SmartDec: Developer's Guide

Yegor Derevenets Alexander Fokin

May 11, 2016

Abstract

This document is meant to be a guide through the source code of the decompiler. It gives you an intuition, but not all the details. For the latter, read the sources. Almost every class and every function is documented in doxygen. Complicated functions usually have explanations in the implementation.

Contents

1 Introduction

SmartDec is a project to develop an instrument for analysing programs in a low-level representation. The primary goal of such analysis is producing high-level code having a semantics close to the input program, i.e. decompiling it.

As an input, the decompiler accepts executable images (PE, ELF) and assembly listings (output of dumpbin disassemblers). As an output, it produces a code in a C-like language which is designed to be textually compatible with C.

1.1 Project Structure

SmartDec contains of several components:

nc — a library implementing various kinds of analyses.

nc-gui — a library implementing a set of GUI widgets for displaying analysis results.

nocode — a command-line front-end to the nc library.

smartdec — a GUI front-end to the 'nc' library that uses 'nc-gui' library.

nocode-plugin — a plug-in for the IDA Pro disassembler that uses 'nc' library for performing decompilation and 'nc-gui' library for showing results.

1.1.1 Directory Structure

Project root contains the following directories:

doc — documentation.

doc/developer — documentation for developers (you are reading it).

doc/user — user documentation.

examples — example input files for the decompiler.

modules — additional CMake scripts.

src/3rd-party — third-party libraries.

src/3rd-party/libudis86 — disassembling library for Intel x86.

src/nc — the 'nc' library.

src/nc/common — various convenience and metaprogramming code.

src/nc/core — representation of assembler programs.

src/nc/core/disasm — disassembly support.

src/nc/core/image — representation for executable image.

src/nc/core/input — base interface for input parsers.

src/nc/core/irgen — generation of intermediate representation from assembly.

src/nc/core/mangling — mangling support.

src/nc/crec — reconstruction of C++ classes and exceptions.

src/nc/gui — the 'nc-gui' library.

src/nc/input — parsers for various decompiler's input formats.

src/nc/intel — support for Intel x86 architecture.

src/nc/ir — intermediate representation.

src/nc/ir/calls — calling conventions support.

src/nc/ir/cflow — structural analysis.

src/nc/ir/cgen — C-like code generation.

src/nc/ir/dflow — dataflow analysis.

src/nc/ir/inlining — inlining of functions.

src/nc/ir/misc — miscellaneous algorithms.

src/nc/ir/types — type reconstruction.

src/nc/ir/usage — computing whether certain terms will actually generate C-like code.

src/nc/ir/vars — reconstruction of variables.

src/nc/likec — C-like language used as high-level representation.

src/nc/refs — computing of code cross-references.

src/nc/vsa — value set analysis.

src/ida-plugin — the IDA Pro plug-in.

src/ida-plugin/patches — patches for IDA SDK required to build the plug-in.

src/nocode — the command-line decompiler.

src/smartdec — the GUI decompiler.

winbuild — manually maintained Visual Studio 2010 project files using a clone of nmake called jom for building. Don't use them unless you know what you are doing.

1.2 Use of Programming Language and Libraries

SmartDec is written in C++11 and uses Boost and Qt libraries. SmartDec uses CMake as its build system.

All header files in the project must have `#include <nc/config.h>` as its first include.

Ownership transfers should be signified by passing pointers to objects using `std::unique_ptr`.

When describing function's parameters or return values having (plain or smart) pointer types, phrases "Valid pointer to XXX" and "Pointer to XXX. Can be NULL" must be used.

1.2.1 C++11 Features

In order to make the source code more brief, concise, and robust, we use the following features of C++11:

- Automatic type inference (`auto`);
- Lambda functions;
- Rvalue references;
- `std::unique_ptr` (supersedes `std::auto_ptr`);
- Static assertions;
- Explicit overrides (`<nc/config.h>` #defines `override` to an empty string on compilers not supporting this feature);
- nullptr (`<nc/config.h>` automatically #defines `NULL` to `nullptr` on decent compilers, so always use `NULL` for a null pointer).

Wikipedia contains a good survey of these and other C++11 enhancements:

- `http://en.wikipedia.org/wiki/C++11`,
- `http://ru.wikipedia.org/wiki/C++11`.

1.2.2 C++ Standard Library

Almost everything is used, except I/O. In addition, where possible, `std::string` is superseded by `QString`.

