Introduction to the SUIF 2.0 Compiler System

Holger Kienle and Urs Hölzle
Department of Computer Science
University of California
Santa Barbara, CA 93106
http://www.cs.ucsb.edu/oocsb

Technical Report TRCS97–22

December 10, 1997

Contents

1 Purpose of this Manual **3**

2 Introduction and Overview **4**

3 The *sty* Library **5**

4 The *zot* Library **6**

5 The *pyg* Library **11**

6 The *suif* Library **16**

7 The *suifpasses* Library **22**

8 Assessment of SUIF 2.0 **24**

List of Figures

1 The zot interface 8
2 Simplified zot library class hierarchy 9
3 The pyg library class hierarchy (generic classes omitted) 12
4 SUIF 2.0 class hierarchy (type-related classes) 17
5 SUIF 2.0 instructions class hierarchy 18
6 SUIF 2.0 control flow graph nodes class hierarchy 20
7 SUIF 2.0 statements class hierarchy 21
8 SUIF 2.0 value blocks class hierarchy 22
9 SUIF 2.0 ownership graph example [Wil97c] 23

1 Purpose of this Manual

This manual is intended to give a brief overview of the new *SUIF 2.0 compiler system*[1]. It describes SUIF 2.0 at a very high level, focusing on the concepts without going into the gory details of the interface and the implementation.

The manual is meant to be an introduction to SUIF 2.0 that explains and motivates its new features (especially the extensibility scheme) and motivates why one would want to switch to the new system. It is not meant to be a detailed description of the interface.

The manual is basically a fleshed-out version of the SUIF 2.0 tutorial given by Chris Wilson at the *Second SUIF Compiler Workshop* [Wil97c]. It also contains some insights into SUIF 2.0 that we gained while implementing a Java front end for SUIF 2.0 [KH97]. Many thanks to Chris Wilson for patiently answering all our questions concerning SUIF 2.0.[2]

We don't expect that the reader is familiar with the SUIF 1.0 compiler system, but knowing SUIF 1.0 might be beneficial. For example, many concepts of the IR-specific parts of SUIF 2.0 will sound familiar to SUIF 1.0 users.

[1] Homepage: `http://www-suif.stanford.edu/suif/suif2.0/`.

[2] Many of these questions have made it into the SUIF 2.0 FAQ, which is accessible at the OSUIF homepage: `http://www.cs.ucsb.edu/~osuif/`.

2 Introduction and Overview

In order to overcome several shortcomings of SUIF 1.0, the Stanford Compiler Group developed a new version of SUIF, called *SUIF 2.0* in the following. The basic principles and functionality stays the same, but the old SUIF kernel has been completely redesigned. This means that SUIF 2.0 is not compatible with SUIF 1.0.

The SUIF 2.0 system provides a compatibility library that supports SUIF 1.0 for a smooth migration to the new system. Optimization passes of both versions can coexist. Conversion passes translate from SUIF 2.0 to SUIF 1.0 file format and vice versa. This allows one to use most parts of the old SUIF toolkit in the new system.

The new design of the SUIF kernel introduces 5 layers of abstraction[3]:

1. The *sty* library is the lowest layer of SUIF 2.0. It defines generic data structures (e.g., lists, arrays, and hash tables) that are used by higher layers, string classes, a lexicon class, a command line parser, classes that can hold arbitrary precise numerical values, etc. Relevant parts of the library are explained in more detail in section 3.

2. The *wallow* library provides architecture independent input/output routines. Data can be exchanged between two different architectures, even if they have different byte ordering and different sizes for various integer types. This library is used internally by the zot library. A user of the system does not need to know anything about it.

3. The *zot* library defines the base class of all SUIF objects. This design makes it possible to provide a uniform low-level view of all SUIF objects. Operations on SUIF objects that do not care about the actual type of the object (e.g., reading and writing SUIF objects to a file) can use this interface to handle SUIF objects in a uniform way. This library lays the groundwork for the extensibility of the system; it is explained in more detail in section 4.

4. The *pyg* library defines the abstract framework of the SUIF IR, for example, (abstract) base classes for types, symbol tables, instructions, state-

[3]We feel compelled to give a short explanation of the motivation that lead to the SUIF 2.0 library names—especially for non-native speakers. *pyg* is pronounced the some way as pig. Now, a pig—or rather pyg—uses to *wallow* in its *sty*; got it? *zot* has no meaning in the English language. It is a (recursively defined) abbreviation for "**z**ot **o**bject **t**ype." This on a first, innocent glance rather awkward name has been chosen because the name for the *abstract* third layer of the SUIF 2.0 system does not want to raise any unwanted associations by using a word for it that has a defined semantical meaning (at least in the English language)—simply because this association would probably turn out to be wrong or misleading anyway. Certainly everybody will agree that zot sounds a whole lot better than "thingy." The last library name, *suif*, turns out to stand for precisely what its name suggests, namely the topmost library of the SUIF 2.0 system—pretty disappointing, huh? [Wil97a].

ments, etc. This library is explained in more detail in section 5.

5. The *suif* library defines concrete SUIF IR constructs, for example, classes for integer types, arithmetic instructions, control flow statements, etc. This library is explained in more detail in section 6.

In order to use the existing functionality of SUIF, only the first, fourth, and fifth layer have to be used. Every SUIF class contains a fixed set of methods that realize the interface of the extensibility mechanism. If an application does not need to extend SUIF (which is the common case) it can simply ignore these methods. Extending SUIF, for example with a new instruction or data type, requires an additional understanding of the extensibility mechanism (which is provided by the zot library).

In the following, the actual C++ class name is given (in `typewriter` font) when a SUIF construct is explained in order to make it easier to find the corresponding source code.

3 The *sty* Library

This library implements a wide variety of conventional, linear data structures and provides a uniform interface for them. The abstract base class `ro_tos` (read-only totally ordered set) defines the uniform interface for read-only access. It is possible to access elements by index and ask for the number of elements (array-style) or to arbitrarily iterate over the set by getting the first/last element and asking for the next/previous one (list-style). `tos` is subclassed from `ro_tos` and extends the interface with methods that can modify the data structure (list-, array-, and stack-style). Concrete classes that are subclassed from `tos` implement

- arrays (`array_tos`)
- single linked-list (`slist_tos`)
- double linked-list (`dlist_tos`)
- cached double linked-list (`cdlist_tos`)
- array-style double linked-list (`adlist_tos`)

The implementation of `adlist_tos` is such that if only list-style operations are performed, it is as efficient as a linked-list, and if only array-style operations are performed it is as efficient as a simple array.

Other SUIF libraries make extensive use of these data structures. Because they have a uniform interface it is extremely easy to substitute different implementation strategies.

An iterator class (`tos_iter`) can be used to iterate over an `ro_tos`.

Arbitrary size integers are supported by the `i_integer` class. It has the same operators as a C++ `int`. An `i_integer` is internally represented as a `long` if it fits into it, otherwise a string is used. Strings are supported with the `string` and `lstring` class. A `lstring` is a subclass of `string` that is registered in a lexicon. They have unique identification numbers and string comparison can be done efficiently by pointer comparison.

