⟩

```
#endif /* MACHINE_OPND_H */
```

5 Types

The parlance of compiler infrastructure overloads a number of terms, such as "type", "variable", and "procedure", which makes it difficult to talk clearly about the implementation of a system like Machine SUIF. When we say "OPI type" or "SUIF type" or "C++ type", we are talking about artifacts of our compiler's implementation. But often we use "type" to mean either a property of the program being compiled, or the object in the compiler that represents that property. As the length of the preceding sentence indicates, we don't have nice crisp terminology for the latter type of types, except to call them "source types", which isn't great. It might work to say "type object" when referring to elements of the infrastructure that represent types in the program being compiled, but the OPI attempts to make a distinction between IR objects, which are accessed indirectly via explicity pointers, and potentially lighter-weight things like operands and types, which might not need to involve pointers.

The OPI uses the term *type identifier* (and the name `TypeId`) for this lightweight type representation.

The SUIF system provides us with a rich type system. As much as possible, we attempt to maintain type information when converting from a SUIF IR to the Machine-SUIF IR and when performing optimizations in Machine SUIF. Often, however, we are interested only in some simple type information. For these times, we provide several predefined type variables for your use.

The first set of predefined type variables are those that are target independent. We declare these type variables in ⟨*target-independent types* 32⟩. Each of these types is aligned.

- `type_v0` describes a void value of size 0 bits.
- `type_s8`, `type_s16`, `type_s32`, and `type_s64` describe aligned, signed integers of the specified bit sizes.
- `type_u8`, `type_u16`, `type_u32`, and `type_u64` describe aligned, unsigned integers of the specified bit sizes.
- `type_f32`, `type_f64`, and `type_f128` describe aligned, floating-point values of the specified bit sizes.

The second set are type objects whose definitions depend upon a specific target machine.

- `type_addr` is a function that returns the type "pointer to `type_v0`". The size of this type depends upon the target. This occurs so frequently in code, that we define the macro `type_ptr` to be equivalent to `type_addr()`. As mentioned earlier, this is the type that we use as the type of an address expression.

Target-independent type objects. The following are predefined type variables that are globally available to any Machine-SUIF pass. These target-independent types are aligned. They are initialized in `machine/init.cc`.

32 ⟨*target-independent types* 32⟩≡ (33b)
```
extern TypeId type_v0;  // void
extern TypeId type_s8;  // signed ints
extern TypeId type_s16;
extern TypeId type_s32;
extern TypeId type_s64;
extern TypeId type_u8;  // unsigned ints
extern TypeId type_u16;
extern TypeId type_u32;
extern TypeId type_u64;
extern TypeId type_f32; // floats
extern TypeId type_f64;
extern TypeId type_f128;
extern TypeId type_p32; // pointers
extern TypeId type_p64;

void attach_opi_predefined_types(FileSetBlock*);
void set_opi_predefined_types(FileSetBlock*);
```

Function `set_opi_predefined_types` fills in the target-independent types, which it obtains from the `generic_types` annotation of a `file_set_block`. This note is created once at the time the Machine-SUIF representation of a file set is first created (by function `attach_opi_predefined_types`), and it is carried along as the file set is transformed. Using this note is an efficient way to avoid creating redundant types in a global symbol table.

A pass typically calls `set_opi_predefined_types` during its `do_file_set_block` method.

Target-dependent type object. For now Machine SUIF has one target-dependent type parameter, namely the type of a generic pointer or address on the target machine. You fetch it from the target context by calling `type_addr()`. To permit a uniform style between the target-independent and the target-dependent types, we define a macro, `type_ptr` that expands to `type_addr()`. Thus in both cases a reference looks like a simple constant.

33a ⟨*target-dependent type* 33a⟩≡ (33b)
```
#define type_ptr type_addr()

TypeId type_addr();
```

5.1 Header file for module `types.h`

The interface file has the following layout:

33b ⟨*machine/types.h* 33b⟩≡
```
/* file "machine/types.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_TYPES_H
#define MACHINE_TYPES_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "machine/types.h"
#endif

#include <machine/substrate.h>

⟨target-independent types 32⟩

⟨target-dependent type 33a⟩

void fprint(FILE*, TypeId);

#endif /* MACHINE_TYPES_H */
```

6 Machine opcodes

In general, a machine opcode value uniquely identifies a particular operation on a particular machine. On different machines, a specific value will (most-likely) correspond to different operations. In other words, an opcode value does not encode any target information beyond the target-relative operation. You must interpret the opcode value in the context of a particular target. The same interpretation rules apply for Machine-SUIF opcodes.

In Machine SUIF, an `opcode` is an integer. When you define an opcode for a machine instruction, you must use the opcode enumeration appropriate for your current target. Each target architecture defines its own extensible opcode `enum` in an `opcodes.h` file. All targets of the same architecture family use the same opcode `enum`. We provide an OPI function, called `target_implements`, that allows you to determine if an opcode is supported on your current target.

33c ⟨*opcode OPI* 33c⟩≡ (35c) 34a ▷
```
bool target_implements(int opcode);
```

In Machine SUIF, an opcode has an associated *name*. It is the string by which the current target's assembler recognizes the opcode. The opcode-to-name mapping is one-to-many since an opcode's name may vary from one vender-OS to another. We provide an OPI function, called `opcode_name`, that you can use to get an opcode's target-appropriate name.

34a ⟨*opcode OPI* 33c⟩+≡ (35c) ◁33c 34b▷

```
char* opcode_name(int opcode);
```

Given an opcode `enum`, if you know that you want to generate SUIFvm instructions, you can write:

```
Instr *mi = new_instr_alm(d_opnd, MOV, s_opnd);
```

Or, if you know that you want to generate an Alpha integer move instruction, you can write:

```
Instr *mi = new_instr_alm(d_opnd, MOV, s_opnd);
```

In general, having knowledge of an architecture's opcode `enum` is sufficient to write a target-specific pass or function, but not very useful for the development of parameterized passes. To create an instruction that is specific to a target without having to know the target at pass-development time, the OPI provides some target-specific opcode generators.

For example, `opcode_move` returns the move opcode appropriate for your current target:

```
Instr *mi = new_instr_alm(d_opnd, opcode_move(d_type), s_opnd);
```

This generator is called a typed opcode generator because you must provide a type to get a type-appropriate opcode. In other words, `opcode_move` uses the `d_type` parameter to determine if it should generate an integer or floating-point move opcode.

The OPI calls for provides three typed opcode generators:

34b ⟨*opcode OPI* 33c⟩+≡ (35c) ◁34a 34c▷

```
int opcode_move(TypeId);
int opcode_load(TypeId);
int opcode_store(TypeId);
```

where

- `opcode_move`: generates a type- and target-appropriate move opcode;
- `opcode_load`: generates a type- and target-appropriate load opcode;
- `opcode_store`: generates a type- and target-appropriate store opcode;

and the following untyped opcode generators:

34c ⟨*opcode OPI* 33c⟩+≡ (35c) ◁34b 35a▷

```
int opcode_line();
int opcode_ubr();
```

where

- `opcode_line`: generates a target-appropriate line pseudo-op opcode.
- `opcode_ubr`: generates a target-appropriate unconditional branch opcode.

The OPI also provides the function `opcode_cbr_inverse` which takes a conditional branch opcode and returns the opcode checking the opposite condition.

35a ⟨*opcode OPI* 33c⟩+≡ (35c) ◁34c 35b▷

```
int opcode_cbr_inverse(int opcode);
```

Finally, there are two global opcodes, which are the same across all architectures:

35b ⟨*opcode OPI* 33c⟩+≡ (35c) ◁35a

```
const int opcode_null = 0;
const int opcode_label = 1;
```

Since these are considered part of each architecture's opcode space, the first target-specific opcode in each opcode `enum` should start with value 2.

6.1 Header file for module `opcodes.h`

The interface file has the following layout:

35c ⟨*machine/opcodes.h* 35c⟩≡

```
/* file "machine/opcodes.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_OPCODES_H
#define MACHINE_OPCODES_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "machine/opcodes.h"
#endif

#include <machine/substrate.h>

⟨opcode OPI 33c⟩

int opcode_from_name(const char *name);

#endif /* MACHINE_OPCODES_H */
```

7 Register Descriptions

A Machine SUIF register allocator should be coded without hard-wiring the particulars of any target machine, but it nevertheless needs a way to access information about the target's registers. This section gives the register-description interface that you must implement when developing a target library. Its purpose is to enumerate the hardware registers, and for those that are subject to register allocation, to classify them according to the different purposes they serve in the instruction-set architecture.

A register *class* is a set of registers that are equally suitable for playing a particular role in programs of the target machine. For example, the set of registers that can be operands of an integer `add` instruction comprise a register class. The members of a class are interchangeable: whenever one of them is playing the role for which the class is defined (e.g., the `add`-operand role), then it is legal to use any other member of the class in its place. Usually, the same register class is associated with many related instructions, e.g., the full set of integer arithmetic and logical instructions. But this is not always the case. The Motorola 68000, for instance, has a class of address registers for addressing memory and a class of data registers for doing certain integer computations. A data register can't be the base register for a load instruction, and an address register can't be the operand of a multiply instruction, so there are disjoint classes for those distinct roles. However, the add instruction can operate on either an address register or a data register, so the machine has a third register class that is the union of the other two.

The register-description interface doesn't involve a C++ class of its own. It is made up of functions, all of whose names begin with `reg_`. These correspond to virtual methods in the context interface of the `machine` library, i.e., in class `MachineContext` (see Section 15). The target library provides implementations for those methods.

7.1 Enumerating hardware registers

A register is identified by a non-negative integer whose particular value is only meaningful within the target library.

- `reg_count` returns the number of registers in the target. Every register identifier must be less than this number.
- `reg_name` returns the name of the register designated by its argument. The is the form that the target assembler expects to see.
- `reg_width` returns the size in bits of the designated register.
- `reg_maximal`, returns the maximal-width register that subsumes the register given as its argument.
- `reg_aliases` returns the set of registers that overlap the designated register in whole or in part, including the register itself. Register x is in `reg_aliases(`y`)` if and only if writing a value into y can affect the value in x.
- `reg_allocables` returns the set of identifiers for all registers that are available for register allocation. When called with its optional argument `true`, it returns the subset that have maximal width.
- `reg_caller_saves` returns the subset of allocable registers that obey a *caller-saves* convention, i.e., that need not be preserved by a called procedure. With optional argument `true`, it returns only the maximal-width caller-saves registers.
- `reg_callee_saves` returns the subset of allocable registers that obey a *callee-saves* convention, i.e., that must be restored by a called procedure if it changes them. With optional argument `true`, it returns only the maximal-width callee-saves registers.
- `reg_info_print` prints a description of the registers of the current target.

37a ⟨*register description functions* 37a⟩≡ (39b)
```
int reg_count();
const char* reg_name(int reg);
int reg_width(int reg);
int reg_maximal(int reg);
const NatSet* reg_aliases(int reg);
const NatSet* reg_allocables(bool maximals = false);
const NatSet* reg_caller_saves(bool maximals = false);
const NatSet* reg_callee_saves(bool maximals = false);

void reg_info_print(FILE*);
```

7.2 Enumerating register classes

Every register class is also identified by a non-negative integer that we call the class *identifier*.

37b ⟨*type* `RegClassId` 37b⟩≡ (39b)
```
typedef int RegClassId;
```

The set of classes must be closed under intersection: if two classes have one or more members in common, then their intersection must also be a class that is identified in this machine description. The intersection function recognizes two distinguished class-identifier values. `REG_CLASS_ANY` is the identity element for class intersection, and `REG_CLASS_NONE` is the zero element.

37c ⟨*distinguished class identifiers* 37c⟩≡ (39b)
```
const RegClassId REG_CLASS_ANY  = -1;   // universal class
const RegClassId REG_CLASS_NONE = -2;   // empty class
```

The following functions describe register classes and their relationships.

- `reg_class_count` returns the number of classes. Every class identifier must be less than this number.
- `reg_members` returns a set of register numbers for the members of the designated register class.
- `reg_class_intersection` returns the identifier of the class that is the intersection of its two argument classes. If one argument is `REG_CLASS_ANY`, it returns the other argument. If the intersection is empty, it returns `REG_CLASS_NONE`.

37d ⟨*register class description functions* 37d⟩≡ (39b)
```
int reg_class_count();
const NatSet* reg_members(RegClassId);
RegClassId reg_class_intersection(RegClassId, RegClassId);
```

7.3 Supporting register allocation

Classification. A register allocator needs a way to classify the register-candidate operands of an instruction in order to decide what registers can legally be assigned to each candidate. That is, it wants a map from operands to register-class identifiers that tells it what role each register operand is playing. For example, the effective-address operand of a Motorola 68000 load instruction might be a register candidate. The class map would show that operand's class to be the address-register class.

Instead of mapping directly from operands to class identifiers, the class map is keyed on the operand indices assigned by an operand catalog (see Section 4.2). So it is just a vector of class identifiers, indexed by operand number.

37e ⟨*type* RegClassMap 37e⟩≡ (39b)

```
typedef Vector<RegClassId> RegClassMap;
```

A register candidate normally appears in more than one instruction, and its class should reflect the most constrained role that it plays. In other words, it is the intersection of the classes ascribed to it in the various places where it appears. The process of classification therefore starts by intializing every element of the class map to `REG_CLASS_ANY`, and at each occurrence of a candidate operand, the corresponding map entry is replaced by the intersection of its current value and the class determined by the use of the operand. When the classification id finished, every map entry for a register candidate should neither be the empty class nor the original universal class.

- `reg_classify` scans the operands of an instruction, using an operand catalog to obtain an integer index for each register candidate. It updates a class map that gives the feasible register classes for each operand.

38a ⟨*allocator support functions* 38a⟩≡ (39b) 38b ▷

```
void reg_classify(Instr*, OpndCatalog*, RegClassMap*);
```

Selection. The allocator also needs an efficient method for selecting one register from the class of a candidate while observing some extra constraints on the choice. One kind of constraint is that the register must come from the pool of caller-saves or callee-saves registers. Another is that it cannot already have been assigned to a conflicting candidate. The final kind of constraint is that the register should if possible be the same as registers chosen previously, or perhaps, on the contrary, it should be different from registers chosen recently. To understand this seemingly quirky final constraint, realize that the allocator wants to assign as few callee-saves registers as it can get away with, since they entail some overhead. On the other hand, it is sometimes best to cycle through all of the caller-saves registers before assigning any one a second time. This helps minimize anti-dependences that thwart instruction scheduling, for instance.

- `reg_choice` tries to select one register from a given class that is also a member of a register pool, and that is not in a set of excluded registers. Its final argument is a flag `rotate` indicating whether successive choices satisfying similar criteria should cycle through the feasible values or should be confined to as small a collection as possible. Returns a negative result if there is no register satisfying all the constraints.

38b ⟨*allocator support functions* 38a⟩+≡ (39b) ◁ 38a 39a ▷

```
int reg_choice(RegClassId, const NatSet *pool, const NatSet *excluded,
               bool rotate);
```

Spilling. The register allocator must be able to insert *spillcode*, i.e., instructions that move a value from a register to a memory location ("spilling") or from memory to register ("filling"). For many targets, a single store instruction is sufficient for spilling and likewise a single load will fill. The freedom to insert more than one instruction might be needed when it takes a separate instruction to sign-extend a value after loading it. And on some targets (such as Itanium) there are register files (e.g., branch registers) for which the ISA provides no direct load/store instructions.

