TRAVERSAL, CASE ANALYSIS, AND LOWERING FOR C++ PROGRAM ANALYSIS

A Thesis

by

LUKE A. WAGNER

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2009

Major Subject: Computer Science

TRAVERSAL, CASE ANALYSIS, AND LOWERING FOR C++ PROGRAM ANALYSIS

A Thesis

by

LUKE A. WAGNER

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Bjarne Stroustrup
Committee Members,	Lawrence Rauchwerger
	Jaakko Järvi
	Gregory Berkolaiko
Head of Department,	Valerie Taylor

August 2009

Major Subject: Computer Science

ABSTRACT

Traversal, Case Analysis, and Lowering
for C++ Program Analysis. (August 2009)
Luke A. Wagner, B.S., Texas A&M University
Chair of Advisory Committee: Dr. Bjarne Stroustrup

To work effectively, programmers need tools to support their typical development activities, such as the creation, analysis, and transformation of source code. Analysis and transformation tools can be difficult to write for modern programming languages and, without a reusable framework, each tool must separately implement nontrivial algorithms like name lookup and type checking. This thesis describes an extension to one such framework, named Pivot, that focuses on programs written in C++. This extension, named Filter, assists the tool builder in traversal, case analysis, and lowering of the data structure representing C++ programs. Comparisons described in the thesis show a 2-4x code reduction when solving basic problems (e.g., searching for uses of a given declaration) using the extension and a performance overhead that drops below 2x for larger problems (e.g., checking C++ layout compatibility).

To my teammate, Jennifer

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

To work effectively, programmers need an environment that supports their typical development activities. The need for a reusable set of "sharp tools" was an early realization [1] that today is more true than ever. A large class of such tools are programs that take programs as input. These tools read and process source code (in a language like FORTRAN or C++) analogous to how a finance tool might read and process bank statements. Families of examples include:

- Tools that **check** source code:
  - **Coding standard conformance**: Coding standards establish a set of guidelines for naming, formatting, feature use, and design. Standard-checking tools [2–4] can be used to automatically detect and report guideline violations.
  - **Error detection**: These "bug finding" tools [5–7] look for errors such as memory leaks, uses of uninitialized data, invalid pointer dereference, and out-of-bounds array access. In general, while these tools do not *guarantee* the absence of errors, they are automatic and non-intrusive.
  - **Type checking**: Type systems also prevent undesirable program behaviors, often *with* guarantees, by requiring that the user annotate their code in a way that "convinces" a type-checking tool [8–10] that the undesirable behavior cannot occur.
  - **Verification**: For mission-critical applications, the programmer writes

---

The journal model is *IEEE Transactions on Automatic Control.*

not only code, but also a rigorous proof of correctness with respect to a specification. A verification tool [11–13] then reads all three to ensure the proof is valid.

- Tools that **generate** some type of output from source code:
  - **Compilation**: Compilers are the quintessential programming tool for converting source code to executable machine code.
  - **Search indexing**: Cross-indexing tools [14] collect and index the names of all program entities to allow the programmer to immediately search an entire project.
  - **Documentation**: Documentation generation tools [15, 16] automatically create diagrams, collect source code comments, and hyperlink related program entities based on information directly available in the source code.
  - **Boilerplate generation**: Code generation tools [17,18] can automatically write code that is long, tedious and error prone, but otherwise straightforward.
- Tools that **improve** source code:
  - **Optimization**: Using expensive algorithms, or taking advantage of domain-specific information, optimization tools [19–21] can transform source code into a faster, but equivalent, form.
  - **Parallelization**: To take advantage of modern multi-core processors, applications need to be written in a parallel style. Tools that perform automatic parallelization [22–24] examine applications written in a sequential style to find portions that can be run in parallel without changing the meaning of the program.

– **Code refactoring/renovation**: Over time, code tends to accumulate awkward designs and general cruftiness [25] resulting from years of piecemeal growth. Refactoring tools [26–28] expedite tedious maintenance activities aimed at improving code quality.

This thesis focuses on programs that analyze source code and thus the class of tools just described will henceforth be referred to as *program analyses*.

In general, and as seen in the above examples, program analyses differ dramatically in complexity and sophistication. For example, a tool that solves the problem "find all occurrences of the name 'Scott' in the source code" can simply view the source code as a string of characters and use the appropriate string-matching algorithm. On the other hand, problems like "find all variables with a given type" or "find all uses of a given variable" require understanding the *meaning* of the characters in the source code, i.e., applying the rules of the language to interpret the source code.

Even for simple languages, interpreting the source code may take more work than writing the entire rest of the program analysis. Moreover, for a modern complex language like C++, a correct interpretation requires several language experts and years of work [29]. The practical solution to this problem is to write this interpretation logic once, separately, so that it can be reused by all program analyses. In this way, program analyses that perform a simple task can be written simply, even if they are analyzing a complicated language.

The Pivot framework serves program analyses of C++ source code in the way just described. Specifically, Pivot provides reusable functionality which takes in raw source code and produces a data structure in memory that describes the interpreted meaning of the source code. This data structure is called the Internal Program Representation (IPR). A program analysis using Pivot can then work directly with the IPR *instead*

of the raw source, thereby avoiding the complex task of interpretation.

This thesis describes the Filter library, a new addition to the Pivot framework that assists the program analysis writer in using the IPR. The Filter library does this by providing reusable functionality that further decreases the work required to write a program analysis in the Pivot framework. Filter targets a smaller class of program analyses than Pivot. This allows Filter to hide and collapse (i.e., filter) information in the IPR that is irrelevant or overly-detailed with respect to the needs of this smaller class. By supporting the development of program analyses, the Filter library supports the overall goal of providing programmers with more, better, and sharper tools.

A. Filter library overview

The Filter library can be explained at a high level in terms of two functional goals and two stylistic goals. These four initial decisions determine, to a great degree, the rest of the design of Filter. This section explains *what* these goals are and leaves the *why* for the discussion in Chapter III. Also, these goals are introduced by metaphor and example, saving their realization in the Filter library for the tutorial in Chapter II.

1. Traversal and case analysis

The first functional goal is to simplify traversal and case analysis. The traversal problem appears in program analyses any time the algorithm being implemented requires finding all entities in —a part of— the program matching a criterion. Simple examples include finding all functions, to check their bodies, or finding all classes to build an inheritance graph. A more sophisticated traversal might involve searching all function bodies in a given namespace, looking for assignments to a given variable. Regarding the diversity of traversal needs, an additional subgoal of Filter is thus to

avoid locking the user into a fixed traversal strategy.

Somewhat dual to traversal, the case analysis problem appears in program analyses whenever it is necessary to consider every possible situation that may occur for a fixed piece of the program. As recognized by the following epigram, case analysis is a central programming activity.

> Programmers are not to be measured by their ingenuity and their logic but by the completeness of their case analysis.
>
> —Alan Perlis

Examples include analyzing an expression to determine if it can be evaluated at compile-time and syntax-directed type checking.

When performing case analysis, it is necessary to know *a priori* all the cases that need to be handled. This can be seen, for example, in the implementations of type systems throughout Pierce's *Types and Programming Languages* [30]. Thus, an additional subgoal of the Filter library is to make the cases that need to be handled *syntactically* evident, in the same way they are made evident by the Algebraic Data Types in Pierce's ML code.

Although traversal and case analysis are different activities, their usage is often interleaved. For example, an analysis might start with a traversal to find uses of a library of which it has extra semantic knowledge, and then switch to a case analysis of the surrounding code to identify optimizations that can be made based on this additional knowledge. Hence, the last subgoal is to not only simplify these two activities, but also their combined usage.

2. Lowering

The second functional goal is to perform lowering of the IPR. Here, the term "lowering" refers to transformations of the IPR that discard purely syntactic information. Of course, the syntactic/semantic distinction is entirely relative to the analysis; to a low-level optimization, anything above Three-Address Code [31] is syntactic, while to the analysis used by an IDE's automatic completion feature, only macros and `typedef` statements might be considered syntactic. For this reason, the IPR captures practically all of the syntactic information available after pre-processing.

To justify lowering, the Filter library targets a more specific group of analyses, described at the beginning of Chapter III, than the Pivot. For a small example of the type of lowering performed by the Filter, consider the following input:

```
namespace N { class C {}; }
N::C x;
```

Looking only at the representation of the variable declaration on the second line, the resulting tree of nodes produced by the IPR and Filter are shown in Figure 1. Even without knowing the exact meaning of the nodes in these trees, it is evident that, in addition to specifying *which* user-defined type was used, the IPR also represents *how* this type was named and found. The Filter library operates under the assumption that the program analysis is not interested in these details and collapses them into a single node summarizing the result. This allows the Filter user to write less code to achieve the same effect.

3. Lightweight view

The first stylistic goal is to provide a lightweight view of the IPR. A model for this design style are the iterators in the C++ Standard Template Library (STL) [32] and the Boost Iterator library [33]. For example, when using an iterator to examine the

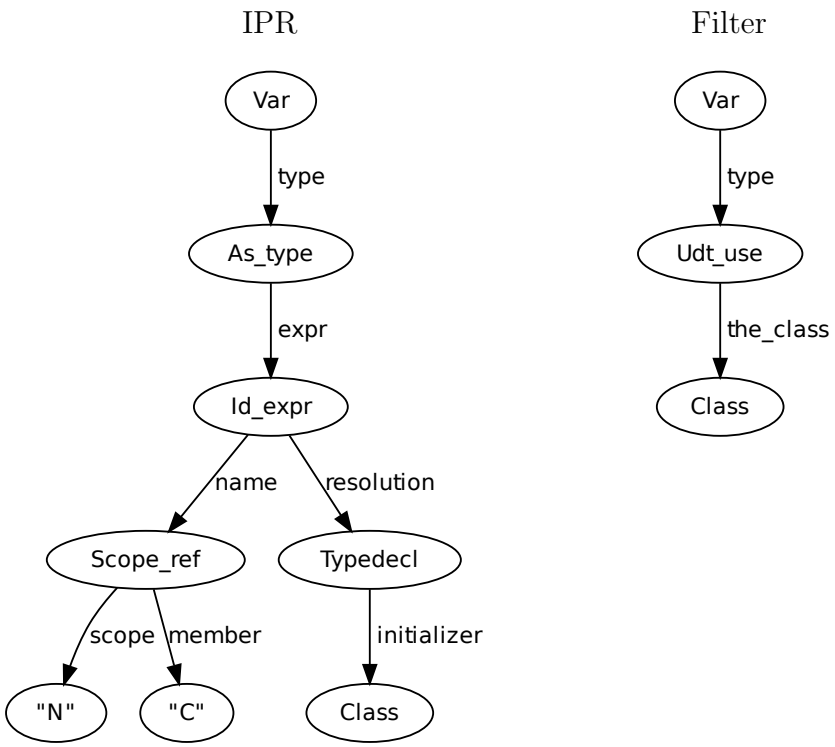

Fig. 1. IPR and Filter representations of a type use

contents of a list:

```
list<Big> ls = ...
for (list<Big>::iterator i = ls.begin(); i != ls.end(); ++i)
  ... use i
```

our understanding of the `iterator` object `i` is that it refers back into `ls`, which holds the `Big` objects. Thus, while copying `Big` objects—and lists thereof—may be expensive, operations like creating, copying, and destroying `iterator` objects are cheap. In this way, iterators provide a "lightweight view" of the underlying list.

Taking the example farther, let's say we wanted to iterate over only the `Big` objects satisfying a certain predicate. One solution would be to first create a copy of `ls` containing only the desired `Big` objects, and then iterate over this "filtered" list. This solution would be expensive up front, but would pay off if we iterated over the same filtered list many times. A "lightweight" solution would be to use the Boost `filter_iterator`:

```
typedef boost::filter_iterator<Pred, list<Big>::iterator> Iter_t;

list<Big> ls = ...
Iter_t beg(ls.begin(), ls.end()), end(ls.end(), ls.end());
for (Iter_t i = beg; i != end; ++i)
  ... use i
```

Here, `filter_iterator` performs filtering on the fly, silently skipping over `Big` objects that do not meet the given predicate `Pred`. This solution is lightweight because there is no large, up-front computation nor is there a large auxiliary data structure required.

In the same spirit as these two examples, the Filter library refers into the IPR and builds its view of the IPR on the fly, without any significant auxiliary data structures. While this goal poses a challenge—which is met—for maintaining an acceptable performance overhead, it allows the Filter to be used with the IPR on an as-needed basis, without penalty for switching back and forth. Alternatively, if a separate data structure was desired, perhaps for performance reasons, the Filter

could be used in the algorithm which created this data structure.

4. Inversion of Control considered harmful

The second stylistic goal is to avoid imposing any Inversion of Control (IoC) [34] on user code. IoC refers to a library design whereby, to use the library, the user must provide functions *to* the library that get called *by* the library. When used in the design of software frameworks, IoC has been called the "Hollywood Principle" ("Don't call us, we'll call you") [35, 36].

Essentially, IoC changes the classic library use:

```
int i = library_query(); /* perform query and receive result */
... use i
```

into:

```
void query_result_function(int i) /* step 2: receive result */
{
  ... use i
}
library_query(&query_result_function); /* step 1: perform query */
```

While the second solution may seem like an exercise in obfuscation, in some problem domains (as described in the above references) the use of IoC has been recognized to promote software engineering ideals like modularity and encapsulation.

Central to any program analysis using a framework like Pivot is a *graph* (a collection of *nodes* and *edges* between nodes) that represents the interpreted meaning of the program. Because a program may be composed of many different types of entities, this graph must contain a *heterogeneous* set of nodes, i.e., nodes of different types. This leads to the *type discovery* problem: after following an edge to a new node, how does the analysis determine the new node's type?

A common solution in object-oriented frameworks [37–40] is the Visitor pattern [41], which uses IoC to reveal a node's type to the analysis in a "safe" way. For

reasons presented on pg. 81 and measured in Chapter IV, the Filter library avoids the Visitor pattern because of its negative effects on the resulting user code. In general, while the recursion induced by IoC seems acceptable or even desirable, based on the mathematical nature of recursion, it can also lead to an unnecessarily complex solution to simple problems. To quote an observation made by Dijkstra on the subject [42]:

> I found a confirmation in the temptation—I myself yielded to it once, in a moment of weakness!—to define the semantics of
>
> "`while B do S od`"
>
> as that of the call
>
> "`while(B, S)`"
>
> of the recursive procedure
>
> "`proc while(B, S): if B then S; while(B, S) fi corp`"
>
> but is not that cracking an egg with a sledgehammer?

Indeed, even in functional programming, where recursion is the general *modus operandi*, the type discovery problem for Algebraic Data Types is solved by using special IoC-free `match/case` statements. (As this second reference indicates, in addition to iterators, the design of the Filter library also resembles aspects of Algebraic Data Types.)

B. Thesis overview

The main work described by this thesis is the design, implementation, and evaluation of the Filter library. Breaking this into independent problems, the contributions of this thesis are:

- the design of a lowered view of the IPR suitable for high-level semantic analysis (pg. 59);
- an integrated solution to the case analysis and traversal problems described above (pg. 48, pg. 125, pg. 128, pg. 146);
- an implementation scheme for efficient, lightweight iteration over dynamically transformed and composed ranges (pg. 88); and
- an evaluation of the library (Chapter IV).

To evaluate Filter, a comparison is made to the "traditional approach" by solving six problems and comparing the solutions' code complexity and performance. The results show that Filter allows a consistent 2-4x reduction in Logical Lines of Code (LLOC) and produces an average 2-4x slowdown on small, synthetic problems. On larger, more realistic problems, this slowdown drops below 2x, and when Filter allows the user to write more precise traversals, the result can be up to 230x faster. Thus, we can conclude that, for many analyses, using Filter will be a win-win situation.

The rest of the thesis is organized as follows. Chapter II is a tutorial of the Filter library, providing the necessary Pivot background. Chapter III describes the reasoning behind the Filter design and related implementation details. Chapter IV explains the evaluation mentioned above and goes over the results. Chapter V describes related libraries and approaches. The conclusion revisits the design decisions listed on pg. 4 and describes future work. Finally, the attached appendix provides

input listings for all the tests described in Chapter IV as well as the implementations of a few large functions used in the solutions to the test described on pg. 134.

CHAPTER II

TUTORIAL

This tutorial introduces the Pivot framework using the Filter library. The tutorial begins with an overview of what the Pivot framework is and what it provides. Next, the tutorial explains how to build a basic Pivot application. Once these basics have been established, the tutorial gives a tour of Pivot by demonstrating how to write a small "semantic grep."

A. Motivation

Pivot is a framework for analyzing and transforming C++ programs. This framework consists of tools and libraries that allow the user to work with C++ code at the abstract syntax level. This abstract syntax representation preserves almost all of the information available about the source code, and thus is higher-level than the internal representations used by compiler back-end frameworks like LLVM [43], Phoenix [44], Open64 [45], and GCC's GIMPLE and RTL phases [46]. However, this abstract syntax representation also contains more information than the raw output of a parser by storing the results of complicated language rules like name lookup, overload resolution, and template instantiation. Thus, in a traditional compilation pipeline, Pivot fits in the middle-end, after front-end syntax analysis and before back-end optimization and code generation.

To see an example of an analysis that would benefit from the Pivot framework, consider trying to replace the following sequential loop with a version that runs each iteration in parallel:

```
void mult_copy(vector<int> &v1, vector<int> &v2, T x)
{
  vector<int>::iterator i1 = v1.begin(), i2 = v2.begin();
```

```
  for (; i1 != v1.end() && i2 != v2.end(); ++i1, ++i2)
    *i2 = x * *i1;
}
```

In order to justify this parallelization, a back-end would need to know that the range of elements iterated over by `i1` did not overlap with that of `i2`. However, these iterators will be compiled down to raw pointers, and thus the back-end will have to do a full alias analysis. In doing so, the analysis will find that `i1` and `i2` receive their values from the fields of the records pointed to by `v1` and `v2`. Without an expensive field-sensitive pointer analysis [47], the optimizer will probably be forced to take the conservative route and give up.

However, with full source information, we can see that `v1` and `v2` both have a type that is an instantiation of the standard C++ `vector`. Using this extra semantic information, we know that no two vectors' `[begin,end)` ranges overlap and thus we can immediately see that the ranges of `i1` and `i2` either overlap completely (if `v1` and `v2` are aliased) or do not overlap at all. In either case, the parallelization is valid. With the introduction of concepts in C++0x [48], we can extend this semantic knowledge to arbitrary user-defined containers that model standard C++ concepts, e.g., doubly-linked lists. However, to do any of this requires preserving source code information like nominal type, instantiation relationships, and modeling relationships.