The abstract base class `ion` can be used to create human-readable output. It is used to generate error/warning messages and output for the `print()` method defined in the zot class.

Error and warning reporting functionality are provided by global functions. They allow SUIF passes to optionally specify file and line number information and automatically put in line breaks to keep the output below 80 characters per line. Using this interface yields a more uniform "look" of the output of a SUIF pass. Output text is generated with the `ion` interface and can contain SUIF specific structures.

The class `therapist` provides an alternate, more flexible interface for error reporting. The test condition is passed as an argument to a `want()` or `need()` method along with an optional diagnosis message. These methods are used like assertions but they can be specialized by subclassing from therapist. The want method is used to indicate an assertion failure that is non-fatal and allows the program to proceed, whereas the need method is used for assertion failures the program cannot recover from. This behavior is, for example, useful for verification methods that check the state of an object because the verification need not to be aborted after the first assertion failure that is encountered.[4]

Finally, a powerful command line parser (`command_line_parser`) is available that can handle all standard usages of boolean, integer, and string arguments.

The sty library is not compiler-specific, but designed for interfaces between modules. It makes extensive use of C++ templates. The question why SUIF 2.0 does not make use of the C++ *Standard Template Library* (STL) [Str97] comes naturally to mind. At design time, STL was not widely available. Furthermore, STL uses exceptions, but many C++ compilers have trouble supporting them. There are other—more subjective—reasons, for example, STL makes extensive use of standard operator overloading, whereas SUIF seldom uses it.[5] [Wil97c]

4 The *zot* Library

This library is used in the SUIF system to model a flexible extensibility scheme that makes it possible to put new kinds of objects into the preexisting hierarchy of SUIF constructs. For example, this scheme allows one to add new kinds of

[4] Surprisingly often not the first assertion failure, but a subsequent one indicates the actual problem that caused the inconsistent object state.

[5] The usefulness of overloading standard operators is questionable (e.g., extensive and unintuitive usage can degrade program readability).

symbols, types, symbol tables (e.g., to model classes for object-oriented PLs), statements (e.g., to model exception handling or concurrency constructs), etc.

The abstract base class `zot` provides the interface that gives a single, logical view to all SUIF constructs that inherit from this base class. Because every object supports this interface, each object can be seen as a black box (even if the particular semantics of its components is not known).

The logical view of a `zot` object consists of (1) a string tag for introspection, and (2) a finite sequence of *brick* building blocks.

The string tag tells the position of the object in the SUIF hierarchy. For example, the SUIF instruction “add” has the following tag:

`"azot/szot/instruction/binary_arithmetic_instruction/add"`.

Note that the SUIF hierarchy does not necessarily reflect the C++ class hierarchy of the implementation, but it often does.[6] A tag tells how to interpret a specific object. Even if a user does not understand all of the tag, he/she might understand a certain prefix of it.

A brick, implemented in class `brick`, is one of the following:

- A reference to another object. IR objects form a directed graph, i.e., each object has some number of references to other objects. For example, an `add` instruction refers to the result type, the two source operands, and the destination operand.

- An *ownership link* to another object. Each SUIF object has exactly one parent, which can be accessed with the `parent()` method. This structure yields an “ownership tree” of the IR objects that is conceptually embedded into the directed graph structure of the object references. For example, an instruction in an expression tree is owned by another instruction, a statement is owned by an enclosing statement, etc. Having a distinguished owner for every object has several useful applications, for example, iterating over a node, reading/writing of a node, or cloning a node and everything that it owns. Note that leaf objects in the ownership tree do not have an ownership link brick. A variable symbol, for example, does never have an ownership link brick among its components.

- An integer, a string, or a sequence of bits.

Each object's data can be represented with a sequence of these bricks.

In summary, tags represent the semantical information and bricks store the raw data of an object. Figure 1 shows the zot interface.

File I/O at the zot level is abstracted in terms of tags and bricks. Usually a class X registers its tag and an instance of its *builder class* X`_builder` with

[6] For example, this is the case for all concrete SUIF arithmetic instructions, which are represented by either `binary_arithmetic_instruction` or `unary_arithmetic_instruction` in the C++ class hierarchy.

```
class zot {
private:
  zot *_parent;
  ...
public:
  virtual lstring tag(void) const = 0;

  virtual s_count_t num_components(void) const = 0;
  virtual brick component(s_count_t component_num) const = 0;
  virtual void set_component(s_count_t component_num, brick the_brick) = 0;

  virtual void verify_invariants(therapist *doc) const;
  virtual void apply_zot_visitor(zot_visitor *the_visitor);
  ...
};
```

Figure 1: The zot interface

the zot library. When X's tag is found when reading in from a file, the right in-memory representation of the object can be constructed by calling the associated *builder method*, namely `X_builder::default_builder()`. If X's tag has not been registered (i.e., no specific builder method is available) an object of class `generic_C` is created, whereas C is the "closest" subclass to X in the class hierarchy that has a registered builder method. The builder method calls a constructor that takes a list of bricks as data. This functionality is realized with the `zot_builder` class.

Every class X that is (transitively) subclassed from `nfzot` is required to implement a `generic_X` class as well. This makes it possible to represent objects of classes even when the class is unknown to the system (and hence no factory method for them exists). For example, a user might want to build a new library on top of SUIF and subclass Y from a SUIF class X. When class Y is read in but the system cannot find the specific library of the user, SUIF needs a mechanism to represent the addition information (i.e., bricks) of class Y that are not in class X. The class `generic_X` provides this functionality.

When reading in from a file we don't always want to read in the whole file and create the whole in-memory representation of it, but rather would like to do I/O on subtrees of the ownership tree of zot objects. For example, a pass that does intra-procedural analysis wants one procedure body at a time. Thus, it is not necessary to have all procedure bodies in memory simultaneously. In order to handle such cases, a very flexible scheme is provided that groups sub-trees of zot objects into I/O units. A group of I/O units is represented with the `zio` class. The user can read, write and deallocate zot objects in units of zios. For example, a file set block contains a list of procedure bodies. By grouping every procedure body subtree into a separate zio we can read in one procedure at a time. The zio functionality is used for several different memory models that the suifpasses library provides (refer to section ??).

It can happen that a zot object in memory contains a reference to another object that is currently not in memory. For example, a inter-procedural analysis might store information that points from a zot object in one procedure to an object in another procedure. If a subsequent intra-procedural pass reads in one procedure at a time it is very likely that references are out-of-memory. These cases are handled by the library with a "forwarder" mechanism. A `forwarder` is a special zot class that serves as proxy for something that is not in memory. When a class is read in from a file that contains a reference to on object that is out-of-memory and in a different zio, a reference to a forwarder object is created.[7] The forwarder allows the user to transparently see the object the forwarder is referring to.