The OPI functions `reg_spill` and `reg_fill` allow you to define target-specific spill-code injectors. It is usually acceptable for these functions to introduce new virtual registers, which the allocator recognizes as additional register candidates. Sometimes an allocator must call `reg_spill` or `reg_fill` after register assignments have all been made. In that case, it passes a flag (`post_reg_alloc`) indicating that no new register candidates may be mentioned in the inserted code.

- `reg_spill` inserts a sequence of instructions after `marker` whose effect is to store a spilled value from `src` (a register) to `dst` (a memory address). It returns the handle of the last instruction inserted. When the optional argument post_reg_alloc is true, the inserted sequence is not allowed to introduce register candidates, such as virtual registers.

- reg_fill inserts a sequence of instructions before marker whose effect is to load a spilled value from src (a memory address) to dst (a register). It returns the handle of the first instruction inserted. When the optional argument post_reg_alloc is true, the inserted sequence is not allowed to introduce register candidates, such as virtual registers.

When implementing either of these functions for a particular target, you should assume that the marker argument refers to an instruction that is part of a CfgNode object.[3] You can reach that CFG node using the get_parent_node function provided in the CFG library.

39a ⟨*allocator support functions* 38a⟩+≡ (39b) ◁38b

```
InstrHandle reg_fill (Opnd dst, Opnd src, InstrHandle marker,
                      bool post_reg_alloc = false);
InstrHandle reg_spill(Opnd dst, Opnd src, InstrHandle marker,
                      bool post_reg_alloc = false);
```

7.4 Upgrading from the earlier register-description interface

Prior to version 2.02.07.15, Machine SUIF used a different register-description interface. We believe the changes justify the nuisance for authors of existing target libraries by making the interface easier to explain and to use.

In the old scheme, you identified register classes like those described above, but you never got to say explicitly which registers belonged to each class. In the new version, you simply generate sets representing the membership of each class (reg_members).

In the old scheme, the alias relationships between registers had to be expressed via "models" representing register resources that might or might not correspond to architecural registers. In the new version, you just list for each register the other registers with which it has some overlap (reg_aliases).

The old scheme had vector-valued functions like reg_names. To get the name of a register, one called reg_names to get a vector and then used the register number to lookup the name in the vector. This arrangement was intended to make repeated lookups more efficient, since the vector could be saved and reused. In the new register-description interface, we eliminated the extra level of indirection, replacing reg_names with the more direct reg_name function. On the rare occasions when caching is warranted, it is easy enough to implement.

7.5 Header file for module reg_info.h

The interface file has the following layout:

39b ⟨*machine/reg_info.h* 39b⟩≡

```
/* file "machine/reg_info.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_REG_INFO_H
#define MACHINE_REG_INFO_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
```

```
#pragma interface "machine/reg_info.h"
#endif

#include <machine/substrate.h>
#include <machine/machine_ir.h>
#include <machine/nat_set.h>
#include <machine/opnd.h>

⟨type RegClassId 37b⟩

⟨type RegClassMap 37e⟩

⟨register description functions 37a⟩

⟨distinguished class identifiers 37c⟩

⟨register class description functions 37d⟩

⟨allocator support functions 38a⟩

extern int reg_lookup(const char *name);


#endif /* MACHINE_REG_INFO_H */
```

8 Assembly-Language Printing

Class `Printer` controls printing of Machine-SUIF intermediate files to assembly language. Its virtual methods are hooks that allow target-specific subclasses to customize printing according to the syntax expected by the assembler for the target system.

8.1 Class Printer

Interface. `Printer` is an abstract class. You only construct `Printer` objects for specific targets, for which there are corresponding `Printer` subclasses.

41 ⟨*class* `Printer` 41⟩≡ (43b)

```
class Printer {
  public:
    virtual ~Printer() { }

    FILE* get_file_ptr() const { return out; }
    void set_file_ptr(FILE* the_file_ptr) { out = the_file_ptr; }

    int  get_Gnum() const { return Gnum; }
    void set_Gnum(int n) { Gnum = n; }

    bool get_omit_unwanted_notes() const { return !omit_unwanted_notes; }
    void set_omit_unwanted_notes(bool omit) { omit_unwanted_notes = omit; }

    virtual void print_notes(FileBlock*);
    virtual void print_notes(Instr*);

    virtual void start_comment() = 0;

    virtual void print_instr(Instr *mi) = 0;
    virtual void print_opnd(Opnd o) = 0;

    virtual void print_extern_decl(VarSym *v) = 0;
    virtual void print_file_decl(int fnum, IdString fnam) = 0;

    virtual void print_sym(Sym *s);
    virtual void print_var_def(VarSym *v) = 0;

    virtual void print_global_decl(FileBlock *fb) = 0;
    virtual void print_proc_decl(ProcSym *p) = 0;
    virtual void print_proc_begin(ProcDef *pd) = 0;
    virtual void print_proc_entry(ProcDef *pd, int file_no_for_1st_line) = 0;
    virtual void print_proc_end(ProcDef *pd) = 0;

  ⟨Printer protected part 42⟩
};
```

Notes on the public methods:

- `get_file_ptr` and `set_file_ptr` fetch and store the output stream (`FILE*`) for the assembly file being generated.
- `get_Gnum` and `set_Gnum` fetch and store the size threshold (in bytes) at which an initialized value begins to be considered "large". Some targets put small static data in a different object-file section

from large data.

- get_omit_unwanted_notes and set_omit_unwanted_notes control whether annotation printing is selective. Normally, annotations on instructions are printed as comments in the assembly file, but those whose keys are in the set nonprinting_notes are skipped. By calling set_omit_unwanted_notes with argument false, you turn off selective printing, so that all annotations appear.

- print_notes prints the annotations on the object to which it it applied. Global annotations such as those recording the history of a file's compilation are attached to the FileBlock object. Local annotations are attached to individual instructions. Separate overloading of print_notes lets you control these cases separately.

- start_comment is called to print the opening string of an assembly-language "line" comment, i.e., one that is implicitly terminated by the next end-of-line.

- print_instr and print_opnd are the workhorse methods. They are called to print each instruction and each operand, respectively.

- print_extern_decl prints the directive that tells the assembler a symbol is externally defined, i.e., that its definition cannot be seen until link time.

- print_file_decl is called to print a directive identifying one of the source files contributing to the current object file. Its arguments are the number and the name of a source file. (The numbering gives the order of the files' appearance. For an obscure reason, it starts at 2.)

- print_sym prints a symbol reference. The default implementation distinguishes between global and local symbols. For those local to a procedure, it prepends the procedure name and a separating dot (.). Names of global symbols are printed without adornment.

- print_var_def prints the definition of a non-automatic variable, which may or may not have an initializer.

- print_global_decl is a hook for printing directives that condition the assembly of a whole file. For example, it may control the level of assembler optimization.

- print_proc_decl is a hook allowing predeclaration of a procedure (*not* the procedure's definition).

- print_proc_begin, print_proc_entry and print_proc_end are hooks for the key points in the printing of a procedure's code. The _begin hook typically adjusts the object section (e.g., with a .text directive) and perhaps the prevailing alignment. The _entry hook typically prints the procedure's prologue and the _end hook prints its epilogue.

Specialization. Each Printer object contains a dispatch table that maps from an opcode to a virtual method for printing an instruction having that opcode. This table, called print_instr_table, is initialized when the target-specific library creates an instance of its own subclass of Printer. The methods used as table entries correspond to the four kinds of instruction in the Machine SUIF implementation; their names are print_instr_alm, print_instr_cti, print_instr_dot, and print_instr_label. To allow for reuse of a target library by an extender targeting the same machine, there is an extra virtual method called print_instr_user_defd, which is called when an opcode lies outside the range covered by the particular Printer subclass. This can be defined by more-derived classes to print additional instructions.

42 ⟨Printer *protected part* 42⟩≡ (41)
```
  protected:
    // table of Instr printing functions -- filled in by derived class
    typedef void (Printer::*print_instr_f)(Instr *);
    Vector<print_instr_f> print_instr_table;

    // printing functions that populate the print_instr_table
    virtual void print_instr_alm(Instr *) = 0;
    virtual void print_instr_cti(Instr *) = 0;
    virtual void print_instr_dot(Instr *) = 0;
    virtual void print_instr_label(Instr *) = 0;

    virtual void print_instr_user_defd(Instr *) = 0;

    Printer();

    // helper
    virtual void print_annote(Annote *the_annote);

    // remaining state
    FILE *out;
    int Gnum;
    bool omit_unwanted_notes;
```

The `print_annote` protected method prints a SUIF `Annote` object, which is expected to be a `BrickAnnote`.

The remaining instance fields are exposed to subclass methods. For brevity, printing methods typically use the `out` field directly.

The `Printer` object for the current target. To access the `Printer` object in the prevailing context, call this function:

43a ⟨*function* target_printer 43a⟩≡ (43b)
```
Printer* target_printer();
```

8.2 Header file for module `Printer.h`

The interface file has the following layout:

43b ⟨*machine/printer.h* 43b⟩≡
```
/* file "machine/printer.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_PRINTER_H
#define MACHINE_PRINTER_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "machine/printer.h"
#endif
```

```
#include <machine/substrate.h>
#include <machine/opnd.h>
#include <machine/machine_ir.h>

⟨class Printer 41⟩

⟨function target_printer 43a⟩

#endif /* MACHINE_PRINTER_H */
```

9 C-Language Printing

Class `CPrinter` is analogous to `Printer` (Section 8), but its role is to help generate C-language, rather than assembly-language, output files. The structure and many of the methods of `CPrinter` are the same as those of `Printer`.

9.1 Class CPrinter

Interface. Though `CPrinter` is abstract, it has fewer methods with no default definition than class `Printer`, since the output language is fixed.

44 ⟨*class* CPrinter 44⟩≡ (47b)

```
class CPrinter {
  public:
    virtual ~CPrinter();

    void clear();
    FILE *get_file_ptr() { return out; }
    void set_file_ptr(FILE *the_file_ptr) { out = the_file_ptr; }

    bool get_omit_unwanted_notes() const { return !omit_unwanted_notes; }
    void set_omit_unwanted_notes(bool omit) { omit_unwanted_notes = omit; }

    virtual void print_notes(FileBlock*);
    virtual void print_notes(Instr*);

    virtual void print_instr(Instr *mi) = 0;
    virtual void print_opnd(Opnd o);
    virtual void print_type(TypeId t);

    virtual void print_sym(Sym *s);
    virtual void print_sym_decl(Sym *s);
    virtual void print_sym_decl(char *s, TypeId t);
    virtual void print_var_def(VarSym *v, bool no_init);

    virtual void print_global_decl(FileBlock *fb) = 0;
    virtual void print_proc_decl(ProcSym *p);
    virtual void print_proc_begin(ProcDef *pd);

⟨CPrinter protected part 46⟩
};
```

Notes on the public methods:

- get_file_ptr and set_file_ptr fetch and store the output stream (FILE*) for the C file being generated.
- get_omit_unwanted_notes and set_omit_unwanted_notes control whether annotation printing is selective. Normally, annotations on instructions are printed as comments in the output C file, but those whose keys are in the set nonprinting_notes are skipped. By calling set_omit_unwanted_notes with argument false, you turn off selective printing, so that all annotations appear.
- print_notes prints the annotations on the object to which it it applied.
- print_instr and print_opnd are called to print each instruction and each operand, respectively.
- print_type prints a C type. It keeps track of struct, union, and enum types that have already been printed and abbreviates them to avoid illegal duplication.
- print_sym prints a symbol.
- print_sym_decl prints the declarator for a symbol, which includes its name and type, but no terminating punctuation. It is used for formal parameters as well as global and local declarations.
- print_var_def prints the defining declaration of a variable symbol. If the variable has an initializer and the second argument no_init is not true, the initializer is included in the printed definition.
- print_global_decl, like the corresponding method of class Printer, is a hook for printing things at the start of the output file. If the C output file needs a special #include directive, it can be printed by this method.
- print_proc_decl prints the declaration of a procedure, including a terminating semi-colon and end-of-line string.
- print_proc_begin prints the header of a procedure definition, including the opening brace of its body.

Specialization. Like a Printer object, an instance of CPrinter contains a dispatch table that maps from an opcode to a virtual method for printing an instruction that has the opcode. This table, called print_instr_table, is initialized when the target-specific library creates an instance of its own subclass of CPrinter. The methods print_instr_alm, print_instr_cti, print_instr_dot, print_instr_label, and print_instr_user_defd are used for C-language printing in a manner that's analogous to assembly-language printing. The main difference is that print_instr_label has a definition, since label printing in C is pretty standard.

The remaining protected methods are helpers that are exposed to the target-library writer in case some aspect of the target requires special treatment when translating to C.

- print_annote method prints a SUIF Annote object, which is expected to be a BrickAnnote.
- print_immed prints an immediate operand.
- print_addr prints an address expression. Its second argument, if non-zero, is the desired referent type; the method applies a cast if necessary to achieve it. The third argument is a syntactic context indicator that helps to control parenthesization.
- print_value_block is a recursive helper for print_var_def that prints the initializer part of a variable definition.

- `print_decl` is a recursive helper for printing a declarator in C's distinctive syntax. The structure of the declarator is described in the second argument, which may include a symbolic identifier or may leave it out, for printing types.
- `print_pointer_cast` prints a cast to the pointer type that has the argument type as its referent.
- `print_type_ref` prints an abbreviated compond type, on the assumption that the full definition of the type has already been seen by the C compiler. The second argument is one of the keywords `struct`, `union`, or `enum`.
- `print_group_def` prints the full definition of a `struct` or `union` type.
- `print_enum_def` prints the full definition of an `enum` type.
- `print_atomic_type` prints a non-compound type.
- `print_string_literal` prints a string literal suitably quoted and with interior escapes for input to a C compiler.

