LLVM Programmer's Manual

1. Introduction
2. General Information
    - The C++ Standard Template Library
3. Important and useful LLVM APIs
    - The `isa<>`, `cast<>` and `dyn_cast<>` templates
    - The `DEBUG()` macro & `-debug` option
        - Fine grained debug info with `DEBUG_TYPE` and the `-debug-only` option
    - The `Statistic` template & `-stats` option
4. Helpful Hints for Common Operations
    - Basic Inspection and Traversal Routines
        - Iterating over the `BasicBlocks` in a `Function`
        - Iterating over the `Instructions` in a `BasicBlock`
        - Iterating over the `Instructions` in a `Function`
        - Turning an iterator into a class pointer
        - Finding call sites: a more complex example
        - Iterating over def-use & use-def chains
    - Making simple changes
        - Creating and inserting new `Instructions`
        - Deleting `Instructions`
        - Replacing an `Instruction` with another `Value`
5. The Core LLVM Class Hierarchy Reference
    - The `Value` class
        - The `User` class
            - The `Instruction` class
                -
            - The `GlobalValue` class
                - The `BasicBlock` class
                - The `Function` class
                - The `GlobalVariable` class
            - The `Module` class
            - The `Constant` class
                -
                -
        - The `Type` class
        - The `Argument` class
    - The `SymbolTable` class
    - The `ilist` and `iplist` classes
        - Creating, inserting, moving and deleting from LLVM lists
    - Important iterator invalidation semantics to be aware of

Written by Chris Lattner, Dinakar Dhurjati, and Joel Stanley

Introduction

This document is meant to highlight some of the important classes and interfaces available in the LLVM source-base. This manual is not intended to explain what LLVM is, how it works, and what LLVM code looks like. It assumes that you know the basics of LLVM and are interested in writing transformations or otherwise analyzing or manipulating the code.

This document should get you oriented so that you can find your way in the continuously growing source code that makes up the LLVM infrastructure. Note that this manual is not intended to serve as a replacement for reading the source code, so if you think there should be a method in one of these classes to do something, but it's not listed, check the source. Links to the doxygen sources are provided to make this as easy as possible.

The first section of this document describes general information that is useful to know when working in the LLVM infrastructure, and the second describes the Core LLVM classes. In the future this manual will be extended with information describing how to use extension libraries, such as dominator information, CFG traversal routines, and useful utilities like the `InstVisitor` template.

General Information

This section contains general information that is useful if you are working in the LLVM source-base, but that isn't specific to any particular API.

The C++ Standard Template Library

LLVM makes heavy use of the C++ Standard Template Library (STL), perhaps much more than you are used to, or have seen before. Because of this, you might want to do a little background reading in the techniques used and capabilities of the library. There are many good pages that discuss the STL, and several books on the subject that you can get, so it will not be discussed in this document.

Here are some useful links:

1. Dinkumware C++ Library reference - an excellent reference for the STL and other parts of the standard C++ library.
2. C++ In a Nutshell - This is an O'Reilly book in the making. It has a decent Standard Library Reference that rivals Dinkumware's, and is actually free until the book is published.
3. C++ Frequently Asked Questions
4. SGI's STL Programmer's Guide - Contains a useful Introduction to the STL.
5. Bjarne Stroustrup's C++ Page

You are also encouraged to take a look at the LLVM Coding Standards guide which focuses on how to write maintainable code more than where to put your curly braces.

Important and useful LLVM APIs

Here we highlight some LLVM APIs that are generally useful and good to know about when writing transformations.

The isa<>, cast<> and dyn_cast<> templates

The LLVM source-base makes extensive use of a custom form of RTTI. These templates have many similarities to the C++ `dynamic_cast<>` operator, but they don't have some drawbacks (primarily stemming from the fact that `dynamic_cast<>` only works on classes that have a v-table). Because they are used so often, you must know what they do and how they work. All of these templates are defined in the `Support/Casting.h` file (note that you very rarely have to include this file directly).

`isa<>`:
: The `isa<>` operator works exactly like the Java "`instanceof`" operator. It returns true or false depending on whether a reference or pointer points to an instance of the specified class. This can be very useful for constraint checking of various sorts (example below).

`cast<>`:
The `cast<>` operator is a "checked cast" operation. It converts a pointer or reference from a base class to a derived cast, causing an assertion failure if it is not really an instance of the right type. This should be used in cases where you have some information that makes you believe that something is of the right type. An example of the `isa<>` and `cast<>` template is:

```
static bool isLoopInvariant(const Value *V, const Loop *L) {
  if (isa<Constant>(V) || isa<Argument>(V) || isa<GlobalValue>(V))
    return true;

  // Otherwise, it must be an instruction...
  return !L->contains(cast<Instruction>(V)->getParent());
```

Note that you should **not** use an `isa<>` test followed by a `cast<>`, for that use the `dyn_cast<>` operator.

`dyn_cast<>`:
The `dyn_cast<>` operator is a "checking cast" operation. It checks to see if the operand is of the specified type, and if so, returns a pointer to it (this operator does not work with references). If the operand is not of the correct type, a null pointer is returned. Thus, this works very much like the `dynamic_cast` operator in C++, and should be used in the same circumstances. Typically, the `dyn_cast<>` operator is used in an `if` statement or some other flow control statement like this:

```
  if (AllocationInst *AI = dyn_cast<AllocationInst>(Val)) {
    ...
  }
```

This form of the `if` statement effectively combines together a call to `isa<>` and a call to `cast<>` into one statement, which is very convenient.

Another common example is:

```
  // Loop over all of the phi nodes in a basic block
  BasicBlock::iterator BBI = BB->begin();
  for (; PHINode *PN = dyn_cast<PHINode>(BBI); ++BBI)
    cerr << *PN;
```

Note that the `dyn_cast<>` operator, like C++'s `dynamic_cast` or Java's `instanceof` operator, can be abused. In particular you should not use big chained `if/then/else` blocks to check for lots of different variants of classes. If you find yourself wanting to do this, it is much cleaner and more efficient to use the InstVisitor class to dispatch over the instruction type directly.