The non-forwarder classes are distinguished from the forwarder class by inheriting from the abstract class `nfzot` (non-forwarder zot). All SUIF constructs from the pyg and suif library inherit from this class. Figure 2 shows a simplified class hierarchy of the zot library. Note that no `generic_zot` class exists because the only legal subclasses of zot are forwarder and nfzot. Hence, every zot object can be represented as either a forwarder or a nfzot. The `generic_forwarder` class has not been implemented because there seems to be no application that requires to subclass from `forwarder` [Wil97b]. Typically, a user will extend azot or a class that inherits from azot.

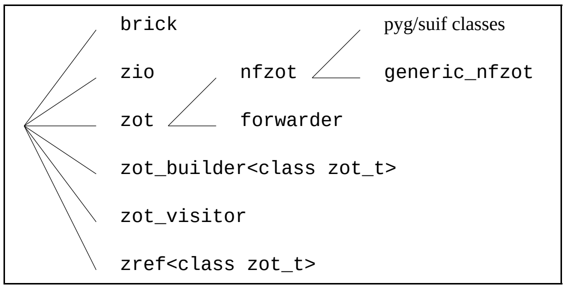

Figure 2: Simplified zot library class hierarchy

The `zref` class is a more convenient wrapper for an object reference that can either be a forwarder or a "normal" zot object. A zref always return the forwarded zot object (going along the chain of forwarders if necessary) or nil if the forwarded object is not in memory.

[7] References to objects that are out-of-memory and in the same zio are first initialized to nil and "backpatched" as soon as the object is in memory.

The zot level also introduces the interface for the visitor pattern [GHJV94] (see figure 1). The zot visitor is implemented with the class `zot_visitor`. A visitor class contains methods to visit all SUIF IR constructs that the library defines. The default implementation is to do nothing. A visitor can be customized by subclassing from it. Every library L that builds on top of zot extends the visitor framework by defining `apply_`L`_visitor()` methods and the L`_visitor` class. The default behavior for a visitor method that implements a new IR construct is to call the method for the closest underlying IR construct.

Furthermore, an interface to clone zot objects is provided. Every class X that inherits from the zot hierarchy must implement the method `clone_tree_`X`()`, which performs a partial deep clone, and the method `clone_local_`X`()`, which performs a shallow clone. In order to get a complete deep clone the method `clone()`, which is implemented by the zot class, is called. A deep clone replicates all SUIF objects that it owns, but no references to objects that do not live in the subtree that is about to be cloned. The clone method first uses the `clone_tree_*()` methods to get a deep clone with unresolved references and then backpatches the unresolved references. The last step of the cloning process calls `close_clone()` for every object of the cloned tree. This method allows the implementation of class-specific cloning semantics.[8]

Finally, the library provides interfaces for useful functionality that should be implemented by every class that inherits form the zot hierarchy, namely, printing and verification. A new class will usually redefine the `print_full()` method in order to provide a specific printout of its state. (Printing can be seen as a debugging-friendly, high-level representation of bricks.) Because classes are not forced to redefine the print method, it is possible that a print method, after pretty-printing the state of its own class, has bricks left it knows nothing about (from the chain of subclasses that did not redefine the print method). If this happens, the remaining bricks (i.e., the state of the subclasses) are printed at a low-level with the `print_brick()` global function. The print methods use the ion class to generate string output and the `zot_print_helper` class. The helper class manages indentation and prints a zot (by invoking `print_full()` on the zot) or just a reference to a zot (usually represented by a number). Verification of the state of an object can be done by invoking the `verify_invariants()` method. Only the local state of the object is verified; the verify method is not intended to walk a tree and verify everything in the tree. It is checked, for example, that if there is an ownership reference to an object that this referenced object has its parent pointer pointing back. Furthermore, it is verified that references to other objects have the correct type. In order to verify the whole (sub-)tree, the user must iterate over it. This means that if the verified object contains references to other zot objects these must be verified in turn. A similar situation to the print method arises when this verify method encounters state

[8] For example, whenever a `label_location_instruction` is cloned, the corresponding code label symbol needs to be explicitly cloned as well if the symbol table where the label lives has not been cloned.

in the form of bricks. In this case, the verify method can look for bricks that represent a reference to another zot object and invoke the verify method on them. The verify method uses the `therapist` class (see section 3) to issue diagnosis messages.

The zot framework is nothing compiler specific; the same design could be used for other graph structures. The zot interface is orthogonal to the high-level interface of a SUIF construct. A typical user will only use the high-level interface and ignore the low-level zot interface.

5 The *pyg* Library

The pyg library is the first compiler-specific library of the system. The (abstract) constructs of the SUIF IR are implemented in this library. For example, the pyg library defines an abstract base class `instruction` to model the concept of an instruction. We call such classes that are subclasses of zot, but not leafs in the inheritance hierarchy *intermediate classes*. Intermediate classes have intermediate interfaces. For example, instruction defines the interface for an arbitrary number of source and destination operands, and result types. All concrete instructions have these components. Every subclass gives a more concrete interface. For example, `binary_arithmetic_instruction` restricts the number of source operand to two. Usually intermediate classes are defined in pyg, and leaf classes are defined in the suif library. The complete pyg class hierarchy is shown in figure 3.

The `azot` (annotatable zot) class inherits from zot. All SUIF IR constructs inherit from azot. An *annotation* consist of (1) a string to identify it, and (2) data. The abstract base class `annote` implements annotations. The `generic_annote` class keeps the annotation data as bricks. If the user wishes a high-level view of the annotation data he/she can derive a custom subclass from annote. The annotation name is the last part of the tag.[9] The annote class itself is a subclass of azot, which makes it possible to annotate annotations. An arbitrary number of annotations can be attached to an azot.

The `sto` (symbol table object) class defines objects that can be stored in symbol tables. The sto interface allows symbol tables to treat their contents uniformly. The pyg library has two kinds of symbol table objects: (1) symbols, and (2) types.

A `type` can be one of the following:

- A data type (`data_type`) gives structure and semantical interpretation to otherwise “raw” data.

[9] This subtle design decision makes it possible that annotations that are instances of the `generic_annote` class will be “magically” restored to instances of this class when read back from a file.