This example illustrates just one family of analyses that could benefit from a typed abstract syntax representation of C++. Other examples include coding guideline conformance, automatic syntax completion, and tools for program understanding. Often, such analyses are based on parsers and support simple C++ usage, but break down when sophisticated C++ features are used. For example, consider the type of `x` in the following code which uses C++ template meta-programming techniques:

```
template <class T> struct pairify { typedef pair<T,T> ret; };
template <class> struct const_first {};
```

```
template <class T, class U> struct const_first<pair<T,U> > {
  typedef pair<const T,U> ret;
};
const_first<pairify<int>::ret>::ret x;
```

Here, determining that `x`'s type is `pair<const int, int>` involves significant work. In Pivot, given the abstract syntax node representing `x`, it is possible to either go directly to its type, or recover the intermediate templates that were used to generate this type.

B. Overview

Central to the Pivot framework is a C++ data structure called the Internal Program Representation (IPR). The IPR represents the program's typed abstract syntax in memory, so that it can be analyzed and manipulated. Most of the tools in Pivot are concerned with producing, consuming, and transforming the IPR. Similarly, most of the libraries provided aim to simplify common usage of the IPR. The primary goals of the IPR are:

1. to be extensible, by allowing future standard C++ features and existing non-standard C++ extensions to be added to the IPR without disrupting existing code;
2. to be efficient in both space and time so as to be able to represent entire programs in memory, thereby allowing whole-program analysis; and
3. to be independent of the front-end that was used to generate the IPR. Currently, both EDG [29] and GCC [49] front-ends are used.

Essentially, the IPR is a labeled directed graph whose nodes represent the atomic pieces of a C++ program and whose edges represent the relationships between these pieces. A subset of these edges form the classic typed abstract syntax tree that is

often used to describe the nesting of program elements [31]. The full graph can be understood as this basic tree augmented with edges to provide useful or necessary information, e.g. the result of name lookup and type evaluation.

Each node in the IPR has a C++ interface which enumerates the node's properties and out-edges. All node types are defined in the public header file `pivot/include/ipr/interface` which also includes documentation for each type. The abstract nature of some IPR nodes can make the IPR difficult to learn from this documentation alone, so a better way is to visualize graphs generated from small examples. To help do this, the Pivot includes the `ipr2dot` tool, located in `pivot/bin`. This tool describes the IPR graph in DOT, a graph language which can be rendered in many formats by the GraphViz package [50]. Using `ipr2dot` and the GraphViz `dot` command, the following Unix pipeline will render the C++ code in `test.cpp` as a PNG image:

```
ipr2dot test.cpp | dot -Tpng > test.png
```

To see an example, we start with the following code:

```
int main()
{
  int a = 1;
  return a + 2 - 3;
}
```

Even with this small program, the output of `ipr2dot` can be overwhelming at first, so we will tip-toe in by looking at graphs with nodes and edges removed. We begin with Figure 2, which shows only the IPR nodes and edges that map directly to the source. In this graph, we almost have a classical abstract syntax tree; only the unification of the identifier "a" breaks the tree property.

In Figure 2, the labels of nodes are the names of IPR types (except for the quoted strings) and the labels of edges are the operations of the source's IPR type that were used to get to the destination node. For example the member function `ipr::Return::value` was called on the node labeled "Return" to get the node labeled "Minus."

Note that nodes represent logical entities in the code and are not simply a record of which C++ grammar rules were used to parse the tokens.

The root of every IPR graph is always a node of type `ipr::Unit`. A `Unit` represents either a single translation unit or multiple translation units that have been merged together. A *translation unit*, as defined by Section 2.1 of the C++ standard [32], roughly means a source file after preprocessing. Leveraging the partial compilation model of C++, an IPR graph does not necessarily have to contain an entire program. However, if all the units of a program are merged together, it may.

From a `Unit`, it is possible to begin traversing the program by calling `Unit::get_global_scope` to get a node representing the global namespace. A `Unit` serves other purposes too, such as providing a central point of resource management. Below the `Unit`, the meaning of each node is fairly self-explanatory except for the node labeled "Id_expr." The `ipr::Id_expr` type represents the use of a name, like a variable name, in an expression. In the example, we can see the use of the name "a." However, with only the information in the above DAG, we do not know what "a" refers to.

To answer this question, we move on to Figure 3, where we show the IPR for the same code, except with fewer auxiliary nodes and edges elided. Two types of additional information are represented in this graph: types and resolution. We can see that the question of the resolution of "a" is answered by the `Id_expr::resolution` member function. Additionally, all nodes are given their C++ type as assigned by the C++ typing rules. Here we can see how unification (e.g., the sharing of "Int") can save considerable memory since many nodes will have the same type.

As a general rule, the IPR provides edges like `resolution` and `type` above when recovering the target of the edge would involve understanding complicated C++ rules. In this way, Pivot allows its users to avoid much of the difficulty associated with static analysis of C++ over, say, C.

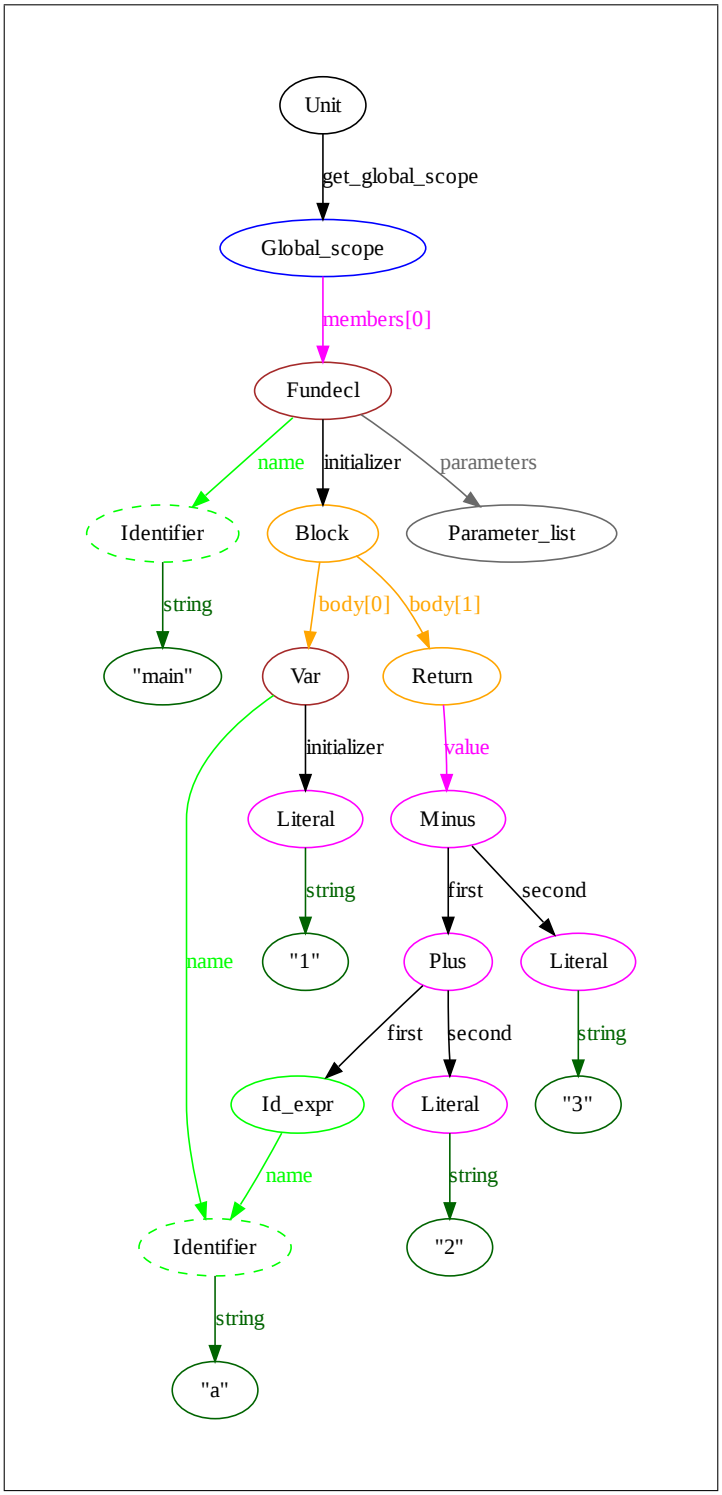

Fig. 2. Essential nodes and edges of `ipr2dot` output

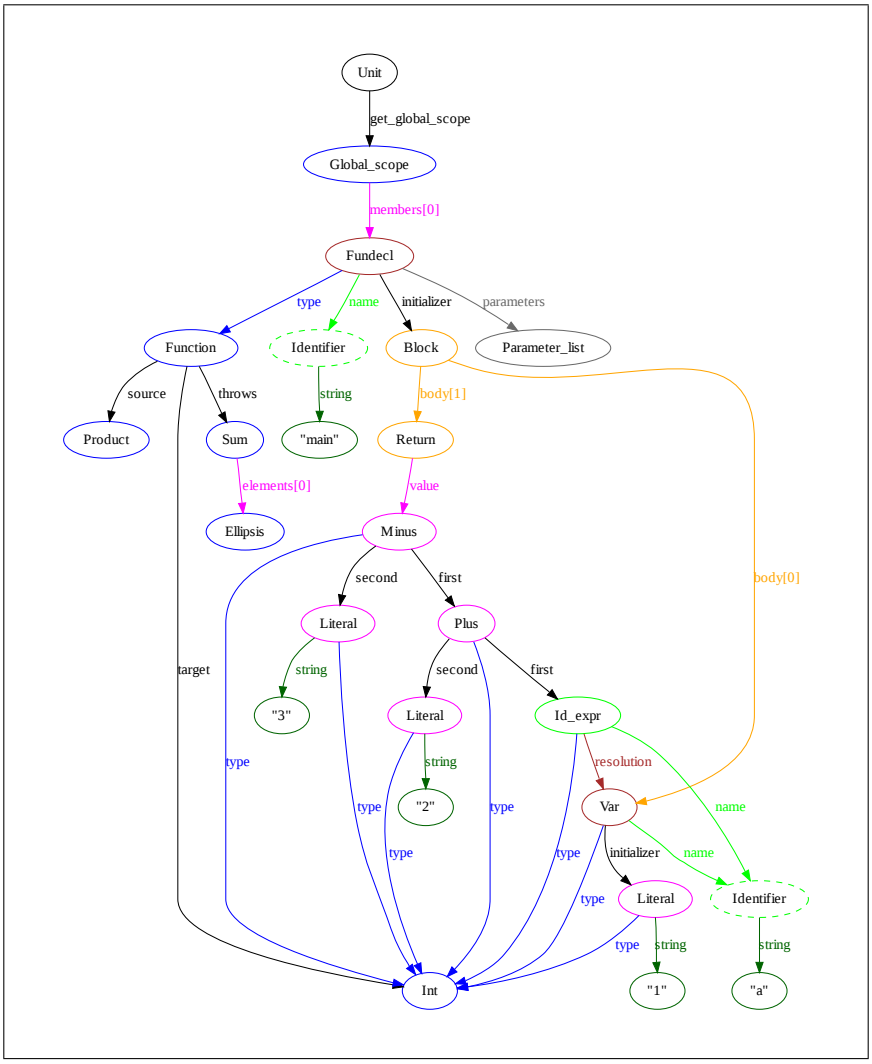

Fig. 3. `ipr2dot` output with auxiliary nodes and edges removed

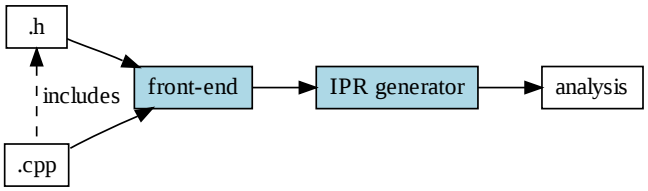

Fig. 4. The Pivot pipeline

C. "Hello, Unit!"

As mentioned earlier, an IPR graph is rooted and contained in an object of type `ipr::Unit`. Thus, the first step to analyze a piece of code is to acquire the `Unit` of that code. The Pivot framework allows `Unit` objects to be created from various sources by providing a separate interface, in the `ipr::impl` namespace, for generating IPR nodes. Additionally, a few ready-to-use generators are provided.

The most direct way to generate a `Unit` is by transforming the output of a C++ compiler front-end. Pivot officially contains one such implementation, using the EDG front-end [29], but there is also an experimental front-end based on GCC. In either case, the pipeline followed by the analyzed code is shown in Figure 4. From this figure, we can see that the front-end takes in a source file, and any other files included by pre-processing, and passes the results to Pivot code that generates a `Unit` which is passed to user's analysis.

The official entry point of an analysis which obtains a `Unit` in this manner is a function `ipr_main` with the following signature:

```
int ipr_main(const ipr::Unit &unit, int argc, char **argv, int sep)
```

where sep is an integer in the range `[0, argc]` that indicates the position, in `argv`, of an optional `--pivotargs` flag. This flag separates command-line arguments read by the front-end from arguments intended for the analysis. Thus, `--pivotargs` is like an

escape character that tells the front-end to ignore all subsequent arguments.

Using `ipr_main`, we can write a minimal first Pivot application:

```
#include <ipr/interface>
#include <iostream>
int ipr_main(const ipr::Unit &u, int argc, char **argv, int sep)
{
  std::cout << "Hello, Unit with its ";
  std::cout << u.get_global_scope().members().size();
  std::cout << " global namespace declarations!\n";
}
```

To compile this file, the include path must be set to the `pivot/include` directory so that `ipr/interface` is found:

```
g++ -c -I PIVOT_INSTALL_FOLDER/include analysis.cpp
```

To generate an executable, the object file must be linked with two libraries: `ipr` and `edg`, both of which are found in the `pivot/lib` folder:

```
g++ -L PIVOT_FOLDER/lib analysis.o -ledg -lipr -o analysis
```

Note that, for some linkers, the order in which libraries are listed is important. The `edg` library depends on definitions in the `ipr` library, so for GCC, `-lipr` must be after `-ledg`.

Finally, the executable can be run with the standard compiler command-line arguments:

```
> echo "int a; void b();" > test.cpp
> analysis test.cpp
Hello, Unit with its 2 global namespace declarations!
```

Here, `argc` will be `2` and `sep` will be 2. If we make the following call:

```
> analysis test.cpp -DNDEBUG --pivotargs foo bar
```

then `argc` will be `6` and `sep` will be `3`.

D. The Filter library

Given an `ipr::Unit`, we can call its `get_global_scope` member to obtain a node representing the global namespace, and then go on our merry way traversing the graph. Before we get very far, however, we are faced with the practical question encountered below:

```
int ipr_main(const ipr::Unit &unit, int, char **, int)
{
  const ipr::Namespace &gns = unit.get_global_scope();
  const ipr::Sequence<ipr::Decl> &glo_decls = gns.members();
  for (int i = 0, end = glo_decls.size(); i != end; ++i) {
    const ipr::Decl &d = glo_decls[i];
    // how do we find out which specific type of declaration d is?
  }
}
```

Here, we have a reference to an `ipr::Decl`, but `ipr::Decl` is just a base class shared by all the nodes that can appear in a namespace. While `Decl` has some useful members common to all declarations, like `Decl::name`, to do anything useful with the node, we will need a more specific interface. For example, if we want to look at the bodies of functions, we will need a reference to an `ipr::Fundecl` so that we can use its `Fundecl::initializer` member (shown in Figure 3).

This type of problem is not specific to global declarations; throughout the IPR, there are many situations where an edge can point to one of many different types of nodes. Examples include the branches of an `if` statement, the operands of an addition, the members of a class, and the parameter types of a function type. In general, this question will have to be answered whenever there is a tree of heterogeneous nodes. As described on pg. 81, there are many ways to solve this problem, each with different trade-offs. In this section, and for the rest of the tutorial, we will focus on only one solution: the one provided by the Filter library.

Table I. logical correspondence between iterators and Filter types

	C++ STL	IPR + Filter
Data structure	container (e.g. `std::set`)	Program (`ipr::Unit`)
View and traversal	iterator	Filter nodes

The Filter library provides a lightweight view of the IPR that makes navigation among nodes easier. Filter is considered *lightweight* because it has no persistent data structures. Thus, there is no upfront cost to using Filter and it may be used and discarded on an as-needed basis. Filter is said to provide a *view* of the IPR because the library does not hold any information, it just presents a transformed version the information already in the IPR.

A familiar example of the relationship between the IPR and the Filter library is the relationship between a C++ Standard Template Library (STL) container and its iterators. The iterators are lightweight objects and provide a way of viewing the internal nodes of the container. Similarly, Filter provides lightweight objects for looking at the IPR nodes inside an `ipr::Unit`. This metaphor is summarized in Table I.

Returning to the example at the beginning of the section, we can use the Filter library to write:

```
int ipr_main(const ipr::Unit &unit, int, char **, int)
{
  Env env(unit);
  Namespace gns(env, unit.get_global_scope());
  for (ns_ctx::Range ds = gns.members(); !ds.empty(); ++ds.first) {
    ns_ctx::Variant d = *ds.first;
    if (d.which() == ns_ctx::glo_func_e) {
      Glo_func gf = d.get<Glo_func>();
      // ...
    }
```

```
  }
}
```

Thus, the question "which type of `Decl` is `d`?" is answered by the Filter library's `ns_ctx::Variant` type using its `which` member. This approach is based on the common "type switch" pattern that can be found in object-oriented programming, starting with the Simula 67 `inspect` statement [51], and functional programming, for example, using the ML `match` operation [52].

We now consider the above code in more detail. The first statement creates an object of type `filter::Env` ("`using namespace filter`" is assumed for this and subsequent code samples). This is an "environment object" that is needed to construct any other Filter type, such as `gns` on the next line. The environment is used to store a reference to the `ipr::Unit`, hold configuration options, and provide caching for common IPR queries. To construct a `filter::Namespace`, we need to pass it an `ipr::Namespace` which we obtain from `get_global_scope`. In general, each Filter node type has a corresponding IPR type which is needed to explicitly construct the Filter node. These two steps—creating an environment and constructing an initial Filter node—are all that is needed to get started with the Filter library.

Given an initial Filter object (`gns`), we can use its members to obtain other Filter nodes which represent different underlying IPR nodes. In this manner, we can navigate through the IPR. We do this in the above code by using `Namespace::members` to get a `Range`, then a `Variant`, and then a `Glo_func`. To stop using the Filter library, we simply extract the underlying IPR node of any Filter node we are using. For example, continuing the above code at the "`// ...`", we can write:

```
const ipr::Fundecl &fd = gf.ipr();
```

Once the underlying IPR node has been extracted, the Filter nodes may be used further or ignored and destroyed.