1.2.3 Exceptions

The decompiler project uses exceptions for error handling. The `nc::Exception` class (see `<nc/common/Exception.h>`) derives from `std::exception` and `boost::exception` and provides Unicode error messages. All classes of exceptions inside the project must be derived from `nc::Exception`.

1.2.4 Boost

The following Boost header-only libraries are used:

- array
- config
- dynamic_bitset
- exception
- foreach
- function
- functional
- icl
- integer
- iterator
- lambda
- lexical_cast
- mpl
- numeric
- operators
- optional
- preprocessor
- range
- type_traits
- unordered
- utility

Documentation on these libraries is available online at `http://www.boost.org/doc/libs/1_50_0/libs/libraries.htm`.

1.2.5 Qt

The following Qt libraries are used:

- QtCore (QString, I/O, containers);
- QtGui (widgets).

Documentation on Qt is available online at `http://doc.qt.digia.com/qt/`.

1.2.6 Language Extensions

The following language extension is used:

- `foreach` — a statement for iterating over a range (array, container). Effectively, it's an alias for `BOOST_FOREACH` defined in `<nc/common/Foreach.h>`. Refer to Boost.Foreach documentation for usage details.

- `std::make_unique` — a function for creating unique pointers which is not yet included in the standard, but is very useful for writing exception-safe code. It is defined in `<nc/common/make_unique.h>`.

1.2.7 Metaprogramming techniques

SmartDec uses some metaprogramming techniques to make the implementation of core interfaces cleaner and more concise. These techniques are used to implement:

- Compile-time registration of different kinds of statements, operands, terms (and not only them) for fast dynamic casts. See `<nc/common/Kinds.h>` for details.
- Domain-specific language for human-readable definitions of instructions' semantics. Implemented in `<nc/core/irgen/InstructionAnalyzerExpressions.h>`, example usage is available in `<nc/intel/irgen/IntelInstructionAnalyzer.cpp>`.

Before digging into implementation details, please make sure you know and understand how the following C++ features and implementation techniques work:

- Partial and full template specialization.
- Argument-dependent lookup. A good explanation is given at Wikipedia: `http://en.wikipedia.org/wiki/Argument-dependent_lookup`.
- Curiously recurring template pattern. Again, Wikipedia can help with it: `http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern`.
- Expression templates. An explanation at Wikipedia (`http://en.wikipedia.org/wiki/Expression_templates`) contains a lot of code and almost no comments, so it is not for the faint of heart. Another good explanation is given at `http://www.angelikalanger.com/Articles/Cuj/ExpressionTemplates/ExpressionTemplates.htm`.

There are some typical questions that programmers ask when they stumble upon code that makes use of metaprogramming techniques, and some of them are better not to be left unanswered. A short FAQ follows.

Q : Why are you using expression templates? Isn't there a simpler way?

A : There is. Construction of expression trees can be written by hand using a couple of constructor calls. However, the resulting code is verbose, difficult to parse and hard to maintain. Compare

```
zf() = operand(0) == operand(1)
```

and

```
new ir::Assignment(
    createTerm(operands->zf()),
    new ir::BinaryOperator(
        ir::BinaryOperator::EQUAL,
        createTerm(instr->operand(0)),
        createTerm(instr->operand(1))))
```

The difference becomes even more apparent with more complex expressions.

Q : Why reinvent the wheel? Why don't use `boost::proto` for constructing your domain-specific language?

A : There are not that many people in this world who can use `boost::proto`, and those who understand how it works can all fit into an office of a typical startup company, and there will still be space left. That is, `boost::proto` code is a nightmare to maintain for those who didn't spend a year or two writing metaprograms in C++. And for those who did, it is still a nightmare, they're just used to it.

1.3 Building

For build instructions, refer to the document `doc/build.txt` under the project root.

1.4 Testing

The project uses CTest framework for regression testing. CTest is a testing tool distributed as a part of CMake. For directions on how to run tests, refer to the build instructions (section 1.3). For more thorough documentation on using the CMake+CTest bundle, see `http://www.vtk.org/Wiki/CMake_Testing_With_CTest`.

Developers are encouraged to run existing tests before pushing changes to the central repository. They are also welcome to define new tests: see `src/nocode/tests/CMakeLists.txt` for an example of how to do it.

1.5 Using

User documentation is located in the directory `doc/user` under the project root.

1.6 Todo

See the tickets in Redmine: `http://smartdec.ru/redmine/`.