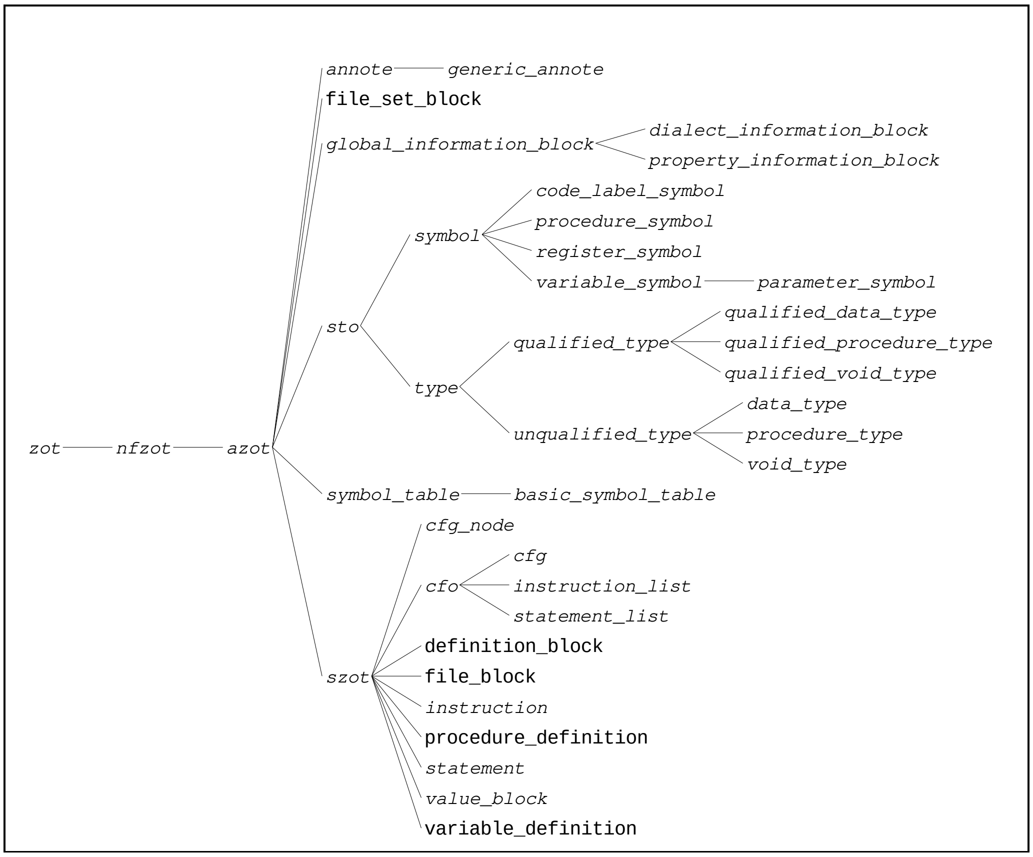

Figure 3: The pyg library class hierarchy (generic classes omitted)

- The type of a function is expressed with the class `procedure_type`. A procedure type can have multiple return types. (A SUIF procedure can have multiple return values.)
- The void type (`void_type`).

Types can be qualified or unqualified. A *qualified type* is a unqualified type with additional *qualifications* about storage (not data/values).[10] For example, expressions always return unqualified types because they give a value; there is no storage location involved. But a pointer type refers to a qualified type because we want to be able to represent information about the storage that

[10] This scheme has been adapted from the ANSI C standard.

the pointer points to.[11] SUIF supports qualifications such as `volatile` and `const`. The user is allowed to add other qualifications, but they should follow the above reasoning. For example, there could be qualifications `local memory` and `remote memory` to distinguish memory with different physical access characteristics. This information can be useful to optimize code based on knowing that some memory accesses are cheaper than others.[12] The type hierarchy for qualified types mirrors the type hierarchy for unqualified types. The class `qualified_`X is implemented as a wrapper that refers to the underlying unqualified class X.

All `symbols` can be—unlike certain types—conveniently identified by their name.[13] Every symbol can be asked if its address has been taken. This is useful for alias analysis. There are four kinds of symbols:

- Variable symbols (`variable_symbol`) represent data storage. They are typed and hence must contain a reference to a SUIF type. Variable symbols that represent non-stack data have a reference to a variable definition (see below). Typically a variable symbol is used as an instruction operand.
- Procedure symbols (`procedure_symbol)` have a reference to a qualified procedure type, and if the procedure is not external a reference to a procedure definition (see below).
- A `code_label_symbol` denotes an specific program point in the IR. Usually a label is used as a jump target for control transfer instructions (e.g., `jump_instruction`).
- The formal arguments of a procedure definition are represented by the `parameter_symbol` class. It is subclassed from a variable symbol and contains additionally a reference to its procedure definition.
- A `register_symbol`—unlike a variable symbol—has no type. An explicit type is given when the symbol is used. Standard SUIF does not make use of register symbols, but a lowering library that uses a machine independent, low-level IR (similar to Machine SUIF for SUIF 1.0) can express machine registers in a natural way.

A `symbol_table` is a container for symbol table objects. Its interface defines methods to retrieve, remove, and insert sto objects.

[11] One might wonder why qualifications are not attached to symbols. Pointer variables are one of the reasons, because sometimes storage is not directly associated with a variable, but can be referenced indirectly through a pointer.

[12] Note that we want to know this information for all storage accesses, direct uses of variables as well as indirect references through pointers. That is why it is appropriate to use these qualifications on types.

[13] Names are not required to be unique in a symbol table. Lookup by name can retrieve multiple symbols.

The abstract base class `cfo` (*control flow objects*) represents a chunk of executable code, such as a procedure body or the `then`-part of an `if` instruction. There are three different choices to represent code, namely,

- as a control flow graph (`cfg`). A cfg is a directed graph. Vertices are represented with the `cfg_node` class. Outgoing edges of a node are (conceptually) modeled as references to other nodes in the cfg. Every cfg node contains a cfo object. Every CFG must have a `start node` and can have an `exit node`. The exit node is used to model nested CFGs, but it can be used for flat CFGs[14] as well. (An explicit exit node often simplifies CFG algorithms.)
- as a list of instructions (`instruction_list`). SUIF instructions (`instruction`) are used for low-level, explicit control flow. They are atomic in a sense that they cannot contain other cfo objects.
- as a list of statements (`statement_list`). Statements (`statement`) are used for high-level structuring of the control flow. They are similar to ASTs (e.g., control flow is implicit). SUIF statements are very similar to statements as they can be found in high-level programming languages. For example, a `while` statement has an operand that represents the condition and a cfo object that represents the loop body.

The above mentioned representations can be freely mixed. For example, it is legal to have a cfg whose nodes contain sub-`cfg`s or statement lists. SUIF provides conversion passes that transform between any of the cfo representations.

The class `constant` is used to represent a compile-time constant. For instance, it is used as the operand of the `load_constant_instruction` and for the `case`-labels of the `multi_way_branch_instruction`. A constant can be an integer value, a bit-block, or a string. Constant strings are used to encode floating-point values. (This is machine-independent and allows the encoding of "NaN.") Furthermore, it can be used for application-specific constants. For example, a front end might wish to represent a nil reference with a string constant that is lowered to a concrete representation by a subsequent machine-specific pass.

Operands in SUIF are represented with the abstract base class `operand`. Source and destination operands of instructions are distinguished with instances of the classes `source_op` and `destination_op`, respectively. An operand holds

- a variable symbol.
- a register symbol and its data type.
- a reference to an instruction and the number[15] of a source or destination operand that is part of this instruction. More precisely, if it is a source

[14] Typically, the nodes of a CFG contain a list of instructions.

[15] The numbering starts with zero, i.e., number 0 refers to the first operand and so on.

operand the number denotes the destination operand of the instruction; if it is a destination operand the number denotes the number of the source operand of the instruction.