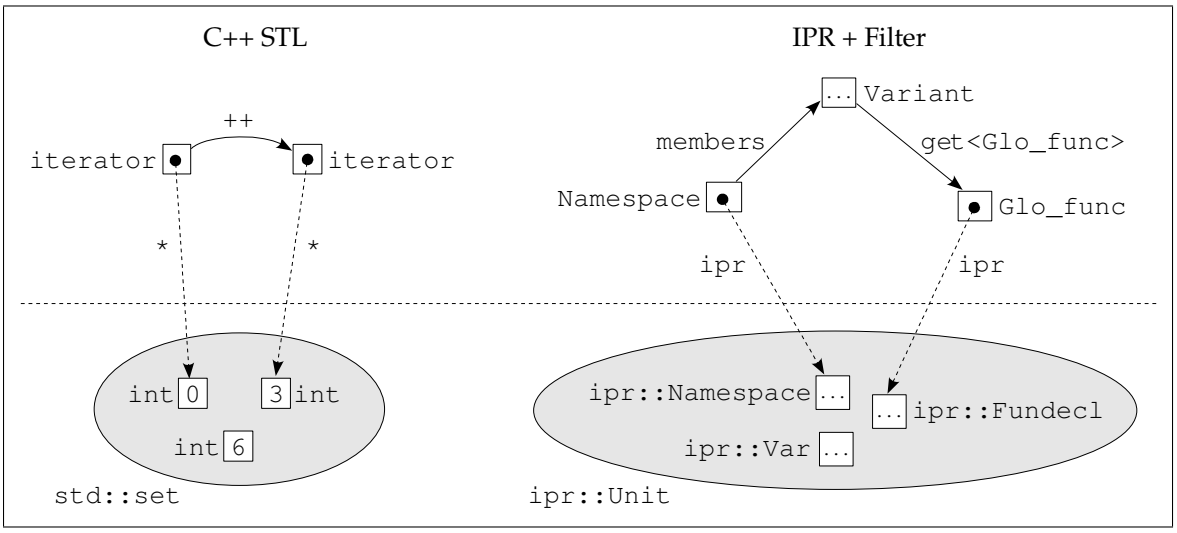

Fig. 5. A more concrete correspondence between iterators and Filter types

With this example code, we can return to the metaphor summarized in Table I and look at the more concrete correspondence given in Figure 5. The left and right diagrams show the "data structure" below the dotted line and the "view and traversal" above. On the left-hand side, since the elements are homogeneous and the order is fixed, there is only one iterator type and traversal operation used. On the right-hand side, the elements are heterogeneous, so an intermediate `Variant` object is used to moderate the branching between the possible types. Also, since the view is a tree, several different operations are available at each node to get to other nodes. In both cases, there is an operation to get to the underlying element.

From this type of usage, we can view the Filter library as a highway for travelling around the IPR. In the sample code, we exited the highway immediately after we entered, but, as we will see in subsequent examples, we can go much farther. One important thing to remember, though, is that the environment object is being implicitly passed to each new Filter node. Thus, the environment object passed to the initial Filter node *must* be kept alive for as long as any Filter node, obtained directly

or indirectly from the first, is alive.

From a performance perspective, the expensive operations of the Filter library are performed during the construction of fresh `Variant` objects (which is done indirectly by `Range` construction and iterator increment). Thus, iterator dereference, `Variant ::which`, `Variant::get`, all copy, and all assignment operations are cheap and can be used freely. Chapters III and IV go into more detail about the implementation and performance of these operations.

One last thing to point out in the code above is the `ns_ctx::` prefix in front of the `Range` and `Variant` types. This prefix stands for the *context* of these types, which, in this case, is the "namespace context." A context represents where the choice of nodes occurred and implies the set of possibilities presented by the `Variant`. Without context information, there would be only one `Variant` which would necessarily contain a choice of every possible node, and thus the user would have to decide for themselves which of the cases actually needed to be handled. As we will see, this is not always apparent, hence the presence of context in types is an important feature of the Filter library.

Currently, the Filter library identifies 5 contexts: namespaces, user-defined aggregates (viz. classes and unions), function bodies, expressions, and types. The first three of these have a high degree of overlap, while the last two are almost disjoint from the rest and each other. Like `ns_ctx`, each other context has its own namespace (`uda_ctx`, `func_ctx`, `expr_ctx`, and `type_ctx`) which contains a `Variant`, `Which` enumeration type (returned by `Variant::which`), `Iter`, and `Range`.

[NOTE: when the Pivot project structure stabilizes for release, an additional paragraph should be added here explaining how to include and link the Filter library.]

E. Semantic grep

Moving past the IPR and Filter basics, we shall now look at how the variety of different C++ elements are represented in Pivot (using the Filter library) and how Pivot assists the user in writing analyses. To illustrate the concepts covered, we guide the discussion with an extended example, which will be developed throughout the next five sections. In walking through the implementation, the goal is not to build an in-depth understanding of every piece covered, but to give an overall picture of how the pieces fit together. Thus, the discussion will move quickly, saving a topical discussion for subsequent chapters.

The extended example we will use is a tool for searching source code for uses and declarations of variables that meet a few conditions. A classic tool for this job is the standard Unix `grep` command, which treats source code as text and allows regular expression-based matching. However, `grep` is limited to describing the lexical properties of a variable and its surroundings. The tool we shall build uses the semantics of the text, viz. that it is C++ code, to allow the programmer to specify additional constraints on occurrences of variables such as their type and whether they are declarations or uses. Hence, we are building a kind of "semantic grep." In addition to a real tool with the same name [53], this example represents a class of program understanding tools as well [54–58].

1. Skeleton

To keep the example short, we focus on only a few properties, and only for global and member variables. The intended usage of the tool is roughly as follows:

```
sgrep FILE --pivotargs NAME [-type NAME] [-only [use|decl]]
```

The first argument is the C++ source file, the second is the argument separator that was mentioned on pg. 20, the third is a regular expression for the variable name, and the last two are optional constraints. The `-type` flag is a regular expression to match the variable's type. The `-only` flag allows only uses or declarations to be found.

To go directly to the use of Filter, we will assume that we have already written the boilerplate code to parse the incoming command-line argument vector. This code takes the form of a function `parse`:

```
bool parse(int argc, char **argv);
```

which populates a global structure with the interpreted command-line options:

```
struct Cmdline_args {
  Pattern name;
  bool has_type;
  Pattern type;
  bool find_decls, find_uses;
} cmdline;
```

We also assume we already have a class that implements regular-expression matching:

```
class Pattern {
  public:
    bool match(const string &) const;
};
```

Using this, we can write the skeleton for our `sgrep` application as follows, putting the body of work in `analyze`:

```
int ipr_main(const ipr::Unit &u, int argc, char **argv, int sep)
{
  if (sep == argc || !parse(argc - (sep + 1), argv + (sep + 1)))
    return -1;
  Env env(u);
  Namespace gns(env, u.get_global_scope());
  analyze(gns);
  return 0;
}
```

The offset added to `argc` and `argv` is necessary to skip all the arguments up to and including the `--pivotargs` flag.

Instead of working directly towards the final goal, we will grow the solution from sub-problems. This will allow us to see the various challenges in analyzing C++. Hence, the reader should suspend judgment if the code used to solve a sub-problem seems ungainly in the context of the larger problem; it is, and it will probably be fixed.

2. Collecting global declarations

We can now begin to implement `analyze` by breaking it into the following steps:

```
void analyze(Namespace gns)
{
  vector<const ipr::Decl *> decls;
  collect_decls(gns, decls); // find matches in ‘gns’, insert into ‘decls’
  if (cmdline.find_decls)
    print_decls(decls);
  if (cmdline.find_uses)
    print_uses(gns, decls); // find and print matches in ‘gns’ using ‘decls’
}
```

Thus, the first sub-problem is to collect a list of global variables that match the criteria in `cmdline`. This list is then used to print declarations and efficiently find uses.

Focusing now on `collect_decls`, we start by writing the following recursive function:

```
void collect_decls(Namespace ns, vector<const ipr::Decl *> &decls)
{
  for (ns_ctx::Range r = ns.members(); !r.empty(); ++r.first)
    switch (r.first->which()) {
      case ns_ctx::namespace_e:
        collect_decls(r.first->get<Namespace>(), decls); // recurse
        break;
      case ns_ctx::glo_var_e:
        if (match(r.first->get<Glo_var>()))
          decls.push_back(&r.first->get<Glo_var>().ipr());
        break;
    }
}
```

Here, we look for global variables which match the user's pattern, using the `match` function we will write shortly. Since global variables can be found in nested namespaces, the function must recursively examine all namespaces found as well.

We now consider this code in more detail. As seen before, we use an `ns_ctx ::Range` to iterate over the contents of namespaces. This type is simply a pair of `ns_ctx::Iter` objects with a few convenience functions, like `Range::empty`, which returns `(first == last)`. The `ns_ctx::Iter` type mostly models the Standard C++ bi-directional iterator concept [32], where the "element type" of `Iter` is a `Variant` in the same context. The difference with Standard C++ iterators, however, is that the pointer returned by the dereference operators (`*` and `->`) is only valid until the next increment/decrement operation. Thus, the following code:

```
vector<ns_ctx::Variant *> vs;
for (ns_ctx::Range r = gns.members(); !r.empty(); ++r.first)
  vs.push_back(&*r.first);
```

collects a list of dangling pointers, as opposed to the behavior of a standards-conforming C++ iterator. Thus, to hold onto the current element of an iterator, a copy should be made (which, as already mentioned, is an inexpensive operation):

```
vector<ns_ctx::Variant> vs;
for (ns_ctx::Range r = gns.members(); !r.empty(); ++r.first)
  vs.push_back(*r.first);
```

In `collect_decls`, we switch on the return value of `ns_ctx::Variant::get`. The type of the value returned by this member function is an enumeration `ns_ctx:: Members` with the following enumerators:

```
namespace_e,
glo_var_e, class_var_e,                       /* variables */
class_e, union_e, enum_e,                     /* user-defined types */
glo_func_e, class_func_e, mem_func_e,         /* functions */
class_family_e, union_family_e, func_family_e  /* templates */
```

This list contains everything that can be found in the context of a namespace including some non-obvious things, like out-of-body member function definitions and static member variable definitions (class variables). The three "families" refer to C++ templates (this viewpoint will be described more on pg. 59). Associated with every enumerator `foo_bar_e` is a Filter wrapper type `Foo_bar`. The `Variant::get<T>` function template can be instantiated with only this set of types. Hence, the following code is incorrect (and will assert in debug builds):

```
ns_ctx::Variant nsv = ...
if (nsv.which() == ns_ctx::glo_var_e) {
  Class_var cv = nsv.get<Class_var>();
  ...
}
```

We now return to the task of implementing the `match` function called in `collect_decls`. The approach we will use is to convert the name and type of a variable to strings, and then perform matching on these strings. To get the name and type of a global variable, we use `filter::Glo_var`'s underlying IPR node, `ipr::Var`:

```
bool match(Glo_var gv)
{
  const ipr::Var &iv = gv.ipr();
  const ipr::Name &name = iv.name();
  const ipr::Type &type = iv.type();
  // ...
}
```

Since names and types can be complex expressions, we use the Pivot Pretty-printer library to give us their textual representation. Describing the Pretty-printer library is beyond the scope of this library, so we will assume that the following stream insertion operators have been defined:

```
std::ostream &operator<<(std::ostream &, const ipr::Name &); // print name
std::ostream &operator<<(std::ostream &, const ipr::Type &); // print type
```

With these, we can continue writing `match` at the "`// ...`" above:

```
std::stringstream ss;
```

```
ss << name;
if (!cmdline.name.match(ss.str()))
  return false;
if (!cmdline.has_type)
  return true;
ss.str("");
ss << type;
return cmdline.type.match(ss.str());
```

This completes the implementation of `collect_decls` for the case of global variables.

3. Collecting member variables

We now extend our analysis to match member variables. To do this, we can use an approach similar to that of global variables, this time searching through the bodies of classes and unions instead of namespaces. By design, classes and unions in C++ are very similar, including what members they may have and where they may be defined. To reflect this, the Filter library has a common base class, `Uda` (UDA is an acronym for User Defined Aggregate), that is derived by the Filter `Class` and `Union` types. Using `Uda`, we can write a new function `uda_collect_decls`:

```
void uda_collect_decls(Uda uda, vector<const ipr::Decl *> &decls)
{
  if (!uda.has_def()) // declared, but not defined
    return;
  for (uda_ctx::Range r = uda.def_members(); !r.empty(); ++r.first)
    switch (r.first->which()) {
      case uda_ctx::class_e:
      case uda_ctx::union_e:
        uda_collect_decls(get_uda(*r.first), decls); // recurse into class/union
        break;
      case uda_ctx::mem_var_e:
        if (match(r.first->get<Mem_var>()))
          decls.push_back(&r.first->get<Mem_var>().ipr());
        break;
    }
}
```

This code does a few new things. First, as opposed to namespaces, where there is no distinction between declarations and definitions, it is possible to find a declaration of a class and union with no definition. Thus, `Uda::has_def` must be checked before accessing parts of the definition. The Filter library distinguishes parts of the definition by the prefix `def_` as a reminder that calling `def_x` assumes there is a definition.

Because classes and unions are so similar, we often want to handle both their cases at once, as in the above code. The `Uda` base class helps us do this, but due to implementation limitations, `Variant` does not allow us to write "`Uda u = v.get< Uda>()`". To get around this, the Filter library provides the `get_uda` function, used above, which achieves the same thing. Similar functions, `get_func` and `get_var`, are provided for other reusable bases which will soon be used.

The only remaining problem is that `match`, as written on pg. 29, takes a `Glo_var` and above we are passing a `Mem_var`. To fix this problem, we can change `match` to take a `filter::Var`, which is a base class inherited by all Filter variable types. Strangely, the underlying IPR node of a `filter::Var` is not an `ipr::Var`, but the more general `ipr::Decl`. This is no matter, though, as `ipr::Decl` provides the required `name` and `type` members, making the change to `match` trivial:

```
bool match(filter::Var v)
{
  const ipr::Decl &d = v.ipr();
  ... same as before
}
```

Notice that in passing by value, we are slicing Filter nodes at the `match` call-sites in `collect_decls` and `uda_collect_decls`. While slicing is generally a dangerous operation, the Filter library takes care to ensure that no problems arise from slicing Filter nodes, and the above code is fine.

Having written `uda_collect_decls`, we now hook it into `collect_decls` by adding the following cases to the main switch statement:

```
case ns_ctx::class_e: case ns_ctx::union_e:
  uda_collect_decls(get_uda(*r.first), decls);
  break;
```

With these changes, we can successfully find even deeply-nested member variables like:

```
namespace A { namespace B { class A { class B { int x; }; }; } }
```

However, trickier cases remain, namely, local classes:

```
class A { void foo() { class B { int x; }; } };
```

In fact, looking at `func_ctx::Members`, we can see that global variable declarations can also appear in function bodies. Thus, we must also find and examine functions. While these may seem like corner cases that could be ignored for our semantic grep example, we pursue it here as it leads to a more general point. Furthermore, as a matter of correctness, some analyses require that *all* functions (and therefore classes) are found.

Continuing the pattern of code we have been writing, we write a new function `func_collect_decls` that gets called whenever we find a function in a namespace or UDA. Like UDAs and variables, there are several Filter types for functions (`Glo_func` , `Class_func`, and `Mem_func`) and all share a common base class `Func`. Using this, we can add the following cases to `collect_decls` and `uda_collect_decls` (with the context prefixes changed accordingly):

```
case ns_ctx::glo_func_e:
case ns_ctx::class_func_e:
case ns_ctx::mem_func_e:
  func_collect_decls(get_func(*r.first), decls);
  break;
```

Moving on to the implementation of `func_collect_decls`, like namespaces and UDAs, functions have a context associated with the discovery of nodes in vari-

ants. However, if we were to use the same "switch and recurse" technique as in the last two collection functions, `func_collect_decls` would require a lot of—seemingly boilerplate—code. For example, consider finding the class buried in the following input:

```
void foo() {
  if (true)
    switch (1) {
      case 1: while (false) {
          class A { int x; };
        }
    }
}
```

Conceptually, this is no different than if `A` were nested inside three namespaces or classes: the algorithm is just tree traversal. Practically, however, traversing this tree requires handling every type of node in the tree, including curly-brace blocks, `if`, `else`, `while`, `for`, `do`, `switch`, `try`, and `catch`. For our analysis, this means littering our main switch statement with cases like:

```
case func_ctx::if_e:
  func_collect_decls(v.get<If>().then_branch(), decls);
  func_collect_decls(v.get<If>().else_branch(), decls);
  break;
```

with only one case where we find what we are actually looking for, classes. This problem will be further exacerbated when examining the expression and type contexts, where there are even more node types to consider. If we take a step back, we can see that any analysis that wanted to look, more or less, over the whole program would have to write the same boilerplate code.

The Filter library allows this mess to be factored out by providing functions to retrieve all the children of a given Filter node without knowing that node's exact type. Specifically, for each context `X`, each `X_ctx::Variant` has a set of `child_Y` member functions that return a `Y_ctx::Range` of child nodes, for each other context

Y. For example, if v1 and v2 are func_ctx::Variant objects holding the following statement-trees:

```
// v1 =                          // v2 =
for (e1; e2; e3)                 if (e4)
  S1;                              S2;
                                 else
                                   S3;
```

then v1.child_expr will return the expr_ctx::Range [e1,e2,e3], v1.child_func will return the func_ctx::Range [S1], v2.child_expr will return the expr_ctx::Range [e4], and v2.child_func will return the func_ctx::Range [S2,S3]. Any other child_Y function will return an empty Y_ctx::Range.

Collectively, these child_Y functions are called *polymorphic child-functions*, since they act like virtual functions. Using child-functions, func_collect_decls becomes much easier to write. We start with the following recursive function, which does most of the work:

```
void func_impl(func_ctx::Variant v, vector<const ipr::Decl *> &decls)
{
  switch (v.which()) {
    case func_ctx::class_e: case func_ctx::union_e:
      uda_collect_decls(get_uda(v), decls);
      break;
    case func_ctx::glo_var_e:
      if (match(v.get<Glo_var>()))
        decls.push_back(&v.get<Glo_var>().ipr());
      break;
    default:
      for (func_ctx::Range r = v.child_func(); !r.empty(); ++r.first)
        func_impl(*r.first, decls);
  }
}
```

and kick off the recursion in func_collect_decls:

```
void func_collect_decls(Func f, vector<const ipr::Decl *> &decls)
{
  if (!f.has_def())
```

```
    return;
  func_impl(b.def_body().variant(), decls);
}
```

Notice that all the boilerplate cases mentioned earlier have been reduced to the single `default` case in `func_impl`.

The expression `b.def_body().variant()` has two interesting features. First, it returns the body of the function as a `func_ctx::Variant`, instead of a `Range`. This is because the root of a function is either a `try` statement or a curly-brace block, not a list of statements. The second interesting feature is `def_body`. Looking at the class `Func` in `filter.h`, we can see that `Func::def_body` returns an object of type `Func::Body`. Nested classes are used in the Filter library to signify that the returned object is elaborating the same node as the parent class. Nested classes are used to allow lazy evaluation and improve performance by eliminating unnecessary calls or checks.