`cast_or_null<>`:
The `cast_or_null<>` operator works just like the `cast<>` operator, except that it allows for a null pointer as an argument (which it then propagates). This can sometimes be useful, allowing you to combine several null checks into one.

`dyn_cast_or_null<>`:
The `dyn_cast_or_null<>` operator works just like the `dyn_cast<>` operator, except that it allows for a null pointer as an argument (which it then propagates). This can sometimes be useful, allowing you to combine several null checks into one.

These five templates can be used with any classes, whether they have a v-table or not. To add support for these templates, you simply need to add `classof` static methods to the class you are interested casting to. Describing this is currently outside the scope of this document, but there are lots of examples in the LLVM source base.

The `DEBUG()` macro & `-debug` option

Often when working on your pass you will put a bunch of debugging printouts and other code into your pass. After you get it working, you want to remove it... but you may need it again in the future (to work out new bugs that you run across).

Naturally, because of this, you don't want to delete the debug printouts, but you don't want them to always be noisy. A standard compromise is to comment them out, allowing you to enable them if you need them in the future.

The "Support/Debug.h" file provides a macro named `DEBUG()` that is a much nicer solution to this problem. Basically, you can put arbitrary code into the argument of the `DEBUG` macro, and it is only executed if '`opt`' (or any other tool) is run with the '`-debug`' command line argument:

```
...
DEBUG(std::cerr << "I am here!\n");
...
```

Then you can run your pass like this:

```
$ opt < a.bc > /dev/null -mypass
  <no output>
$ opt < a.bc > /dev/null -mypass -debug
  I am here!
$
```

Using the `DEBUG()` macro instead of a home-brewed solution allows you to now have to create "yet another" command line option for the debug output for your pass. Note that `DEBUG()` macros are disabled for optimized builds, so they do not cause a performance impact at all (for the same reason, they should also not contain side-effects!).

One additional nice thing about the `DEBUG()` macro is that you can enable or disable it directly in gdb. Just use "`set DebugFlag=0`" or "`set DebugFlag=1`" from the gdb if the program is running. If the program hasn't been started yet, you can always just run it with `-debug`.

Fine grained debug info with `DEBUG_TYPE()` and the `-debug-only` option

Sometimes you may find yourself in a situation where enabling `-debug` just turns on **too much** information (such as when working on the code generator). If you want to enable debug information with more fine-grained control, you define the `DEBUG_TYPE` macro and the `-debug` only option as follows:

```
...
DEBUG(std::cerr << "No debug type\n");
#undef  DEBUG_TYPE
#define DEBUG_TYPE "foo"
DEBUG(std::cerr << "'foo' debug type\n");
#undef  DEBUG_TYPE
#define DEBUG_TYPE "bar"
DEBUG(std::cerr << "'bar' debug type\n");
#undef  DEBUG_TYPE
#define DEBUG_TYPE ""
DEBUG(std::cerr << "No debug type (2)\n");
...
```

Then you can run your pass like this:

```
$ opt < a.bc > /dev/null -mypass
  <no output>
$ opt < a.bc > /dev/null -mypass -debug
  No debug type
  'foo' debug type
```

```
  'bar' debug type
  No debug type (2)
$ opt < a.bc > /dev/null -mypass -debug-only=foo
  'foo' debug type
$ opt < a.bc > /dev/null -mypass -debug-only=bar
  'bar' debug type
$
```

Of course, in practice, you should only set `DEBUG_TYPE` at the top of a file, to specify the debug type for the entire module (if you do this before you `#include "Support/Debug.h"`, you don't have to insert the ugly `#undef`'s). Also, you should use names more meaningful that "foo" and "bar", because there is no system in place to ensure that names do not conflict: if two different modules use the same string, they will all be turned on when the name is specified. This allows all, say, instruction scheduling, debug information to be enabled with `-debug-type=InstrSched`, even if the source lives in multiple files.

The `Statistic` template & `-stats` option

The "Support/Statistic.h" file provides a template named `Statistic` that is used as a unified way to keeping track of what the LLVM compiler is doing and how effective various optimizations are. It is useful to see what optimizations are contributing to making a particular program run faster.

Often you may run your pass on some big program, and you're interested to see how many times it makes a certain transformation. Although you can do this with hand inspection, or some ad-hoc method, this is a real pain and not very useful for big programs. Using the `Statistic` template makes it very easy to keep track of this information, and the calculated information is presented in a uniform manner with the rest of the passes being executed.

There are many examples of `Statistic` users, but this basics of using it are as follows:

1. Define your statistic like this:

   ```
   static Statistic<> NumXForms("mypassname", "The # of times I did stuff");
   ```

   The `Statistic` template can emulate just about any data-type, but if you do not specify a template argument, it defaults to acting like an unsigned int counter (this is usually what you want).