This last features makes it possible to build *expression trees*. They do not require temporary variables and are often straightforward to build. For example, it is possible to build the expression $x + 42$ without introducing temporary variables when an expression tree is used. The `add` instruction expects two source operands. The first operand holds a variable symbol (which represents x), the second operand holds a `load_constant_instruction` and the number zero, which denotes the first (and only) destination operand of the load instruction (which holds the value 42). Finally, this tree, which represents a (sub-)expression, could be combined with another expression tree or assigned to an l-value.

Note that constants and operands are not strictly part of the IR because they are not derived from zot and hence not explicitly written out to a file. Constants are written out as bricks and an operand is a wrapper that can reference to different IR constructs.

The `variable_definition` class is used to represent statically allocated variables (i.e., the memory address is known during compile time). The storage represented by a variable definition can be explicitly initialized with a `value_block`. A variable symbol can have at most one variable definition.

Similarly, a procedure is defined with a `procedure_definition`. It contains a symbol table (which represents the procedural scope), the procedure body (which is a cfo object), a list of formal parameters (which are represented with the `parameter_symbol` class), and a `definition_block` that holds statically allocated locals (e.g., non-`auto` variables in C/C++).

At the highest level of the IR resides the `file_set_block`. It contains

- a list of file blocks (`file_block`). A file block represents a file scope (i.e., a single source file). Note that SUIF does not support namespaces, which can be orthogonal to file scopes (e.g., C++). Furthermore, a file set block has a symbol table and a definition block that holds a list of variable and procedure definitions. If the source language has a flat namespace (which means, for example, that a file scope does not introduce a new namespace), the file set block's symbol table is empty. It the source language does not support multiple source files (e.g., original Pascal and SIMULA 67) the file set block contains a single file block.

- a file set symbol table that holds objects that are visible for all file scopes.

- an external symbol table that hold objects for external linkage.

- information blocks (see below).

Information that is invariant for the entire file set is kept in an *information block* system. Any of the following blocks, which are subclassed from `global_information_block`, can be hang off a file set block:

- The `dialect_information_block` lists *dialect properties* for a particular file set. This information is used by the suifpasses library (refer to section ??) to ensure that passes are run in the right order and that passes make valid assumptions about the input that they receive from a previous pass.
- The `property_information_block` keeps information about SUIF IR constructs. In the information block a class is denoted by its tag and properties are stored as strings. For example, for an instruction the property "has no side effects," and for a binary arithmetic instruction the property "is associative" can be given. In general, it is possible to put any kind of property information on a tag, similar to the annotation mechanism. Properties are useful for optimization passes that do not know the precise semantics of a (user defined) IR construct.

To summarize, the pyg library provides the conceptual framework of the IR. The suif library, discussed in the next section, implements a concrete IR on top of this conceptual framework.[16]

6 The *suif* Library

The top-level library of the SUIF system contains all the IR constructs that are needed to represent imperative languages, such as C and Fortran. In order to support these languages, the suif library extends the system with (concrete) types, instructions, statements, and control flow nodes. Furthermore, subclasses of `value_block` and additional information blocks are defined.

Figure 4 shows the unqualified data types that the library implements. This figure omits the class hierarchy for qualified types because it is identical to the unqualified type hierarchy. (See section 5 and figure 3 for a more detailed explanation.)

Most types are self-explanatory. Integer and floating-point types can be defined with arbitrary precision (and alignment restrictions) at the bit level.

A `group_type` corresponds to a `struct` or `union`[17] in C. A boolean flag indicates if the type is *incomplete*, i.e., not all fields of the group type have been specified. Standard SUIF requires that an incomplete group type is empty (i.e., it does not contain any fields).[18] Incomplete group types in standard SUIF correspond to *forward declarations* in C++ [Str97]. The *SUIF linker*,

[16]This distinction is not always intuitive and somewhat in the eye of the beholder.

[17]Fields in a group type can overlap arbitrarily.

[18]Note that on the other hand an empty group types does not imply that the type is incomplete (as opposed to SUIF 1.0.)

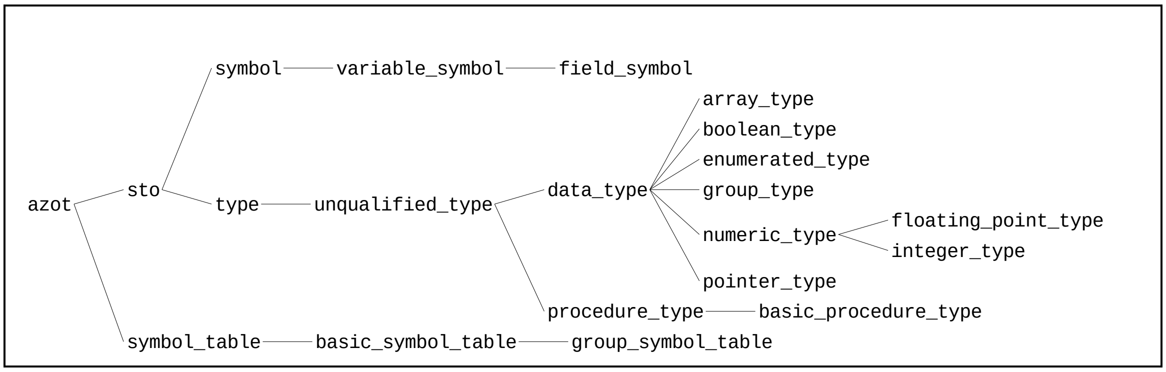

Figure 4: SUIF 2.0 class hierarchy (type-related classes)

which takes an arbitrary number of SUIF files as input, resolves incomplete types by replacing them with the actual type. A group type contains a `group_symbol_table`. This symbol table is subclasses form `basic_procedure_table` and can hold only `field_symbols`. Field symbols are used to represent fields in a group type. In addition to a variable symbol they contain a field offset.

The `basic_procedure_type` implements a procedure type that is similar to C functions in the sense that they can have a variable number of arguments and need not have their argument types specified.[19] Unlike C functions, they can have multiple return values. All types have an alignment restriction (specified in multiples of bits).

It is not required to specify a type completely. For example, one can leave the alignment restriction for a new type unspecified. In this case, a subsequent (target-specific) pass will fill in this information before code is generated. Similarly, the upper and lower bounds for an array type[20] and the offset for a field in a group type can be omitted.

SUIF distinguishes between structural equivalence and name equivalence of types [ASU86]. Name equivalence is used for group types[21] and enumerations (`enumerated_type`). These types have names (like symbols). All other types use structural equivalence and hence have no name. Every type X that uses structural equivalence defines global functions `get_`X`()`, which check if the type to be entered into the global symbol table already exists. If so, the existing type is returned; otherwise, a new type is constructed.

[19] Such a type corresponds to a function prototype with empty parameter list in K&R C.

[20] This does not mean that SUIF supports *dynamic arrays* (like Ada and Java) or *flexible arrays* (like APL and CLU) [GJ87]. A subsequent pass must specify at least the lower bound before code can be generated for an array.