4. Traversal

The code described thus far implements the declaration-finding stage of our semantic grep. Before moving on, in this section we consider the code we have written and look at a new Filter feature that can simplify the code further.

To find all member and global variables, we first identified all the contexts in which they could occur, and then wrote code to search *for* these contexts and then to search *through* these contexts. In doing so, we only examined contexts which might possibly contain the target nodes, ignoring a large percentage of the program that exists in type and sub-statement expressions. This "pruning" of the whole program graph is desirable because it improves the performance of our analysis.

The problem is, we had to write a lot more code than if we had a reusable traversal

algorithm at our disposal. While a discussion of traversal strategies is outside the scope of this traversal, the basic pattern is that the user first writes code to handle each type of node, and then passes this code (by a pointer, type, or function value) to an algorithm that applies the code to every node in the program graph. Thus, if we could have used a traversal algorithm to do declaration-finding, the only code we would have had to write would be the cases to accept global and member variables and the `match` function.

Instead of providing a set of traversal algorithms, the Filter library takes a different route. The basic idea has already been demonstrated with child-functions; these functions allow us to generically walk over the nodes in a tree without handling every node along the way. Using them, we can approximate a traversal function with only a few lines of code:

```
template <class Variant, class F>
void traverse(Variant v, F f)
{
  f(v);
  for (ns_ctx::Range r = v.child_ns(); !r.empty(); ++r.first)
    traverse(*r.first, p);
  for (uda_ctx::Range r = v.child_uda(); !r.empty(); ++r.first)
    traverse(*r.first, p);
  for (func_ctx::Range r = v.child_func(); !r.empty(); ++r.first)
    traverse(*r.first, p);
  for (expr_ctx::Range r = v.child_expr(); !r.empty(); ++r.first)
    traverse(*r.first, p);
  for (type_ctx::Range r = v.child_type(); !r.empty(); ++r.first)
    traverse(*r.first, p);
}
```

Here, `f` is a callable value that accepts `Variant` objects, which can then be queried for the desired node. Since `traverse` is a template, it is actually a family of five functions, one for each context's `Variant`.

While `traverse` is a good start, it breaks one of the Filter's design goals by imposing an inversion of control on the user: the user must package their code into a

function/object that gets called by `traverse`. An alternative design that avoids this inversion would be an iterator that lazily walks the tree each time it is incremented. Using this iterator (named `Traverse`), we could replace our three mutually recursive collect-functions with:

```
void collect_decls(Namespace ns, vector<const ipr::Decl *> &decls)
{
  for (Traverse t(ns); !t.done(); t.step())
    switch (t->which())
      case ?_ctx::glo_var_e:
      case ?_ctx::mem_var_e:
        if (match(get_var(*t)))
          decls.push_back(&get_var(*t).ipr());
}
```

However, as shown by the `?` in the above code, we run into the problem that we don't have a variant "big enough" to hold all possible Filter nodes. In fact, the whole goal of putting variants into contexts was to enumerate the *precise* subset of nodes that could actually occur.

To solve these and a family of related problems, Filter provides the *Any* context, which represents the union of the five other contexts. As one might expect, the Any context has a namespace, `any_ctx`, which contains `Variant`, `Iter`, `Which`, and `Range` types. Using the Any context, we can implement a pre-order depth-first `Traverse` as shown in Figure 6. Notice that the constructor of `Traverse` takes an `any_ctx::Variant` while the calling code above passes a `Namespace`. This works because the Any context provides a constructor for every node type.

Considering the relatively concise implementation of `Traverse`, we can see that, by encapsulating the complexity of child-enumeration in the Any context's child-functions, the user is free to design custom traversals that fit their needs. In contrast, while pre-packaged, library-supplied traversal algorithms can meet the common needs (depth-first, breadth-first, nested, etc.), they ultimately constrain the user to a

```
class Traverse {
  vector<any_ctx::Range> st_;

 public:
  Traverse(any_ctx::Variant v) { st_.push_back(v); }
  bool done() const { return st_.empty(); }
  any_ctx::Variant &operator*() const { return *st_.back().first; }
  any_ctx::Variant *operator->() const { return &*st_.back().first; }

  void step()
  {
    any_ctx::Range r = st_.back().first->child_any();
    if (!r.empty())
      st_.push_back(r);
    else {
      ++st_.back().first;
      while (st_.back().empty()) {
        st_.pop_back();
        if (!st_.empty())
          ++st_.back().first;
        else
          break;
      }
    }
  }
};
```

Fig. 6. A class for iterative pre-order depth-first traversal

framework. This comparison is discussed with additional details in Chapter IV and on pg. 146.

As it is presented above, using the Any context will forfeit the performance advantages mentioned at the beginning of this section. That is, `Traverse` will visit all reachable children of the node given on construction. To provide fine-tuning of this traversal, `child_any` has an optional "pruning" parameter which takes the following record:

```
struct Prune {
  bool ns, uda, func, expr, type;
  Prune()
   : ns(true), uda(true), func(true), expr(true), type(true) {}
  Prune(bool n, bool u, bool f, bool e, bool t)
   : ns(n), uda(u), func(f), expr(e), type(t) {}
};
```

Next, `Traverse` can be modified, as shown in Figure 7, to forward the user's pruning choices to `child_any`. With this new `Traverse`, the user can avoid visiting expressions and types by clearing the corresponding flags in the `Prune` argument:

```
Prune pr(true,true,true,false,false)
for (Traverse t(ns, pr); !t.done(); t.step())
  ... as before
```

The end result is that we are able to visit the same nodes in the graph as before, but with a fraction of the code. Hopefully the reader will forgive the indirect route to this solution. Altogether, we can see that the Filter library offers three levels of navigation: coarse-grained, through the Any context's child-functions; medium-grained, through the five specific contexts' child-functions; and fine-grained, through the direct use of Filter node member functions.

```
class Traverse {
  Prune pr_;
  vector<any_ctx::Range> st_;
 public:
  Traverse(any_ctx::Variant v, Prune p) : pr_(p) { st_.push_back(v); }
  ... unchanged
  void step()
  {
    Range_t r = st_.back().first->child_any(pr_);
    ... unchanged
  }
};
```

Fig. 7. `Traverse` modified to prune edges

5. Finding and printing uses

Having finished and refined the declaration-finding stage, we now turn to the use-finding stage, where the goal is to find and print all uses of variables that match the pattern in `cmdline`. Using the developments of the first four parts, we can accomplish this in just one section.

One approach to implementing use-finding would be to first find all uses of variables and then use the `match` function to decide what uses to print. As there are often many uses in a program for every declaration, this would be a wasteful approach, as `match` would be performed on the same variable many times. Instead we will use the list of declarations found in the previous stage.

The key feature for efficient use-finding is the identity property of IPR nodes. Consider Figure 3, shown earlier in this tutorial, and the `resolution` edge from `Id_expr` to `Var`. Since the edges of the graph represent C++ pointers/references in the IPR, we can see that the `ipr::Var` object found at the declaration site (as a child of `Block`) is the same object found at the use site (as the `resolution` of `Id_expr`). This means that C++ pointer equality suffices to check whether a declaration and use

refer to the same entity.

The Filter library represents uses of declarations (through their names) in expressions with the `Name_expr` node. Using `Name_expr`, we can start implementing `print_uses` as follows. (Recall from pg. 29 that `print_uses` is called by `analyze`.)

```
void print_uses(Namespace ns, const vector<const ipr::Decl *> &vec)
{
  unordered_set<const ipr::Decl *> decls(vec.begin(), vec.end());
  for (Traverse t(ns); !t.done(); t.step())
    switch (t->which()) {
      case any_ctx::name_expr_e: {
        Name_expr ne = t->get<Name_expr>();
        if (ne.which() == Name_expr::glo_var_e &&
            decls.find(&ne.glo_var().ipr()) != decls.end())
          print_use(t);
        break;
      }
    }
}
```

The basic approach is to look through the whole program for `Name_expr` nodes, see if the declaration used is a global variable, and then test whether its underlying `ipr::Decl` is in the set of `ipr::Decl` objects found in the first stage. To improve the performance of this lookup, a hash table is used. Once a match is found, the use is printed by the `print_use` function, which is shown further below.

We now consider this code in more detail, starting with the `Name_expr` node type. Its abbreviated declaration (from `filter.h`) is shown below:

```
struct Name_expr : Variant_node
{
  enum Which {
    this_e, enum_val_e, loc_var_e, glo_var_e, class_var_e, parm_e,
    glo_func_e, class_func_e
  };
  Which which() const;

  cross_edge::Enum::Val enum_val() const;
  cross_edge::Loc_var loc_var() const;
  cross_edge::Glo_var glo_var() const;
  ...
```

```
};
```

Here we can see that `Name_expr` is like the `Variant` types: it provides a `which` member and an enumeration of possibilities. However, instead of providing a `get` member, `Name_expr` provides a member `foo_bar()`, for each enumerator `foo_bar_e`. This member is used to get more information about the declaration that was used, such as, which global variable was named.

What about the `cross_edge` prefix? Looking at the `cross_edge` namespace in `filter.h`, we can see that `cross_edge::X` is just a typedef for `X`. So why the subterfuge? The `cross_edge` typedefs allow a Filter member to syntactically specify that the node being returned is not a child, but another node's child. Thus, the consistent use of `cross_edge` prefixes throughout the Filter library give a *syntactic* characterization of the tree structure inherent in the source code.

One last note about the above code is that, by using polymorphic child-functions (through `Traverse`), we easily find all the subtle places expressions can be used, such as the five uses of `G` in:

```
const int G = 5;
template <int I = G> struct A {
  int m : 2*G;
};

struct B : A<G+1> {
  int m;
  B(int i = G) : m(4*G-3) {}
};
```

To catch all cases like these, a feature like polymorphic child-functions, or a library-supplied traversal algorithm, is absolutely necessary.

Broadening the search to include member variables, we can see that member variables are conspicuously absent from `Name_expr::Which`. The reason for this is that member variables can only appear in a limited context. In particular, they may

only be used to access the member of another expression or have their address taken.[1] These two cases are represented by the `Member` and `Address` node types, respectively. Using these, we can add the following two cases to the switch statement in `print_uses`:

```
case any_ctx::member_e:
  if (t->get<Member>().which() == Member::mem_var_e &&
      decls.find(&t->get<Member>().mem_var().ipr()) != decls.end())
    print_use(t);
  break;
case any_ctx::address_e:
  if (t->get<Address>().which() == Address::mem_var_e &&
      decls.find(&t->get<Address>().mem_var().ipr()) != decls.end())
    print_use(t);
  break;
```

Looking at `filter.h`, we can see that `Member` and `Address` follow the same `which`-switch style as `Name_expr`. In the case of `Member`, `which` indicates whether the member is accessed by its declared name or by an expression resulting in pointer-to-member. For `Address`, `which` indicates whether the address is taken of a normal expression or a member.

With these cases added, our analysis will find all global and member variables. All that remains is to implement `print_use`. However, there is a difficulty in doing this: the IPR only stores source-position information for nodes of type `ipr::Stmt` , which is inherited by `ipr::Decl` but not by `ipr::Expr`. Furthermore, there are no direct links from `ipr::Expr` nodes to the `ipr::Stmt` that contains them. These decisions were made to save space: since there are a large number of expression and type nodes, the space required for each should be minimal, so each pointer counts.

To solve the problem, we take advantage of the tree-like structure of the program. Specifically, the `first` members of all the `Range` objects stored in the internal stack of

[1] In the upcoming C++ Standard, member variables may also appear as the argument of a `sizeof` or `typeid` expression, but, for brevity, we will ignore these cases in our analysis.

`Traverse` represent a path from the root of the traversal to the current node. Using this, we can find an expression's parent statement by simply walking backwards over this path until we reach a node derived from `ipr::Stmt`. Thus, if searching for uses of "x" in the following code:

```
int x;
void foo() {
  if ((1 < 2 ? x + 1 : 0) == 1)
    ;
}
```

our algorithm will first find the addition, then the conditional, then the equality, until it finally finds the `if` statement.

To implement this algorithm, we first need to expose the internal `Traverse` stack:

```
class Traverse {
  ...
  const vector<any_ctx::Range> &path() const { return st_; }
};
```

On pg. 81, we will see an efficient way to test inheritance using the Visitor pattern [41]. For now, we can simply use `dynamic_cast`. Thus, we implement `print_use` as follows:

```
void print_use(const Traverse &t)
{
  typedef vector<any_ctx::Range>::const_reverse_iterator Iter_t;
  for (Iter_t i = t.path().rbegin()+1, end = t.path().end(); i != end; ++i) {
    any_ctx::Variant v = *i->first; // pull the current node from the Range
    const ipr::Node *n = v.node().ipr(); // pull the IPR out of the Variant
    if (const ipr::Stmt *s = dynamic_cast<const ipr::Stmt *>(n)) {
      cout << "Use at line: " << s->unit_location().line << '\n';
      return;
    }
  }
}
```

Looking at this code, there are a few new bits that deserve explanation. First, `Variant` types have a member `Variant::node` which returns a reference to the Filter node held inside the `Variant`. Using this reference, we can ask the Filter node for its underlying IPR node. As indicated by the `ipr::Node*` return type, this pointer can be `0`, which

indicates that there is no underlying IPR node and the node has been synthesized as part of the lowering performed by the Filter.

Putting all these pieces together, we are finished with the use-finding stage. The only remaining code to write is `print_decls`:

```
void print_decls(const vector<const ipr::Decl *> &decls)
{
  for (int i = 0; i < decls.size(); ++i)
    std::cout << "Decl at: " << decls[i]->unit_location().line << '\n';
}
```

This concludes the implementation of `sgrep`.

6. Postmortem

Using the IPR with the Filter and Pretty-printer libraries, only about 60 lines of code were required to implement `sgrep`. In this code, we saw examples of how to analyze the declarations, statements, and expressions of C++ programs. Missing from this list are types in C++ programs. Analyzing types is described in more detail onpg. 59.

Part of the reason we ultimately had to write only a small amount of code was the direct IPR representation of the *results* of name lookup, overload resolution, specialization matching, type evaluation, and other hard C++ problems. For example, in the following code:

```
namespace A {
  struct S {};
  void f(S);
}
int main() {
  A::S s;
  f(s);
}
```

an analysis using `Id_expr::resolution` will be able to find the variable declaration named by "s" just as easily as the function declaration named by "f," despite the

latter dependency on the more complex Argument Dependent Lookup mechanism. The same argument applies, *mutatis mutandis*, for the other hard C++ problems.

Another feature that clearly shortened the code was the polymorphic child-functions. These allowed us to easily write a custom `Traverse` iterator to fit our needs (which changed over time), instead of using a pre-packaged algorithm with a fixed mode of traversal.

A last noteworthy feature, that didn't shorten code but assisted in writing it, is the context maintained by Filter types. Without context, there would be only one `Variant` in the Filter library—the `any_ctx::Variant`. Knowing the context in which a `Variant` is found, however, allows the `Variant` to enumerate a more precise set of nodes that it may actually contain. In this way, the `X_ctx::Members` enumerations save the user the trouble of looking at the C++ language definition to determine what may actually occur in every context. Thus, the Filter library effectively encapsulates this logic in one place, instead of having it scattered throughout every analysis.

F. Big picture

To conclude the tutorial, this section builds a simple, coherent picture of the various Filter concepts that have been introduced. While there are many classes in the Filter library, making the task of learning how to effectively use the library seem onerous, the user can "compress" their understanding by keeping this picture—literally, Figure 8, shown later—in mind.

First, we need to be precise about some terminology that has been used loosely up to this point. The term *Filter type* refers to a C++ class in the Filter library, and thus is a static entity. A *Filter object* refers to an object of a Filter type, and thus is a runtime entity. We can sort the Filter types into groups based on their role:

variants, *ranges*, *iterators*, *nodes*, and *node bases*. The variant, range, and iterator groups consist of all the `Variant`, `Range`, and `Iter` types, respectively, found in the six `X_ctx` namespaces. The node group consists of all the Filter types that represent individual IPR nodes. The nodes that have been described so far are `Namespace`, `Glo_func`, `Glo_var`, `Class`, `Union`, and `Name_expr`. Additionally, there are more than 40 types that collectively complete Filter's view of the IPR. Lastly, the node base group contains the types `Func`, `Var`, and `Uda` introduced so far as well as one other base, `Initable_var`.

With these groups defined, we now enumerate the functions between them, i.e., the ways to take a Filter object of one Filter type and produce an object of another. In this list, statements involving X and Y can be seen as applying to each of the six contexts, by substituting for X and Y.

1. *IPR node* $\leftrightarrow$ *Filter node* : The left-to-right direction is accomplished by the various node constructors, which take an appropriately-typed IPR node and a reference to the environment. The right-to-left direction uses the `ipr` member function found in most node types. (pg. 22)

2. *IPR node* $\rightarrow$ *X-context variant/range* : Although not shown in the tutorial, the Filter library contains the following five functions for going directly from an IPR node to a variant or range:

   ```
   maybe<ns_ctx::Variant> ns_ctx::discern(Env_ct &, const ipr::Decl &);
   maybe<uda_ctx::Variant> uda_ctx::discern(Env_ct &, const ipr::Decl &);
   func_ctx::Range func_ctx::discern(Env_ct &, const ipr::Stmt &);
   expr_ctx::Variant expr_ctx::discern(Env_ct &, const ipr::Expr &);
   type_ctx::Variant type_ctx::discern(Env_ct &, const ipr::Type &);
   ```

   The `discern` functions returning a `maybe` type indicate that the given IPR node may be found to be purely syntactic, and thus not represented by any Filter

node. The function context returns a `Range` since a single IPR statement may be lowered into zero or more Filter nodes.

3. *X-context variant* ↔ *node* : The left-to-right direction is accomplished by the `Variant::get<T>` member, which returns the node object currently inside the given variant object. The type `T` must correspond to an enumerator in the `X_ctx::Which` enumeration and must match the actual runtime return value of `X_ctx::Variant::which`. (pg. 22)

   The right-to-left direction uses the fact that `X_ctx::Variant` types have constructors for each node in `X_ctx::Members`. Thus, a `ns_ctx::Variant` can be constructed from a `Glo_func`, while trying to construct from a `Name_expr` will fail at compile time. (pg. 37)

4. *X-context variant* → *X-context range* : Although not shown in the tutorial, it is possible to construct a range from a variant. The resulting range will be a singleton sequence containing a copy of the variant.