46 ⟨CPrinter *protected part* 46⟩≡ (44)

```
protected:
  // table of instr printing functions -- filled in by derived class
  typedef void (CPrinter::*print_instr_f)(Instr*);
  print_instr_f *print_instr_table;

  // instr printing functions that populate the print_instr_table
  virtual void print_instr_alm(Instr*) = 0;
  virtual void print_instr_cti(Instr*) = 0;
  virtual void print_instr_dot(Instr*) = 0;
  virtual void print_instr_label(Instr*);

  virtual void print_instr_user_defd(Instr*) = 0;

  // helper methods

  virtual void print_annote(Annote*);

  virtual void print_immed(Opnd);
  virtual void print_addr(Opnd, TypeId goal = 0, int context = ANY);
  virtual void print_addr_disp(Opnd addr, Opnd disp, TypeId goal,
                               int context, char *op);

  virtual bool process_value_block(ValueBlock*, TypeId);

  virtual void print_decl(TypeId, const Declarator&);
  virtual void print_pointer_cast(TypeId referent);
  virtual bool print_type_ref(TypeId, const char *keyword);
  virtual void print_group_def(TypeId);
  virtual void print_enum_def (TypeId);
  virtual void print_atomic_type(TypeId);

  virtual void print_string_literal(IdString literal);

  CPrinter();

  FILE *out;
  bool omit_unwanted_notes;
```

```
    List<TypeId> noted_types;
    int next_type_tag;

    // syntactic contexts
    enum { ANY, ASSIGN, BINARY, UNARY, PRIMARY };
```

The `CPrinter` object for the current target. To access the `CPrinter` object in the prevailing context, call this function:

47a ⟨*function* `target_c_printer` 47a⟩≡ (47b)

```
  CPrinter* target_c_printer();
```

9.2 Header file for module `CPrinter.h`

The interface file has the following layout:

47b ⟨*machine/c_printer.h* 47b⟩≡

```
  /* file "machine/c_printer.h" */

  ⟨Machine-SUIF copyright 95⟩

  #ifndef MACHINE_CPRINTER_H
  #define MACHINE_CPRINTER_H

  #include <machine/copyright.h>

  #ifdef USE_PRAGMA_INTERFACE
  #pragma interface "machine/cprinter.h"
  #endif

  #include <machine/substrate.h>
  #include <machine/machine_ir.h>

  class Declarator;

  ⟨class CPrinter 44⟩

  ⟨function target_c_printer 47a⟩

  #endif /* MACHINE_CPRINTER_H */
```

10 Machine code finalization

Code finalization is the phase of a back end that allocates space in a procedure-activation record (usually a stack frame) and introduces certain prologue and epilogue code that is best deferred till the very end of compilation. The Machine-SUIF `fin` pass is responsible for this final translation work. It typically takes care of adding code to save and restore callee-saved registers, for example. Earlier translation and optimization passes use symbolic references to stack-frame locations and leave it to the `fin` pass to replace these with effective-address operands based on a stack- or frame-pointer.

This section introduces two classes used by the `fin` pass. Class `CodeFin` is the framework on which we hang the target-dependent aspects of code finalization. Its virtual methods are hooks that allow machine-specific subclasses to customize the finalization phase. Class `StackFrameInfoNote` is a custom annotation class that provides a way for earlier passes affecting the code at a procedure's entry and exits to record their effects so that the `fin` pass knows what to do. The key for notes of class `StackFrameInfoNote` is `k_stack_frame_info`.

10.1 Class `CodeFin`

We present two views of the `CodeFin` class. Readers interested in the OPI's code finalization facilities for an existing target should read only the first three subsections. The remaining subsections contain information relevant to readers interested in defining a `CodeFin` subclass for a new target.

10.1.1 OPI for code finalization

The following class defines a `CodeFin` object.

48 ⟨*class* `CodeFin` 48⟩≡ (50b)

```
class CodeFin {
  public:
    virtual ~CodeFin() { }

    virtual void init(OptUnit *unit);
    virtual void analyze_opnd(Opnd) = 0;
    virtual void layout_frame() = 0;
    virtual Opnd replace_opnd(Opnd) = 0;
    virtual void make_header_trailer() = 0;
    virtual void finalize() { }

    List<Instr*>& header()  { return _header; }
    List<Instr*>& trailer() { return _trailer; }

⟨CodeFin protected parts 50a⟩
};
```

A rundown of the methods declared above:

- `init` is called when treatment of a procedure begins. It initializes per-procedure variables.
- `analyze_opnd` is called on each operand in a pre-scan of the IR. It can be used to detect which callee-saved registers are in use, and to catalog local variables that need to be allocated in the activation record.
- `layout_frame` is the hook that performs dialect-specific layout of the activation record.
- `replace_opnd` is used in rewriting instructions to replace variable occurrences by effective addresses in the frame. Its argument is such a variable operand, and its result is the replacement operand.
- `make_header_trailer` is called to perform dialect-specific creation of prologue and epilogue code sequences. Methods `header` and `trailer` can then retrieve the respective code sequences.
- `finalize` is invoked just before processing of a procedure ends. Normally, the target library doesn't need to define this method, but it can be used for any final cleanup of the procedure's code.

10.1.2 Generating a target-specific CodeFin object

To access the `CodeFin` object in the prevailing context, call this function:

49a ⟨*function* `target_code_fin` 49a⟩≡ (50b)
```
CodeFin *target_code_fin();
```

10.1.3 Class StackFrameInfoNote

We attach a `k_stack_frame_info` note to the body of each `OptUnit`. This note has a custom class:

49b ⟨*class* `StackFrameInfoNote` 49b⟩≡ (50b)
```
class StackFrameInfoNote : public Note {
 public:
    StackFrameInfoNote() : Note(note_list_any()) { }
    StackFrameInfoNote(const StackFrameInfoNote &other)
        : Note(other) { }
    StackFrameInfoNote(const Note& note) : Note(note) { }

    bool get_is_leaf() const;
    void set_is_leaf(bool is_leaf);

    bool get_is_varargs() const;
    void set_is_varargs(bool is_varargs);

    int get_frame_size() const;
    void set_frame_size(int frame_size);

    int get_frame_offset() const;
    void set_frame_offset(int frame_offset);

    int get_max_arg_area() const;
    void set_max_arg_area(int max_arg_area);
};
```

Conceptually, a `StackFrameInfoNote` instance has the following fields:

- `is_leaf` is `true` if this procedure does not contain any call instructions.
- `is_varargs` is `true` if this procedure is a is declared with a variable number of arguments (either through `varargs.h` or `stdarg.h`).
- `framesize` records the total size of the stack frame in bytes, if known; `0` otherwise.
- `frameoffset` records the frame offset, also in bytes. It's interpretation is architecture dependent.
- `max_arg_area` records the maximum size of the call argument area in bytes for procedures that are not leaves.

The `framesize` and `frameoffset` cannot actually be known before the finalization pass, so it may seem pointless to record them in an annotation if the `fin` pass is the last to run. Once in a while, it is useful to reoptimize after finalization has taken place. In that case, a second pass of finalization is needed, and these fields of the `StackFrameInfoNote` help `fin` restart where it left off.

10.1.4 Specializing CodeFin for a target

As explained above, the CodeFin class does not provide a public constructor. This is because the base class does not actually define a CodeFin object for any specific target. You develop a CodeFin object for a specific target (or class of targets) by creating a derived class of CodeFin. The CodeFinAlpha class found in the alpha library is an example of such a derived class.

The non-public parts of CodeFin contain only instance fields for use by the public methods. There are variables to hold values from the StackFrameInfoNote during the fin pass. There is a map giving the frame location for each local symbol. There is a set per register bank that is used for collecting the registers that need to be saved, if any, and there are the lists that will carry prologue and epilogue code during finalization.

50a ⟨CodeFin *protected parts* 50a⟩≡ (48)

```
  protected:
    // Properties of current procedure, initialized by init()
    //
    OptUnit *cur_unit;

    bool is_leaf;
    bool is_varargs;
    int max_arg_area;

    Map<Sym*,int> frame_map;                    // variables -> frame offsets

  private:
    List<Instr*> _header;                       // header instructions, reversed
    List<Instr*> _trailer;                      // trailer instructions
```

10.2 Header file for module code_fin.h

The interface file has the following layout:

50b ⟨*machine/code_fin.h* 50b⟩≡

```
/* file "machine/code_fin.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_CODE_FIN_H
#define MACHINE_CODE_FIN_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "machine/code_fin.h"
#endif

#include <machine/substrate.h>
#include <machine/machine_ir.h>
#include <machine/note.h>

⟨class StackFrameInfoNote 49b⟩

⟨class CodeFin 48⟩
```

⟨*function* target_code_fin 49a⟩

```
#endif /* MACHINE_CODE_FIN_H */
```

11 Sets of Natural Numbers

Class NatSet represents sets of natural numbers. An instance of NatSet is either a finite set or the complement of a finite set. At present, there are two subclasses of NatSet that allow you to pick different space and time cost characteristics.

We view sets as not being constrained by size limits. One major client for these set classes is the BVD library for data-flow analysis [2]. In that application, it is useful to be able to begin developing data-flow information before we know the maximum number of bit-vector slots that we'll need to represent it. Furthermore, some applications of the BVD framework make incremental extensions of existing data-flow information in which they increase the sizes of the original bit vectors.

To create an instance of NatSet, you must choose one of the following subclasses. They have different underlying representations and therefore different performance characteristics.

NatSetDense — A NatSet represented using a bit vector:

- Insertion and removal of elements takes place in constant time.
- Storage is in general proportional to the value of the largest element of the finite-set part.
- Iteration produces elements in increasing order, but not necessarily in time proportional to the set cardinality.
- Union, intersection, subtraction, and equality testing can take time proportional to set storage, but they exploit bitwise parallel operations.

NatSetSparse — A NatSet represented using a sorted linked list:

- Insertion and removal of elements depends on the finite-set cardinality.
- Storage is proportional to the cardinality of the finite-set part.
- Iteration produces elements in increasing order and in time proportional to the number of elements produced.

We also declare an iterator type compatible with all of the NatSet types.

51 ⟨*class* NatSetIterPure 51⟩≡ (56b)

```
class NatSetIterPure {
  public:
    virtual ~NatSetIterPure() { }

    virtual unsigned current() const = 0;
    virtual bool is_valid() const = 0;
    virtual void next() = 0;

    virtual void insert(unsigned) = 0;
    virtual void remove(unsigned) = 0;
};
```

Its methods are the usual ones for SUIF iterators, except that it supports insertion and removal of elements into/from the set being traversed.

`current()`	Return the current element.
`is_valid()`	Return `true` unless the iterator is exhausted.
`next()`	Advance to the next element, if any.
`insert(element)`	Inserts `element` in the set under iteration.
`remove(element)`	Removes `element` from the set under iteration.

The `insert` and `remove` methods are provided both for convenience. It is convenient to be able to modify a set via its iterator without having to pass around both the set and the iterator. It is often more efficient to modify a sorted set at the point currently under the iterator's attention. However, the `insert` and `remove` methods have a precondition to simplify implementations. The element being inserted or removed must not exceed the element currently under scan.

The iterator class used in actual programs is a pointer-based realization of the interface class `NatSetIterPure`:

52a ⟨*class* `NatSetIter` 52a⟩≡ (56b)

```
class NatSetIter : public NatSetIterPure {
  public:
    NatSetIter(NatSet&);
    NatSetIter(const NatSetIter&);
    NatSetIter(NatSetIterRep *rep) : rep(rep) { }
    virtual ~NatSetIter();

    virtual NatSetIter& operator=(const NatSetIter&);

    virtual unsigned current() const;
    virtual bool is_valid() const;
    virtual void next();

    virtual void insert(unsigned);
    virtual void remove(unsigned);

  protected:
    NatSetIterRep *rep;
};
```

11.1 Class `NatSet`

The `NatSet` class interface expresses all of the functionality of the natural-set types except initial construction. For that, see the subclasses `NatSetDense` and `NatSetSparse` just below.

52b ⟨*class* `NatSet` 52b⟩≡ (56b)

```
class NatSet {
  public:
    virtual ~NatSet() { }
    virtual NatSet& operator=(const NatSet&);

    virtual bool is_finite() const = 0;
    virtual bool is_empty() const = 0;

    virtual int size() const = 0;

    virtual bool contains(unsigned element) const = 0;
```

```
    virtual bool contains(const NatSet&) const;
    virtual bool overlaps(const NatSet&) const;

    virtual void insert(unsigned element) = 0;
    virtual void remove(unsigned element) = 0;
    virtual void accommodate(unsigned element) = 0;

    virtual void insert_all() = 0;
    virtual void remove_all() = 0;

    virtual void complement() = 0;

    virtual bool operator==(const NatSet&) const;
    virtual bool operator!=(const NatSet &that) const
        { return !(*this == that); }

    virtual void operator+=(const NatSet&);
    virtual void operator*=(const NatSet&);
    virtual void operator-=(const NatSet&);

    virtual NatSetIter iter(bool complement = false) = 0;
    virtual NatSetIter iter(bool complement = false) const = 0;

    virtual void print(FILE* = stdout, unsigned bound = UINT_MAX) const;

⟨NatSet extender's interface 54a⟩
};
```

Here's a rundown on the `NatSet` methods.

`operator=(`s`)`	Copies s into the current set, preserving the representation style of the current set.
`is_finite()`	Returns `true` if the current set is finite, i.e., is not the complement of a finite set.
`is_empty()`	Returns `true` if the current set is empty.
`size()`	Must only be applied when the current set `is_finite`. Returns the number of its elements.
`contains(`e`)`	Returns `true` if e is an element of the current set.
`contains(`s`)`	Returns `true` if set s is fully contained by current set.
`overlaps(`s`)`	Returns `true` if the intersection of set s with the current set is not empty.
`insert(`e`)`	Inserts e into the current set.
`remove(`e`)`	Removes e from the current set.
`accommodate(`e`)`	Ensures the set's representation accommodates e.
`insert_all()`	Makes the current set represent the whole universe (the complement of the empty set).
`remove_all()`	Makes the current set empty.
`complement()`	Replaces the current set by its complement.
`operator==(`s`)`	Returns `true` if the current set equals s.
`operator!=(`s`)`	Returns `true` if the current set does not equal s.
`operator+=(`s`)`	Unions set s into the current set.
`operator*=(`s`)`	Intersects set s into the current set.
`operator-=(`s`)`	Subtracts set s from the current set.
`iter()`	Produces an iterator (of type `NatSetIter`) over the current set. This iterator doesn't terminate unless the set is finite.
`print(`f`, `b`)`	Prints the elements of the current set to file f, excluding those greater than bound b.

Extender's view of `NatSet`. The implementation of a concrete `NatSet` must provide the following:

54a ⟨`NatSet` *extender's interface* 54a⟩≡ (52b)

```
  protected:
    friend class NatSetCopy;

    virtual int us_size() const = 0;
    virtual NatSet* clone() const = 0;
```

where

`us_size()` returns the size of the underlying finite set.
`clone()` copies the set into the heap and returns a pointer to the copy.

11.2 Class `NatSetDense`

The `NatSetDense` variant of the natural-number sets has the same functionality as `NatSet`, but you can construct one from whole cloth. It is a represented as a `BitVector`, which is capable of representing infinite sets: the complement of a finite set has an "infinity bit" of one, meaning that all the bit positions not explicitly represented are treated as being one.

54b ⟨*class* `NatSetDense` 54b⟩≡ (56b)

```
class NatSetDense : public NatSet {
  public:
    NatSetDense(bool complement = false, int size_hint = 0);
    NatSetDense(const NatSet&);
    NatSet& operator=(const NatSet& that);

    bool is_finite() const;
    bool is_empty() const;

    int size() const;

    bool contains(unsigned element) const;
    bool contains(const NatSet&) const;
    bool overlaps(const NatSet&) const;

    void insert(unsigned element);
    void remove(unsigned element);
    void accommodate(unsigned element);

    void insert_all();
    void remove_all();

    void complement();

    bool operator==(const NatSet&) const;

    void operator+=(const NatSet&);
    void operator*=(const NatSet&);
    void operator-=(const NatSet&);

    NatSetIter iter(bool complement = false);
    NatSetIter iter(bool complement = false) const;
```

```
  protected:
    BitVector us;               // underlying set

    int us_size() const;
    NatSet* clone() const;
};
```

11.3 Class NatSetSparse.

The NatSetSparse class is analogous to NatSetDense but uses a sorted linked-list representation. Apart from its constructors and assignment operator, NatSetSparse inherits all methods from NatSet.