[21] Note that SUIF's notion of type equivalence for group types resembles Pascal `records`, but differs from C `structs`.

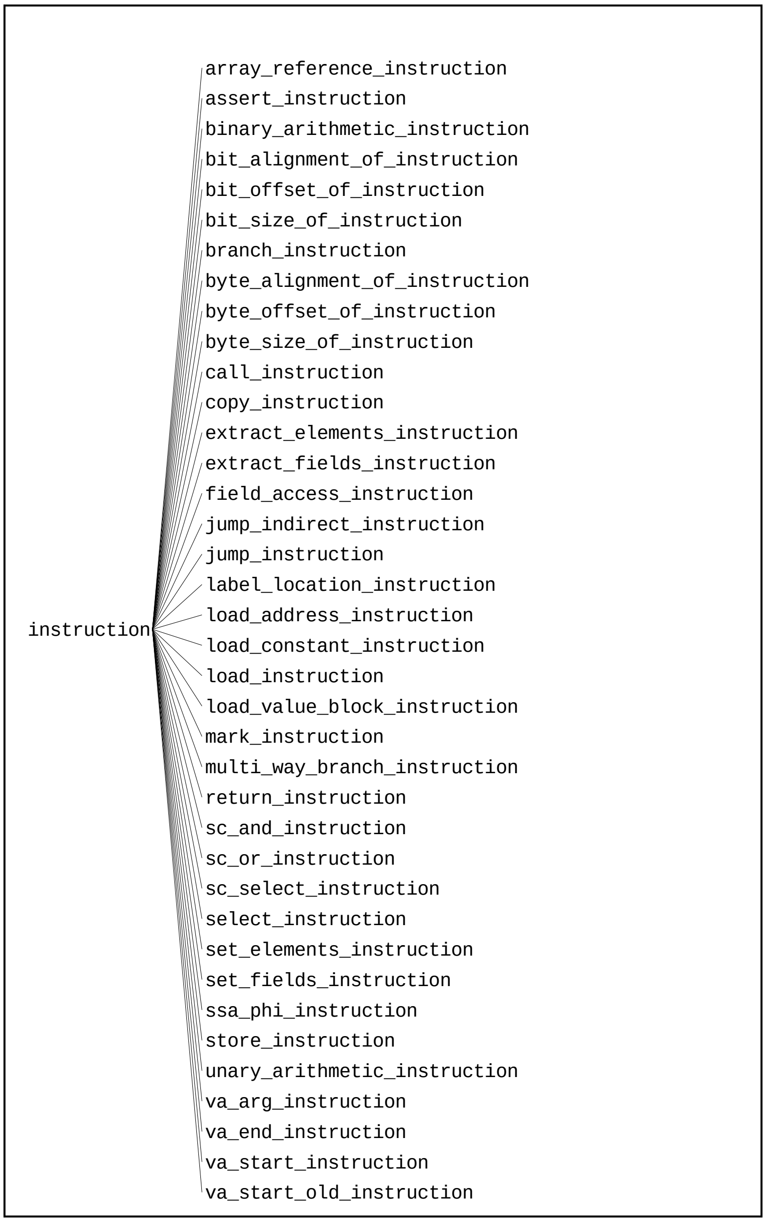

Figure 5: SUIF 2.0 instructions class hierarchy

Furthermore, the suif library defines concrete instructions (see figure 5). Instructions take any number of source operands, constants, etc. as input and have zero or more destination operands[22] as output. Thus, an instruction can be seen as a function with multiple return values. In the following, we give a short explanation of instructions whose semantics might not be obvious:

- The `label_location_instruction` defines a label. (Note that labels are separate instructions in SUIF, which are not attached to instructions.) A `branch_instruction`/`jump_instruction` takes such a label as its branch/jump target. Every label instruction must have a unique `code_label_symbol`.

- The `copy_instruction` is used to assign a source operand to one or more destination operands. It must not be confused with the `store_instruction`, which stores a source operand into a storage location (which is denoted by a source operand[23]).

- The `field_access_instruction` takes the base address of a variable symbol, which represents a group type, and a field symbol. The address of the field is returned. In order to store a value into a field the store instruction is used subsequently; in order to access the field value the `load_address_instruction` is used subsequently. The `select_fields_instruction` and `set_fields_instruction` can be used to conveniently access the fields in a group type without using addresses.

  Instructions for arrays are almost analogous to group types, with the only difference that an index must be given instead of a field symbol.

- The `ssa_phi_instruction` represents the SSA ϕ operator [CFRW91]. This instruction is only legal in code that uses the control flow graph representation. (SUIF offers passes that transform code from non-SSA form to SSA form and vice versa.)

- The `sc_*` instructions are used for short-cut boolean operators. (Standard boolean operators are defined as `binary_arithmetic_instructions`.)

- The `mark_instruction` does not perform any operation. It can be used to attach annotations (e.g., for line number information).

SUIF allows the construction of expression trees by using an instruction as a source operand for another instruction.

In the CFG representation, SUIF distinguishes between the following types of control flow graph nodes, which are subclasses from `cfg_node` (see figure 6):

[22] Certain SUIF dialects restrict the number of destination operands to zero or one.

[23] Note that a source operand is used because the store instruction does not return a value. The actual store is just a "side effect" of the instruction.

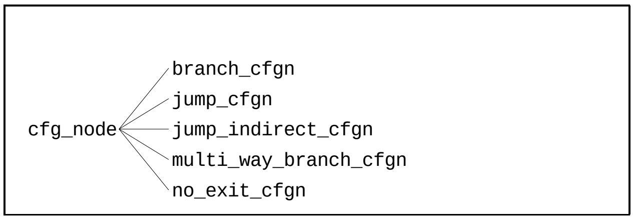

Figure 6: SUIF 2.0 control flow graph nodes class hierarchy

- A `branch_cfgn` is used for conditional jumps, thus the node has two out edges.
- A `jump_cfgn` is used for unconditional jumps, thus the node has one out edge. This node is also used for control flow that "falls through" to another node.[24]
- A `jump_indirect_cfgn` is used for an indirect jump to a code label symbol. (A `load_address_instruction` can be used to get the address of a code label symbol.[25])
- A `no_exit_cfgn` has no out edges. Typically, this block transfers control flow back to a caller (e.g., with a `return_instruction`) or marks the end of the program.
- A `multi_way_branch_cfgn` corresponds to a `switch` statement in C.

Statements (see figure 7) are used in SUIF to express control flow at a high level, as opposed to the low-level representation that instructions and CFGs offer. Statements are used only to structure control flow; instructions must be used to perform computations. The `eval_statement`, which holds a sequence of instructions, is used to wrap instructions into a statement.[26] Conceptually, all expression are evaluated first before any values are written.

Concrete subclasses of `value_block` (see figure 8) are used to initialize symbols with constants (e.g., a numerical value), or expressions (e.g., a numerical computation that uses the value or address of another symbol). A `multi_value_block` wraps a sequence of arbitrary value blocks. It is used to initialize group

[24] Often a separate fall-through node is used to distinguish a basic block that ends with an unconditional jump from a block that ends with a non-transfer instruction, but this is not strictly necessary if the original code sequence needs not to be recovered.