5. *X-context variant* → *Y-context range* : The member functions `X_ctx::Variant ::child_Y` return the children of the node object contained by the variant object that are in the Y-context. (pg. 32)

6. *X-context iterator* ↔ *X-context range* : Simply, each range can be constructed from two iterators in the same context, and iterators can be extracted from a range by its `first` and `last` members. (pg. 27)

7. *X-context iterator* → *X-context variant* : Dereferencing an X-context iterator returns a X-context variant. (pg. 29)

8. *X-context variant* → *Any-context variant* : Following from the fact that the

Any context logically represents the union of the other five contexts, `any_ctx::Variant` has constructors taking each of the other five `Variant` types:

```
expr_ctx::Variant ev = ...
any_ctx::Variant av(ev);
assert(ev.which() == static_cast<any_ctx::Members>(ev.which()));
```

The final assertion only makes sense because, as can be seen in `filter.h`, enumerators in `X_ctx::Members` are defined to have the same integral values as the corresponding enumerators in `any_ctx::Members`.

9. *X-context range* → *Any-context range* : Building on the previous conversion in the obvious way, there is an `any_ctx::Range` constructor that accepts any other `X_ctx::Range`. This can be seen as a special case of covariant subtyping, which is safe when ranges are immutable [59].

These conversions are summarized in Figure 8. Ignoring the arrow for item 5, this graph is actually a *commutative* diagram, i.e., [with a little hand-waving and the appropriate projection/concatenation operations inserted where ranges are involved] any two paths between two nodes will take equal objects to equal objects, and any path that forms a loop will yield the same object.

One consequence of the diagram in Figure 8 is that the Filter library user can easily "tweak" the type they have in order to fit their needs. For example, assume the user has written a recursive traversal over ranges:

```
void examine_stmts(func_ctx::Range r)
{
  ... // examine the linear sequence of statements
  examine_stmts(...); // recursively descend into a nested sequence
}
```

but wants to iniate the recursion given a single node, say, a an `ipr::Stmt`, `Block`, or `func_ctx::Variant`. Looking at Figure 8, we can see a clear path for converting what we have into a `func_ctx::Range`:

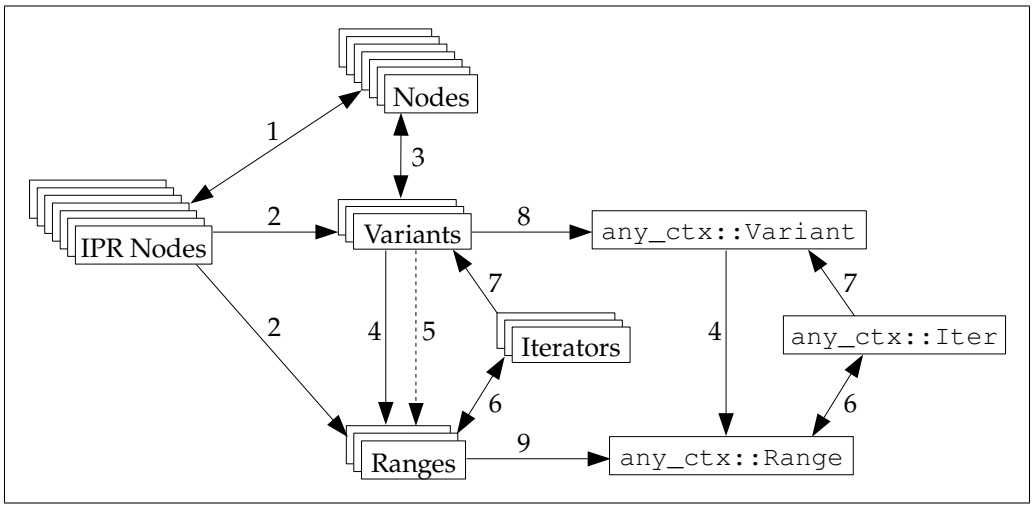

Fig. 8. Overall organization of the Filter library

```
const ipr::Stmt &stmt = ...
examine_stmts(func_ctx::discern(env, stmt));

Block b = ...
examine_stmts(func_ctx::Range(func_ctx::Variant(b)));

func_ctx::Variant v = ...
examine_stmts(v); // implicit conversion to Range via constructor
```

Moreover, when there are two ways to do the conversion, except for uses of `child_func` (the arrow for item 5 in Figure 8), both paths will produce the same results.

Finally, we now revisit the earlier claim (pg. 11) that the Filter integrates traversal and case analysis. Recall that traversal using the Filter is achieved through variants and polymorphic child-functions. The nodes considered during such a traversal are controlled by the context of the variants used. Thus, choosing the context of a traversal is an effective way to "prune" large parts of the program tree. Using the abovementioned conversions, we can also see how the context of a traversal can be broadened, by injecting into an Any-context variant, or refined, by using a non-Any child-function. Combined with the ability to use polymorphic child-functions in an *ad hoc* manner, this allows the user to easily build precise, custom traversals.

As an example of this integration, consider the following solution to the problem: "find all uses of a given variable in the conditions of if-statements in a given function."

```
void find_all_x_in_cond(Func f, Loc_var x)
{
  for (Traverse<func_ctx::Range> t1(f.def_body().variant());
       !t1.done(); t1.step())
    if (t1->which() == func_ctx::if_e)
      for (Traverse<expr_ctx::Range> t2(t1->get<If>().condition());
           !t2.done(); t2.step())
        if (t2->which() == expr_ctx::name_expr_e &&
            t2->get<Name_expr>().which() == Name_expr::loc_var_e &&
            &t2->get<Name_expr>().loc_var().ipr() == &x.ipr())
          ... found one!
}
```

Here, we assume that we have parameterized `Traverse` over range type, instead of fixing `any_ctx::Range` as in Figure 6.

Now, we consider how this might be accomplished with a "traditional" traversal library, i.e., without (1) the *ad hoc* nesting of traversals used above or (2) the automatic pruning performed based on context. The approach that first comes to mind would be to write a traversal that searches for all uses of the given variable and then, when one is found, walks up the ancestors of the use site to determine whether the use occurs as the condition of an `if` statement. This assumes that the traversal maintains and exposes such a stack of ancestors to the user, which is not always the case [38,60]. One downside of this approach is that it will visit *all* types and expressions, instead of just the runtime expressions in conditions. On average, this will more than double the running time compared to the Filter solution. To regain efficiency, the user would need to match on every statement node to carefully control descent into types and expressions. This also requires that the library traversal provide the user with fine-grained traversal control (viz. the ability to descend into some, but not all, child nodes).

CHAPTER III

DESIGN AND IMPLEMENTATION

Having introduced the Filter library, this chapter describes the design and implementation in more detail. On pg. 4, four basic goals—two functional, two stylistic—were outlined. The way Filter achieves the first goal (simplify traversal and case analysis) was demonstrated in the tutorial in Chapter II. The second functional goal (perform lowering on the IPR) was used implicitly throughout the tutorial and is outlined explicitly in this chapter by §A and §B. The two stylistic goals (provide a lightweight view and avoid inversion of control) were also used implicitly throughout the tutorial and are detailed in §C and §D. These four sections build a list of functional and performance requirements for the implementation of ranges and the final section (§E) concludes by describing this implementation in detail.

As mentioned at the beginning of Chapter I, Filter simplifies the IPR for the user by focusing on a more specific set of use cases than Pivot. In particular, the class of analyses targeted by the Filter library are the "high-level semantic analyses." Examples include the automatic parallelization example from pg. 13, semantically enhanced library languages [61], safety analyses [6, 62], and code generation to an IR like LLVM [43]. Non-examples include code-style checking and pretty printing. Thus, statements made in this chapter about what information is necessary and what information is just syntactic detail are made with respect to the class of high-level semantic analyses.

Lastly, there is one syntactic convention to discuss. In general, the IPR is a graph, as shown in Figure 3. However, in many cases we are only interested in viewing a tree-like subgraph of the overall graph. In these cases, a concise LISP-like representation can be used instead of an image. For example, to show the IPR rooted

at the `Return` node in Figure 3, the following syntax is used, ignoring children with "..." and dropping fields when they are not relevant:

```
(Return value:(Minus first:(Plus first:(Id_expr resolution:(Var ...)
                                        second:(Literal "2"))
                           second:(Literal "3")))
```

A. Templates

In C++, there are two ways to look at a template. In the first view, a template is a single concrete entity with associated syntax. In the second view, a template is a family of concrete entities, each with associated syntax, that are generated by instantiations of the template. To illustrate the difference between these two views, consider the following example:

```
template <class T>
bool reflexive(const T& x, const T& y) { return x == y && y == x; }

bool b1 = reflexive(1, 2);
bool b2 = reflexive(string(), string());
```

We now consider this code from both points of view. In the first view, the sample contains three declarations: a template and two global variables. The template declaration has a body, which contains an expression-statement. Looking at this expression, it is not exactly clear what operations are being performed. Is the "`==`" a built-in operation or a call to a user-defined function? Even the call to "`&&`" is unresolved, since the result type of equality comparison might not be `bool`. As for the initializers of `b1` and `b2`, both are calls to a template function which pass two runtime arguments and one compile-time template argument.

Despite these unknowns in the representation of `reflexive`, this information is useful. For example, a *concept analysis* can look at this code and derive the syntactic requirements on the parameter `T` [63]. Alternatively, assuming `T` was known to provide

certain minimal syntactic and semantic guarantees, it is possible to prove properties about the generic algorithm itself [64]. Additionally, on the caller side, we can examine the template arguments to determine whether they satisfy all the semantic properties required by the generic algorithm.

Changing to the second view, the same code looks different. There are still three declarations, but the first is a "family of functions." Looking at this family, we don't consider the syntax of its body, just the family members, which look like normal non-template functions. In the sample code, we can find two different instantiations, `reflexive<int>` and `reflexive<string>`, thus the family has two members. Because these members are fully-instantiated, we can be sure that all the unknowns mentioned in the first view have been resolved. Thus, in `reflexive<int>`, the call to "`==`" shows up as a C++ built-in, while in `reflexive<string>`, "`==`" is a call to `string::operator ==`. Lastly, the calls to `reflexive` in the initializers of `b1` and `b2` are viewed as calls to fully-instantiated functions.

With this view of templates, we can now more easily apply traditional analysis frameworks like monotonic data-flow analysis [65] and abstract interpretation [66]. In fact, we can almost ignore the presence of templates in C++ altogether. To do this, we just view instantiations, like `reflexive<int>`, as normal functions with strange names.

Clearly, both views of the code are useful and, for certain applications, necessary. While the second view could technically be derived from the first by simulating template instantiation as needed, this process is complicated and, since it is already done by the front-end, wasteful. Thus, Pivot represents both views in the IPR, simultaneously. Furthermore, these views are tightly integrated and they allow the user to shift back and forth between templates and their instantiations. A more detailed description of how templates are represented in the IPR is beyond the scope of this

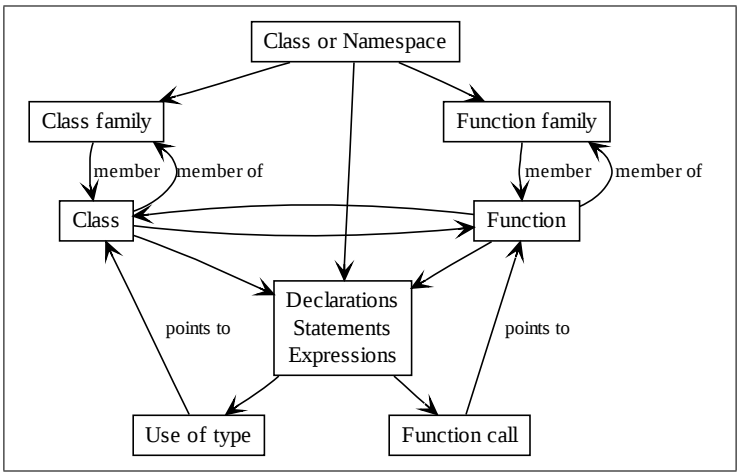

Fig. 9. Filter model of templates

thesis; see the Pivot framework documentation.

With this background, we can now examine how templates are treated in the Filter library. The basic model corresponds to the second view described above and is shown in more detail in Figure 9. In this figure, edges without labels correspond to the ordinary "child" or "contains" relationship. The important edges are the "member" and "member of" edges between families and members. The key property of this graph is that it avoids entering the uninstantiated context of a template body. This is key because it allows Filter to provide a simpler view of the program than the raw IPR can. Specific instances of this simplification are described on pg. 59 and pg. 63.

We conclude by showing how to analyze templated C++ code with the Filter library. *Unfortunately, as opposed to the rest of the discussion in this thesis, this demonstration is hypothetical: these interfaces have not been fully implemented in Filter. This is due to the fact that the exact representation of templates in the IPR currently lacks a detailed specification and hence is not fully implemented by the underlying IPR generation tools.*

Templates in Filter are represented by the `Class_family`, `Union_family`, and `Func_family` Filter nodes. These nodes can be used to obtain their respective family members as follows:

```
void analyze_class(Class);

void analyze_ns(Ns ns)
{
  for (ns_ctx::Range nr = ns.members(); !r.empty(); ++r.first)
    switch (nr.first->which()) {
      case ns_ctx::class_e:         // found single [non-template] class
        analyze_class(nr.first->get<Class>());
        break;
      case ns_ctx::class_family_e: { // found family of classes
        Class_family cf = nr.first->get<Class_family>();
        for (Class_family::Range cfr = cf.members();
             !cfr.empty();
             ++cfr.first)
          analyze_class(*cfr.first);
        break;
      }
    }
}
```

As shown, `Class_family::members` corresponds to the "member" edge of Figure 9. Note that `Class_family::Range` contains a pair of iterators that, when dereferenced, return a `Class`. Altogether, this code shows how an analysis that only cares about non-template classes (`analyze_class`) can be "lifted" to analyze template classes by analyzing their members.

Another way to find the instantiations of a template is through the polymorphic child-functions, based on the view that instantiations are child nodes of the template. Thus, an analysis using the `Traverse` class defined onpg. 37, which internally uses `child_any` to traverse the tree, would examine all instantiations.

The other direction, corresponding to the "member of" edge in Figure 9, is provided by the members of `Class`, `Union`, and `Func`:

```
for (Traverse t(...); !t.done(); t.step())
  if (t->which() == any_ctx::class_e)
```

```
if (t->get<Class>().has_family()) {
  Class_family cf = t->get<Class>().family();
  ...
}
```

The uses of templated classes and functions stays the same as the use of non-templated classes and functions, which are discussed on pg. 59 and pg. 63, respectively. Thus, in the following code:

```
class C {};
template <class> class T {};
C x;
T<int> y;
```

both `x` and `y`'s types will appear as uses of a `Class`. To determine that `y`'s type is actually an instantiation of a template, the analysis can take the extra step of testing `Class::has_family`.

B. Lowering

In this section we consider a set of lowering operations that are commonly needed for high-level semantic analysis. Not a great deal of code is required for most of the individual lowering operations performed and each could therefore be written by an IPR user as needed. However taken together, they represent a significant amount of code and knowledge of C++ and the IPR. Thus, one of the goals of the Filter library is factor out this common lowering code by providing a view of the IPR where lowering has already been performed.

1. Types

In C++, types can be thought of as expression trees, where "`*`", "`&`", "`[]`", and "`()`" are operators that construct compound types from their arguments. The IPR represents this expression tree directly by using nodes to represent each of these type

constructors. For example, the types of the variables in the following code:

```
int ar[4];
float (*pf)(char);
```

are represented by the following IPR nodes:

```
ar : (Array element_type:(Int) bound:(Literal "4"))
pf : (Pointer points_to:(Function source:((Char)) target:(Float)))
```

Additionally, there is a loose one-to-one correspondence between the IPR and Filter for these node types. In fact, the only reason for the Filter library to wrap these nodes at all is to support seamless use of the library when traversing the IPR.

However, the representation of the use of user-defined types is less straightforward in the IPR. For example, given the following code:

```
class C {};
C c;
```

one might guess that the IPR for `c`'s type would look like:

```
c : (Class ...)
```

However, the following is the actual IPR produced:

```
c : (As_type expr:(Id_expr name:(Identifier "C")
                           resolution:(Typedecl initializer:(Class ...))))
```

This is for a good reason; the meaning of these nodes is as follows. `As_type` represents the use of a general expression as a type. Here, the expression is an `Id_expr`, which represents a name that has been resolved to a declaration. The declaration is a `Typedecl`, which represents the named declaration of some type which, in this case, is a `Class`. Together, this representation captures many of the variations that can appear in C++ source code: for example, names can qualified or unqualified and types can be named or unnamed.

For the purpose of high-level semantic analysis, however, we often only care to find out the type being used. Furthermore, for new users, understanding each IPR node of an expression can be daunting. To both simplify usage and factor out common

code, the Filter library begins by providing the `Udt_use` node type. Its abbreviated signature is as follows:

```
struct Udt_use {
  enum Which { class_e, union_e, enum_e };
  Which which() const;

  cross_edge::Class the_class() const;
  cross_edge::Union the_union() const;
  cross_edge::Enum the_enum() const;
  ...
};
```

Recall, from pg. 42, that the `cross_edge` prefix means that, with respect to the program tree, the returned node is not a child of the node whose member was called.

With `Udt_use`, picking out uses of user-defined types is as easy as picking out the type constructors introduced at the beginning of this section. For example, the following code harvests the classes of all the variables in the given namespace:

```
void harvest_classes(Namespaces ns, vector<Class> &out)
{
  for (Traverse t(ns); !t.done(); t.step())
    if (t->which() == any_ctx::udt_use_e &&
        t->get<Udt_use>().which() == Udt_use::class_e)
      out.push_back(t->get<Udt_use>().the_class());
}
```

Additionally, in the process of discerning the right Filter node type for a given `ipr::Type`, Filter's algorithm handles the following intricacies for the user:

- There are two ways to determine the kind of user-defined types being declared by a `Typedecl`: looking at `Typedecl::type` or the type of the node returned by `Typedecl::initializer`. The former method can always be used, but it requires several virtual function calls. The latter method is faster, wasting no virtual calls if the initializer is needed anyway, but only possible when `Typedecl::has_initializer` is `true`. Thus, to be efficient and correct, a hybrid strategy employing both should be used.

- A user-defined aggregate may be declared but not defined, or defined but not declared:

```
class C; // Typedecl with no initializer
struct { int i; } x; // type of Var is Class with no Typedecl
```

  Filter allows uniform treatment by merging both cases.

- In addition to representing the use of user-defined types, `As_type` also represents primitive types by holding an `Identifier` containing the name of the primitive type (e.g. `int` becomes `(As_type expr:(Identifier "int"))`). Filter can identify standard and non-standard primitive types without looking into this `Identifier`: Standard C++ primitive types are picked out by visitation, leaving non-standard primitive types as the only possible meaning of an `As_type` holding an `Identifier`. These two cases are presented to the user as the Filter node types `Cpp_builtin` and `Non_cpp_builtin`.