55 ⟨*class* NatSetSparse 55⟩≡ (56b)

```
class NatSetSparse : public NatSet {
  public:
    NatSetSparse(bool complement = false, int size_hint = 0)
        : is_complemented(complement) { }
    NatSetSparse(const NatSet&);

    NatSet& operator=(const NatSet &rhs);

    bool is_finite() const;
    bool is_empty() const;

    int size() const;

    bool contains(unsigned element) const;

    void insert(unsigned element);
    void remove(unsigned element);
    void accommodate(unsigned element) { }

    void insert_all();
    void remove_all();

    void complement();

    bool operator==(const NatSet&) const;

    NatSetIter iter(bool complement = false);
    NatSetIter iter(bool complement = false) const;

  protected:
    Set<unsigned> us;
    bool is_complemented;

    int us_size() const;
    NatSet* clone() const;
};
```

11.4 Class NatSetCopy.

The NatSetCopy class plays a different role from those above. You use it when you want to make a private copy of an existing set, giving it the same representation, and hence the same performance characteristics, as that original set.

56a ⟨*class* NatSetCopy 56a⟩≡ (56b)

```
class NatSetCopy : public NatSet {
  public:
    NatSetCopy(const NatSet&);
    NatSet& operator=(const NatSet&);

    bool is_finite() const;
    bool is_empty() const;

    int size() const;

    bool contains(unsigned element) const;
    bool contains(const NatSet&) const;
    bool overlaps(const NatSet&) const;

    void insert(unsigned element);
    void remove(unsigned element);
    void accommodate(unsigned element);

    void insert_all();
    void remove_all();

    void complement();

    bool operator==(const NatSet&) const;

    void operator+=(const NatSet&);
    void operator*=(const NatSet&);
    void operator-=(const NatSet&);

    NatSetIter iter(bool complement = false);
    NatSetIter iter(bool complement = false) const;

    void print(FILE* = stdout, unsigned bound = UINT_MAX) const;

  protected:
    int us_size() const;
    NatSet* clone() const;

  private:
    NatSet *own;
};
```

11.5 Header file nat_set.h

The header file for module nat_set has the following layout.

56b ⟨*machine/nat_set.h* 56b⟩≡
```
/* file "machine/nat_set.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_NAT_SET_H
#define MACHINE_NAT_SET_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "machine/nat_set.h"
#endif

#include <machine/substrate.h>

class NatSet;
class NatSetIterRep;

⟨class NatSetIterPure 51⟩

⟨class NatSetIter 52a⟩

⟨class NatSet 52b⟩

⟨class NatSetDense 54b⟩

⟨class NatSetSparse 55⟩

⟨class NatSetCopy 56a⟩

#endif /* MACHINE_NAT_SET_H */
```

12 Annotations

An annotation attaches one or more values to an IR object. Each such association is identified by a *key*. An object can have one or more annotations under a particular key. You supply both the object pointer and the key when you attach, detach, or inspect an annotation.

Type `NoteKey`. The OPI uses type `NoteKey` as the type of annotation keys. In Machine SUIF, `NoteKey` is defined to be `IdString`, since these values are conveniently mnemonic and they can be tested for equality efficiently.

57 ⟨*typedef* `NoteKey` 57⟩≡ (63b)
```
typedef IdString NoteKey;
```

12.1 Class `Note`

In Machine SUIF, class `Note` is implemented as a wrapper for a base SUIF annotation. The only storage underlying a `Note` consists of a value of type `Annote*` and a pointer to a reference count. You must use subclasses of `Note` to construct useful annotations; the only public `Note` constructors are the null-note constructor and the copy constructor. If you create a custom subclass of `Note`, you should record all of their state in the SUIF `Annote`, and not add more instance fields. That allows the note to be written to and read from intermediate files transparently (once a `set_note` call has attached the SUIF annotation to the IR object), and it also enables easy interconversion between the base class and your subclass, because there is no risk of "slicing" bugs, i.e., loss of instance data through upcasting.

58 ⟨*class* `Note` 58⟩≡ (63b)

```
class Note {
  public:
    Note();                             // null-note constructor
    Note(const Note&);                  // copy constructor
    ~Note();

    const Note& operator=(const Note&);
    bool operator==(const Note&) const;
    operator bool() const { return annote != NULL; }

  protected:

    Annote *annote;                     // underlying SUIF annotation
    mutable int *ref_count;             // nullified by set_note

    friend Note clone(const Note&);
    friend Note get_note (IrObject*, NoteKey);
    friend Note take_note(IrObject*, NoteKey);
    friend void set_note (IrObject*, NoteKey, const Note&);
    friend Note note_flag();
    friend Note note_list_any();

    // Protected constructor, only invoked by friends.
    Note(Annote*, bool owned);

    void clear();

    // Methods that access the value sequence of a BrickAnnote
    int _values_size() const;

    void _insert_before(int pos, long value);
    void _insert_before(int pos, Integer value);
    void _insert_before(int pos, IdString value);
    void _insert_before(int pos, IrObject *value);

    void _append(long value);
    void _append(Integer value);
    void _append(IdString value);
    void _append(IrObject *value);

    void _replace(int pos, long value);
    void _replace(int pos, Integer value);
    void _replace(int pos, IdString value);
    void _replace(int pos, IrObject *value);

    void _remove(int pos);

    long _get_value(int pos, long const&) const
        { return _get_c_long(pos); }
```

```
    Integer _get_value(int pos, Integer const&) const
        { return _get_integer(pos); }
    IdString _get_value(int pos, IdString const&) const
        { return _get_string(pos); }
    IrObject* _get_value(int pos, IrObject* const&) const
        { return _get_ir_object(pos); }

    long      _get_c_long(int pos) const;
    Integer   _get_integer(int pos) const;
    IdString  _get_string(int pos) const;
    IrObject* _get_ir_object(int pos) const;
};
```

Public methods. The public methods of class `Note` allow you to construct a null note, to copy an existing note, to compare notes for equality, and to test whether a note is null. What does it mean to copy a note? In Machine SUIF, it means to make a new reference to the same underlying SUIF `Annote`. So if you assign one `Note`-valued variable to another, then changes to the underlying value of either variable *does* affect the other. (To override this reference-semantics behavior, you have to use the `clone` function.)

Two notes are equal when their underlying SUIF annotations are isomorphic and have pairwise equal components. A null note is recognized by the fact that its internal pointers (the `Annote*` and the reference-count pointer) are null. The main purpose of a null note is to indicate the absence of an attachment to a particular object, for example, if the result of evaluating `get_note(mi, k_line)` is a null note value, it means that object `mi` has no note under the key `k_line`.

When a note is converted to type `bool`, the result is false only when the note is null; otherwise, the result is true. The purpose of this conversion rule is allow you declare, initialize, and test a note variable in one statement. For example,

```
if (LineNote note = get_note(mi, k_line)) {
    const char *file = note.get_file().chars();
      ...
}
```

If object `mi` has no `k_line` note, the "then" clause of the `if` statement is skipped.

Friends of `Note`. Since the only way to construct a non-null note or to look at its underlying representation is through protected methods, several OPI functions are declared as friends of class `Note`.

Protected methods. The way most notes come into being is via the protected constructor that takes an `Annote*` and a Boolean. The first of these arguments becomes the underlying annotation in the new note. The second is true if that underlying `Annote` is already attached to (and hence, owned by) a SUIF object. If so, the new note needs no reference count; it will be reclaimed when its owning object is reclaimed. Otherwise, the annotation is not yet attached; the new note gets a reference count to be sure that the `Annote*` is properly `deleted` if it happens never to be attached to an object.

The `clear` method is used when one of a note's references is discarded. If the note has a reference count, this method decrements it, and it takes care of reclaiming storage if the count reaches zero.

The rest of the methods in class `Note` are for the common case in which the note associates a tuple of values with the object to which it is attached. You use them when building a specialized subclass of `Note`; they allow you to manipulate a tuple of values that comprise note's contents. We call it a "tuple" because it is not homogeneous: the tuple methods allow for four kinds of values: C `long`, `Integer`, `IdString`, and `IrObject*`.[6]

The names of the tuple methods follow the OPI's conventions for a sequence whose (conceptual) name is `values`, but there's an extra `_` at the front of each to avoid precluding the use of the actual OPI name as a public method of some derived class. Because the `values` tuple can have multiple element types, the `_get_`... methods are unusual. When writing a subclass method, if you know that the tuple element you want to extract is, say, a string, you use `_get_string` to fetch it. But when writing a templated subclass, the type of the tuple element may be a template parameter, e.g., `T`. In that case, you can write `_get_value(pos, T())`, and the C++ overloading mechanism will choose the correct method for you.

Atomic notes. There are two kinds of *atomic* notes: null notes, which can't be attached to objects, and flag notes, which can be attached, but carry no other information than that conveyed by their presence on an object. There is an OPI function to create each of these kinds, and a predicate to test for nullness.

60a ⟨*functions for atomic notes* 60a⟩≡ (63b)
```
Note note_null();
bool is_null(const Note&);

Note note_flag();
```

As mentioned above, the representation for a null note contains a null `Annote*` field. A flag annote holds a pointer to a `GeneralAnnote`, a subclass of `Annote`.

Tuple notes. A non-atomic note contains a pointer to a `BrickAnnote`. That's a kind of `Annote` that can carry a heterogeneous series of values ("bricks"). The methods of `Note` for manipulating these values are protected, so you must use or create a derived note class in order to take advantage of them. Derived classes use the following function to generate a tuple-note value.

60b ⟨*creator for non-atomic notes* 60b⟩≡ (63b)
```
Note note_list_any();   // used only by Note classes
```

You should only need this function when deriving a note subclass, however.

Attaching, detaching, and testing for notes. The OPI functions that you use to test for the presence of notes on an object, to access or remove a note from an object, or to attach a note to an object, are declared as follows.

60c ⟨*note-object association functions* 60c⟩≡ (63b)
```
bool has_notes(IrObject*);
bool has_note (IrObject*, NoteKey);
Note get_note (IrObject*, NoteKey);
Note take_note(IrObject*, NoteKey);
void set_note (IrObject*, NoteKey, const Note&);
```

[6] The C `long` values are of course a subset of the `Integers`, but because relatively small numbers are so common, it is convenient to have the extra case for `longs`.

Functions `get_note` and `take_note` return a null note if the object passed to them has no associated note under the given note key. You can test for nullness with the `is_null` predicate or by using an `if` or `while` statement that declares, binds, and tests a note variable all at once. (See the example of an `if` statement above.)

12.2 Specialized Note Classes

Single-valued notes. Class `OneNote<ValueType>` allows attachment of a single value to an object. The type of the value is the template parameter `ValueType`.

61a ⟨*class* `OneNote` 61a⟩≡ (63b)

```
template <class ValueType>
class OneNote : public Note {
 public:
    OneNote() : Note(note_list_any()) { _replace(0, ValueType()); }
    OneNote(ValueType value) : Note(note_list_any()) { set_value(value); }
    OneNote(const OneNote<ValueType> &other) : Note(other) { }
    OneNote(const Note &note) : Note(note) { }

    ValueType get_value() const
        { return _get_value(0, ValueType()); }
    void set_value(ValueType value)
        { _replace(0, value); }
};
```

The first (parameterless) constructor gives the note a default value. To give it a non-default value, you can either use the second constructor, which accepts a value, or you can create the note and then use the `set_value` method. To fetch the value from the note, you apply the `get_value` method.

List-carrying notes. Class `ListNote<ValueType>` allows you to attach zero or more values, all of which have type `ValueType`.

61b ⟨*class* `ListNote` 61b⟩≡ (63b)

```
template <class ValueType>
class ListNote : public Note {
  public:
    ListNote() : Note(note_list_any()) { }
    ListNote(const ListNote<ValueType> &other) : Note(other) { }
    ListNote(const Note &note) : Note(note) { }

    int values_size() const
        { return _values_size(); }
    ValueType get_value(int pos) const
        { return _get_value(pos, ValueType()); }
    void set_value(int pos, ValueType value)
        { _replace(pos, value); }
    void append(ValueType value)
        { _append(value); }
    void insert_before(int pos, ValueType value)
        { _insert_before(pos, value); }
    void remove(int pos)
        { _remove(pos); }
};
```

When you first create a `ListNote`, is contains no values; that is, its `value_size` method returns zero. The methods of the class allow you to add, remove, update, and access values.

Custom notes for source-code location. This is a typical custom note class. It is mentioned as an example in *The Extender's Guide*. You often want to be able to attach a few values of different kinds to an object. In this case, the object is usually an instruction, and the values are the source file name (a string) and the source line number (an integer) that gave rise to the instruction. Here's the entire derived class:

62a ⟨*class* `LineNote` 62a⟩≡ (63b)

```
class LineNote : public Note {
 public:
    LineNote() : Note(note_list_any()) { }
    LineNote(const LineNote &other) : Note(other) { }
    LineNote(const Note &note) : Note(note) { }

    int get_line() const        { return _get_c_long(0); }
    void set_line(int line)     { _replace(0, line); }
    IdString get_file() const   { return _get_string(1); }
    void set_file(IdString file){ _replace(1, file); }
};
```

Here, we chose to make the line number the first element of the underlying tuple and the file name the second element. But no users of class `LineNote` need to know that.

Custom note for multiway branches. Notes of class `MbrNote` attach information about a multiway branch to the instruction that implements it, which for some target is not distinguishable from an ordinary indirect jump instruction without this annotation.

62b ⟨*class* `MbrNote` 62b⟩≡ (63b)

```
class MbrNote : public Note {
 public:
    MbrNote() : Note(note_list_any()) { }
    MbrNote(const MbrNote &other) : Note(other) { }
    MbrNote(const Note &note) : Note(note) { }

    int get_case_count() const;
    VarSym* get_table_sym() const
        { return to<VarSym>(_get_ir_object(0)); }
    void set_table_sym(VarSym* var)
        { _replace(0, var); }

    LabelSym* get_default_label() const
        { return to<LabelSym>(_get_ir_object(1)); }
    void set_default_label(LabelSym* label)
        { _replace(1, label); }

    int get_case_constant(int pos) const
        { return _get_c_long((pos << 1) + 2); }
    void set_case_constant(int constant, int pos)
        { _replace((pos << 1) + 2, constant); }

    LabelSym* get_case_label(int pos) const
        { return to<LabelSym>(_get_ir_object((pos << 1) + 3)); }
    void set_case_label(LabelSym* label, int pos)
```

```
        { _replace((pos << 1) + 3, label); }
};
```

The contents of a MbrNote include a sequence of (case_constant, case_label) pairs, where the case_constant is the integral value of the tested expression that marks a particular case and the case_label is the label of the code to which the instruction jumps in that case. In addition to the sequence of pairs, there are fixed "fields": case_count is the number of non-default cases, i.e., the length of the sequence of pairs just mentioned; table_sym is the variable symbol holding the dispatch table used to compute the target of the indirect jump; and default_label is the code label to which control transfers when the tested expression doesn't evaluate to one of the explicit case constants.