2 Nc Library

The library is capable of doing the following kinds of analyses, in the order of execution:

1. Parsing of input files. If input file is an executable image, a convenient representation of it is built. If it is an assembly listing, the input is translated into a sequence of instructions. Program architecture is detected at this stage too.
2. Disassembly of the executable image into a sequence of instructions (if the input file was an executable image).
3. Translation of the sequence of instructions into intermediate representation (IR) making instructions' semantics explicit. The IR has a form of a control flow graph (CFG) with simple statements and expressions inside its basic blocks.
4. Isolation of functions. Control flow graphs of functions are created.
5. Creation of call graph and identification of calling conventions.
6. Joint reaching definitions and constant propagation/folding analysis.
7. Liveness analysis.
8. Type reconstruction.
9. Reconstruction of local variables.
10. Structural analysis (reconstruction of high-level control flow statements).
11. Recovery of exception handling information, virtual functions, and class hierarchies.
12. Code generation.

Fig. 1 presents a scheme of decompiler's workflow in a form of a Petri net. Algorithms are drawn as boxes. Input and output of the algorithms are drawn as ellipses. Labels of the nodes name the classes actually implementing the algorithms or data structures the algorithms work on.

Figure 1: Decompiler's Workflow Graph

All library code is located under the `nc` namespace.

2.1 Core: Representation of Input Program

The module called 'core' resides in `nc::core` namespace and is responsible for storing input program and decompilation state in a convenient form.

2.1.1 Instructions

`InstructionSet` class contains the set of instructions. Instructions are represented as instances of the `Instruction` class. Each instruction has an address, size, mnemonic and a list of operands. Address and size are integers. Mnemonic of an instruction is stored as a pointer to an instance of the `Mnemonic` class. Operands of an instruction are implemented as a vector of pointers to the `Operand` class.

The `Mnemonic` class contains integer id of the mnemonic, uppercase and lowercase names, and description of the mnemonic.

`Operand` is a base class for all classes implementing various kinds of instruction's operands. Every operand has a size measured in bits and integer id of its kind.

Operands generally constitute an expression tree. There are following basic kinds of the tree nodes:

- register (`RegisterOperand` class),

- addition of two operands (`AdditionOperand` class),
- multiplication of two operands (`MultiplicationOperand` class),
- dereference of an operand (`DereferenceOperand` class),
- bit range of an operand (`BitRangeOperand` class),
- integer constant (`ConstantOperand` class).

Defining custom operand types is possible. See `<nc/intel/IntelOperands.h>` for an example of this.

Mnemonic and Operand classes are immutable, so they can be shared among instructions. Instructions are also immutable and therefore can be shared among InstructionSet instances. Actually, InstructionSet owns instructions via shared pointers, which makes creating copies of the set cheap and easy.

2.1.2 Architecture

`Architecture` class contains general information about the architecture:

- bitness — bit size of a pointer on this architecture,
- pointer to a storage of mnemonics (`Mnemonics` class instance),
- pointer to a storage of registers (`Registers` class instance).

Also, `Architecture` works as a cache for operands. Due to the fact that operands are immutable, they can be shared among instructions. The methods `registerOperand` and `constantOperand` provide instances of operands of respective types. Returned instances are taken from cache or created as necessary.

2.1.3 Image

Executable image is handled in `nc::core::image` namespace. `Image` class contains a list of sections. `Section` class provides with a section name, its address and size, type, and access permissions. Both classes provide methods for reading data from the image by implementing the `Reader` interface. As the data source, they use `ByteSource` instances. Use `setExternalByteSource` method of `Image` and `Section` to assign a content to the image or section.

2.1.4 Mangling

Mangling is handled in `nc::core::Mangling` namespace. `Demangler` class is a base class for all demanglers and cannot demangle anything by itself. The only currently existing implementation of demangler is `MemorizingDemangler` class which supports remembering demangled versions of certain strings.

2.1.5 Module

`Module` class incorporates the architecture, image, demangler.

2.1.6 Context

`Context` class incorporates the module, list of instructions, and decompilation results. The class is immutable in the sense that every property can be initialized only once. It has a copy constructor which copies everything but the decompilation results from the previous instance.

2.1.7 Input

In order to fill in the `Context` with the architecture, image, and list of instructions, some input files must be parsed. This is where the classes from the `nc::core::input` namespace come into play.

Parser is a base class for any parser. This class has `canParse` and `parse` methods. The former checks whether given input stream can be parsed by the parser. The latter performs parsing and fills passed Context object with information. Exception of the `ParseError` class is thrown when a file cannot be parsed.