- The special case of parameter types breaks the assumption stated in the previous bullet, since `Ellipsis` derives `As_type` and has an `Identifier` expression containing "`...`". To keep this special case out of `type_ctx::Variant`, ellipses are caught where they occur: in function signatures and catch statements.

- Lastly, `As_type` may refer to an `Alias` node (which represents the C++ `typedef`). These are iteratively followed to an eventual non-`Alias` type.

As this list illustrates, `As_type` can be used in a number of ways. On the other hand, this is a fairly restricted set of possibilities compared to `As_type`'s description as "the use of a general expression as a type." For Standard C++, this list is a complete characterization of all legal uses of `As_type`. Thus, the Filter library encapsulates the case analysis needed to interpret `As_type`. More than just saving code, though, by

replacing `As_type` with all its "semantic" cases, Filter saves the user from the need to consider all possible C++ types that might be represented as a "general expression." Instead, when considering the type of a variable or expression, the Filter user is simply given a `type_ctx::Variant` whose `which` member function returns one of the following enumerators:

```
enum type_ctx::Members {
  cpp_builtin_e, non_cpp_builtin_e, udt_use_e,
  qualified_e, ptr_e, ref_e, array_e, ptr_to_member_e, func_t_e
}
```

On a final note, `ipr::Product` and `ipr::Sum` are also missing from this list. This is because these types only show up in specialized circumstances, like the parameter list of an `ipr::Function`. To keep them out of `type_ctx::Variant` where they make little C++ sense, they are represented by a `type_ctx::Range` in the Filter interface wherever they may occur.

2. Runtime expressions

We now consider the portion of the IPR which represents *runtime expressions*. The qualification "runtime" is necessary because the IPR notion of "expression" represented by `ipr::Expr` is very broad and includes declarations, statements, and types. Since runtime expressions are key to semantic analyses—used, for example, to determine the transfer functions in data-flow analysis—Filter focuses on this subset.

In many cases, the representation of runtime expressions in the IPR mirrors traditional ASTs. For example, the expression "`1+2*3`" is represented as:

```
(Plus first:(Literal "1")
      second:(Mul first:(Literal "2") second:(Literal "3")))
```

The representation of function calls also has a straightforward mapping to the syntax. For example, the expression "`p->foo()`" is represented:

```
(Call function:(Arrow base:(Id_expr name:(Identifier "p")
```

```
                                    resolution:(...) )
                          member:(Id_expr name:(Identifier "foo")
                                          resolution:(...)))
        args:())
```

This is a good choice of representation because it can be uniformly applied in templated and non-templated contexts. For example, in this function:

```
template <class T> void call_foo_member(T *p) { p->foo(); }
```

the structure of the IPR representation of "`p->foo()`" stays the same, replacing the second `Id_expr` node with a weaker `Identifier` node since the resolution of "`foo`" is unclear. Notice that, from the point of the view of the template, "`foo`" can legally be a member function, member variable of type pointer-to-function, or member variable with an overloaded `operator()`.

While this representation is good for both pretty-printing and uninstantiated contexts, it is not ideal for analyzing the *execution* of an expression. For example, `Arrow` has no behavioral meaning by itself; it has to be interpreted in the context of its parent and children. In instantiated contexts, all this information is present in the IPR, so Filter lowers the view of these IPR nodes so as to give each Filter node an operational meaning: member access, member function call, and non-member function call.

Member access is represented by the `Member` node, whose abbreviated signature is shown here:

```
struct Member {
  expr_ctx::Variant object() const;

  enum Which { mem_var_e, ptr_expr_e };
  Which which() const;

  cross_edge::Mem_var mem_var() const;
  expr_ctx::Variant ptr_expr() const;
};
```

The `object` member returns the runtime expression that evaluates to the object whose

member is being accessed. The `Which` values respectively indicate whether the member is accessed directly, with an identifier, or indirectly, through an expression that results in a pointer-to-member.

The abbreviated signature for `Mem_call` (the Filter node for a member function call) resembles `Member`, with the addition of an argument list:

```
struct Mem_call {
  expr_ctx::Variant object() const;

  enum Which { mem_func_e, ptr_expr_e };
  Which which() const;

  cross_edge::Mem_func mem_func() const;
  expr_ctx::Variant ptr_expr() const;

  expr_ctx::Range args() const;
};
```

Notice here that, in the program tree, `Mem_call` has up to three classes of child edges: an edge to the receiver (`object`), a possible edge to an expression resulting in a pointer-to-member-function (`ptr_expr`), and an edge to each function argument (`args`). Thus, if an `expr_ctx::Variant` holds an expression "`(a->*b)(c,d)`", which is rooted at a `Mem_call`, `child_expr` must return `[a,b,c,d]`. To implement this, since there is no underlying `ipr::Sequence` with the elements `[a,b,c,d]`, the Filter library must be able to synthesize this sequence.

Compared to member invocations, non-member invocations take away the receiver expression and add new choices for what function is called:

```
struct Call {
  enum Which { builtin_e, glo_func_e, class_func_e, ptr_expr_e };
  Which which() const;

  ipr::Category_code builtin() const;
  cross_edge::Glo_func glo_func() const;
  cross_edge::Class_func class_func() const;
  expr_ctx::Variant ptr_expr() const;

  expr_ctx::Range args() const;
};
```

Notice that, with the `builtin_e` enumerator, `Call` treats expressions like "`a+b`" as calls to a built-in function "`+`". Thus, in addition to hiding the syntactic IPR nodes `Arrow`, `Dot`, `Arrow_star`, and `Dot_star`, this node unifies the more than 50 expression nodes like `Plus`, `Minus`, `Div`, etc. A benefit of this design is that, when the actual operation described by a node is not important, to get to the arguments, the user does not have to handle every type of node.

As a result of lowering to these three node types, semantic analysis of calls and member access is much easier. For example, the task of finding all calls to a given member function in a given namespace can be written:

```
void find_calls_to(Namespace ns, Mem_func mf)
{
  for (Traverse t(ns); !t.done(); t.step())
    if (t->which() == any_ctx::mem_call_e &&
        t->get<Mem_call>().which() == Mem_call::mem_func_e &&
        &t->get<Mem_call>().mem_func().ipr() == &mf.ipr())
      ... found!
}
```

This one short piece of code actually matches three very different IPR patterns represented by the *three* member function invocations written below:

```
struct P { A *operator->(); };
struct A { B foo(); };
struct B { B operator+(B); };

void uses() {
  P p;
  B b;
  b + p->foo();
}
```

Notice that `find_calls_to` did not have to bother with the language rules, or different IPR representations, for operator overloading and the unique case of `operator->`. Moreover, the following calls, which superficially resemble member function invocations in the IPR, will be correctly recognized as `Call` nodes by Filter:

```
typedef void (*Ptr_func_t)();
```

```
struct A { Ptr_func_t m; void foo(); };
Ptr_func_t operator->*(A, void (A::*)());

void uses() {
  A a, *pa = &a;
  a.m(); // pointer-to-function call
  pa->m(); // pointer-to-function call
  (a->*(&A::foo))(); // then pointer-to-function call
}
```

Ultimately, the case analysis required to correctly categorize the IPR along these lines is rather complex; the optimized algorithm in the Filter library requires around 200 lines of code. Figure 10 shows the basic decision tree used. Thus, the lowering of runtime expressions done by Filter can offer a significant reduction in user code.

Moving beyond calls and member access, another type of runtime expression that is lowered is uses of declarations, represented by the Filter node `Name_expr`. Thus, `Name_expr` replaces `ipr::Id_expr` in runtime expressions much like `Udt_use` replaced `ipr::As_type`/`ipr::Id_expr` in type expressions. `Name_expr` has already been described on pg. 42.

A choice of runtime expressions in Filter is represented by the `expr_ctx::Variant`. Thus, the condition of an `if` statement, the expression returned by a `return` statement, and the arguments of a function call all return `expr_ctx::Variant` objects. The `expr_ctx::Members` enumeration contains:

```
name_expr_e, member_e, call_e, mem_call_e, literal_e, datum_e, new_e,
delete_e, address_e, cast_e, typeid_e, sizeof_e, throw_e, conditional_e
```

Comparing this set to the set of derived classes of `ipr::Expr`, we can see how `expr_ctx ::Variant` simplifies case analysis by significantly reducing the set of possibilities that must be considered.

The expert C++ user may have noticed that certain nodes that can appear in expressions in special cases, such as local variables, are missing. This is not incompleteness; these special cases are kept out of the `expr_ctx::Variant` by introducing

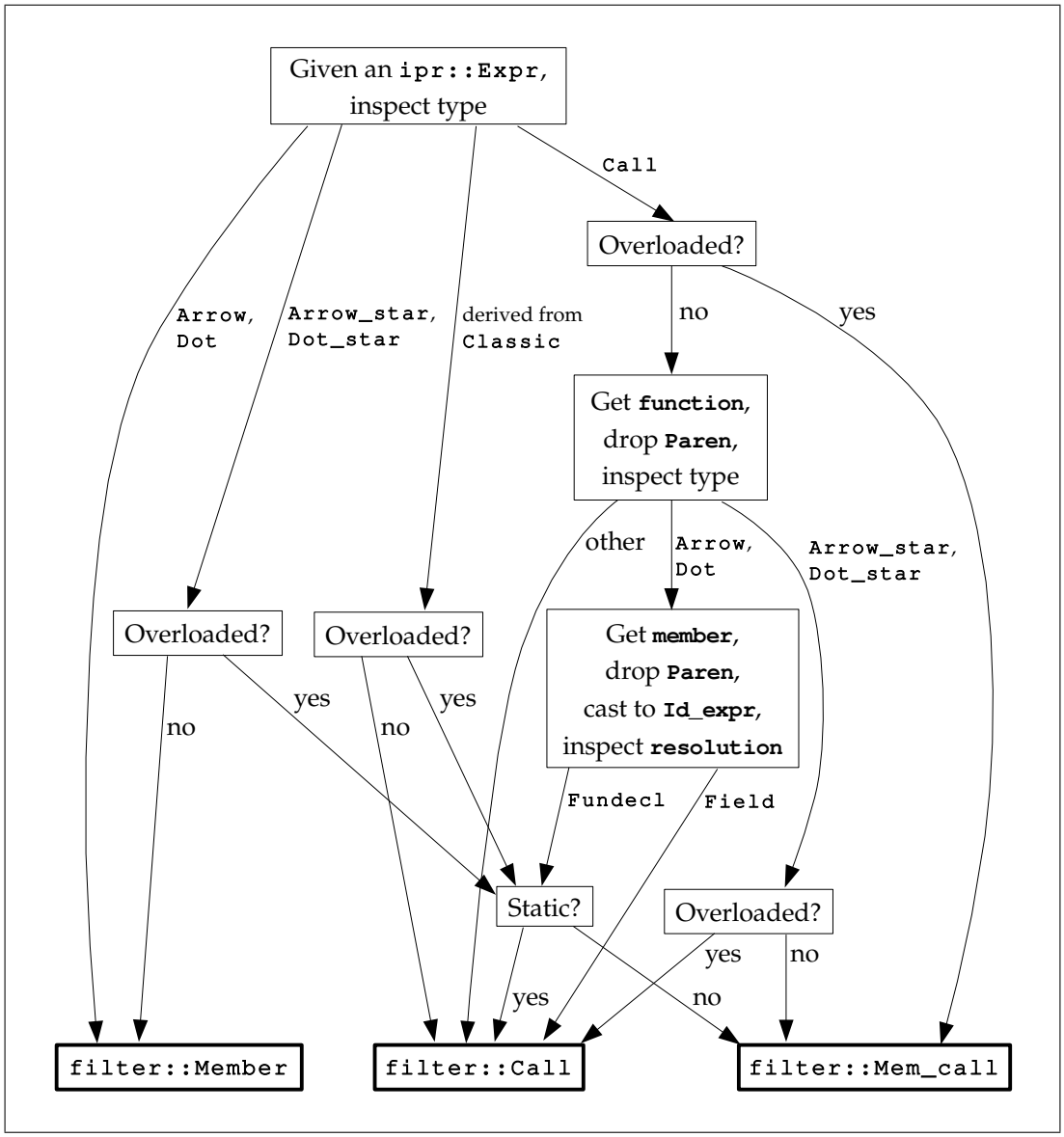

Fig. 10. Decision tree for expression lowering

the possibility *at the special case*. For example, variable declarations with initializers can be used in the conditions of statements. Thus, the interface for `If` has a special pair of members:

```
struct If {
  bool cond_is_var() const;
  Var cond_var() const; // precondition: cond_is_var is true

  expr_ctx::Variant condition() const;
  ...
};
```

that describe the variable, while `condition` returns an expression using this declared variable.

By keeping these special cases out of `expr_ctx::Variant`, the general case is kept more concise and the user does not have to consider the meaning of a `Var` that occurs in the middle of an ordinary expression. Another example is `ipr::Phantom`. These nodes represent special situations where an expression is optional, like the increment of a `for` statement. Thus, `Phantom` is kept out of `expr_ctx::Variant` by catching it at each of the places where expressions are optional (e.g., by adding `Loop::has_inc` and `Return::has_expr` which must return `true` before asking for the expression). Again, this prevents the user from having to consider optional expressions *in the general case*.

As a general rule, Filter tries to maintain the property that the nodes enumerated by a variant are *actual possibilities* anywhere the variant is used, although clearly there are limits to this precision. This assists the user in case analysis since the user is reminded of special cases by corresponding special cases in the Filter interface.

3. Default arguments

For a high-level semantic analysis, it often is not important whether an argument was supplied or a default was used. Thus, the following two calls mean the same thing:

```
void foo(int i = 1);
foo();
foo(1);
```

In the more general usage scenarios supported by the Pivot framework, this syntactic information could be useful, thus it is preserved in the IPR. For example, the above code is represented as:

```
(Fundecl parameters:((Parameter default_value:(Literal "1") ...)) ...)
(Call function:(Id_expr name:"foo" resolution:...) args:())
(Call function:(Id_expr name:"foo" resolution:...) args:((Literal "1")))
```

To recover the actual arguments passed, it is necessary to follow `Call::function` through the `Id_expr` to the `Fundecl`, get its parameter list, and find the corresponding `Parameter`'s `default_value`.

To simplify this common task, Filter performs a slight lowering, inside `Call` and `Mem_call`, whereby the argument list is transparently extended to include the default values that are used. Thus, the Filter library will show the two calls to `foo` above as having equivalent argument lists. In addition to saving code, this prevents the possible mistake of assuming all runtime arguments are present in the `Call::args` list.

The same lowering also applies to arguments in template identifiers:

```
template <class T = int> class C {};
C<> c1;
C<int> c2;
```

As before, the types of `c1` and `c2` are the same although the argument lists of their types are different; Filter fills in the missing arguments using the defaults.

These default-completed argument lists (exposed to the Filter user as `expr_ctx::Range` objects) exist in the IPR as two `ipr::Sequence` objects (one for the arguments, one for the parameters). Thus, they must be transparently merged by the Filter `Range` implementation.

4. Labeled statements

In C++, labels are not complete statements by themselves, but are instead attached to other statements, forming compound statements without the need for curly braces:

```
int x;
A: x = 1; // single labeled statement
if (b)
  B: x = 2; // legal without curly-braces
switch (i)
  case 1: case 2: x = 3; // also legal
```

The IPR preserves this information about the syntax by making `ipr::Labeled_stmt` hold the statement it labels as a child node. On the other hand, `Labeled_stmt` can be frustrating when trying to iterate over a linear list of statements. For example, the loop:

```
ipr::Sequence<ipr::Stmt> &stmts = ...
for (int i = 0, sz = stmts.size(); i < sz; ++i)
  use(stmt[i]);
```

will skip over the "`x = 1`" in the statement "`A: x = 1;`". To fix this, the loop can be changed to:

```
ipr::Sequence<ipr::Stmt> &stmts = ...
for (int i = 0, sz = stmts.size(); i < sz; ++i) {
  ipr::Stmt *s = &stmts[i];
  while (true) {
    use(*s);
    if (s->category == ipr::labeled_stmt_cat)
      s = &static_cast<const ipr::Labeled_stmt *>(s)->stmt();
    else
      break;
  }
}
```

or an analogous version using a visitor. Reverse iteration is much trickier.

To alleviate this inconvenience, the Filter library lowers `ipr::Labeled_stmt` by treating its child as its next sibling. Thus, the following loop:

```
func_ctx::Range r = ...
```

```
for (; !r.empty(); ++r.first)
  use(*r.first);
```

will examine the same statements as the previous nested loop.

This lowering represents yet another case where Filter ranges do not map directly to IPR sequences. A particular challenge is supporting bidirectional iteration efficiently and without penalizing the average case.

5. Aliases

The last and smallest lowering to describe is the hiding of `ipr::Alias`, which represents `typedef` statements, `using` declarations, `using` directives, and `namespace` aliases. These statements are purely syntactic as they only affect the names which can be used to refer to declarations. In general, this information might be useful to an analysis, so it is preserved in the IPR. For example, an analysis might want to give a warning for the following misuse:

```
typedef int Policy_number;
typedef int Policy_date;
void remove_policy(Policy_number);

void do_work(Policy_date pd, Policy_number pn)
{
  ...
  remove_policy(pd); // legal C++, but probably intended ‘pn’
}
```

However, for a semantic analysis, we often want to ignore this syntactic information. Thus, the Filter library slims down the number of cases to analyze by hiding `ipr::Alias` both when it is declared and used. On pg. 59, it is mentioned that uses of `Alias` are unfolded during type lowering. For example, in the following code:

```
namespace N {
  typedef const char *Str_t;
  Str_t name;
}
```

Filter views the namespace `N` as containing only one declaration and views the type of `name` as "`const char *`". Considering the implementation of the `ns_ctx::Range` which enumerates the members of `N`; there is an underlying `ipr::Sequence`, but to use it, the implementation must filter out elements.

6. The discern functions

While raw IPR nodes can be wrapped in Filter nodes and then injected into a `Variant`, the Filter library exposes another way to inject IPR nodes into variants (mentioned on pg. 48) which can be significantly more useful. Every context has a `discern` function which takes a raw IPR base type and returns some form of `Variant` for the context directly. In the case of expressions and types, `discern` has a simple signature:

```
typedef const Env Env_ct;
expr_ctx::Variant discern(Env_ct &, const ipr::Expr &);
type_ctx::Variant discern(Env_ct &, const ipr::Type &);
```

because there is a mapping from every runtime expression and type to a Filter node. Note that, as described on pg. 63, `ipr::Expr` includes more nodes than just runtime expressions. Thus, it is the caller's responsibility to ensure the given node does not just derive `ipr::Expr`, but is actually an expression found in a runtime context.