Custom note for attaching a natural-number set. Class NatSetNote connects a NatSet to an IR object. One use is to record the sets of registers implicitly used or defined by an instruction for the benefit of data-flow analyzers.

63a ⟨*class* NatSetNote 63a⟩≡ (63b)

```
class NatSetNote : public Note {
  public:
    NatSetNote(bool dense = false);
    NatSetNote(const NatSet*, bool dense = false);
    NatSetNote(const NatSetNote &other) : Note(other) { }
    NatSetNote(const Note &note) : Note(note) { }

    void get_set(NatSet*) const;
    void set_set(const NatSet*);
};
```

Just as there are "dense" (bit-vector based) and "sparse" (linked-list based) representations for an in-memory NatSet, you can chose between dense and sparse NatSetNote representations. By default, the sparse-set form is used, meaning that each set element is stored individually in the values tuple of the note. On the other hand, if you pass the argument true to the NatSetNote constructor, it will choose a dense-set form and represent the set as a bit vector, stored in a single Integer value. In either case, the set is allowed to be the complement of a finite set (the result of applying the NatSet::complement method).

You can the supply the set of values to attach by passing a NatSet* to the NatSetNote constructor, or you can call the set_set method of an existing note. When you want to read back the set contained in a note, you must pass a pointer to a NatSet to the get_set method, which clears it and then fills it from the note. Here is one way to do this:

```
if (NatSetNote regs_used_note = get_note(instr, k_regs_used)) {
    NatSetDense regs_used;
    regs_used_note.get_set(&regs_used);
      ...                              // regs_used holds register set
}
```

You don't have to know whether the note was represented in sparse or dense form when you read it back. We could just as well have declared regs_used to have type NatSetSparse in the example above.

12.3 header file opnd.h

The header file for operands has the following outline:

63b ⟨*machine/note.h* 63b⟩≡
```
/* file "machine/note.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_NOTE_H
#define MACHINE_NOTE_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "machine/note.h"
#endif

#include <machine/substrate.h>
#include <machine/nat_set.h>

⟨typedef NoteKey 57⟩

⟨class Note 58⟩

⟨functions for atomic notes 60a⟩

⟨creator for non-atomic notes 60b⟩

⟨note-object association functions 60c⟩

⟨class OneNote 61a⟩

⟨class ListNote 61b⟩

⟨class LineNote 62a⟩

⟨class MbrNote 62b⟩

⟨class NatSetNote 63a⟩

#endif /* MACHINE_NOTE_H */
```

13 Problems

This section contains problem-reporting utilities for Machine SUIF.

13.1 Progress diagnostics and warning messages

Function `debug`. `debug` prints diagnostic messages provided the user has set `debuglvl` to the appropriate level of verbosity. Most Machine-SUIF passes take a command-line option for setting this variable.

The first argument must be less than or equal to the current value of `debuglvl` for the call on `debug` to have any effect. If that condition holds, the second and subsequent arguments are used as they would be in a call to `fprintf` with `stderr` as the output stream.

We condition the definition of `debug` so that its calls can be completely omitted from compiled passes and libraries if the preprocessor token NDEBUG is defined and optimizations are turned on at the time Machine SUIF is built.

65a ⟨*function* debug 65a⟩≡ (66)
```
extern int debuglvl;            /* user defined diagnostic print level */

#ifndef NDEBUG
extern void debug(const int, const char * ...);
#else
#define debug if (false)
#endif
```

Function warn. warn is similar to debug except that it is unconditional.

65b ⟨*function* warn 65b⟩≡ (66)
```
#ifndef NDEBUG
extern void warn(const char * ...);
#else
#define warn if (false)
#endif
```

Macros if_debug. The macro if_debug(lvl) causes the subsequence statement to be executed only if debuglvl equals or exceeds lvl. The reason for using it instead of testing debuglvl directly is that if_debug can be redefined to erase the subsequent statement, thereby making production code smaller and a bit faster.

65c ⟨*macro* if_debug 65c⟩≡ (66)
```
#ifndef NDEBUG
#define if_debug(lvl) if (debuglvl >= lvl)
#else
#define if_debug(lvl) if (false)
#endif
```

13.2 Assertions

In Machine SUIF, the assertion primitive is called claim. You state your assertion as a claim that a given Boolean expression holds. If the expression evaluates to false the program prints a message, optionally containing formatted text supplied with the claim, and it aborts.[7] It is as if function claim had the following declarations.

```
extern void claim(bool assertion);
extern void claim(bool assertion, const char *format, ...);
```

The implementation adds source-code location information to help identify a claim that is violated. As with the debug and warn functions, we can conditionally omit calls to claim in an installation of Machine SUIF by defining NDEBUG and turning on compiler optimizations.

[7] Our assertion machinery is adapted from an earlier version of SUIF.

65d ⟨*function* `claim` 65d⟩≡ (66)

```
extern char *__assertion_file_name;
extern int __assertion_line_num;
extern char *__assertion_module_name;

extern void _internal_assertion_failure(void);
extern void _internal_assertion_failure(const char *format, va_list ap);

inline void __do_assertion(bool assertion)
  {  if (!(assertion))  _internal_assertion_failure(); }

inline void __do_assertion(bool assertion, const char *format, ...)
  {
    if (!(assertion))
      {
        va_list ap;
        va_start(ap, format);
        _internal_assertion_failure(format, ap);
        va_end(ap);
      }
  }

#ifndef NDEBUG

#ifndef _MODULE_
#define _MODULE_ NULL
#endif

#define claim  __assertion_file_name = __FILE__, \
               __assertion_line_num = __LINE__, \
               __assertion_module_name = _MODULE_, \
               __do_assertion
#else
#define claim if (false)
#endif
```

13.3 Header file `problems.h`

The header file for module `problems` has the following layout.

66 ⟨*machine/problems.h* 66⟩≡

```
/* file "machine/problems.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_PROBLEMS_H
#define MACHINE_PROBLEMS_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "machine/problems.h"
#endif
```

```
#include <machine/substrate.h>

⟨function debug 65a⟩

⟨function warn 65b⟩

⟨macro if_debug 65c⟩

⟨function claim 65d⟩

#endif /* MACHINE_PROBLEMS_H */
```

14 Utilities

This section contains utilities for Machine SUIF.

14.1 Operand scanning and replacement

Utility map_opnds applies a functional object to the operands of an instruction and replaces each by the result returned (which may of course be the same as the original). The related utilities map_src_opnds and map_dst_opnds do the same for the source and destination subsets of an instruction's operands, respectively.

The object applied to each operand is of class OpndFilter. The only significant method of this class is its "apply" operator (operator()), which takes and returns an Opnd. You can subclass OpndFilter so that the apply operator examines and/or transforms operands as you wish and so that an instance of your subclass carries whatever state might be needed when each operand is visited. The base class is quite simple:

67a ⟨*class* OpndFilter 67a⟩≡ (75d)

```
class OpndFilter {
  public:
    OpndFilter(bool thru_addr_exps = true)
        { _thru_addr_exps = thru_addr_exps; }
    virtual ~OpndFilter() {}

    typedef enum { IN, OUT } InOrOut;

    virtual Opnd operator()(Opnd, InOrOut) = 0;

    bool thru_addr_exps() const { return _thru_addr_exps; }
  protected:
    bool _thru_addr_exps;
};
```

The map_opnds, map_src_opnds, and map_dst_opnds functions take the instruction whose operands are to be scanned and a concrete instance of OpndFilter.

67b ⟨*function* map_opnds 67b⟩≡ (75d)

```
void map_opnds(Instr *instr, OpndFilter &filter);
void map_opnds(Opnd addr_exp, OpndFilter &filter);
void map_src_opnds(Instr *instr, OpndFilter &filter);
void map_dst_opnds(Instr *instr, OpndFilter &filter);
```

The `map_opnds` function applies the filter to each operand, indicating via a second argument whether the operand is an input (`IN`) or an output (`OUT`) of the instruction. It replaces the original operand by the filter's result.

When `map_opnds` reaches an address-expression operand, it applies the filter to the whole operand before doing any of the constituent operands. If the filter returns the address-expression operand unchanged, then the filter is applied to, and may replace, the constituent operands of the address expression. Recall that all constituent operands of an address expression are instruction inputs, even if the address expression is a destination operand. The filter therefore always receives the indicator `IN` when applied to a suboperand of an address expression.

Function `map_src_opnds` is just like `map_opnds`, but it only maps the direct source operands of the instruction. Likewise, `map_dst_opnds` maps only the destination operands.

A typical use for `map_opnds` is to rewrite operands that have been assigned to physical registers. The filter simply checks whether its argument is one of those to be rewritten. If so, it returns the appropriate hard-register operand; if not, it returns its argument unchanged.

14.2 Annotation help

Certain annotations are used often enough to make it worthwhile defining utilities to query and update them.

Formal parameters assigned to registers. The `param_reg` annotation records the assignment of a formal-parameter symbol to a specific argument register. The only data stored is the abstract register number. The following functions manage `param_reg` annotations.

68a ⟨*parameter-register utilities* 68a⟩≡ (75d)

```
bool is_reg_param(VarSym*);
int get_param_reg(VarSym*);
void set_param_reg(VarSym*, int reg);
```

`is_reg_param(`p`)`	Returns `true` if parameter symbol p is assigned to a register.
`get_param_reg(`p`)`	Returns the abstract number of the register to which parameter p is assigned, or -1 if it's not assigned to a register.
`set_param_reg(`p`, `r`)`	Records the assignment of parameter p to register r.

14.2.1 Annotation transfer

Sometimes when you replace one annotated `IrObject` with another, you want to transfer the annotations of the former to the latter. For this we have:

68b ⟨*note-transfer functions* 68b⟩≡ (75d)

```
extern void move_notes(IrObject *from, IrObject *to);
extern void copy_notes(IrObject *from, IrObject *to);
```

The `move_notes` function removes all annotations from the one object and attaches them to another, while `copy_notes` clones the annotations before attaching them, rather than removing them from the original.

14.2.2 Annotation suppression during printing

The following list contains annotation keys. This list is used by the Machine-SUIF printing utilities to keep output free of tedious annotations. Most printing passes allow you to override this list and print all annotations.

68c ⟨*non-printing-note keys* 68c⟩≡ (75d)
```
extern Set<IdString> nonprinting_notes;
```

14.3 Symbol, symbol-table, and type utilities

Symbol predicates. The following set of predicates on symbols distinguish their scopes.

69a ⟨*symbol predicates* 69a⟩≡ (75d)
```
bool is_global(Sym*);
bool is_external(Sym*);
bool is_defined(Sym*);
bool is_private(Sym*);
bool is_auto(VarSym*);
```

Predicate `is_global` is true of a symbol whose scope is not local to a procedure. It works by finding the symbol table to which the symbol belongs and checking whether its parent is a `file_set_block` or a `file_block`. The latter objects own the global symbol tables.

A symbol satisfying `is_external` lives in the external symbol table at the `file_set_block` level, which means it is visible externally. It may be defined within the file set or outside of it.

A symbol satisfying `is_defined` is a global symbol whose defining declaration is in the current file set.

A symbol satisfying `is_private` lives in the symbol table of a `file_block`, which means that it is global, but isn't visible outside of the file.

An "automatic" variable, one whose symbol satisfies `is_auto`, is local to a procedure.

Symbol finders. `lookup_local_var` looks up a variable symbol in the current local scope, while `lookup_external_var` looks up a variable symbol in the external scope, i.e., one that is either exported from or imported by the file currently being compiled. `lookup_external_proc` does the same for a procedure symbol. In each case, if the symbol is not found, the routine returns `NULL`.

69b ⟨*symbol finders* 69b⟩≡ (75d)
```
VarSym* lookup_local_var(IdString name);
VarSym* lookup_external_var(IdString name);
ProcSym* lookup_external_proc(IdString name);
```

Symbol creators. These functions each generate a new symbol in the local scope with a name guaranteed not to clash in that scope.

69c ⟨*symbol creators* 69c⟩≡ (75d) 70a ▷
```
VarSym* new_named_var(TypeId, IdString name);
VarSym* new_unique_var(TypeId, const char *prefix = "_var");
VarSym* new_unique_var(Opnd init, const char *prefix = "_var");
LabelSym* new_unique_label(const char *prefix = "_label");

VarSym* new_empty_table_var(TypeId elem_type, int length);
VarSym* new_dispatch_table_var(MbrNote&);
```

With new_named_var, the resulting variable has exactly the name given. With the others, the caller may provide a prefix string for the new symbol's name. The name of the resulting symbol has a numeric suffix that assures uniqueness in the local scope.

new_named_var and new_unique_var each generate a variable symbol. If given a type, new_unique_var creates an uninitialized variable with that type. When an operand is passed instead of a type, it must be a numeric-immediate operand. In that case, the type of the operand becomes the type of the new variable, and the immediate value becomes its initial value.

new_unique_label generates a new local code-label symbol.

new_empty_table_var creates a new, unique, uninitialized, local array variable with a given element type and element count. Such a variable might be used for saving and restoring a set of registers, for example.

new_dispatch_table_var creates and initializes a new unique variable to serve as the dispatch table for a multiway branch instruction. The annotation passed as its argument carries all the necessary information about the size of the table and the code labels that become its contents. As a side effect, this function stores the new variable symbol back into the argument annotation.

The following function returns the external procedure symbol having the given type and name.

70a ⟨*symbol creators* 69c⟩+≡ (75d) ◁69c

```
ProcSym* find_proc_sym(TypeId, IdString name);
```

If such a symbol already exists, it's simply returned. Otherwise, a new symbol is created and entered in the external symbol table.

Accessing and changing symbol properties. To fetch or store the type of a variable or procedure symbol:

70b ⟨*symbol-property functions* 70b⟩≡ (75d) 70c▷

```
TypeId get_type(VarSym*);
void set_type(VarSym*, TypeId);

TypeId get_type(ProcSym*);
void set_type(ProcSym*, TypeId);
```

To fetch or set the address-taken attribute of a variable:

70c ⟨*symbol-property functions* 70b⟩+≡ (75d) ◁70b 70d▷

```
bool is_addr_taken(VarSym*);
void set_addr_taken(VarSym*, bool);
```

Function update_dispatch_table_var updates one target label in the dispatch table for a multiway branch. It takes a MbrNote and the zero-based index of the multiway branch case. The annotation supplies both the variable symbol whose definition holds the dispatch table and the new label for the case at the given index.

70d ⟨*symbol-property functions* 70b⟩+≡ (75d) ◁70c 71a▷

```
void update_dispatch_table_var(MbrNote&, int index);
```

Function strip_dispatch_table_var decommisions the dispatch table for a multiway branch, leaving only a single entry that contains a null label value. Again, the MbrNote for the branch is the source of a variable whose definition gets stripped.

71a ⟨*symbol-property functions* 70b⟩+≡ (75d) ◁70d

```
void strip_dispatch_table_var(MbrNote&);
```

Formal parameter helpers. These functions inspect the formal parameters of an optimization unit.