[25] By convention, the type of the label's address should be a `void*`.

[26] Usually a single expression tree per eval statement is used.

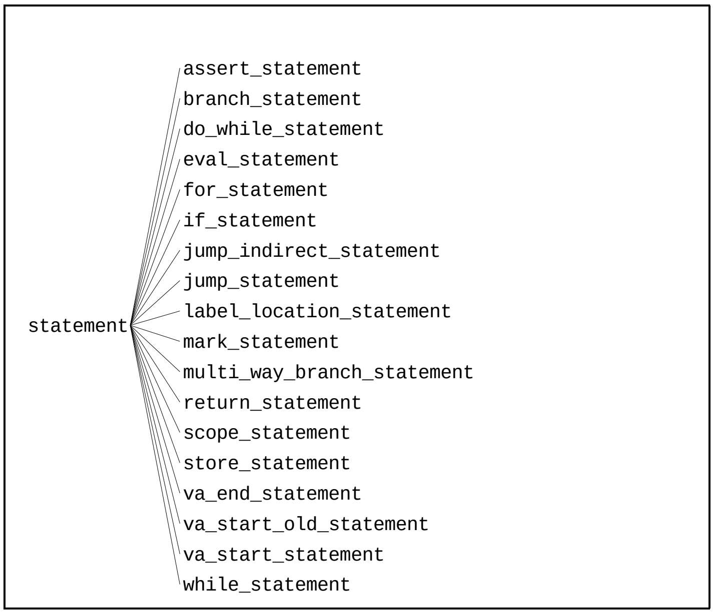

Figure 7: SUIF 2.0 statements class hierarchy

and array types. The `undefined_value_block` explicitly specifies that a symbol is uninitialized, but this is not mandatory.[27]

Finally, the library defines two additional information blocks:

- The `c_information_block` contains predefined C data types, which are useful for calls to the C library (e.g., `printf()`), or to external C code.
- The `target_machine_block` contains machine specific information (e.g., big or little endian, and alignment restrictions) that general high-level passes might need to know.

It should be noted that the discussed libraries provide only the basic functionality that is required. The SUIF philosophy is to keep the interface and the implementation at this stage as simple as possible. A more user-friendly interface is provided by auxiliary utility libraries on top of the five core libraries.[28]

Last but not least—to wrap up the SUIF libraries—figure 9 gives an (incomplete) example of an IR ownership graph.

[27] This value block is not strictly needed for symbols with scalar types, but for (partially) initialized symbols with array or group types.

[28] As of this writing, these libraries are not implemented yet.

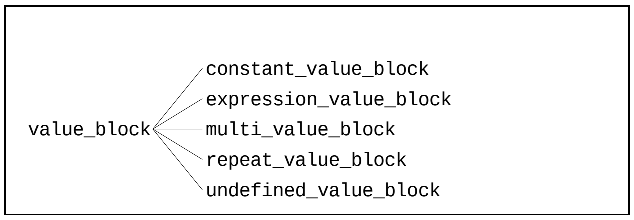

Figure 8: SUIF 2.0 value blocks class hierarchy

7 The *suifpasses* Library

SUIF passes are usually independent units that run on or create SUIF code. They can be composed in memory or through files. A generic driver offers both behaviors for all passes that use the suifpasses library.

The suifpasses library distinguishes between the following types of passes (which are subclassed from the abstract base class `suif_pass`):

- A front-end pass (`suif_front_end_pass`) is used to represent passes that produce SUIF code of a program as output but do not take SUIF code as input.
- A SUIF-to-SUIF pass (`suif_to_suif_pass`) is used to represent SUIF-to-SUIF transformations of the SUIF code. This is typically an optimization pass. Several standard memory models are supported. For example, the "one definition at a time" model is used by passes that only want to see one procedure or variable definition at a time. This might be a useful model for an intra-procedural optimization pass. On the other extreme, the "all in memory" model reads in the complete SUIF code before the pass is run.
- A back-end pass (`suif_back_end_pass`) takes SUIF as input but does not produce any SUIF code as output. For instance, a pass that prints a textual representation of the SUIF code would be implemented as a back-end pass.

Each pass is registered by name and specifies its *dialect properties*. The transformations on the SUIF code that are made by the pass and the properties of the SUIF code that the pass expects are described in terms of these dialect properties. The *standardsuifproperties library* has a collection of useful standard dialect properties. These properties can be checked with the *dialect checking framework* (if a certain dialect has a registered checker). The checker does static checks of rules for a dialect on SUIF code (similar to `checksuif` for SUIF 1.0).