2. Whenever you make a transformation, bump the counter:

   ```
   ++NumXForms;   // I did stuff
   ```

That's all you have to do. To get '`opt`' to print out the statistics gathered, use the '`-stats`' option:

```
$ opt -stats -mypassname < program.bc > /dev/null
 ... statistic output ...
```

When running `gccas` on a C file from the SPEC benchmark suite, it gives a report that looks like this:

```
  7646 bytecodewriter  - Number of normal instructions
   725 bytecodewriter  - Number of oversized instructions
129996 bytecodewriter  - Number of bytecode bytes written
  2817 raise           - Number of insts DCEd or constprop'd
  3213 raise           - Number of cast-of-self removed
  5046 raise           - Number of expression trees converted
    75 raise           - Number of other getelementptr's formed
   138 raise           - Number of load/store peepholes
    42 deadtypeelim    - Number of unused typenames removed from symtab
   392 funcresolve     - Number of varargs functions resolved
    27 globaldce       - Number of global variables removed
     2 adce            - Number of basic blocks removed
   134 cee             - Number of branches revectored
```

```
  49 cee             - Number of setcc instruction eliminated
 532 gcse            - Number of loads removed
2919 gcse            - Number of instructions removed
  86 indvars         - Number of canonical indvars added
  87 indvars         - Number of aux indvars removed
  25 instcombine     - Number of dead inst eliminate
 434 instcombine     - Number of insts combined
 248 licm            - Number of load insts hoisted
1298 licm            - Number of insts hoisted to a loop pre-header
   3 licm            - Number of insts hoisted to multiple loop preds (bad, no loop pre-header)
  75 mem2reg         - Number of alloca's promoted
1444 cfgsimplify     - Number of blocks simplified
```

Obviously, with so many optimizations, having a unified framework for this stuff is very nice. Making your pass fit well into the framework makes it more maintainable and useful.

Helpful Hints for Common Operations

This section describes how to perform some very simple transformations of LLVM code. This is meant to give examples of common idioms used, showing the practical side of LLVM transformations.

Because this is a "how-to" section, you should also read about the main classes that you will be working with. The Core LLVM Class Hierarchy Reference contains details and descriptions of the main classes that you should know about.

Basic Inspection and Traversal Routines

The LLVM compiler infrastructure have many different data structures that may be traversed. Following the example of the C++ standard template library, the techniques used to traverse these various data structures are all basically the same. For a enumerable sequence of values, the `XXXbegin()` function (or method) returns an iterator to the start of the sequence, the `XXXend()` function returns an iterator pointing to one past the last valid element of the sequence, and there is some `XXXiterator` data type that is common between the two operations.

Because the pattern for iteration is common across many different aspects of the program representation, the standard template library algorithms may be used on them, and it is easier to remember how to iterate. First we show a few common examples of the data structures that need to be traversed. Other data structures are traversed in very similar ways.

Iterating over the `BasicBlock`s in a `Function`

It's quite common to have a `Function` instance that you'd like to transform in some way; in particular, you'd like to manipulate its `BasicBlock`s. To facilitate this, you'll need to iterate over all of the `BasicBlock`s that constitute the `Function`. The following is an example that prints the name of a `BasicBlock` and the number of `Instruction`s it contains:

```
// func is a pointer to a Function instance
for (Function::iterator i = func->begin(), e = func->end(); i != e; ++i) {

    // print out the name of the basic block if it has one, and then the
    // number of instructions that it contains

    cerr << "Basic block (name=" << i->getName() << ") has "
         << i->size() << " instructions.\n";
}
```

Note that i can be used as if it were a pointer for the purposes of invoking member functions of the `Instruction` class. This is because the indirection operator is overloaded for the iterator classes. In the above code, the expression `i->size()` is exactly equivalent to `(*i).size()` just like you'd expect.

Iterating over the `Instruction`s in a `BasicBlock`

Just like when dealing with `BasicBlock`s in `Function`s, it's easy to iterate over the individual instructions that make up `BasicBlock`s. Here's a code snippet that prints out each instruction in a `BasicBlock`:

```
// blk is a pointer to a BasicBlock instance
for (BasicBlock::iterator i = blk->begin(), e = blk->end(); i != e; ++i)
   // the next statement works since operator<<(ostream&,...)
   // is overloaded for Instruction&
   cerr << *i << "\n";
```

However, this isn't really the best way to print out the contents of a `BasicBlock`! Since the ostream operators are overloaded for virtually anything you'll care about, you could have just invoked the print routine on the basic block itself: `cerr << *blk << "\n";`.

Note that currently operator<< is implemented for `Value*`, so it will print out the contents of the pointer, instead of the pointer value you might expect. This is a deprecated interface that will be removed in the future, so it's best not to depend on it. To print out the pointer value for now, you must cast to `void*`.

Iterating over the `Instruction`s in a `Function`

If you're finding that you commonly iterate over a `Function`'s `BasicBlock`s and then that `BasicBlock`'s `Instruction`s, `InstIterator` should be used instead. You'll need to include `llvm/Support/InstIterator.h`, and then instantiate `InstIterator`s explicitly in your code. Here's a small example that shows how to dump all instructions in a function to stderr (**Note:** Dereferencing an `InstIterator` yields an `Instruction*`, *not* an `Instruction&`!):

```
#include "llvm/Support/InstIterator.h"
...
// Suppose F is a ptr to a function
for (inst_iterator i = inst_begin(F), e = inst_end(F); i != e; ++i)
  cerr << **i << "\n";
```

Easy, isn't it? You can also use `InstIterator`s to fill a worklist with its initial contents. For example, if you wanted to initialize a worklist to contain all instructions in a `Function` F, all you would need to do is something like:

```
std::set<Instruction*> worklist;
worklist.insert(inst_begin(F), inst_end(F));
```

The STL set `worklist` would now contain all instructions in the `Function` pointed to by F.