71b ⟨*formal parameter helpers* 71b⟩≡ (75d)

```
int get_formal_param_count(OptUnit*);
VarSym* get_formal_param(OptUnit*, int pos);
```

Symbol-table accessors.

71c ⟨*symbol-table accessors* 71c⟩≡ (75d)

```
SymTable* external_sym_table();
SymTable* file_set_sym_table();
SymTable* get_sym_table(FileBlock*);
SymTable* get_sym_table(ProcDef*);
```

Symbol-table predicates. The following set of predicates on symbol tables identify their level in the hierarchy.

71d ⟨*symbol-table predicates* 71d⟩≡ (75d)

```
bool is_global(SymTable*);
bool is_external(SymTable*);
bool is_private(SymTable*);
```

Taxonomy of types. To keep dependence on SUIF's machinery for composing and decomposing type objects localized, we define functions that operate on a TypeId. We make no attempt to cover all of different kinds of type objects that SUIF provides. Back end work doesn't make too much use of types. When others features are needed, we'll add them.

SUIF views the type kingdom as consisting of data types, procedure types, and qualified types. A data type is the type of a data value other than code. A procedure type describes a piece of code that you can call. A qualified type is based on a data type, but it describes a storage location, rather than a value. To the underlying value type, it adds qualifiers like const and volatile. In SUIF, variable symbols, array elements, and record fields must have qualified type.

Most Machine SUIF code does not need to make these distinctions, because the helpers we define here take care of coercing type-object pointers to the appropriate subclasses.

Predicates on TypeId's. Here are functions that identify different kinds of types.

71e ⟨*type helpers* 71e⟩≡ (75d) 72a ▷
```
bool is_void(TypeId);
bool is_scalar(TypeId);         // data type other than array or record
bool is_boolean(TypeId);
bool is_integral(TypeId);
bool is_signed(TypeId);         // apply only to an integral type
bool is_floating_point(TypeId);
bool is_pointer(TypeId);
bool is_enum(TypeId);
bool is_record(TypeId);         // e.g., struct or union type
bool is_struct(TypeId);
bool is_union(TypeId);
bool is_array(TypeId);
```

Properties of typed values. These functions return the size and the alignment requirement, both expressed in bits, of the value described by a `TypeId`. (To say that a value must have, e.g., 32-bit alignment means that its address in bytes must be a multiple of four, i.e., 32/8.)

72a ⟨*type helpers* 71e⟩+≡ (75d) ◁ 71e 72b ▷
```
int get_bit_size(TypeId);
int get_bit_alignment(TypeId);
```

Once in a while, a back end needs to construct a type identifier. Here are functions for creating a pointer type, given the type pointed to (which we call the *referent* type), and for extracting the referent type from a `TypeId` that is known to be a pointer type.

72b ⟨*type helpers* 71e⟩+≡ (75d) ◁ 72a 72c ▷
```
TypeId pointer_type(TypeId referent);
TypeId get_referent_type(TypeId);
```

Here's a function that creates an array type from the type of its elements plus the lower and upper bounds of its index range, and one that extract the element type of an array type.

72c ⟨*type helpers* 71e⟩+≡ (75d) ◁ 72b 72d ▷
```
TypeId array_type(TypeId element_type, int lower_bound, int upper_bound);
TypeId get_element_type(TypeId array_type);
```

And this one extracts the result type from a procedure type.

72d ⟨*type helpers* 71e⟩+≡ (75d) ◁ 72c 72e ▷
```
TypeId get_result_type(TypeId type);
```

Creating a C procedure type. Function `find_proc_type` uses the SUIF type factory to obtain the C procedure type for a two-argument procedure, given its argument types and its result type.

72e ⟨*type helpers* 71e⟩+≡ (75d) ◁ 72d
```
TypeId find_proc_type(TypeId result, TypeId arg0, TypeId arg1);
```

14.4 Cloning

Function deep_clone copys one IR object, including all of the subobjects that it owns. It is implemented as a template function so that the result type will always be the same as the argument type.

73a ⟨*cloning functions* 73a⟩≡ (75d)
```
template<class T>
T*
deep_clone(T *object)
{
    return to<T>(object->deep_clone(the_suif_env));
}
```

Function renaming_clone is used to clone a local IR object (one that is part of an optimization unit) into a new scope, i.e., one governed by a different local symbol table. It takes the object and the symbol table of the destination scope as arguments. It returns the cloned object.

73b ⟨*cloning function* 73b⟩≡
```

IrObject* renaming_clone(IrObject*, SymTable *receiving_scope);
```

Any local symbols found in the object being cloned are themselves cloned and inserted in the receiving_scope table. If necessary, their names are changed to avoid clashes with existing symbols of that scope. Non-local symbols of the cloned object are not cloned, since they remain in scope.

A typical application for renaming_clone is in inlining, where the body of one procedure is cloned and inserted into the body of another.

14.5 A string utility

Since the strdup function, which returns a fresh copy of a C string, is not standard, we provide strdupe as a substitute.

73c ⟨*function* strdupe 73c⟩≡
```
inline char *
strdupe(const char *s)
{
    int n = strlen(s) + 1;
    char *r = new char[n];
    return (char *)memcpy(r, s, n);
}
```

14.6 A hashing utility

The hash code generator for long integers, needed by SUIF's hash-map utility, will eventually be part of base SUIF. For now:

73d ⟨*hashing helpers* 73d⟩≡ (75d)
```
size_t hash(const unsigned long);
```

14.7 Printing utilities

Our convention in the OPI is to overload the function `fprint` for printing to a `FILE*`. The following variants of `fprint` print `IrObjects` (at least at a quality suitable for debugging) and extended-precision integers.

74a ⟨*printing utilities* 74a⟩≡ (75d)

```
void fprint(FILE*, IrObject*);
void fprint(FILE*, Integer);
```

14.8 Sequence utilities

Some utilities that apply to STL container objects. Function `clear` makes its list argument empty.

74b ⟨*sequence utilities* 74b⟩≡ (75d) 74c ▷

```
template <class Item>
void clear(list<Item> &l)
{
    l.clear_list();
}
```

Function `get_last_handle` returns a handle on the last element of its list argument, or else the sentinal handle of that list.

74c ⟨*sequence utilities* 74b⟩+≡ (75d) ◁ 74b 74d ▷

```
template <class Item>
list<Item>::iterator
get_last_handle(list<Item> &l)
{
    return l.get_nth(l.size() - 1);
}
```

74d ⟨*sequence utilities* 74b⟩+≡ (75d) ◁ 74c 74e ▷

```
template <class Item>
list<Item>::iterator
find(list<Item> &l, const Item &item)
{
    for (list<Item>::iterator h = l.begin(); h != l.end(); ++h)
        if (*h == item)
            return h;
    return l.end();
}
```

74e ⟨*sequence utilities* 74b⟩+≡ (75d) ◁ 74d 75a ▷

```
template <class Item>
bool
contains(list<Item> &l, const Item &item)
{
    return find(l, item) != l.end();
}
```

Function `maybe_expand` makes sure that a vector's size is large enough to cover an entry at `index`. If not, it resizes the vector to cover `index`. Argument `init` provides an initial value for any added elements.

75a ⟨*sequence utilities* 74b⟩+≡ (75d) ◁74e 75b▷

```
template <class Item>
void
maybe_expand(Vector<Item> &v, size_t index, const Item &init)
{
    if (index >= v.size())
        v.resize(index + 1, init);
}
```

Function `end_splice` works around the lack of a `splice` method in the `suif_list` template. It moves the elements of its second argument to the end of the first, leaving the donor list empty. In the real STL, this can be done in constant time.

75b ⟨*sequence utilities* 74b⟩+≡ (75d) ◁75a

```
template <class Item>
void
end_splice(List<Item> &l, List<Item> &x)
{
    while (!x.empty())
    {
        l.insert(l.end(), x.front());
        x.pop_front();
    }
}
```

14.9 Miscellany

The following are handy when changing the representation of a optimization unit from instruction-list to, say, CFG form.

75c ⟨*optimization-unit body functions* 75c⟩≡ (75d)

```
AnyBody* get_body(OptUnit*);
void set_body(OptUnit*, AnyBody*);
```

14.10 Header file `util.h`

The header file for module `util` has the following layout.

75d ⟨*machine/util.h* 75d⟩≡

```
/* file "machine/util.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_UTIL_H
#define MACHINE_UTIL_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
```

```
#pragma interface "machine/util.h"
#endif

#include <machine/substrate.h>
#include <machine/machine_ir.h>
#include <machine/opnd.h>
#include <machine/note.h>

⟨class OpndFilter 67a⟩

⟨function map_opnds 67b⟩

⟨parameter-register utilities 68a⟩

⟨formal parameter helpers 71b⟩

⟨note-transfer functions 68b⟩

⟨non-printing-note keys 68c⟩

⟨symbol predicates 69a⟩

⟨symbol finders 69b⟩

⟨symbol creators 69c⟩

⟨symbol-property functions 70b⟩

⟨type helpers 71e⟩

⟨symbol-table predicates 71d⟩

⟨symbol-table accessors 71c⟩

⟨cloning functions 73a⟩

⟨hashing helpers 73d⟩

⟨printing utilities 74a⟩

⟨sequence utilities 74b⟩

⟨optimization-unit body functions 75c⟩

#endif /* MACHINE_UTIL_H */
```

15 Contexts

As mentioned in Section 2.5, a *context* gathers all the items that characterize the target machine into one record. OPI programmers don't need to concern themselves with the context data structure; they only need to keep it set correctly for the current target. But users who add target-machine descriptions do so by developing a refined context that holds the characteristics of the machine they want to add.

Machine SUIF deals with only one context at a time, because compilation typically deals with only one kinds of machine at a time. For the rare cases in which two machine descriptions might be needed simultaneously, you divide the task into two phases, one for each machine.

As an OPI programmer, you see target characteristics through global functions and through the data structures that they return. Function `target_printer`, which returns a `Printer` pointer, is typical. But all this function does is to access `the_context`, a global variable of type `Context*`. This global context has a method named `target_printer` that provides the implementation of the global function, which calls that method.

So why not call such methods of `the_context` directly? For one thing, it's shorter to write calls on global functions, and these things occur quite frequently. More importantly, the context is not needed in some compilation settings, such as run-time optimization. So for portability, code written against the OPI should be almost "context free".

But there is still another reason why context methods are not accessed in OPI code. It has to do with the way contexts are built for extensibility. Class `Context` itself has no interesting functionality. The `target_printer` method is actually declared in another class, called `MachineContext`. It is an interface class: `target_printer` and other public methods are pure virtual methods. `MachineContext` is not a subclass of `Context`. It is a chunk of interface that target libraries use when composing an actual context instance that is right for the target in question. The method of composition is multiple inheritance. A typical concrete context inherits from `Context`, `MachineContext`, and `SuifVmContext`. (Context-interface classes are named for the libraries that define them: `MachineContext` is defined in the `machine` library, `SuifVmContext` in the `suifvm` library, and so on.)

So while the declared type of `the_context` is `Context*`, it normally points to an object whose class inherits not only from `Context`, but also from several interface classes. A method such as `target_printer` is invoked by casting `the_context` to type `MachineContext*`:

```
dynamic_cast<MachineContext*>(the_context)->target_printer();
```

Further motivation and a description of how we use multiple inheritance from independent context-interface classes, together with dynamic casting based on run-time type information (RTTI), can be found in the *Extender's Guide*.

15.1 Establishing context

Any Machine-SUIF pass that calls one or more target-characterization functions must first set `the_context`. This is accomplished by calling `focus(FileBlock*)`, which sets `the_context` using the `target_context` function.

77a ⟨*context creation* 77a⟩≡ (80e)

```
extern Context *the_context;

Context* target_context(FileBlock*);
Context* find_context(IdString lib_name);
```

Function `target_context` expects to find a `target_lib` note on its file-block argument giving the name of the target-specific library for the file. It calls `find_context` to obtain the corresponding context. If necessary `find_context` loads that library. The library's initialization code registers a function that creates a new target-characterization context. It registers this context-creator function in the following map:

77b ⟨*context-creator registry* 77b⟩≡ (80e)

```
extern Map<IdString,Context*(*)()> the_context_creator_registry;
```

Most passes have no need to be aware of `the_context` or the `target_lib` annotation; the latter is attached by a code-generation pass and is usually retained through subsequent phases. There is no need to delete `the_context` at the end of the pass; `target_context` builds a cache, so that context records are reused while compilation is going on and then deleted all at once when a pass terminates.

15.2 Class `Context`

`Context` is an empty interface class. It is used as a completely library-independent handle on the current library-created context. Although it is not expected to have subclasses, we give it a virtual destructor. That's because the C++ RTTI machinery can only navigate to so-called "polymorphic" classes, which means those having virtual methods.

78a ⟨*class* `Context` 78a⟩≡ (80e)
```
class Context {
  public:
    Context() { }
    virtual ~Context() { }
};
```

15.3 Class `MachineContext`

Class `MachineContext` is a non-trivial interface class. It declares the virtual methods for instruction properties and register properties. Note that it is *not* derived fron `Context`.

78b ⟨*class* `MachineContext` 78b⟩≡ (80e)
```
class MachineContext {
  public:
    MachineContext();
    virtual ~MachineContext();

    ⟨MachineContext generic-pointer method 78c⟩

    ⟨MachineContext register-info methods 79a⟩

    ⟨MachineContext printer methods 79b⟩

    ⟨MachineContext code-finalizer method 79d⟩

    ⟨MachineContext instruction-predicate methods 80a⟩

    ⟨MachineContext opcode-generator methods 80b⟩

    ⟨MachineContext opcode query methods 80c⟩

  protected:
    ⟨MachineContext protected matter 80d⟩
};
```

Target-characterization methods. The target's generic-pointer type, the value produced by function `type_addr` (see Section 5), is fetched by:

78c ⟨`MachineContext` *generic-pointer method* 78c⟩≡ (78b)
```
    virtual TypeId type_addr() const = 0;
```

In addition, the following methods implement like-named functions for use by register allocators (Section 7):

79a ⟨MachineContext *register-info methods* 79a⟩≡ (78b)
```
virtual int reg_count() const
    { claim(false); return -1; }
virtual const char* reg_name(int reg) const
    { claim(false); return NULL; }
virtual int reg_width(int reg) const
    { claim(false); return -1; }
virtual const NatSet* reg_aliases(int reg) const
    { claim(false); return NULL; }

virtual int reg_class_count() const
    { claim(false); return -1; }
virtual const NatSet* reg_members(RegClassId) const
    { claim(false); return NULL; }
virtual const NatSet* reg_allocables(bool maximals = false) const
    { claim(false); return NULL; }
virtual const NatSet* reg_caller_saves(bool maximals = false) const
    { claim(false); return NULL; }
virtual const NatSet* reg_callee_saves(bool maximals = false) const
    { claim(false); return NULL; }
virtual int reg_maximal(int reg) const
    { return reg; }
virtual InstrHandle reg_fill(Opnd dst, Opnd src, InstrHandle marker,
                             bool post_reg_alloc = false) const
    { claim(false); return marker; }
virtual InstrHandle reg_spill(Opnd dst, Opnd src, InstrHandle marker,
                              bool post_reg_alloc = false) const
    { claim(false); return marker; }
virtual void reg_classify(Instr*, OpndCatalog*, RegClassMap*) const
    { claim(false); }
virtual RegClassId reg_class_intersection(RegClassId, RegClassId) const
    { claim(false); return REG_CLASS_NONE; }
virtual int reg_choice(RegClassId, const NatSet *pool,
                       const NatSet *excluded, bool rotate) const
    { claim(false); return -1; }
```

The target's `Printer` pointer, which corresponds to global variable `printer` (Section 8), is fetched by:

79b ⟨MachineContext *printer methods* 79b⟩≡ (78b) 79c ▷
```
virtual Printer* target_printer() const = 0;
```

And similarly for its `CPrinter` pointer:

79c ⟨MachineContext *printer methods* 79b⟩+≡ (78b) ◁ 79b
```
virtual CPrinter* target_c_printer() const = 0;
```

The target's `CodeFin` generator, which corresponds to global function `target_code_fin` (Section 10), is fetched by the method:

79d ⟨MachineContext *code-finalizer method* 79d⟩≡ (78b)
```
virtual CodeFin* target_code_fin() const = 0;
```

The target-specific instruction predicates described in Section 3.3 are implemented by like-named methods of `MachineContext`.