The `discern` functions for the other three contexts are not so simple:

```
maybe<ns_ctx::Variant> discern(Env_ct &, const ipr::Decl &);
maybe<uda_ctx::Variant> discern(Env_ct &, const ipr::Decl &);
func_ctx::Range discern(Env_ct &, const ipr::Stmt &);
```

The first two signatures reflect the fact that, because of the lowering described on pg. 72, some `ipr::Decl` nodes are not represented by any Filter node. The `maybe` type supports boolean tests and dereference to access its contents. The last signature reflects the situation, described in pg. 71, where a single `ipr::Stmt` refers to multiple Filter nodes.

Using these functions, the Filter user can immediately jump into the Filter library without knowing the exact type of the IPR node used to construct the initial Filter object. For example, this allows the following code to use the case analysis encapsulated by the Filter library, while working directly with the IPR before and after:

```
const ipr::Decl &ipr_in = ...
maybe<ns_ctx::Variant> mv = ns_ctx::discern(env, ipr_in);
if (mv && mv->which() == ns_ctx::class_func_e) {
  const ipr::Fundecl &ipr_out = mv->get<Class_func>().ipr();
  ...
}
```

C. A lightweight library design

As already stated, one of the design goals of the Filter library is to be a *lightweight* library on top of Pivot. This lightweight design is already demonstrated in Chapter II, particularly on pg. 22, where Filter nodes are created and destroyed as needed without any significant initialization or finalization work. This section describes how this design choice interacts with the lowering described on pg. 59 and the library's performance.

1. Lowering

In trying to achieve the goals of being a lightweight library and doing lowering there is an apparent conflict: the natural way to describe lowering is as a transformation from an un-lowered data structure to a lowered one, and the natural way to implement this is as a giant recursive function that does all the lowering in one shot. However, this strategy is clearly not lightweight: such a transformation would be a large up-front cost to using the Filter and the resulting data structure would consume memory commensurate with the original `ipr::Unit`.

In addition to breaking the "lightweight" design goal, the separate-data-structure approach raises another design question which does not seem to have a good answer: after the lowering transformation, should both data structures be kept in memory? If they are, then memory usage will [roughly] double and performance will be reduced due to increased paging. However, if the IPR is released, the user will be cut off from the syntactic origins of the code. These origins may still be useful; it is not unreasonable for an analysis writer to want to work on the lowered model offered by Filter while occasionally requiring the syntactic details offered by the IPR.

For these reasons, the classic separate-data-structure route is undesirable. Fortunately, there are several properties of lowering that can be used to our advantage. First, compared to the lowering of IRs described in [67] or the code generation described in [31], the logical distance between the un-lowered and lowered forms is small. Furthermore, the information required to lower is local and the computation required is light—usually just a case analysis on a few reachable nodes.

Taking advantage of these properties, an alternative to the upfront creation of a separate data structure is a model where IPR nodes are lowered to Filter nodes as each individual node is requested by the user. For example, consider the implementation of `filter::If`. Its abbreviated signature is:

```
struct If {
  expr_ctx::Variant condition() const;
  func_ctx::Variant then_branch() const;
  func_ctx::Variant else_branch() const;
};
```

In the on-demand model, `filter::If` simply stores a pointer to the underlying IPR node. When the Filter user calls `If::condition`, the implementation first calls `condition` on the underlying `ipr::If_then` or `ipr::If_then_else` node to get an `ipr::Expr` node. Next, Filter performs lowering on this IPR node, as described on pg. 63,

to determine what Filter node to put inside the returned `expr_ctx::Variant`. Thus, using the `expr_ctx::discern` function described on pg. 73, the implementation can be written something like this:

```
expr_ctx::Variant If::condition() const
{
  return expr_ctx::discern(
           ipr_->category == ipr::if_then_cat
             ? static_cast<const ipr::If_then *>(ipr_)->condition()
             : static_cast<const ipr::If_then_else *>(ipr_)->condition());
}
```

(The actual Filter implementation pushes the type test earlier and catches errors.) Importantly, inside `discern`, the Filter node that gets created does not immediately lower *its* children. This means that the only lowering performed by calling `If::condition` is on the IPR node[s] at the root of `If`'s condition. In this way, the Filter library acts as a tool for selective, on-demand lowering.

Another, more figurative, way to picture the on-demand lowering done by the Filter library is to imagine a boulder being transported over land by rolling it over logs. Here, only a small number of logs are needed to cover large distances by reusing the same logs. Going back to Filter, the boulder is the analysis, the ground is the graph of IPR nodes, and the logs are the memory used by the Filter library. Filter nodes are thus represented by a *single use* of a log to carry the boulder a short distance. Just like how, when the boulder moves past a log, the log can be reused to cover new ground by placing it in front, when an analysis no longer needs a given Filter node, it can be deleted and the memory reused to construct new Filter nodes. By keeping Filter nodes on the stack, this is the strategy used in the examples throughout this thesis.

2. Performance

While the on-demand model addresses the goals of being lightweight, it appears to do so at the cost of performance. Specifically, if the Filter user creates and destroys several Filter nodes from the same IPR nodes, a lowering cost is incurred each time. For example, consider the following algorithm which counts the number of variables in a list that are objects of a given class:

```
int count_class_objects(Class c, const vector<Loc_var> &vars)
{
  int ct = 0;
  for (auto i = vars.begin(); i != vars.end(); ++i)
    if (i->type().which() == type_ctx::udt_use_e &&
        i->type().get<Udt_use>().which() == Udt_use::class_e &&
        i->type().get<Udt_use>().the_class().ipr() == c.ipr())
      ++ct;
  return ct;
}
```

Here, each expression "`i->type()`" invokes the lowering of the same IPR nodes. That means that each iteration calls `ipr::Var::type` followed by `type_ctx::discern` three times! Because, as described on pg. 59, several IPR nodes are used to represent the use of a user-defined type, and each node access implies a virtual function call, we can expect this sloppiness to more than double the running time of the algorithm. Thus, a more efficient approach is to hold on to the `type_ctx::Variant`:

```
int count_class_objects(Class c, const vector<Loc_var> &vars)
{
  int ct = 0;
  for (auto i = vars.begin(); i != vars.end(); ++i) {
    type_ctx::Variant tv = i->type();
    if (tv.which() == type_ctx::udt_use_e &&
        tv.get<Udt_use>().which() == Udt_use::class_e &&
        tv.get<Udt_use>().the_class().ipr() == c.ipr())
      ++ct;
  }
  return ct;
}
```

As was briefly mentioned on pg. 22 and will be described on pg. 81, `get<Udt_use>` is very fast, so its redundancy is negligible.

As this example shows, the value semantics of Filter nodes are crucial to allowing the user to extract good performance from the Filter library. Unfortunately, it is up to the user to detect logically redundant sources of lowering. Even assuming user cooperation, though, the Filter still needs to take extra steps to avoid performance pitfalls.

The first case to guard against is unnecessary node access. To achieve a binary interface, the IPR hides all fields behind virtual functions. Thus, if the Filter accesses fields that would not have been needed in a traversal without the Filter, the overhead can be significant. Consider the implementation of `filter::Return`, whose abbreviated interface is:

```
struct Return : Normal_node<ipr::Return> {
  bool has_expr();
  expr_ctx::Variant expr();
};
```

There is a temptation to check for `ipr::Phantom`, as described on pg. 63, in the `Return` 's constructor. This would allow the result to be computed and stored once, rather than accessing `ipr::Return::value` in *both* `has_expr` and `expr`. However, if the Filter user does not care about the returned expression—perhaps they are only looking for local variables—then the Filter will have imposed the cost of an extra virtual function call unnecessarily on any such analysis that creates a `Return` node, but does not call `has_expr` or `expr`.

The solution to the `Return` question, and the many situations like it, is to use lazy evaluation and caching. That is, avoid any up-front field access in the constructor, but also avoid duplicate field access in members by caching the results of each member's query. For example, applying this strategy to `Return` yields:

```
class Return : Normal_node<ipr::Return> {
  mutable const ipr::Expr *value_;
 public:
  Return(Env_ct &e, Ipr_ct &i) : /* base constructor */, value_(0) {}

  bool has_expr() const
  {
    if (value_ == 0)
      value_ = &ipr().value();
    return value_->category != ipr::phantom_cat;
  }

  expr_ctx::Variant expr() const
  {
    if (value_ == 0)
      value_ = &ipr().value();
    assert(has_expr());
    return expr_ctx::discern(env(), *value_);
  }
};
```

Altogether, this strategy imposes a small overhead from the extra branching involved in cache checks but prevents the performance hits associated with unnecessary virtual function calls.

The other performance concern is dynamic allocation. While individual Filter node types can simply use member variables to store their data, the `Variant` and `Range` types need to store data of varying size: an object of a single type (e.g. `expr_ctx::Variant`) must store data describing one of many types (viz. all the types listed in `expr_ctx::Members`).

Making excessive calls to `malloc` and `free` is a simple way to dominate the execution time of an otherwise efficient library [68, 69]. To avoid this cost, a logical first attempt would be to make more sophisticated use of the in-place storage provided by member variables. This could take the form of a C++ `union` or just raw memory:

```
namespace expr_ctx {
  class Variant {
    char mem_[max<sizeof(Member), sizeof(Mem_call), ...>::result];
    ...
```

```
  };
}
```

Here, the `max` template meta-program is used to calculate the maximum memory required to store a fully constructed copy of whatever node ends up inside the variant.

While this approach escapes `malloc` and `free`, it has several drawbacks. First, the complexity of variants and ranges increases drastically. Second, for an average-sized node (around 5 words), much of the space in the variant is wasted since the variant must be large enough for the biggest node (around 11 words). The discrepancy for ranges is even worse (3 words compared to more than 10, for each iterator). If the user stores the resulting variant or range in a `std::vector`, as `Traverse` does, this waste would then be magnified.

To motivate the solution used by Filter, we can observe a few properties of the data that needs to be stored:

1. except for the lazy evaluation mentioned above, the data is read-only;
2. while there are many types allocated, they come only in a few fixed sizes; and
3. there is a high turn-over rate of objects.

Observation 1 immediately suggests a reference-counting scheme for variants and ranges. Thus, the variant copy operation can require only a few instructions, compared to the polymorphic clone required for in-place allocation. Observation 2 suggests a classic fixed-size pool allocator, like the one described in §19.4.2 of [70] and implemented in [71]. As shown in Chapter IV, these two techniques allow Filter to do limited dynamic allocation without significant overhead.

On the other hand, in a concurrent environment, both of these techniques can create race-conditions in seemingly race-free user code. Fortunately, both techniques can be made thread-safe without adding a large synchronization overhead. For ref-

erence counting, this issue is well-understood from copy-on-write implementations of C++ `string` classes [72]. One low-cost solution is to use atomic increment and decrement primitives, as described in [73]. This does not, however, synchronize the lazy-cache update operations. For pool allocation, a more sophisticated algorithm may be used to avoid locking the entire pool during allocation and deallocation. Examples include the *slab* allocator [74] used in Linux and the *magazine* allocator [75] used in GNOME. However, neither of these options have been tested for this thesis.

D. Type discovery

A key use case of the Filter library is the case analysis commonly found in syntax-directed algorithms like type systems [30] and structured flow analysis [76]. In any heterogeneous data structure, a necessary part of the case analysis is discovering the type of a node. The Filter library's solution to the type discovery problem is described in Chapter II and revolves around the use of `Variant` types. This section describes why this solution was chosen from among the alternatives and how variants are implemented.

1. Alternatives

Although it can take many literal forms, the type discovery problem always starts the same way: the program follows a reference from a source node to a target, and the target node does not have a single, statically-determined type. This can be seen by following the path from an `ipr::Namespace` to its members:

```
const ipr::Namespace &ns = ...
ipr::Sequence<const ipr::Decl> &decls = ns.members();
for (int i = 0, sz = decls.size(); i != sz; ++i) {
  const ipr::Decl &d = decls[i];
  ... now what?
}
```

Hence the question is: once we have `d`, what can we do with it?

By nature of being a typed abstract-syntax graph, the IPR gives us a partial answer. Looking at the signature of `ipr::Decl`, we can see members like `name`, `type`, and `initializer`, so we can call those directly. However, none of these operations, or the types they return, reveal to us what exactly we have found in the global namespace. However, this established commonality between all nodes, represented by `ipr::Decl`, can be useful. For example, if we were simply looking for all entities named "foo," we would be satisfied knowing only that we had found an `ipr::Decl`.

In general, though, we need to find out more about the node pointed to by `decl`, which means first discovering its most-derived [public interface] type. One way is to use the built-in C++ `dynamic_cast` operation:

```
if (const ipr::Fundecl *fd = dynamic_cast<const ipr::Fundecl *>(&decl))
  ... now we can use fd
```

However, this is not an efficient route when performing many such tests in a row. This is common for many analyses, like type systems, that intend to analyze *every* case:

```
if (const ipr::Fundecl *fd = dynamic_cast<const ipr::Fundecl *>(&decl))
  ... now we can use fd
else if (const ipr::Typedecl *td = dynamic_cast<const ipr::Typedecl *>(&decl))
  ... now we can use td
else if (const ipr::Var *v = dynamic_cast<const ipr::Var *>(&decl))
  ... now we can use v
```

What we would rather do is ask a node "what type are you?" instead of asking "are you an X," for every X. With some indirection, the Visitor pattern [41] lets us do exactly that. To receive the answer, we write a class that overrides one member function for every possible answer:

```
struct Question : ipr::Visitor {
  void visit(const ipr::Fundecl &fd) { ... now we can use fd }
  void visit(const ipr::Typedecl &td) { ... now we can use td }
  void visit(const ipr::Var &v) { ... now we can use v }
```

```
};
```

Note that `Question` must derive `ipr::Visitor` so that we can ask our specific question through the node's general interface. In Pivot, the question is asked with the `accept` member of any IPR node:

```
Question q;
decl.accept(q);
```

From inside `accept`, the node calls the appropriate `visit` overload of `Question`, based on the most-derived IPR type.

Despite the more convoluted route, the Visitor pattern solves the problem efficiently. The reason is that, no matter how many different `visit` overloads are present, `accept` only costs 2 virtual function dispatches. In general, this is faster than a single `dynamic_cast`, thus, the Visitor method outperforms `dynamic_cast` even when only a single test is needed. Furthermore, in the specific case of Pivot, the deep single-inheritance of IPR nodes (much of it in the implementation) makes the performance advantage of the Visitor method even greater. This is due to the fact that the running time of `dynamic_cast` can be observed (in GCC [49]) to be proportional to the distance between the source type of the cast and the *most-derived* type of the object.

Unfortunately, visitors can be awkward to use for several reasons. Underlying all these reasons is the Inversion of Control (IoC, also known as the "Hollywood Principle" as in, "Don't call us; we'll call you" [36]) required by the Visitor pattern. While IoC is often used in object-oriented frameworks to protect invariants, protocols, or other high-level properties encapsulated by the framework, these advantages do not transfer over to type discovery, while the awkwardness does.

A first source of frustration appears if the operations inside a visitor are part of a larger algorithm. In this case, the algorithm's state must be "threaded" into and out of the visitor. For example, to implement the following pseudo-code:

```
int print_var_names(std::ostream &os, const ipr::Sequence<ipr::Decl> &seq)
{
  int num = 0;
  for (int i = 0, sz = seq.size(); i != sz; ++i)
    if (... seq[i] is an ipr::Var ...)
      if (... seq[i].name() is an ipr::Identifier ...) {
        os << seq[i].name().string() << ' ';
        ++num;
      }
  return num;
}
```

two visitors are needed, one for each type-discovery question:

```
struct Var_id_vis : ipr::Visitor {
    std::ostream &os;
    int &num;
    Var_id_vis(std::ostream &o, int &n) : os(o), num(n) {}
    void visit(const ipr::Var &v) { v.name().accept(Id_vis(os, num)); }
};

struct Id_vis : ipr::Visitor {
    std::ostream &os;
    int &num;
    Id_vis(std::ostream &o, int &n) : os(o), num(n) {}
    void visit(const ipr::Identifier &id) { os << id.string() << ' '; ++num; }
};

int print_var_names(std::ostream &os, const ipr::Sequence<ipr::Decl> &seq)
{
  int num = 0;
  for (int i = 0, sz = seq.size(); i != sz; ++i)
    seq[i].accept(Var_id_vis(os, num));
  return num;
}
```

Furthermore, the local state used in the inner loop must be manually threaded into both visitors so that `Id_vis` can perform the print and increment operations.

Of course, this code is not entirely realistic since it does not use any of the abstractions that immediately leap to mind; pg. 153 describes an entire line of Visitor-based research that minimizes the amount of code the user must write. However, even with these developments, the IoC remains and thus some amount of threading

is required.

Another problem is the fragmentation of logic that results from application of the Visitor pattern. To see this we need only compare the pseudo-code with the implementation using visitors. This effect can be further seen throughout the tests described in Chapter IV.

The last problem with visitors is that they make it hard to be in two places at the same time. That is, they favor analyses where there is only a single focal point that moves through the program. As a counter-example, consider an analysis which compares user-defined types to see if they are compatible, in some sense like structural subtyping [30] or the C++ One Definition Rule ([32], §3.2). Without a language feature like multi-methods [77] or an explicit design emulating multi-methods, the Visitor pattern cannot visit two trees simultaneously. This assumption tends to be enforced by abstractions built on top of the Visitor pattern such that explicit support, like the `Data.Generics.Twins` module in SYB [60], is required for simultaneous traversal. A clear example of this effect can be seen in the layout-compatible test on pg. 134.

For all of these reasons, the Filter library went instead with the type-switch design that has already been presented in Chapter II. However, one downside of this design choice is that, without language support, the type-switch offered by the Filter cannot catch a specific type of error at compile time. Specifically, if the user makes the following error:

```
func_ctx::Variant v = ...
switch (v.which()) {
  case func_ctx::loop_e: {
    If i = v.get<If>(); // v holds a Loop, not an If!
    ...
  }
}
```

the Filter can only report a run-time assertion. In contrast, the language-supported type-switches described on pg. 146 both catch the error and allow the `switch` to be written more concisely. However, as far as programmer errors go, this one is fairly easy to catch due to its locality.

2. Implementation

As mentioned on pg. 22, the `Variant` members `get` and `which`, as well as copy and assignment, are meant to be efficient operations. To implement this, a `Variant` holds a pointer to a fully-constructed node. This node is dynamically allocated and managed by `Variant` using the techniques described on pg. 77, namely reference counting and pool allocation.

The smart pointer `filter::detail::shared_node` is used to encapsulate the management of nodes contained in variants and allow RAII techniques ([70], §14.4.1). To save memory, the smart pointer embeds the reference count in the node. However, to avoid imposing this overhead on Filter nodes that are not stored inside a `Variant`, this is done non-intrusively, as shown in Figure 11. By controlling construction, `shared_node` is able to embed the node in a composite that also contains the reference count. The pool allocation is handled by deriving `pool_alloc_base`, which overloads `new` and `delete` to allocate from the pool, instead of `malloc` and `free`.