Turning an iterator into a class pointer (and vice-versa)

Sometimes, it'll be useful to grab a reference (or pointer) to a class instance when all you've got at hand is an iterator. Well, extracting a reference or a pointer from an iterator is very straightforward. Assuming that `i` is a `BasicBlock::iterator` and `j` is a `BasicBlock::const_iterator`:

```
Instruction& inst = *i;   // grab reference to instruction reference
Instruction* pinst = &*i; // grab pointer to instruction reference
const Instruction& inst = *j;
```

However, the iterators you'll be working with in the LLVM framework are special: they will automatically

convert to a ptr-to-instance type whenever they need to. Instead of dereferencing the iterator and then taking the address of the result, you can simply assign the iterator to the proper pointer type and you get the dereference and address-of operation as a result of the assignment (behind the scenes, this is a result of overloading casting mechanisms). Thus the last line of the last example,

```
Instruction* pinst = &*i;
```

is semantically equivalent to

```
Instruction* pinst = i;
```

It's also possible to turn a class pointer into the corresponding iterator. Usually, this conversion is quite inexpensive. The following code snippet illustrates use of the conversion constructors provided by LLVM iterators. By using these, you can explicitly grab the iterator of something without actually obtaining it via iteration over some structure:

```
void printNextInstruction(Instruction* inst) {
    BasicBlock::iterator it(inst);
    ++it; // after this line, it refers to the instruction after *inst.
    if (it != inst->getParent()->end()) cerr << *it << "\n";
}
```

Of course, this example is strictly pedagogical, because it'd be much better to explicitly grab the next instruction directly from inst.

Finding call sites: a slightly more complex example

Say that you're writing a FunctionPass and would like to count all the locations in the entire module (that is, across every `Function`) where a certain function (i.e., some `Function`*) is already in scope. As you'll learn later, you may want to use an `InstVisitor` to accomplish this in a much more straightforward manner, but this example will allow us to explore how you'd do it if you didn't have `InstVisitor` around. In pseudocode, this is what we want to do:

```
initialize callCounter to zero
for each Function f in the Module
    for each BasicBlock b in f
      for each Instruction i in b
        if (i is a CallInst and calls the given function)
          increment callCounter
```

And the actual code is (remember, since we're writing a `FunctionPass`, our `FunctionPass`-derived class simply has to override the `runOnFunction` method...):

```
Function* targetFunc = ...;

class OurFunctionPass : public FunctionPass {
  public:
    OurFunctionPass(): callCounter(0) { }

    virtual runOnFunction(Function& F) {
        for (Function::iterator b = F.begin(), be = F.end(); b != be; ++b) {
            for (BasicBlock::iterator i = b->begin(); ie = b->end(); i != ie; ++i) {
                if (CallInst* callInst = dyn_cast<CallInst>(&*i)) {
                    // we know we've encountered a call instruction, so we
                    // need to determine if it's a call to the
                    // function pointed to by m_func or not.

                    if (callInst->getCalledFunction() == targetFunc)
                        ++callCounter;
                }
            }
    }
```

```
  private:
    unsigned  callCounter;
};
```

Iterating over def-use & use-def chains

Frequently, we might have an instance of the Value Class and we want to determine which `Users` use the `Value`. The list of all `Users` of a particular `Value` is called a *def-use* chain. For example, let's say we have a `Function*` named `F` to a particular function `foo`. Finding all of the instructions that *use* `foo` is as simple as iterating over the *def-use* chain of `F`:

```
Function* F = ...;

for (Value::use_iterator i = F->use_begin(), e = F->use_end(); i != e; ++i) {
    if (Instruction *Inst = dyn_cast<Instruction>(*i)) {
        cerr << "F is used in instruction:\n";
        cerr << *Inst << "\n";
    }
}
```

Alternately, it's common to have an instance of the User Class and need to know what `Values` are used by it. The list of all `Values` used by a `User` is known as a *use-def* chain. Instances of class `Instruction` are common `Users`, so we might want to iterate over all of the values that a particular instruction uses (that is, the operands of the particular `Instruction`):

```
Instruction* pi = ...;

for (User::op_iterator i = pi->op_begin(), e = pi->op_end(); i != e; ++i) {
    Value* v = *i;
    ...
}
```

Making simple changes

There are some primitive transformation operations present in the LLVM infrastructure that are worth knowing about. When performing transformations, it's fairly common to manipulate the contents of basic blocks. This section describes some of the common methods for doing so and gives example code.

Creating and inserting new `Instructions`

Instantiating Instructions

Creation of `Instructions` is straightforward: simply call the constructor for the kind of instruction to instantiate and provide the necessary parameters. For example, an `AllocaInst` only *requires* a (const-ptr-to) `Type`. Thus:

```
AllocaInst* ai = new AllocaInst(Type::IntTy);
```

will create an `AllocaInst` instance that represents the allocation of one integer in the current stack frame, at runtime. Each `Instruction` subclass is likely to have varying default parameters which change the semantics of the instruction, so refer to the doxygen documentation for the subclass of Instruction that you're interested in instantiating.

Naming values

It is very useful to name the values of instructions when you're able to, as this facilitates the debugging of your transformations. If you end up looking at generated LLVM machine code, you definitely want to have logical names associated with the results of instructions! By supplying a value for the `Name` (default) parameter of the `Instruction` constructor, you associate a logical name with the result of the instruction's execution at runtime. For example, say that I'm writing a transformation that dynamically allocates space for an integer on the stack, and that integer is going to be used as some kind of index by some other code. To accomplish this, I place an `AllocaInst` at the first point in the first `BasicBlock` of some `Function`, and I'm intending to use it within the same `Function`. I might do:

```
AllocaInst* pa = new AllocaInst(Type::IntTy, 0, "indexLoc");
```

where `indexLoc` is now the logical name of the instruction's execution value, which is a pointer to an integer on the runtime stack.