For keeping record of available parsers, the `ParserRepository` class exists. It's a singleton class maintaining a list of all available parsers. New parsers must have a unique name and be registered using the `registerParser` method.

Typical way of parsing a file is to get a list of all parsers, find (using the `canParse` method) a parser that reports to be able to parse given file, and let him do the job.

Note that in principle you can use separate parsers for an assembly listing and for executable image, so there is some sense in having more than one input file for a single decompilation task. In theory, nothing even stops you from having a parser for an architecture description.

2.1.8 Disassembly

Disassembly is handled in `nc::core::disasm` namespace. There is a main class `Disassembler` capable of disassembling a sequence of instructions. By default, it uses an instance of `InstructionDisassembler` class for performing disassembly of a single instruction. `Architecture` usually has a pointer to the right instance.

2.1.9 Decompilation: UniversalAnalyzer

`Architecture` contains a pointer to an instance of `UniversalAnalyzer` class. This class provides methods for performing all kinds of analyses as well as the decompilation as a whole. If a particular implementation of an analysis does not work for you architecture, make a subclass and make your architecture use it by calling `Architecture::setUniversalAnalyzer`. The analyzer stores all intermediate and final results of analyses in the `Context`.

2.1.10 Implementing new architecture

Extending the decompiler to support new architecture is pretty straighforward once you know how to do it. Here is a short guideline:

- Create instruction table for the new architecture. For example of an instruction table, see `<nc/intel/IntelInstructionTable.i>`. This table must contain instruction upper- and lowercase names and textual descriptions.
- Create register table for the new architecture. For example of a register table, see `<nc/intel/IntelRegisterTable.i>`. This table must contain description of the architecture's registers — their names, sizes, and locations.
- Create static mnemonic and register containers for the new architecture. Basic blocks for them are provided in `<nc/core/Mnemonics.i>`, `<nc/core/MnemonicsConstructor.i>`, `<nc/core/Registers.i>` and `<nc/core/RegistersConstructor.i>`. For example usage, see `<nc/intel/Mnemonics.h>` and `<nc/intel/Registers.h>`.
- Once all the convenience classes dealing with instructions and registers are in place, implement the `nc::core::InstructionAnalyzer` interface for your architecture. This is the class that will convert architecture-specific instructions into IR. For example implementation, see `<nc/intel/IntelInstructionAnalyzer.h>`.

- Implement the `nc::core::Architecture` interface for your architecture. In its constructor, initialize instruction analyzer and mnemonic and register tables. For example implementation, see `<nc/intel/IntelArchitecture.h>`.
- Implement a `nc::core::input::Parser` for the format in which input low-level programs are provided on your platform. In its `parse` method, initialize the `Module`'s architecture object to a new instance of your architecture.

2.2 IR: Intermediate Representation

The 'ir' module is responsible for conveying the exact semantics of assembler program in a form suitable for further analyses. The code of this module is located under the `nc::ir` namespace.

Intermediate Representation (IR) of a program or a function is their control flow graph (CFG). CFG of a program is implemented in the `CFG` class. CFG of a function is implemented in the `Function` class.

Basic blocks of both graphs are implemented in the `BasicBlock` class. A basic blocks *can* have a start address (basic blocks arising from complex instructions or, in some cases, during inlining don't have an address). They also have a list of predecessors and a list of successors.

2.2.1 Statements and Terms

A basic block consists of a sequence of *statements*. Statement is a simple operation like an assignment or a jump. Statements can have side effects. Statements are flat, i.e. statements can't contain other statements. `Statement` is a base class in the hierarchy of statements.

The following kinds of statements (subclasses of `Statement`) exist:

Comment — textual comment, useful mainly for debugging.

Assignment — assignment of one statement's argument to another.

Kill — killing of a reaching definition.

Jump — conditional jump to another basic block (destination basic block can be described either by its address or by a pointer to a `BasicBlock` object).

Call — call of a function by address.

Return — return from a function.

Jump instruction can stay only at the end of a basic block. The basic block being visited if jump doesn't happen is called *direct successor*; the `BasicBlock` class stores an additional pointer to it.

Arguments of statements are expressions. Expressions can't have side effects. An expression is represented as a tree. `Term` is a base class for the tree nodes. Each term has some size (measured in bits), and flags (whether this term is meant for reading, writing or killing).

The following kinds of terms (subclasses of `Term`) exist:

Constant — integer constant.

Intrinsic — some function computable in assembler, but which is hard or impossible to express in C.

Undefined — undefined value.

MemoryLocationAccess — access to a fixed memory location, such as a register.