80a ⟨`MachineContext` *instruction-predicate methods* 80a⟩≡ (78b)
```
virtual bool is_ldc(Instr*) const = 0;
virtual bool is_move(Instr*) const = 0;
virtual bool is_cmove(Instr*) const = 0;
virtual bool is_predicated(Instr*) const { return false; }
virtual bool is_line(Instr*) const = 0;
virtual bool is_ubr(Instr*) const = 0;
virtual bool is_cbr(Instr*) const = 0;
virtual bool is_call(Instr*) const = 0;
virtual bool is_return(Instr*) const = 0;
virtual bool is_binary_exp(Instr*) const = 0;
virtual bool is_unary_exp(Instr*) const = 0;
virtual bool is_commutative(Instr*) const = 0;
virtual bool is_two_opnd(Instr*) const = 0;
virtual bool reads_memory(Instr*) const = 0;
virtual bool writes_memory(Instr*) const = 0;
virtual bool is_builtin(Instr*) const = 0;
```

The target-specific opcode generators described in Section 6 are implemented by like-named methods of `MachineContext`.

80b ⟨`MachineContext` *opcode-generator methods* 80b⟩≡ (78b)
```
virtual int opcode_line() const = 0;
virtual int opcode_ubr() const = 0;
virtual int opcode_move(TypeId) const = 0;
virtual int opcode_load(TypeId) const = 0;
virtual int opcode_store(TypeId) const = 0;
virtual int opcode_cbr_inverse(int cbr_opcode) const = 0;
```

Likewise for the functions that ask target-specific questions about opcodes.

80c ⟨`MachineContext` *opcode query methods* 80c⟩≡ (78b)
```
virtual bool target_implements(int opcode) const = 0;
virtual char* opcode_name(int opcode) const = 0;
```

Protected fields. Class `MachineContext` is not quite a pure abstract interface. For the convenience of target-library implementors, it contains fields for caching the heap-allocated data objects that may be produced through the interface. These are owned by the context and are deleted when it is destructed.

80d ⟨`MachineContext` *protected matter* 80d⟩≡ (78b)
```
mutable Printer *cached_printer;
mutable CPrinter *cached_c_printer;
mutable CodeFin *cached_code_fin;
```

15.4 Header file `contexts.h`

The header file for contexts has the following outline:

80e ⟨*machine/contexts.h* 80e⟩≡

```
/* file "machine/contexts.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_CONTEXT_H
#define MACHINE_CONTEXT_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "machine/contexts.h"
#endif

#include <machine/substrate.h>
#include <machine/problems.h>
#include <machine/opnd.h>
#include <machine/instr.h>
#include <machine/reg_info.h>
#include <machine/code_fin.h>
#include <machine/printer.h>
#include <machine/c_printer.h>

⟨class Context 78a⟩

⟨class MachineContext 78b⟩

⟨context creation 77a⟩

⟨context-creator registry 77b⟩

#endif /* MACHINE_CONTEXT_H */
```

16 Substrate Encapsulation

As has mentioned earlier, Machine SUIF can be viewed as an implementation of the OPI on a substrate consisting of C++ library facilities and the core of the Stanford SUIF infrastructure. The `machine` library serves as the layer that encapsulates the substrate for the rest of Machine SUIF, so that libraries and passes are easy to move to OPI implementations that are based on different substrates.

This section collects a number of declarations that help with substrate encapsulation. The header file (`substrate.h`) that it generates doesn't depend on any OPI types defined elsewhere. It is meant to be included by other modules in the `machine` library, including the hoof-generated module that defines many of the IR types in the OPI.

The `substrate.h` header file is also included by `machine.h`, which collects the exports of the `machine` library. Clients of the library therefore see these declarations when they include `machine.h`.

16.1 OPI Types

Several OPI types are implemented in Machine SUIF as synonyms for SUIF types.

81 ⟨*renamed types* 81⟩≡ (86e)
```
typedef AnnotableObject IrObject;
typedef ProcedureSymbol ProcSym;
typedef ProcedureDefinition ProcDef;
typedef ProcedureDefinition OptUnit;
typedef PointerType PtrType;
typedef Symbol Sym;
typedef SymbolTable SymTable;
typedef SymbolTable ScopeTable;
typedef VariableSymbol VarSym;
typedef VariableDefinition VarDef;
typedef CodeLabelSymbol LabelSym;
```

Since the OPI's type abstraction, `TypeId`, is used without an explicit level of indirection, we make it a synonym for `Type*` instead of `Type`.

82a ⟨*type* `TypeId` 82a⟩≡ (86e)
```
typedef Type* TypeId;
```

16.2 Class `Integer`

In order to support cross-compilation without being limited by the host's numeric characteristics, the OPI makes use of extended-precision integers. The implementation of OPI-class `Integer` derives from SUIF's infinite-integer class.

82b ⟨*class* `Integer` 82b⟩≡ (86e)
```
class Integer : public IInteger {
  public:
    Integer() { }                               // returns the empty string
    Integer(const char *chars)                  : IInteger(chars) { }
    Integer(const IInteger &i_integer)          : IInteger(i_integer) { }

    Integer(signed char integral)               : IInteger(integral) { }
    Integer(unsigned char integral)             : IInteger(integral) { }
    Integer(short integral)                     : IInteger(integral) { }
    Integer(unsigned short integral)            : IInteger(integral) { }
    Integer(int integral)                       : IInteger(integral) { }
    Integer(unsigned int integral)              : IInteger(integral) { }
    Integer(long integral)                      : IInteger(integral) { }
    Integer(unsigned long integral)             : IInteger(integral) { }
#ifdef LONGLONG
    Integer(LONGLONG integral)                  : IInteger(integral) { }
    Integer(unsigned LONGLONG integral)         : IInteger(integral) { }
#endif
    Integer(const char *initial_string, int base = 10);

    const char* chars() const { return c_string_int(); }
};
```

16.3 Class `IdString`

The OPI class `IdString` derived directly from SUIF's "lexicon string" class. Values are hashed into a global lexicon, so that comparison for equality takes place in constant time.

82c ⟨*class* IdString 82c⟩≡ (86e)

```
class IdString : public LString {
  public:
    IdString() { }                              // returns the empty string
    IdString(const IdString &id_string) : LString(id_string) { }
    IdString(const LString &l_string)   : LString(l_string) { }
    IdString(const String &string)      : LString(string) { }
    IdString(const char *chars)         : LString(chars) { }

    const char* chars() const { return c_str(); }
    bool is_empty()     const { return length() == 0; }
};

extern const IdString empty_id_string;
```

We provide a less-than operator on `IdString` values in a manner that allows them to be used in STL sets and maps.

83a ⟨*string function* 83a⟩≡ (86e)

```
namespace std {
class less<IdString> : public binary_function<IdString, IdString, bool> {
  public:
    bool operator()(const IdString &s1, const IdString &s2) const
        { return s1.get_ordinal() < s2.get_ordinal(); }
};
}
```

16.4 Living with C++ Container Classes

SUIF's STL substitutes. To counteract the vagaries of C++ implementations, SUIF has its own STL-like container classes. In order to be free to switch between the native STL classes and SUIF's substitutes, Machine SUIF uses its own container names.

83b ⟨*container-class defines* 83b⟩≡ (86e)

```
#define List list
#define Vector vector
#define Set set
#define Map map
#define HashMap suif_hash_map
```

Iterator arithmetic. A certain kind of C++ container-class "iterator" can be incremented or decremented but can't be used in other pointer arithmetic. The following functions can be applied to such an iterator (which in OPI jargon we call a *handle*) to produce a handle on the preceding or following element without side-affecting the argument.

83c ⟨*iterator helpers* 83c⟩≡ (86e)
```
template <class Iterator>
Iterator
before(Iterator iterator)
{
    return --iterator;
}

template <class Iterator>
Iterator
after(Iterator iterator)
{
    return ++iterator;
}
```

16.5 C++ and Base SUIF Header Files

This module takes care of including header C++ and base SUIF header files that are widely used in Machine SUIF. When adding a new C++ module to the system, it is usually not necessary to worry about its dependence any specific substrate header files, since they come in via inclusion of `<machine/machine.h>`.

84 ⟨*substrate header includes* 84⟩≡ (86e)
```
#include <stdlib.h>
#include <set.h>
#include <map.h>
#include <functional>
                                // Following suppresses inclusion of <vector>
                                // under SGI STL, which defines bit_vector,
                                // which conflicts with a basesuif typedef.
#define __SGI_STL_VECTOR
#include <vector.h>

#include <common/i_integer.h>
#include <common/formatted.h>
#include <common/suif_vector.h>
#include <common/suif_list.h>
#include <common/suif_map.h>
#include <common/suif_indexed_list.h>
#include <bit_vector/bit_vector.h>
#include <utils/type_utils.h>
#include <utils/symbol_utils.h>
#include <utils/expression_utils.h>
extern "C" void init_utils(SuifEnv*);

#include <suifkernel/suif_env.h>
#include <suifkernel/dll_subsystem.h>
#include <suifkernel/command_line_parsing.h>
#include <basicnodes/basic.h>
#include <basicnodes/basic_constants.h>
#include <suifnodes/suif.h>
#include <typebuilder/type_builder.h>
#include <suifcloning/cloner.h>

#include <basicnodes/basic_factory.h>
```

```
#include <suifnodes/suif_factory.h>
```

16.6 Accessing SUIF Value Descriptors

SUIF has a number of IR classes devoted to describing static data values such as those that appear in C initializers. These are rarely used in Machine SUIF, and they ought to be better encapsulated than they are at present. Meanwhile, here are some functions used for access SUIF's value descriptors. The important classes are `ValueBlock`, which is the root class for value descriptions, and `VarDef` which connects a variable symbol with its statically-defined value (if it has one). A `MultiValueBlock` describes an aggregate of values, while a `RepeatValueBlock` represents a series of copies of the same value. An `ExpressionValueBlock` used an expression to describe an initial value. In this case, the easiest way to extract a description of the value is to use a Machine SUIF operand.

85a ⟨*accessing value descriptors* 85a⟩≡ (86e)

```
VarDef* get_def(VarSym*);
ProcSym* get_proc_sym(ProcDef*);

ValueBlock* get_init(VarDef* d);
int get_bit_alignment(VarDef* d);

int subblocks_size(MultiValueBlock *mvb);

ValueBlock* get_subblock(MultiValueBlock *mvb, int i);

Integer get_subblock_offset(MultiValueBlock *mvb, int i);

int get_repetition_count(RepeatValueBlock *rvb);

ValueBlock* get_subblock(RepeatValueBlock *rvb);

TypeId get_type(ValueBlock *vb);

class Opnd;
Opnd get_value(ExpressionValueBlock *svb);
```

16.7 Miscellany

Current compilation environment. The following global variable connects to the substrate by recording the current "SUIF environment", which holds the current file set, modules loaded, object factories, and so on.

85b ⟨*SUIF environment* 85b⟩≡ (86e)

```
extern SuifEnv* the_suif_env;
```

Extracting the name of a named object. The overloaded function `get_name` provides a uniform way to extract the name of a source file or procedure or to get the name of a symbol or type. (Symbols and types are instances of SUIF's `SymbolTableObject` class.)

85c ⟨*function* `get_name` 85c⟩≡ (86e)

```
IdString get_name(FileBlock*);
IdString get_name(ProcDef*);
IdString get_name(SymbolTableObject*);
```

SUIF object-class identification. SUIF objects are built on a reflection system that allows easy run-time identification of their classes. It is useful during debugging to the able to print the class name of a SUIF object.

86a ⟨*class identification* 86a⟩≡ (86e)
```
const char* get_class_name(SuifObject*);
```

An IR-class forward reference. Class `AnyBody` requires a forward reference to `InstrList`. Since `substrate.h` is one of the few headers included before the hoof-generated OPI types are defined, we include this forward reference here.

86b ⟨*forward reference* 86b⟩≡ (86e)
```
class InstrList;
```

Command-line option helpers. Fetching the strings accumulated in an `OptionString` (one of the non-terminal descriptor in a SUIF command-line grammar) is a bit clumsy. This helper abbreviates the procedure a bit.

86c ⟨*command-line-option helpers* 86c⟩≡ (86e) 86d ▷
```
IdString get_option_string_value(OptionString*, int pos = 0);
```

The next helper processes zero, one or two files names from a command line. It uses `the_suif_env` to decide whether an input file is appropriate and behaves accordingly. If no input is in the environment already and there is at least one name given, it reads the SUIF file into the environment. If an output file is provided, it returns its name. Otherwise, it returns the empty string.

86d ⟨*command-line-option helpers* 86c⟩+≡ (86e) ◁ 86c
```

IdString process_file_names(OptionString *file_names);
```

16.8 Header file `substrate.h`

The substrate header file has the following outline:

86e ⟨*machine/substrate.h* 86e⟩≡
```
/* file "machine/substrate.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_SUBSTRATE_H
#define MACHINE_SUBSTRATE_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "machine/substrate.h"
#endif

⟨substrate header includes 84⟩

⟨forward reference 86b⟩

⟨iterator helpers 83c⟩
```

```
⟨renamed types 81⟩

⟨type TypeId 82a⟩

⟨class Integer 82b⟩

⟨class IdString 82c⟩

⟨string function 83a⟩

⟨accessing value descriptors 85a⟩

⟨SUIF environment 85b⟩

⟨function get_name 85c⟩

⟨class identification 86a⟩

⟨command-line-option helpers 86c⟩

⟨container-class defines 83b⟩

⟨SUIF environment 85b⟩

⟨class identification 86a⟩

#endif /* MACHINE_SUBSTRATE_H */
```

17 Library initialization

Before you can start using the facilities of the `machine` library, the library must initialize some parts of itself. In SUIF, this is performed by defining an `init_`*libname* routine.