Inserting instructions

There are essentially two ways to insert an `Instruction` into an existing sequence of instructions that form a `BasicBlock`:

- Insertion into an explicit instruction list

  Given a `BasicBlock* pb`, an `Instruction* pi` within that `BasicBlock`, and a newly-created instruction we wish to insert before `*pi`, we do the following:

  ```
  BasicBlock *pb = ...;
  Instruction *pi = ...;
  Instruction *newInst = new Instruction(...);
  pb->getInstList().insert(pi, newInst); // inserts newInst before pi in pb
  ```

- Insertion into an implicit instruction list

  `Instruction` instances that are already in `BasicBlocks` are implicitly associated with an existing instruction list: the instruction list of the enclosing basic block. Thus, we could have accomplished the same thing as the above code without being given a `BasicBlock` by doing:

  ```
  Instruction *pi = ...;
  Instruction *newInst = new Instruction(...);
  pi->getParent()->getInstList().insert(pi, newInst);
  ```

  In fact, this sequence of steps occurs so frequently that the `Instruction` class and `Instruction`-derived classes provide constructors which take (as a default parameter) a pointer to an `Instruction` which the newly-created `Instruction` should precede. That is, `Instruction` constructors are capable of inserting the newly-created instance into the `BasicBlock` of a provided instruction, immediately before that instruction. Using an `Instruction` constructor with a `insertBefore` (default) parameter, the above code becomes:

  ```
  Instruction* pi = ...;
  Instruction* newInst = new Instruction(..., pi);
  ```

  which is much cleaner, especially if you're creating a lot of instructions and adding them to `BasicBlocks`.

Deleting `Instructions`

Deleting an instruction from an existing sequence of instructions that form a `BasicBlock` is very straightforward. First, you must have a pointer to the instruction that you wish to delete. Second, you need

to obtain the pointer to that instruction's basic block. You use the pointer to the basic block to get its list of instructions and then use the erase function to remove your instruction.

For example:

```
Instruction *I = .. ;
BasicBlock *BB = I->getParent();
BB->getInstList().erase(I);
```

Replacing an `Instruction` with another `Value`

Replacing individual instructions

Including "llvm/Transforms/Utils/BasicBlockUtils.h" permits use of two very useful replace functions: `ReplaceInstWithValue` and `ReplaceInstWithInst`.

- `ReplaceInstWithValue`

  This function replaces all uses (within a basic block) of a given instruction with a value, and then removes the original instruction. The following example illustrates the replacement of the result of a particular `AllocaInst` that allocates memory for a single integer with an null pointer to an integer.

  ```
  AllocaInst* instToReplace = ...;
  BasicBlock::iterator ii(instToReplace);
  ReplaceInstWithValue(instToReplace->getParent()->getInstList(), ii,
                       Constant::getNullValue(PointerType::get(Type::IntTy)));
  ```

- `ReplaceInstWithInst`

  This function replaces a particular instruction with another instruction. The following example illustrates the replacement of one `AllocaInst` with another.

  ```
  AllocaInst* instToReplace = ...;
  BasicBlock::iterator ii(instToReplace);
  ReplaceInstWithInst(instToReplace->getParent()->getInstList(), ii,
                      new AllocaInst(Type::IntTy, 0, "ptrToReplacedInt"));
  ```

Replacing multiple uses of `Users` and `Values`

You can use `Value::replaceAllUsesWith` and `User::replaceUsesOfWith` to change more than one use at a time. See the doxygen documentation for the Value Class and User Class, respectively, for more information.

The Core LLVM Class Hierarchy Reference

The Core LLVM classes are the primary means of representing the program being inspected or transformed. The core LLVM classes are defined in header files in the `include/llvm/` directory, and implemented in the `lib/VMCore` directory.

The `Value` class

`#include "llvm/Value.h"`
doxygen info: Value Class

The `Value` class is the most important class in LLVM Source base. It represents a typed value that may be used (among other things) as an operand to an instruction. There are many different types of `Values`, such as

Constants, Arguments, and even Instructions and Functions are Values.

A particular Value may be used many times in the LLVM representation for a program. For example, an incoming argument to a function (represented with an instance of the Argument class) is "used" by every instruction in the function that references the argument. To keep track of this relationship, the Value class keeps a list of all of the Users that is using it (the User class is a base class for all nodes in the LLVM graph that can refer to Values). This use list is how LLVM represents def-use information in the program, and is accessible through the use_* methods, shown below.

Because LLVM is a typed representation, every LLVM Value is typed, and this Type is available through the getType() method. In addition, all LLVM values can be named. The "name" of the Value is symbolic string printed in the LLVM code:

```
%foo = add int 1, 2
```

The name of this instruction is "foo". **NOTE** that the name of any value may be missing (an empty string), so names should **ONLY** be used for debugging (making the source code easier to read, debugging printouts), they should not be used to keep track of values or map between them. For this purpose, use a std::map of pointers to the Value itself instead.

One important aspect of LLVM is that there is no distinction between an SSA variable and the operation that produces it. Because of this, any reference to the value produced by an instruction (or the value available as an incoming argument, for example) is represented as a direct pointer to the class that represents this value. Although this may take some getting used to, it simplifies the representation and makes it easier to manipulate.

Important Public Members of the Value class

- Value::use_iterator - Typedef for iterator over the use-list
  Value::use_const_iterator - Typedef for const_iterator over the use-list
  unsigned use_size() - Returns the number of users of the value.
  bool use_empty() - Returns true if there are no users.
  use_iterator use_begin() - Get an iterator to the start of the use-list.
  use_iterator use_end() - Get an iterator to the end of the use-list.
  User *use_back() - Returns the last element in the list.

  These methods are the interface to access the def-use information in LLVM. As with all other iterators in LLVM, the naming conventions follow the conventions defined by the STL.

- Type *getType() const

  This method returns the Type of the Value.

- bool hasName() const
  std::string getName() const
  void setName(const std::string &Name)

  This family of methods is used to access and assign a name to a Value, be aware of the precaution above.

- void replaceAllUsesWith(Value *V)

  This method traverses the use list of a Value changing all Users of the current value to refer to "V" instead. For example, if you detect that an instruction always produces a constant value (for example through constant folding), you can replace all uses of the instruction with the constant like this:

  ```
  Inst->replaceAllUsesWith(ConstVal);
  ```

The `User` class

`#include "llvm/User.h"`
doxygen info: User Class
Superclass: `Value`

The `User` class is the common base class of all LLVM nodes that may refer to `Value`s. It exposes a list of "Operands" that are all of the `Value`s that the User is referring to. The `User` class itself is a subclass of `Value`.