Dereference — access of a memory location determined by an address expressed as an operand.

UnaryOperator — all kinds of unary operations.

BinaryOperator — all kinds of binary operations.

Choice — special binary operator returning its first argument if its definitions reach the choice, and its second argument otherwise. Choice is useful for generating human-friendly high-level code related to arithmetic flags and jumps (see `nc::intel::IntelInstructionAnalyzer` for details).

2.2.2 Memory Model

Memory of an IR machine consists of several unrelated memory domains. For example, each set of non-overlapping registers can be assigned a separate domain. Global memory and stack are two another domains. The `MemoryDomain` class has an enumeration of possible domains. User can define new domains when necessary.

A memory location then is determined by three integers: domain, address, and size. `MemoryLocation` is the class for representing memory locations.

2.2.3 Translation of Assembler Program to IR

IR is constructed out of the low-level representation, i.e. the `core::Module` and `core::InstructionSet` objects. Assembler program is translated directly to the CFG of the whole program, i.e. to an `ir::CFG` object. Translation is done by the `core::IRGenerator` class.

First, instructions are translated to statements. These statements are added to new basic blocks or to the end of existing basic blocks. Translation of specific instruction to a sequence of statements is actually done by the `core::InstructionAnalyzer` class (`core::Architecture` has a pointer to the right instance of this class).

Second (and last), appropriate arcs between created basic blocks are added. For this, a quick dataflow analysis is performed on the basic block level. This dataflow analysis helps to estimate destinations of jumps (especially of those done via a jump table), validity of jump conditions, etc.

2.2.4 Isolation of Functions

Generation of intermediate representations of functions is performed by the `FunctionsGenerator` class. It isolates functions in the program's CFG and adds new `Function` objects to an object of the `Functions` class being the container of functions.

Isolation of functions is done in two steps. First, if a basic block having a start address is found, and this address is used as a call target, then all transitive successors of this basic block are isolated into a new function. Second, if some basic blocks having a start address are still left, then all transitive successors of this basic block are isolated into a new function.

Existence of the first step allows to process functions with multiple entry nodes correctly. Such a function is translated to multiple functions, each having its own entry and a copy of the common body.

When a function is being created, its basic blocks are being cloned. Cloning of a set of nodes is implemented in the `FunctionsGenerator::cloneIntoFunction` method. For cloning statements and terms, the `clone` methods of `Statement` and `Term` classes are used.

As a convenience, function's CFG always has two fake nodes: entry and exit. These nodes do not contain any statements. All actual entry basic blocks of a function have an incoming arc from the fake entry node. All actual exit blocks of a function have an outgoing arc to the fake exit node. So, `function->entry()->successors()` gives all actual entry basic blocks of the function, and `function->exit()->predecessors()` gives all actual exit basic blocks of the function.

The `FunctionsGenerator::makeFunction` creates a new function out of given set of basic blocks and, optionally, an entry basic block. It automatically computes the sets of entry and exit basic blocks and adds appropriate arcs from/to the fake nodes.

2.2.5 Inlining

The library directly supports the basic operation of inlining a given function in place of a given call statement. This functionality is implemented by the class `inlining::CallInliner`, see the `perform` method.

2.2.6 Calling Conventions

Knowing how and which function arguments are passed and how values are returned from the function is crucial for dataflow analysis and code generation. Class `calls::CallGraph` stores the information about which function uses which calling convention. A function in such case is identified by a `calls::FunctionDescriptor` object. The object can identify a function either by its entry address or by the address of the call to this function (in case the real call argument cannot be found out). Calling convention is described by an instance of a class implementing `calls::CallingConvention` interface. Class `calls::GenericCallingConvention` is such an implementation suitable for describing most calling conventions.

A `calls::CallingConvention` can create an `calls::DescriptorAnalyzer`. The latter is responsible for reconstructing signature of a function as well as creating `calls::CallAnalyzer`, `calls::FunctionAnalyzer`, and `calls::ReturnAnalyzer` objects that will be used during the dataflow analysis for collecting calling convention-specific information during the dataflow analysis. When the `CallGraph` knows about the calling convention of a function, it creates suitable *Analyzer objects automatically when requested.

Reconstructed signature of a function is represented as a `calls::FunctionSignature` object.

One can use method `CallGraph::setCallingConventionDetector` to specify the calling convention detector to be used for descriptors, for which no calling convention was set so far.

2.2.7 Dataflow Analysis

IR can be a subject to partial interpretation. The aim of the partial interpretation is to compute a set of reaching definitions for each term.