Each pass specifies the dialect properties that

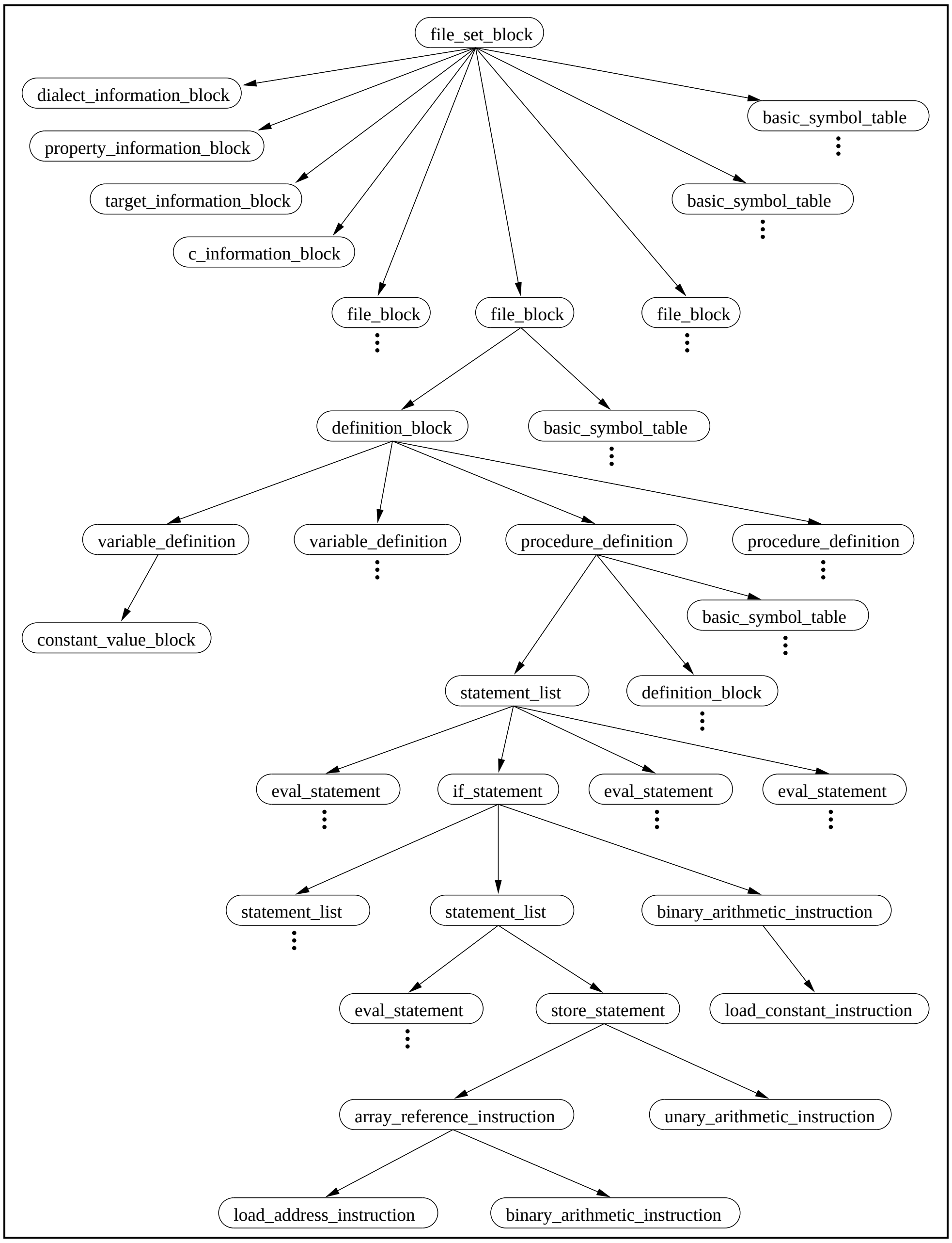

Figure 9: SUIF 2.0 ownership graph example [Wil97c]

- it expects from the input (as prerequisites).
- are preserved (if already present).
- are destroyed (by the transformations that the pass does).
- are generated (by the transformations that the pass does).

This framework makes it easy to describe the effects that a pass has on the SUIF code. Furthermore, it can be used to verify that passes are run in correct order, i.e., that they make correct assumptions about the input that they get from a previous pass.

The suifpasses library organizes algorithms in an object-oriented way that allows modularity, extensibility, and reuse of passes.

8 Assessment of SUIF 2.0

SUIF 2.0 has several benefits compared to SUIF 1.0.:

- It is more extensible and highly orthogonal. The most important part of the new design is the zot library, which models an extensible class hierarchy.
- It uses introspection with string-tags. They convey more semantical information (because they contain the complete SUIF hierarchy) then the SUIF 1.0 implementation based on `enums`.
- It supports nested procedures. (A nested procedure is kept in the `definition_block` of the enclosing procedure.)
- It is implemented in a more object-oriented programming style.
- It allows a more object-oriented style of programming. SUIF passes are now encapsulated into classes, making it easier to write variations on existing passes. Visitors can be used to iterate over SUIF structures (e.g., procedure definitions).
- It provides a new mechanism to manage extensions to the IR. Derivations from the base IR are called dialects. The well-defined "Standard SUIF" is the central dialect. It is possible to register new dialects and variations of existing dialects and keep track of which passes can handle which dialects.
- It additionally offers a CFG representation, which is very useful to describe unstructured control-flow.
- It has new, more expressive instructions (e.g., for short-cut boolean expressions and SSA form).

- It has a more flexible type scheme. The specification of the memory layout of a type can be deferred until it is actual needed. For example, a front end can define a group type without giving any alignment or padding information for the fields. A back end later fills in the missing information in order to generate code for a specific machine architecture. In contrast, SUIF 1.0 assumes that the field offsets are known.

- Its *interpretation framework* allows any piece of SUIF code to be evaluated by providing an interpreter class for each dialect. This functionality might be useful for constant folding and *partial evaluation* [CD93]. Furthermore, the interpreter code serves as the exact definition of what a dialect means.[29]

- SUIF 2.0 passes can be composed in memory; SUIF 1.0 required file I/O in order to sequentially execute passes. It is still possible to write out the result of a pass into a file. This is useful during development and debugging. No intervening file I/O during passes greatly improves the overall speed of compilation and moves SUIF closer to the domain of a production compiler.

- Passes in SUIF 1.0 had to be very conservative when they encountered an instruction that they did not understand because no information about the effect of the instruction is know. SUIF 2.0 allows the definition of *semantic properties* for user defined instructions. Such properties specify, for example, if the instruction has a side effect, or if it is commutative. Thus, passes that do not know about the exact semantics of an instruction are still able to extract useful information for optimizations.

- Development experiences with SUIF 1.0 uncovered a number of interface details that turned out to be inconvenient—SUIF 2.0 remedies these. For example, `if` statements in SUIF 1.0 contain a header part that tests for the condition. This header contains an explicit branch instruction to an implicitly defined `else` label. This turned out to be awkward to handle for optimization passes. SUIF 2.0 defines the condition now as an expression (like C) thus making the control flow implicit.

- The use of annotations is often no longer necessary. The same effect can usually be achieved in a more consistent and intuitive way by extending SUIF 2.0 constructs (e.g., types, symbols or statements) directly. For example, one can introduce a new statement in order to model a C++ try block (presumably derived from the `scope_statement` class). SUIF's goal is to provide extensibility in a way that allows existing code to continue

[29] Using the interpreter code to define the semantics of a SUIF construct is certainly questionable because bugs in the interpreter code become features of the semantics of a SUIF construct, but the practical advantages overshadow this rather theoretical concern.

to work. For example, even it old passes do not understand all about a new construct, they should be able to be conservative about its semantics. One way to achieve this is to use semantic properties.

Note that SUIF 2.0 does not extend the functionality of SUIF 1.0 towards non-imperative languages, but provides far better support for implementing those extensions. *Object SUIF* (OSUIF) [DCI+97], which is currently developed at UCSB, is such an extension for object-oriented languages.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, 1986.

[CD93] C. Consel and O. Danvy. Tutorial notes on partial evaluation. *ACM Symposium on Principles of Programming Languages*, pages 493–501, January 1993.

[CFRW91] R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman. Efficiently computing static single assignment form and the control dependency graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[DCI+97] A. Duncan, B. Cocosel, C. Iancu, H. Kienle, R. Rugina, U. Hölzle, and M. Rinard. OSUIF: SUIF with objects. *Proceedings of the Second SUIF Compiler Workshop*, pages 1–7, August 1997.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[GJ87] Carlo Ghezzi and Medhi Jazayeri. *Programming Language Concepts.* John Wiley & Sons, 1987.

[KH97] Holger Kienle and Urs Hölzle. `j2s`: A SUIF Java compiler. *Proceedings of the Second SUIF Compiler Workshop*, pages 8–15, August 1997. Also available as Technical Report TRCS97–16, Department of Computer Science, University of California Santa Barbara.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, third edition, 1997.

[Wil97a] Chris Wilson. Private email communication, August 1997.

[Wil97b] Chris Wilson. Private email communication, November 1997.

[Wil97c] Chris Wilson. Suif 2.0 tutorial. Given at the Second SUIF Compiler Workshop, August 1997.