The operands of a `User` point directly to the LLVM `Value` that it refers to. Because LLVM uses Static Single Assignment (SSA) form, there can only be one definition referred to, allowing this direct connection. This connection provides the use-def information in LLVM.

Important Public Members of the `User` class

The `User` class exposes the operand list in two ways: through an index access interface and through an iterator based interface.

- `Value *getOperand(unsigned i)`
  `unsigned getNumOperands()`

  These two methods expose the operands of the `User` in a convenient form for direct access.

- `User::op_iterator` - Typedef for iterator over the operand list
  `User::op_const_iterator use_iterator op_begin()` - Get an iterator to the start of the operand list.
  `use_iterator op_end()` - Get an iterator to the end of the operand list.

  Together, these methods make up the iterator based interface to the operands of a `User`.

The `Instruction` class

`#include "llvm/Instruction.h"`
doxygen info: Instruction Class
Superclasses: `User`, `Value`

The `Instruction` class is the common base class for all LLVM instructions. It provides only a few methods, but is a very commonly used class. The primary data tracked by the `Instruction` class itself is the opcode (instruction type) and the parent `BasicBlock` the `Instruction` is embedded into. To represent a specific type of instruction, one of many subclasses of `Instruction` are used.

Because the `Instruction` class subclasses the `User` class, its operands can be accessed in the same way as for other `User`s (with the `getOperand()`/`getNumOperands()` and `op_begin()`/`op_end()` methods).

An important file for the `Instruction` class is the `llvm/Instruction.def` file. This file contains some meta-data about the various different types of instructions in LLVM. It describes the enum values that are used as opcodes (for example `Instruction::Add` and `Instruction::SetLE`), as well as the concrete sub-classes of `Instruction` that implement the instruction (for example `BinaryOperator` and `SetCondInst`). Unfortunately, the use of macros in this file confused doxygen, so these enum values don't show up correctly in the doxygen output.

Important Public Members of the `Instruction` class

- `BasicBlock *getParent()`

Returns the BasicBlock that this Instruction is embedded into.

- `bool mayWriteToMemory()`

  Returns true if the instruction writes to memory, i.e. it is a `call`, `free`, `invoke`, or `store`.

- `unsigned getOpcode()`

  Returns the opcode for the `Instruction`.

- `Instruction *clone() const`

  Returns another instance of the specified instruction, identical in all ways to the original except that the instruction has no parent (ie it's not embedded into a `BasicBlock`), and it has no name.

The `BasicBlock` class

`#include "llvm/BasicBlock.h"`
doxygen info: BasicBlock Class
Superclass: `Value`

This class represents a single entry multiple exit section of the code, commonly known as a basic block by the compiler community. The `BasicBlock` class maintains a list of `Instruction`s, which form the body of the block. Matching the language definition, the last element of this list of instructions is always a terminator instruction (a subclass of the `TerminatorInst` class).

In addition to tracking the list of instructions that make up the block, the `BasicBlock` class also keeps track of the `Function` that it is embedded into.

Note that `BasicBlock`s themselves are `Value`s, because they are referenced by instructions like branches and can go in the switch tables. `BasicBlock`s have type `label`.

Important Public Members of the `BasicBlock` class

- `BasicBlock(const std::string &Name = "", Function *Parent = 0)`

  The `BasicBlock` constructor is used to create new basic blocks for insertion into a function. The constructor simply takes a name for the new block, and optionally a `Function` to insert it into. If the `Parent` parameter is specified, the new `BasicBlock` is automatically inserted at the end of the specified `Function`, if not specified, the BasicBlock must be manually inserted into the `Function`.

- `BasicBlock::iterator` - Typedef for instruction list iterator
  `BasicBlock::const_iterator` - Typedef for const_iterator.
  `begin()`, `end()`, `front()`, `back()`, `size()`, `empty()`, `rbegin()`, `rend()`

  These methods and typedefs are forwarding functions that have the same semantics as the standard library methods of the same names. These methods expose the underlying instruction list of a basic block in a way that is easy to manipulate. To get the full complement of container operations (including operations to update the list), you must use the `getInstList()` method.

- `BasicBlock::InstListType &getInstList()`

  This method is used to get access to the underlying container that actually holds the Instructions. This method must be used when there isn't a forwarding function in the `BasicBlock` class for the operation that you would like to perform. Because there are no forwarding functions for "updating" operations, you need to use this if you want to update the contents of a `BasicBlock`.

- `Function *getParent()`

  Returns a pointer to `Function` the block is embedded into, or a null pointer if it is homeless.

- `TerminatorInst *getTerminator()`

  Returns a pointer to the terminator instruction that appears at the end of the `BasicBlock`. If there is no terminator instruction, or if the last instruction in the block is not a terminator, then a null pointer is returned.

The `GlobalValue` class

`#include "llvm/GlobalValue.h"`
doxygen info: GlobalValue Class
Superclasses: `User`, `Value`

Global values (`GlobalVariable`s or `Function`s) are the only LLVM values that are visible in the bodies of all `Function`s. Because they are visible at global scope, they are also subject to linking with other globals defined in different translation units. To control the linking process, `GlobalValues` know their linkage rules. Specifically, `GlobalValues` know whether they have internal or external linkage.

If a `GlobalValue` has internal linkage (equivalent to being `static` in C), it is not visible to code outside the current translation unit, and does not participate in linking. If it has external linkage, it is visible to external code, and does participate in linking. In addition to linkage information, `GlobalValues` keep track of which `Module` they are currently part of.

Because `GlobalValues` are memory objects, they are always referred to by their address. As such, the `Type` of a global is always a pointer to its contents. This is explained in the LLVM Language Reference Manual.