87a ⟨*machine library initialization* 87a⟩≡ (88)

```
extern "C" void init_machine(SuifEnv* suif_env);
```

We also use the initialization header file as a gathering place for defining string constants used in Machine SUIF. These string constants include those used as the names for annotations. You should consult the indicated section, when appropriate, to learn about the meaning and use of these string constants.

87b ⟨*machine string constants* 87b⟩≡ (88)

```
extern IdString k_target_lib;           // see contexts.h.nw
extern IdString k_target_type_ptr;      // see types.h.nw
extern IdString k_generic_types;        // see types.h.nw
extern IdString k_enable_exceptions;    // see codegen.cc
extern IdString k_stack_frame_info;     // see codegen.h.nw

extern IdString k_empty_string;         // I.e. "" (used for defaults)

extern IdString k_history;              // Note listing compilation history
extern IdString k_line;                 // Note flagging .line directive

extern IdString k_comment;              // Note containing comment text
```

```
extern IdString k_mbr_target_def;       //   "   marking mbr target calculation
extern IdString k_mbr_index_def;        //   "   (deprecated: use preceding key)
extern IdString k_mbr_table_use;        //   "   marking mbr dispatch table use
extern IdString k_instr_mbr_tgts;       //   "   giving mbr case values/labels
extern IdString k_instr_opcode;         //   "   for generic-instr opcode_name
extern IdString k_instr_opcode_exts;    //   "   for opcode extensions
extern IdString k_proc_entry;           //   "   marking procedure entry pt.
extern IdString k_regs_used;            //   "   giving regs used at call
extern IdString k_regs_defd;            //   "   giving regs def'd at call
extern IdString k_instr_ret;            //   "   marking return instruction
extern IdString k_incomplete_proc_exit; //   "   marking incomplete proc exit
extern IdString k_header_trailer;       //   "   on instruction added by fin
extern IdString k_builtin_args;         //   "   giving args for builtin "call"
extern IdString k_param_reg;            //   "   giving hard reg for parameter
extern IdString k_vr_count;             //   "   giving unit's virtual-reg count
extern IdString k_stdarg_offset;        //   "   giving unnamed-arg frame offset

extern IdString k_const;                // keyword "const"

extern IdString k_dense;                // NatSetNote tag
extern IdString k_dense_inverse;        //     "       "
extern IdString k_sparse;               //     "       "
extern IdString k_sparse_inverse;       //     "       "

extern IdString k_call_target;          // Note attaching target symbol to call
extern IdString k_param_init;           //   "  marking instr to init proc param
```

The machine library initialization header has the following layout:

88 ⟨*machine/init.h* 88⟩≡

```
/* file "machine/init.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_INIT_H
#define MACHINE_INIT_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "machine/init.h"
#endif

#include <machine/substrate.h>

⟨machine library initialization 87a⟩

⟨machine string constants 87b⟩

#endif /* MACHINE_INIT_H */
```

18 Header file for the machine library

The following is the header file is for use by other libraries and passes that depend upon the machine library. It is never included in any implementation file within the machsuif/machine directory.

89a ⟨*machine/machine.h* 89a⟩≡
```
/* file "machine/machine.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_MACHINE_H
#define MACHINE_MACHINE_H

#include <machine/copyright.h>

#include <machine/substrate.h>
#include <machine/problems.h>
#include <machine/opnd.h>
#include <machine/machine_ir.h>
#include <machine/machine_ir_factory.h>
#include <machine/init.h>
#include <machine/nat_set.h>
#include <machine/note.h>
#include <machine/util.h>
#include <machine/c_printer.h>
#include <machine/instr.h>
#include <machine/reg_info.h>
#include <machine/opcodes.h>
#include <machine/types.h>
#include <machine/printer.h>
#include <machine/code_fin.h>
#include <machine/contexts.h>

#endif /* MACHINE_MACHINE_H */
```

19 Connection to the Base SUIF Pass Mechanism

As described in Appendix A of the *OPI User's Guide*, Machine SUIF passes wrap OPI passes in a layer that connects them to the base SUIF pass management machinery.

Header file pass.h is for inclusion by the suif_pass module associated with each Machine-SUIF pass. It imports aspects of base SUIF that are only needed in the pass-wrapping modules, so we put it in a header file of its own.

89b ⟨*machine/pass.h* 89b⟩≡
```
/* file "machine/pass.h" */

⟨Machine-SUIF copyright 95⟩

#ifndef MACHINE_PASS_H
#define MACHINE_PASS_H

#include <machine/copyright.h>

#include <common/suif_vector.h>         // needed by command_line_parsing.h
```

```
// Header files for pass construction under SUIF
#include <suifkernel/command_line_parsing.h>
#include <suifkernel/module_subsystem.h>
#include <suifkernel/token_stream.h>
#include <suifpasses/suifpasses.h>

#endif /* MACHINE_PASS_H */
```

20 Hoof Specification of Machine-SUIF IR Classes

The Machine-SUIF realization of the IR classes in the OPI is expressed using SUIF's hoof specification language. Here, we provide only the barest details of our implementation. Please see the documentation of SUIF for a detailed explanation of hoof and its syntax.

20.1 Class `Instr`

Recall the categorization of machine instructions given in Section 3:

- Active instructions (real machine operations)
  - (`alm`) Arithmetic, logical, and memory instructions
    - (`alu`) Arithmetic and logical
    - (`mem`) Load and store
  - (`cti`) Control-transfer instructions
- Inactive instructions
  - (`label`) Label instructions
  - (`dot`) Assembler pseudo-operations[8]

The implementation classes for machine instructions follow the above outline and adhere closely to the explanation in Section 3. As noted there, the only instruction class that the OPI user sees is `Instr`.

90 ⟨*class* `Instr` *and subclasses* 90⟩≡ (94b)

```
abstract Instr : ScopedObject {
    int opcode;
};

concrete InstrAlm : Instr {
    vector<IrOpnd* definer> srcs;
    vector<IrOpnd* definer> dsts;
CPP_DECLARE
  public:
    suif_vector<IrOpnd*>& srcs() { return _srcs; }
    suif_vector<IrOpnd*>& dsts() { return _dsts; }
CPP_DECLARE
};

concrete InstrCti : InstrAlm {
    Symbol* reference target;
```

[8] Many assemblers use a leading period (".") to identify non-code directives.

```
};

concrete InstrLabel : Instr {
    CodeLabelSymbol* definer label in defined_labels;
};

concrete InstrDot : Instr {
    vector<IrOpnd* definer> srcs;
CPP_DECLARE
  public:
    suif_vector<IrOpnd*>& srcs() { return _srcs; }
CPP_DECLARE
};
```

Note that operand vectors within an instruction are treated as defining points for their contents, not as ordinary references. This ensures that operands are cloned when the instruction that contains them is cloned. That is important for mutable operands (currently just address expressions), which must not be left shared between an original instruction and its clone.

20.2 Class IrOpnd

As shown in Section 4.3, the OPI class `Opnd` is implemented in Machine SUIF using a SUIF-like object class called `IrOpnd`. Specifically, the only instance field in an `Opnd` value is a pointer to an `IrOpnd` object.

Class `IrOpnd` is hoof-generated. It has a subclass for each kind of `Opnd`. Each of these subclasses has methods supporting the implementation of the `get_kind`, `get_type`, and equality-testing functions on operands. The subclass for address expressions, `OpndAddrExp` also provides a cloning method. (Since these are the only mutable operands at present, they are the only ones for which cloning is non-trivial.)

91a ⟨*class* `IrOpnd` 91a⟩≡ (94b)

```
abstract IrOpnd : SymbolTableObject {
    virtual int kind;
    virtual Type* reference type;

CPP_DECLARE
  public:
    friend class Opnd;
    virtual bool operator==(const IrOpnd &other) const
        { suif_assert(false); return false; }
CPP_DECLARE
};
```

91b ⟨*class* `OpndReg` 91b⟩≡ (94b)

```
concrete OpndReg : IrOpnd {
    Type* reference type implements type;
    int reg;
CPP_DECLARE
  public:
    int get_kind() const
        { return (_reg < 0) ? opnd::REG_VIRTUAL : opnd::REG_HARD; }
    virtual bool operator==(const IrOpnd &other) const
        { suif_assert(is_kind_of<OpndReg>(&other));
          return _reg == ((const OpndReg&)other)._reg; }
CPP_DECLARE
};
```

92a ⟨*class* OpndVar 92a⟩≡ (94b)

```
concrete OpndVar : IrOpnd {
    VariableSymbol* reference var;
CPP_DECLARE
  public:
    int   get_kind() const { return opnd::VAR; }
    Type* get_type() const { return _var->get_type(); }
    virtual bool operator==(const IrOpnd &other) const
        { suif_assert(is_kind_of<OpndVar>(&other));
          return _var == ((const OpndVar&)other)._var; }
CPP_DECLARE
};
```

92b ⟨*class* OpndImmedInteger 92b⟩≡ (94b)

```
concrete OpndImmedInteger : IrOpnd {
    Type* reference type implements type;
    IInteger immed;
CPP_DECLARE
  public:
    int get_kind() const { return opnd::IMMED_INTEGER; }
    virtual bool operator==(const IrOpnd &other) const
        { suif_assert(is_kind_of<OpndImmedInteger>(&other));
          return _immed == ((const OpndImmedInteger&)other)._immed; }
CPP_DECLARE
};
```

92c ⟨*class* OpndImmedString 92c⟩≡ (94b)

```
concrete OpndImmedString : IrOpnd {
    Type* reference type implements type;
    LString immed;
CPP_DECLARE
  public:
    int get_kind() const { return opnd::IMMED_STRING; }
    virtual bool operator==(const IrOpnd &other) const
        { suif_assert(is_kind_of<OpndImmedString>(&other));
          return _immed == ((const OpndImmedString&)other)._immed; }
CPP_DECLARE
};
```

92d ⟨*class* OpndAddrSym 92d⟩≡ (94b)

```
concrete OpndAddrSym : IrOpnd {
    Symbol* reference sym;
CPP_DECLARE
  public:
    int   get_kind() const { return opnd::ADDR_SYM; }
    Type* get_type() const { return type_addr(); }
    virtual bool operator==(const IrOpnd &other) const
        { suif_assert(is_kind_of<OpndAddrSym>(&other));
          return _sym == ((const OpndAddrSym&)other)._sym; }
CPP_DECLARE
};
```

93a ⟨*class* `OpndAddrExp` *and subclasses* 93a⟩≡ (94b)

```
      abstract OpndAddrExp : IrOpnd {
          int kind implements kind;
          Type* reference deref_type;
          vector<IrOpnd* reference> srcs;
      CPP_DECLARE
        public:
          suif_vector<IrOpnd*>& srcs() { return _srcs; }
          virtual bool operator==(const IrOpnd &) const;
          OpndAddrExp* clone() const;
          Type* get_type() const { return type_addr(); }
      CPP_DECLARE
      CPP_BODY
          bool
          OpndAddrExp::operator==(const IrOpnd &other) const
          {
              suif_assert(get_kind() == other.get_kind());
              const OpndAddrExp &aeo = (const OpndAddrExp&)other;
              if (get_deref_type() != aeo.get_deref_type())
                  return false;
              suif_assert(_srcs.size() == aeo._srcs.size());
              suif_vector<IrOpnd*>::const_iterator sit =     _srcs.begin();
              suif_vector<IrOpnd*>::const_iterator oit = aeo._srcs.begin();
              for ( ; sit != _srcs.end(); sit++, oit++)
                  if (!(*sit == *oit))
                      return false;
              return true;
          }
          OpndAddrExp* OpndAddrExp::clone() const
          {
              OpndAddrExp *clone = create_opnd_addr_exp(the_suif_env, 0, NULL);
              clone->set_kind(get_kind());
              clone->set_deref_type(get_deref_type());
              for (unsigned i = 0; i < get_src_count(); i++)
                  clone->append_src(get_src(i));
              return clone;
          }
      CPP_BODY
      };
```

20.3 Class `AnyBody`

Class `AnyBody` is the generic type for procedure bodies in Machine SUIF. Here is how the class is built on top of SUIF.

93b ⟨*class* `AnyBody` 93b⟩≡ (94b)

```
  concrete AnyBody : ExecutionObject {
      CPP_DECLARE
        public:
          virtual InstrList* to_instr_list() { suif_assert(false); return NULL; }
      CPP_DECLARE
  };
```

20.4 Class InstrList

The most basic subclass of AnyBody is InstrList, which represents a simple instruction list.

94a ⟨*class* InstrList 94a⟩≡ (94b)

```
concrete InstrList : AnyBody {
    list<Instr* owner> instrs;
    CPP_DECLARE
      public:
        list<Instr*>& instrs() { return _instrs; }
        InstrList* to_instr_list();
    CPP_DECLARE
    CPP_BODY
        extern InstrList* instr_list_to_instr_list(InstrList*);
        InstrList*
        InstrList::to_instr_list()
        {
            return instr_list_to_instr_list(this);
        }
    CPP_BODY
};
```

20.5 Overall hoof specification for module machine

The combined hoof grammar for the machine module has the following layout. (The inclusion of suifnodes/suif.h near the start of module machine_ir is just for the benefit of machine/substrate.h.)

94b ⟨*machine/machine_ir.hoof* 94b⟩≡

```
# file "machine_ir.hoof"
#
#    Copyright (c) 2000 The President and Fellows of Harvard College
#
#    All rights reserved.
#
#    This software is provided under the ter described in
#    the "machine/copyright.h" include file.


#include "basicnodes/basic.hoof"

module machine_ir {

    include <functional>;
    include <basicnodes/basic.h>;
    include <suifnodes/suif.h>;
    include <machine/substrate.h>;
    include <machine/types.h>;
    include <machine/opnd.h>;

    import basicnodes;

# Universal machine-level ExecutionObject

⟨class AnyBody 93b⟩
```

```
# Instruction-list class

⟨class InstrList 94a⟩

# Instruction classes

⟨class Instr and subclasses 90⟩

# Operand classes

⟨class IrOpnd 91a⟩
⟨class OpndVar 92a⟩
⟨class OpndReg 91b⟩
⟨class OpndImmedInteger 92b⟩
⟨class OpndImmedString 92c⟩
⟨class OpndAddrSym 92d⟩
⟨class OpndAddrExp and subclasses 93a⟩
}
```

21 Copyright

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

95 ⟨*Machine-SUIF copyright* 95⟩≡ (16b 30 33b 35c 39b 43b 47b 50b 56b 63b 66 75d 80e 86e 88 89)

```
/*
    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.
*/
```

22 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.

References

[1] P. Briggs and L. Torczon. An efficient representation of sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1-4), March-December 1993, pp. 59-70

[2] G. Holloway and A Dimock. *The Machine-SUIF Bit-vector Data-flow Analysis Library.* The Machine SUIF documentation set, Harvard University, 2002.

[3] G. Holloway and M. D. Smith. *The Machine-SUIF Control Flow Graph Library* The Machine SUIF documentation set, Harvard University, 2002.

[4] G. Holloway and M. D. Smith. *An Extender's Guide to the Optimization Programming Interface and Target Descriptions.* The Machine-SUIF documentation set, Harvard University, 2002.

[5] G. Holloway and M. D. Smith. *A User's Guide to the Optimization Programming Interface.* The Machine-SUIF documentation set, Harvard University, 2002.