Partial interpretation consists of integrated reaching definitions [3] and constant propagation/folding analysis [4]. It works on function level and is implemented in the `dflow::DataflowAnalyzer` class. The analysis recomputes reaching definitions and term values until a fixed point is reached. The method `dflow::DataflowAnalyzer::analyze` runs the analysis on a specified function.

The methods `dflow::DataflowAnalyzer::simulate` perform simulation of statement's or term's execution. The simulation methods take an instance of the `dflow::SimulationContext` class. Simulation context owns an object of the `dflow::ReachingDefinitions` class. The latter contains information about the definitions reaching simulated statement. During a call to `simulate`, reaching definitions are updated according to the semantics of given statement or term.

The results of dataflow analysis are stored in objects of the `dflow::Dataflow` class. For a term, they are capable of reporting its reaching definitions and value properties.

Calling Conventions Dataflow analysis respects calling conventions used by functions. When a set of reaching definition leaving the fake entry node of a function is

computed, the `calls::FunctionAnalyzer::simulateEnter` method is executed. Typically it sets registers to their initial values guaranteed by the convention. Similarly, when a return statement is simulated, the `calls::ReturnAnalyzer::simulateExit` method is called. Typically it runs simulation of the registers that can contain return value, in order to have the information about their reaching definitions later, while determining the way how the function returns its value. When a call statement is simulated, the `calls::CallAnalyzer::simulateCall` is called. Typically, it analyzes the definitions reaching the statement, tries to determine the list of actual arguments, and kills definitions of spoiled registers. The respective `*Analyzer` objects are provided by the `calls::CallGraph` (they are created automatically when first requested).

2.2.8 Reconstruction of Local Variables

Here, a variable is a set of terms realizing accesses to the same variable of original high-level program. Local variables are reconstructed as connected components of definition-use graph.

The algorithm of computing the connected components is implemented in the `vars::VariableAnalyzer`. The results of variable reconstruction are stored in `vars::Variables` objects. These objects can report an instance of `vars::Variable` associated with a given term. This pointer uniquely identifies the set of terms representing reconstructed local variable.

2.2.9 Hiding Redundant Computations

Some of the computations visible in intermediate representation should not be visible to the end user. These are, for example, dead computations, adjustments of stack pointers, etc.

Computing of the set of terms, operations on which must be visible in the generated code, is performed by `usage::UsageAnalyzer`. The algorithm works as follows:

1. Every term is marked as unused.
2. If a term represents a write to a global memory or to unknown location, it is marked as used.
3. Jump conditions and jump/call destinations are marked as used.
4. Return value of a function is marked as used too.

When a term is marked as used, all its definitions and child terms are marked as used recursively.

The results of the analysis are stored in objects of the `usage::Usage` class. Assignments to the terms not being marked as used won't produce any code during code generation.

2.2.10 Type Reconstruction

Reconstruction of high-level types is largely based on ideas from [5]. With each term, a `types::Type` object is associated. This object stores computed type traits of the term. These type traits are enough to generate high-level type description.

Type traits are computed by an iterative algorithm. The algorithm stops when a fixed point is reached. Since type traits are boolean flags and they can only be changed from `false` to `true`, the algorithm always terminates.

Type reconstruction algorithm is implemented in the class `types::TypeAnalyzer`. The resulting mapping from terms to their type traits objects is stored in an object of `types::Types` class.

2.2.11 Structural Analysis

For the reconstruction of high-level control flow statements, *structural analysis* [6] is used.

First, `cflow::GraphBuilder` transforms function's CFG to a `cflow::Graph` object. Translation is rather straightforward, since `cflow::Graph` is just yet another representation of a control flow graph. `cflow::Graph` has two kinds of nodes: `cflow::BasicBlockNode` (basic block) and `cflow::Region` (region, i.e. a set of nodes with single entry and zero or more exit nodes). After translation, the graph has a single region containing all the basic blocks.

Next, `cflow::StructureAnalyzer` runs structural analysis: it finds subgraphs matching certain patterns and moves them to newly created regions. Regions are marked by their kind: block, if-then, if-then-else, while, etc.

Result of the analysis is a modified graph with new regions singled out.

2.2.12 Code Generation

After all the analyses are done, the translation of IR into a high-level representation becomes essentially a technical task.

`cgen::CodeGenerator` is the central class doing the code generation. It takes the results of necessary analyses from `core::Context` and builds an AST of high-level program (see subsection 2.3).