Important Public Members of the `GlobalValue` class

- ```
  bool hasInternalLinkage() const
  bool hasExternalLinkage() const
  void setInternalLinkage(bool HasInternalLinkage)
  ```

  These methods manipulate the linkage characteristics of the `GlobalValue`.

- `Module *getParent()`

  This returns the `Module` that the GlobalValue is currently embedded into.

The `Function` class

`#include "llvm/Function.h"`
doxygen info: Function Class
Superclasses: `GlobalValue`, `User`, `Value`

The `Function` class represents a single procedure in LLVM. It is actually one of the more complex classes in the LLVM heirarchy because it must keep track of a large amount of data. The `Function` class keeps track of a list of `BasicBlock`s, a list of formal `Argument`s, and a `SymbolTable`.

The list of `BasicBlock`s is the most commonly used part of `Function` objects. The list imposes an implicit ordering of the blocks in the function, which indicate how the code will be layed out by the backend. Additionally, the first `BasicBlock` is the implicit entry node for the `Function`. It is not legal in LLVM explicitly branch to this initial block. There are no implicit exit nodes, and in fact there may be multiple exit
  ```

nodes from a single `Function`. If the `BasicBlock` list is empty, this indicates that the `Function` is actually a function declaration: the actual body of the function hasn't been linked in yet.

In addition to a list of `BasicBlock`s, the `Function` class also keeps track of the list of formal `Argument`s that the function receives. This container manages the lifetime of the `Argument` nodes, just like the `BasicBlock` list does for the `BasicBlock`s.

The `SymbolTable` is a very rarely used LLVM feature that is only used when you have to look up a value by name. Aside from that, the `SymbolTable` is used internally to make sure that there are not conflicts between the names of `Instruction`s, `BasicBlock`s, or `Argument`s in the function body.

Important Public Members of the `Function` class

- `Function(const FunctionType *Ty, bool isInternal, const std::string &N = "")`

  Constructor used when you need to create new `Function`s to add the the program. The constructor must specify the type of the function to create and whether or not it should start out with internal or external linkage.

- `bool isExternal()`

  Return whether or not the `Function` has a body defined. If the function is "external", it does not have a body, and thus must be resolved by linking with a function defined in a different translation unit.

- `Function::iterator` - Typedef for basic block list iterator
  `Function::const_iterator` - Typedef for const_iterator.
  `begin()`, `end()`, `front()`, `back()`, `size()`, `empty()`, `rbegin()`, `rend()`

  These are forwarding methods that make it easy to access the contents of a `Function` object's `BasicBlock` list.

- `Function::BasicBlockListType &getBasicBlockList()`

  Returns the list of `BasicBlock`s. This is necessary to use when you need to update the list or perform a complex action that doesn't have a forwarding method.

- `Function::aiterator` - Typedef for the argument list iterator
  `Function::const_aiterator` - Typedef for const_iterator.
  `abegin()`, `aend()`, `afront()`, `aback()`, `asize()`, `aempty()`, `arbegin()`, `arend()`

  These are forwarding methods that make it easy to access the contents of a `Function` object's `Argument` list.

- `Function::ArgumentListType &getArgumentList()`

  Returns the list of `Argument`s. This is necessary to use when you need to update the list or perform a complex action that doesn't have a forwarding method.

- `BasicBlock &getEntryBlock()`

  Returns the entry `BasicBlock` for the function. Because the entry block for the function is always the first block, this returns the first block of the `Function`.

- `Type *getReturnType()`
  `FunctionType *getFunctionType()`

This traverses the `Type` of the `Function` and returns the return type of the function, or the `FunctionType` of the actual function.

- `SymbolTable *getSymbolTable()`

  Return a pointer to the `SymbolTable` for this `Function`.

The `GlobalVariable` class

`#include "llvm/GlobalVariable.h"`
doxygen info: GlobalVariable Class
Superclasses: `GlobalValue`, `User`, `Value`

Global variables are represented with the (suprise suprise) `GlobalVariable` class. Like functions, `GlobalVariables` are also subclasses of `GlobalValue`, and as such are always referenced by their address (global values must live in memory, so their "name" refers to their address). Global variables may have an initial value (which must be a `Constant`), and if they have an initializer, they may be marked as "constant" themselves (indicating that their contents never change at runtime).

Important Public Members of the `GlobalVariable` class

- `GlobalVariable(const Type *Ty, bool isConstant, bool isInternal, Constant *Initializer = 0, const std::string &Name = "")`

  Create a new global variable of the specified type. If `isConstant` is true then the global variable will be marked as unchanging for the program, and if `isInternal` is true the resultant global variable will have internal linkage. Optionally an initializer and name may be specified for the global variable as well.

- `bool isConstant() const`

  Returns true if this is a global variable is known not to be modified at runtime.

- `bool hasInitializer()`

  Returns true if this `GlobalVariable` has an intializer.

- `Constant *getInitializer()`

  Returns the intial value for a `GlobalVariable`. It is not legal to call this method if there is no initializer.

The `Module` class

`#include "llvm/Module.h"`
doxygen info: Module Class

The `Module` class represents the top level structure present in LLVM programs. An LLVM module is effectively either a translation unit of the original program or a combination of several translation units merged by the linker. The `Module` class keeps track of a list of `Function`s, a list of `GlobalVariable`s, and a `SymbolTable`. Additionally, it contains a few helpful member functions that try to make common operations easy.

Important Public Members of the `Module` class

- `Module::iterator` - Typedef for function list iterator
  `Module::const_iterator` - Typedef for const_iterator.

`begin()`, `end()`, `front()`, `back()`, `size()`, `empty()`, `rbegin()`, `rend()`

These are forwarding methods that make it easy to access the contents of a `Module` object's `Function` list.

- `Module::FunctionListType &getFunctionList()`

  Returns the list of `Function`s. This is necessary to use when you need to update the list or perform a complex action that doesn't have a forwarding method.

- `Module::giterator` - Typedef for global variable list iterator
  `Module::const_giterator` - Typedef for const_iterator.
  `gbegin()`, `gend()`, `gfront()`, `gback()`, `gsize()`, `gempty()`, `grbegin()`, `grend()`

  These are forwarding methods that make it easy to access the contents of a `Module` object's `GlobalVariable` list.

- `Module::GlobalListType &getGlobalList()`

  Returns the list of `GlobalVariable`s. This is necessary to use when you need to update the list or perform a complex action that doesn't have a forwarding method.

- `SymbolTable *getSymbolTable()`

  Return a reference to the `SymbolTable` for this `Module`.

- `Function *getFunction(const std::string &Name, const FunctionType *Ty)`

  Look up the specified function in the `Module` `SymbolTable`. If it does not exist, return `null`.

- `Function *getOrInsertFunction(const std::string &Name, const FunctionType *T)`

  Look up the specified function in the `Module` `SymbolTable`. If it does not exist, add an external declaration for the function and return it.

- `std::string getTypeName(const Type *Ty)`

  If there is at least one entry in the `SymbolTable` for the specified `Type`, return it. Otherwise return the empty string.

- `bool addTypeName(const std::string &Name, const Type *Ty)`

  Insert an entry in the `SymbolTable` mapping `Name` to `Ty`. If there is already an entry for this name, true is returned and the `SymbolTable` is not modified.

The `Constant` class and subclasses

Constant represents a base class for different types of constants. It is subclassed by ConstantBool, ConstantInt, ConstantSInt, ConstantUInt, ConstantArray etc for representing the various types of Constants.

Important Public Methods

- `bool isConstantExpr()`: Returns true if it is a ConstantExpr

Important Subclasses of Constant

- ConstantSInt : This subclass of Constant represents a signed integer constant.
  - `int64_t getValue() const`: Returns the underlying value of this constant.
- ConstantUInt : This class represents an unsigned integer.
  - `uint64_t getValue() const`: Returns the underlying value of this constant.
- ConstantFP : This class represents a floating point constant.
  - `double getValue() const`: Returns the underlying value of this constant.
- ConstantBool : This represents a boolean constant.
  - `bool getValue() const`: Returns the underlying value of this constant.
- ConstantArray : This represents a constant array.
  - `const std::vector<Use> &getValues() const`: Returns a Vecotr of component constants that makeup this array.
- ConstantStruct : This represents a constant struct.
  - `const std::vector<Use> &getValues() const`: Returns a Vecotr of component constants that makeup this array.
- ConstantPointerRef : This represents a constant pointer value that is initialized to point to a global value, which lies at a constant fixed address.
  - `GlobalValue *getValue()`: Returns the global value to which this pointer is pointing to.

The `Type` class and Derived Types

Type as noted earlier is also a subclass of a Value class. Any primitive type (like int, short etc) in LLVM is an instance of Type Class. All other types are instances of subclasses of type like FunctionType, ArrayType etc. DerivedType is the interface for all such dervied types including FunctionType, ArrayType, PointerType, StructType. Types can have names. They can be recursive (StructType). There exists exactly one instance of any type structure at a time. This allows using pointer equality of Type *s for comparing types.

Important Public Methods

- `PrimitiveID getPrimitiveID() const`: Returns the base type of the type.
- `bool isSigned() const`: Returns whether an integral numeric type is signed. This is true for SByteTy, ShortTy, IntTy, LongTy. Note that this is not true for Float and Double.
- `bool isUnsigned() const`: Returns whether a numeric type is unsigned. This is not quite the complement of isSigned... nonnumeric types return false as they do with isSigned. This returns true for UByteTy, UShortTy, UIntTy, and ULongTy.
- `bool isInteger() const`: Equilivent to isSigned() || isUnsigned(), but with only a single virtual function invocation.
- `bool isIntegral() const`: Returns true if this is an integral type, which is either Bool type or one of the Integer types.
- `bool isFloatingPoint()`: Return true if this is one of the two floating point types.
- `bool isRecursive() const`: Returns rue if the type graph contains a cycle.
- `isLosslesslyConvertableTo (const Type *Ty) const`: Return true if this type can be converted to 'Ty' without any reinterpretation of bits. For example, uint to int.
- `bool isPrimitiveType() const`: Returns true if it is a primitive type.
- `bool isDerivedType() const`: Returns true if it is a derived type.
- `const Type * getContainedType (unsigned i) const`: This method is used to implement the type iterator. For derived types, this returns the types 'contained' in the derived type, returning 0 when 'i' becomes invalid. This allows the user to iterate over the types in a struct, for example, really easily.
- `unsigned getNumContainedTypes() const`: Return the number of types in the derived type.

Derived Types

- SequentialType : This is subclassed by ArrayType and PointerType

        - `const Type * getElementType() const`: Returns the type of each of the elements in the sequential type.
    - ArrayType : This is a subclass of SequentialType and defines interface for array types.
        - `unsigned getNumElements() const`: Returns the number of elements in the array.
    - PointerType : Subclass of SequentialType for pointer types.
    - StructType : subclass of DerivedTypes for struct types
    - FunctionType : subclass of DerivedTypes for function types.
        - `bool isVarArg() const`: Returns true if its a vararg function
        - `const Type * getReturnType() const`: Returns the return type of the function.
        - `const ParamTypes &getParamTypes() const`: Returns a vector of parameter types.
        - `const Type * getParamType (unsigned i)`: Returns the type of the ith parameter.
        - `const unsigned getNumParams() const`: Returns the number of formal parameters.

The `Argument` class

This subclass of Value defines the interface for incoming formal arguments to a function. A Function maitanis a list of its formal arguments. An argument has a pointer to the parent Function.

By: Dinakar Dhurjati and Chris Lattner
The LLVM Compiler Infrastructure
Last modified: Sat Sep 20 09:25:11 CDT 2003