`cgen::DeclarationGenerator` is the class generating function's declarations. `cgen::DefinitionGenerator` is its subclass; it does generate function's high-level code. It descends recursively through the hierarchy of control flow regions, statements and terms, every time `switch`ing on their kind.

2.3 LikeC: High-level Representation

The 'likec' module implements an abstract syntax tree for a C/C++-like language called 'LikeC'. Its code resides in the `nc::likec` namespace.

`Tree` is the central class storing the AST. A tree contains two types of entities: tree nodes and types.

The `TreeNode` class is a base class for the hierarchy of tree nodes. Nodes correspond to syntactical elements of the program: compilation units, function and variable declarations, statements, expressions, etc. Child nodes are owned by parents. Root node is owned by `Tree`.

`Type` is a base class for the hierarchy of high-level classes. Types are immutable. Most of them are created, owned and cached by `Tree`. Such approach implies efficient memory usage and allows of efficient type arithmetic.

LikeC representation implements algorithms for code simplification via rewriting: removal of unnecessary typecasts and unused labels, simplification of expressions, etc. Node-level rewriting is done in the `TreeNode::rewrite` method. The `Tree::rewriteRoot` rewrites the whole tree.

2.4 Crec: Reconstruction of C++ classes and exceptions

'Crec' module implements reconstruction of C++ class hierarchies and expcetion handling constructs. Description of the algorithm used is given in [1, 2]. Note that these algorithms are currently implemented only for MSVC compiler.

Code of 'crec' module resides in `nc::crec` namespace, with `crec::Creq` being the central class that stores all information on class hierarchy and exception handling reconstruction. `creq::Creq::perform` is the entry point for reconstruction algorithms.

2.4.1 Class hierarchy reconstruction

Description of the algorithm is given in [1]. It is recommended that you make yourself familiar with the approach described there before delving into implementation details.

Class hierarchy reconstruction process is performed in several steps.

1. Construction of the neccessary data structures. At this step `crec::Function` objects are constructed for each function in the IR. These objects store additional information that is used in the steps that follow.
2. Scanning of executable image for virtual function tables. This is done by the `crec::VtScanner` class. Note that the algorithm uses cross-reference information from the 'refs' module. At this step for each vtable an instance of `crec::VTable` class is constructed. All accesses to vtables are also found, and description of each access is stored as an instance of `crec::VTableAccess` class.
3. Interprocedural value set analysis of virtual functions for vtable accesses. Most of the job at this step is done by the `crec::ChainAnalyzer` class, which implements a custom analyzer for the value set analysis (see 'vsa' module). At this step vtable access chains and chain bulks (instances of `crec::VtChain` and `crec::VtChainBulk` classes) are constructed, the former representing a chain of consequent overwrites of the same memory location with addresses of different vtables, and the latter being a collection of vtable access chains that access memory locations differing by a constant offset.
4. `crec::ChainReconstructor` class does the rest of the job. Vtable access chains are classified as belonging either to constructors or destructors using the heuristics described in [2] and inheritance relations between vtables are reconstructed. Classes are then constructed from vtable access chain bulks, and inheritance relation between these classes is inferred from the inheritance relation between vtables they contain.

2.4.2 Reconstruction of exception handling constructs

Exception handling constructs are currently reconstructed for MSVC only. It is recommended that you study how exception handling in MSVC works before working with the implementation. A good coverage of the exception handling process is given in an article at OpenRCE: `http://www.openrce.org/articles/full_view/21`.

Reconstruction of exception handling constructs is performed by the `crec::ExceptionAnalyzer` class, which implements a custom analyzer for the value set analysis (see 'vsa' module). This analyzer scans the function, looking for a specific stack layout to find the location of exception handling structures associated with the function, and its exception counter. It then emulates execution of the function, computing the value of exception counter at each instruction. Borders of `try` and `catch` blocks are defined in terms of exception counter intervals, so they are easily reconstructed once its value is known.

2.5 Refs: Code cross-references

This is a helper module that constructs a set of cross-references that can be queried for instructions that reference the given memory location. Single cross-reference is represented by an instance of `refs::XRef` class, a set of all cross-references — by an instance of `refs::XRefSet` class.

Construction of a set of cross-references is performed by the `refs::XRefSetBuilder` class, which runs through the intermediate representation of the whole program and creates cross-references for all memory location accesses it encounters.

2.6 VSA: Value set analysis

This is a helper module that implements a simple form of value set analysis. The main classes of these module are:

- `vsa::DefaultAnalyzer` — class that performs analysis and emulation of statemets.
- `vsa::Context` — data class that represents analyzer state at some point of emulation. As emulation is non-linear, instances of this class are copied and merged whenever necessary by the analyzer.

The algorithm implemented has the following important properties:

- The algorithm tracks the values at known memory locations. Statement emulation changes these values, and classes derived from `vsa::DefaultAnalyzer` can hook into the process by overriding provided virtual methods.
- When encountering the code that was already simulated at some other branch of execution, it simply stops (See implementation of `DefaultAnalyzer::analyzeInner`). This means that it does not roll until reaching a fixed point, and that at each point of execution no more than one value can be stored for each memory location. So, it is not really a value *set* analysis, but such detail level is sufficient for the needs of 'crec' module.
- When encountering a function call, the analyzer emulates it if the target function's code is available (See implementation of `DefaultAnalyzer::analyze(Context *, const ir::Call *)`). Context at the end of function's execution is then merged into the current context. This makes it possible to track how a value at some memory location changes throughout an execution path that spans several functions.

2.7 Intel: Support for Intel x86 Architecture

Support for the family of Intel x86 architectures is implemented in the `nc::intel` namespace. Implementation followed the process described in section 2.1.10. Parsers supporting this architecture are elf, pe, dumpbin.

3 Nocode: Command Line Decompiler

Decompiler's command line front-end is called 'nocode'. Its code lies in the root namespace.

Nocode lets the user specify (on the command line) which files to parse and in which files to print the results of which analyses.

The code is rather straightforward.

4 SmartDec: Decompiler with a GUI

Decompiler's GUI front-end is implemented in the `nc::gui` namespace. It lets the user browse both assembler and decompiled source code. Cursor positions in code views are synchronized, so that the user always knows where the selected decompiled code originated from, and what selected assembler code was decompiled into.

The GUI follows the MVC (model-view-controller) model. `gui::CxxView`, `gui::InstructionsView`, `gui::SectionsView`, `gui::TreeInspector` are the view part. `gui::CxxDocument`, `gui::InstructionsModel`, `gui::SectionsModel`, `gui::TreeModel` are the models shown in the respective views. Actually, the `*Document` and `*Model` classes are just adaptors of another models. So, `CxxDocument` produces its context by printing a `likec::Tree` object. `InstructionsModel` is a wrapper over `core::InstructionSet` object. `SectionsModel` is a wrapper over the `core::image::Image`'s list of sections. `TreeModel` is a wrapper over the `core::Context` class. Every model class owns the underlying data it presents via a shared pointer.

All functionality of the module is wrapped up by the `gui::MainWindow` class, which implements GUI front-end window.

The controller part of MVC is currently realized by `gui::Project`. User issues various commands represented by `gui::Command` objects. These commands are added to a queue, `gui::CommandQueue` object owned by `Project`, and executed in order. Project tracks the state of `CommandQueue` and automatically launches decompilation when all changes requested by a user are done, and the queue is empty.

5 Nocode-plugin: IDA Pro plug-in

SmartDec is available as an IDA Pro plugin, which is implemented in the `nc::ida` namespace. The plug-in works by providing the access to the executable image loaded in IDA. It creates image sections using the information from IDA and implements `core::image::ByteSource` interface for forwarding the data access calls to IDA Pro API. When done, it opens `gui::MainWindow` and loads the new, IDA-based project into it. Disassembly of the code contained in the image is done by decompiler itself, i.e. independently from IDA.

As IDA Pro 5.x is implemented is Delphi, some magic is requred to make it work with Qt-based GUI that is used in the plugin. The neccessary magic is implemented in the `ida::QtSupportPlugin` class. This workaround is unnecessary for IDA Pro 6.x (although it does not hurt).

References

[1] A. Fokin, E. Derevenetc, A. Chernov and K. Troshina. “Reconstruction of C++-specific Constructs for Decompilation”. Never published.

[2] A. Fokin, E. Derevenetc, A. Chernov and K. Troshina. “SmartDec: Approaching C++ Decompilation”, in proceedings of the 18th Working Conference on Reverse Engineering, 2011.

[3] Reaching definition `http://en.wikipedia.org/wiki/Reaching_definition`

[4] Constant folding `http://en.wikipedia.org/wiki/Constant_folding`

[5] Е. Н. Трошина, А. В. Чернов. *Восстановление типов данных в задаче декомпилирования в язык Си.* Прикладная информатика, 2009.

[6] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, chapter 7. Morgan Kaufmann, 1997.