The `Variant_node` base is derived by all Filter node types that can go into a `Variant`. It adds a small virtual interface (needed for the implementation described on pg. 88) and three members that, for performance reasons, are kept non-virtual:

```
class Variant_node {
  Env_ct *env_;
  const ipr::Expr *ipr_;
  any_ctx::Members node_type_;
 public:
  virtual ~Variant_node() {}
```

```
class shared_node {
  struct ref_ct_base : util::pool_alloc_base {
    virtual ~ref_ct_base() {}
    unsigned int ct;
    Variant_node &node;
    ref_ct_base(Variant_node &vn) : ct(1), node(vn) {}
  };

  template <class T>
  struct ref_ct_wrap : T, ref_ct_base {
    ref_ct_wrap(const T &t) : T(t), ref_ct_base(*this) {}
  };

  ref_ct_base *ptr_;

 public:
  template <class T> shared_node(const T &t)
    : ptr_(new ref_ct_wrap<T>(t)) {}
  ... copy, assignment use normal intrusive ref-counting
  Variant_node &operator*() const { return ptr_.node; }
};
```

Fig. 11. A smart pointer class for non-intrusive embedded reference counting

```
  ... virtual interface
  ... inline accessors of the three member variables
};
```

The most important of these fields is `node_type_`, whose value is returned by `Variant ::which`. Note, however, that `node_type_` is of type `any_ctx::Members`, while each `X_ctx::Variant` will return `X_ctx::Members`. To allow this problem to be solved by a simple `static_cast`, all enumerators of `X_ctx::Members` are carefully defined to have the same values as the corresponding enumerators in `any_ctx::Members`. Using this, a runtime-checked `Variant::get` can be implemented efficiently:

```
template <class T> void Variant::get()
{
  assert(node_->node_type() == any_ctx::node_type_to_code<T>::result);
  return static_cast<const T &>(*node_);
}
```

Here, `node_` is the sole member of `Variant` and has type `shared_node`. The "node

type to code" template meta-function is a simple utility for mapping from types to their `any_ctx::Member` value.

Since `get` and most other members of `X_ctx::Variant` are the same, they are factored into a `Variant_base` class. In fact, only one important job is done in each `Variant` class: provide constructors for each type of node, as already mentioned on pg. 48. These constructors simply forward the parameter to `shared_node` member's constructor. Altogether, `Variant` types are little more than smart-pointer wrappers with a few additional operations that catch static and dynamic misuse.

E. Implementing ranges and iteration

This section concludes the chapter by describing the implementation of iterators and ranges in the Filter library. This is the most complicated part of the implementation due to the list of simultaneous constraints and requirements accumulated by the preceding chapters. Summarizing this list:

1. Ranges and iterators must fit in with the overall goal of being lightweight and efficient, as described on pg. 74.
2. The elements of a Filter sequence must be describable by a transformation of several underlying IPR sources, such as the lowering described on pg. 59.
3. Filter sequences must be able to describe the arbitrary concatenation of IPR sequences and IPR node singletons, such as the children of `Mem_call` described on pg. 63.
4. Several elements of a Filter sequence must be able to be extracted from a single underlying IPR node, such as from an `ipr::Labeled_stmt`, as described on pg. 71.

To describe the solution to these goals, the rest of this section is broken into subsections describing the individual parts of the implementation.

1. Overview of sequence composition

To motivate the implementation strategy used, we consider how the `Range` objects returned by polymorphic child-functions are created. As described on pg. 86, a `Variant` is just a thin wrapper around a `Variant_node` pointer. Therefore, the implementation of `Variant::child_x` simply calls a corresponding virtual function in `Variant_node` which gets overridden by each concrete node type. Thus, each node is responsible for listing its own children.

With this design, we now consider how the node types build `Range` objects. To demonstrate the issue, consider `filter::Loop`, whose abbreviated signature is shown here:

```
struct Loop {
  enum Which_init { none_e, expr_e, loc_vars_e };
  Which_init which_init() const; // what type of initializer is used?

  expr_ctx::Variant init_expr() const; // if which_init == expr_e
  Var_range init_loc_vars() const; // if which_init == loc_vars_e

  bool has_cond() const; // does the loop have a condition?
  bool cond_is_var() const; // if so, does condition declare a variable?
  Loc_var cond_var() const; // if so, what is the variable?
  expr_ctx::Variant condition() const; // what expr is tested?

  func_ctx::Range body() const;

  bool has_inc() const;
  expr_ctx::Variant inc() const;
};
```

Looking at this signature, we can see several potential children. For `child_expr`, `Loop` should return an `expr_ctx::Range` containing `init_expr`, if `which_init` is `expr_e`, `condition`, if `has_cond` is `true`, and `inc`, if `has_inc` is `true`. The `func_ctx::Range` returned by `child_func` should contain the variables in `Var_range`, if `which_init` is

`loc_vars_e`, the single variable `cond_var`, if `has_cond` and `cond_is_var` are both `true`, and `body`.

So, how are these `Range` objects built? The first part of the answer is: by composing sequence primitives. For the current Filter implementation, these primitives are:

- a single IPR node,
- a sequence of IPR nodes held by an `ipr::Sequence`,
- a sequence of IPR nodes represented by a dispatch table,
- a sequence of IPR nodes extracted from an `ipr::Stmt` (pg. 71).

Here, *dispatch table* refers to a static array of member-function pointers which, when applied to an object, yield the desired sequence. For example, we can associate the following array with the node `ipr::Plus`:

```
[ &ipr::Plus::first, &ipr::Plus::second ]
```

so that, for a *particular* `ipr::Plus` node, we can get its members by calling the members in the list. Why go through all this trouble? The reason is that these static arrays can be generated *automatically* for any `ipr::Classic` node by a template metaprogram that uses the IPR types `Unary`, `Binary`, and `Ternary` to provide compile-time reflection. This allows us to write a utility function:

```
typedef const ipr::Expr &(* const Dispatch_entry_t)(const ipr::Classic &);
struct Dispatch_table {
  Dispatch_entry_t (&arr)[];
  int arr_sz;
};

Dispatch_table get_dispatch_table(const ipr::Classic &);
```

which takes an `ipr::Classic` and returns an array of pointers to functions which, when given *the same* `ipr::Classic` node, return its children.

With the abovementioned sequence primitives, the next question is how to represent these primitives and their composition in a data structure. An initial idea is to use template meta-programming to generate, for each list of primitives, a custom class representing the composition. For example, assuming `Single` and `Sequence` are the types of sequence primitives and `Compose` is a template meta-program, we could write the following to refer to the composition of two single IPR nodes and one IPR sequence:

```
Compose<Single, Single, Sequence>::result
```

The `result` type could then be allocated with exactly the storage required and with a static knowledge of the composed sequence.

The problem with this approach is that it forces all composition to occur at once. Thus, for any composed sequence with conditionally included members (e.g., the children of `Loop` described above), every different combination of primitives must use a different type. This can lead to a combinatorial explosion of branching in the implementation. This effect is particularly pronounced in the implementation of `child_any`, which conditionally combines children based on the `Prune` parameter.

To overcome this inflexibility, the Filter library composes sequence primitives dynamically using the data structure shown in Figure 12. In this data structure, each sequence primitive is a separately-allocated *segment* that is composed with other segments in a doubly-linked list. The whole composition is contained by a *rope*. Regarding resource management, the iterators are managed by the user, usually being passed by value as part of a `Range`; ropes are pool-allocated and reference-counted by the iterators; and segments are pool-allocated and managed exclusively by a single rope.

Ropes are implemented by the `filter::detail::Rope` class, and are fairly simple: they contain only the first/last pointers shown in Figure 12 and an embedded reference

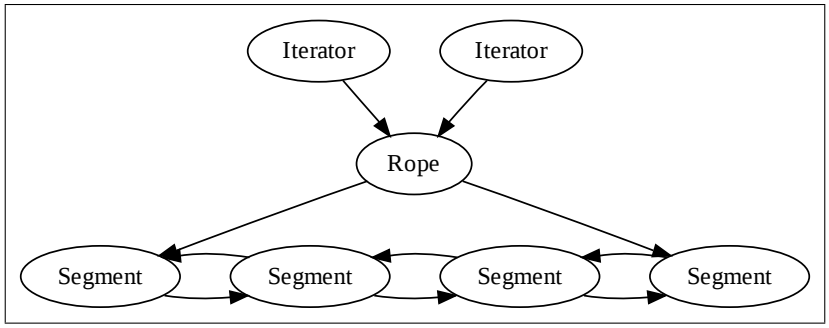

Fig. 12. The `Rope` data structure

count used by the iterators. The key member function provided by `Rope` is `append`, which takes ownership of a dynamically-allocated `Segment` by adding it to the end of the linked list. With this, polymorphic child-functions can be implemented by having nodes append their custom segments. For example, the following code builds the rope for the `func_ctx::Range` returned by `child_func` on a `Loop`:

```
void Loop::append_func(detail::Rope &r) const
{
  if (which_init() == loc_vars_e)
    r.append(new Seq<...>(...)); // append loop init vars
  if (has_cond() && cond_is_var())
    r.append(new Single<...>(...)); // append loop condition vars
  r.append(new Single<...>(...)); // append loop body
}
```

The segments `Seq` and `Single` will be explained in the following subsections. Note that the logic in `append_func` closely matches the earlier informal description of what `Loop`'s children should be.

As shown in the above code for the function context, there are a set of virtual `append_X` functions, for each context `X`, in the `Variant_node` base class that are overridden by each node. These `append_X` functions can then be used to implement the `Variant::child_X` member functions as follows:

```
func_ctx::Range Variant::child_func() const
{
```

```
  Rope r(...);
  node_->append_func(r);
  return range_from_rope<func_ctx::Iter>(r);
}
```

Here, `Rope` is first allocated on the stack and filled with dynamically-allocated segments. If the `Rope` is not empty, `range_from_rope` will dynamically allocate a `Rope` and have it "steal" (or, in C++0x terms, "move construct" [78] from) the segments of the stack-allocated rope. This new `Rope` will then be used to initialize the returned `Range`. Otherwise, if the stack-allocated `Rope` is empty, `range_from_rope` will call a special `Range` constructor that avoids the need for a dynamically-allocated `Rope` altogether.

With this organization, `append_any` can be implemented as follows:

```
void Loop::append_any(Rope &r, Prune p) const
{
  if (p.func)
    Loop::append_func(r);
  if (p.expr)
    Loop::append_expr(r);
}
```

Notice, though, that `append_any` is implemented for each specific node. This is an optimization, made at the cost of extra boilerplate: `Loop::append_any` can call the other `append_x` functions statically. If all these `append_any` calls were factored out into `Variant::child_any`, *each* call to `append_X` would have to be dynamically dispatched.

The other parts of Figure 12 are discussed in the next subsections.

2. Segment interface / move algorithm

More important than the implementation of iterators and segments is the interface between them, which is key to minimizing the number of virtual dispatches to the polymorphic segment while supporting all the features listed at the beginning of this section. Overall, there are two types of operations needed: creating iterators at the

beginning/end of a range, and increment/decrement operations. The challenge with implementing these operations is that, for reasons like those described on pg. 72, each segment represents a lazily-filtered sequence, not unlike the Boost `filter_iterator` [33]. This means that each operation requires a subsequent "settling" phase, to place the iterator on an un-filtered element, which must work correctly between segments.

To support the above usage, `Segment` provides the following five operations:

```
class Segment {
 public:
  Segment *next() const;
  Segment *prev() const;
  int pre() const;
  int post() const;
  virtual shared_node move(Env_ct &, int &i, bool forward) const = 0;
};
```

Notice that the first four members are non-virtual—in fact, they are inline functions. The `next` and `prev` members are the links in the doubly-linked list of segments. The list is acyclic, so these members return `0` when at the ends. The `pre` and `post` operations produce indices that are one-past-the-beginning and one-past-the-end, respectively.

The last operation, `move`, merges increment/decrement and settling into a single operation. The `Env_ct` reference is needed for the underlying calls to `discern`. The `i` parameter is the index into the segment and, as a precondition, must be in the range `[pre, post]`. Note that `i` is passed by mutable reference, which allows `move` to change the index as it skips over IPR nodes in the aforementioned settling process. The `forward` parameter indicates whether to increment or decrement and, when settling, which direction to settle. Lastly, the return value is the smart pointer introduced on pg. 86. This smart pointer has a testable null value; returning this null value is how `move` indicates that a valid Filter node was not found.

Using this interface, we can build an algorithm to "lift" `Segment::move` to the

```
template <class Op>
  Segment *Rope::move(Segment &first_seg, int &i, Op found, bool forward)
{
  for (Segment *seg = &first_seg; true; )
    if (shared_node n = seg->move(env_, i, forward)) {
      found(n);
      return seg;
    }
    else
      if (forward)
        if ((seg = seg->next()) != 0)
          i = seg->pre();
        else
          return 0;
      else
        if ((seg = seg->prev()) != 0)
          i = seg->post();
        else
          return 0;
}
```

Fig. 13. An algorithm to lift `move` from segments to ropes

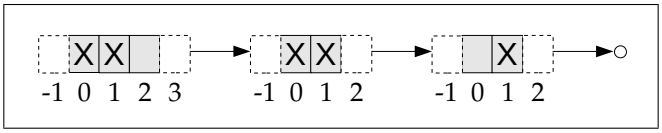

Fig. 14. Example rope for the `move` algorithm

entire rope. This algorithm is a member of `Rope` and shown in Figure 13. We first consider its signature:

```
template <class Op>
  Segment *Rope::move(Segment &first_seg, int &i, Op found, bool forward);
```

The `first_seg` argument is the segment indexed by `i`. The `found` parameter is a callable value that receives a valid element, if one is found, in the form of a non-null `shared_node`. If `found` *is* called, the return value of `move` is a pointer to the segment containing the node and `i` is its index. On the other hand, if no node was found, the return value is `0` and `i` should be regarded as garbage.

From this specification and the interface provided by `Segment::move`, the implementation in Figure 13 follows directly. To illustrate the combination of `Rope::move` and `Segment::move`, we now consider the example rope in Figure 14. To abstract from the implementation of segments, we will view segments simply as linear sequences of "good" and "bad" elements, with bad elements labelled "X." The numbers under each element represent their index in the segment. Thus, numbers under the dotted boxes are the values of `pre` and `post` for the segment. The arrows between segments are the `next` pointers—`prev` pointers have been elided.

We start by calling `Rope::move`, passing the leftmost segment for `first_seg` and `i` set to "`first_seg.pre()`", which is `-1`. Thus, this call represents a request for the first non-filtered element of the sequence. Inside, the call to "`seg->move`" returns a non-null `shared_node` and sets `i` to `2`. Thus, `Rope::move` returns a pointer to the first

segment, keeping `i` set to `2`.

Now we call `Rope::move` again, passing the `Segment` returned from the last call and the same `i`. Inside, the first call to "`seg->move`" reaches the end of the segment and returns null. The algorithm then moves to the next segment and resets `i` to its `pre` value, which is `-1`. This segment also fails to find any valid element, returning null. On the third iteration, a valid element is found and so `Rope::move` returns a pointer to the third segment with `i` set to `0`.

Calling `Rope::move` a third time will first fail to find an element with "`seg->move`" and then fail to find a next segment with "`seg->next`". Thus, `Rope::move` will return `0` and leave `i` in an undefined state.

Altogether, these three calls to `Rope::move` reveal the three good elements distributed through the segments. While "good" here is an abstract property, pg. 102 describes how this abstract interface and algorithm give segment implementations enough flexibility to realize the rest of the required functionality.

3. Iterator implementation

We now consider just the iterator side of the interface described on pg. 93 and how `Rope::move` can be put to work. Before going any further, though, we have to address a problem that appears for any iterator that dynamically creates its elements: how to implement the dereference operations (`operator*` and `operator->`).

The normal strategy for these operations is simply to return a reference or pointer to the element in the underlying data structure. However, as described on pg. 74, there are no permanent elements to which a reference can be made. Returning by value for these operations is also not an option: for `operator*`, it is inefficient, and for `operator->`, even using the proxy pattern, a value type prevents the iterator from being used to build composite iterators, like the `Traverse` class from the tutorial.

One option would be to use an iterator's rope to store permanent elements since, because of reference counting, ropes live longer than their iterators. However, this would either require creating all these elements up-front, breaking the lazy evaluation strategy of the Filter library (pg. 77), or creating them as needed, which would require a dynamic data structure to hold the elements. Either way would make ropes bigger and slower.

The alternative used by Filter is to store a single element inside the iterator itself. Because of the compact representation of variants (a single pointer), the space required is minimal. With this in-place element, the dereference operations can simply return references/pointers. The downside, however, is that these references will be invalidated after any operation that changes the iterator's current element. Thus, we can see the reason behind the limitation, described on pg. 29, that a reference should not be stored to the current element of an iterator.

Since the iterators for each context differ primarily in the type of variant returned, iterators are implemented by a class template:

```
namespace filter {
  namespace detail {
    template <class Policy>
    class Rope_iter {
      typedef typename Policy::Variant_t Variant_t;
      ...
      Variant_t &operator*() const;
      Variant_t *operator->() const;
    };
  }
}
```

Here, `Policy` is a collection of types and functions used by iterators and segments to capture the variation between the contexts.

Inside `Rope_iter`, only four words are required to maintain the state of the iterator, making it the same size as a `std::deque` iterator:

```
Rope_ptr rope_; // reference-counting smart pointer
Segment *seg_; // current segment of the iterator
int i_; // current index within the current segment
Variant_t elem_; // current element
```

The `Rope_ptr` type is a utility class that provides intrusive reference counting and allows RAII techniques ([70], §14.4.1). The `seg_` member is `0` when the iterator is one past the end of the sequence. When `seg_` is not `0`, `elem_` holds the last valid element found and `i_` holds its index. Otherwise, when `seg_` is `0`, these two variables should not be used except for destruction.

With these members, copy construction, assignment, and destruction can all use the compiler-generated defaults. Construction of iterators is implemented as follows, using the empty `End_t` type as a flag to construct the end iterator:

```
Rope_iter(Rope &r) : rope_(r), seg_(0)
{
  if (!r.empty()) {
    i_ = rope_->first().pre();
    seg_ = rope_->move(rope_->first(), i_, Assign_to(elem_.node_), true);
  }
}

Rope_iter(Rope &r, End_t) : rope_(r), seg_(0) {}
```

Here we see code corresponding to the example described at the end of pg. 93: to settle on the first valid element, `i_` is set to the element *before* the first element of the first segment and then the `move` operation is called. When and if a valid element is found, `Assign_to` is the function object, substituted for `found` in Figure 13, which will be called. Its implementation is simply:

```
struct Assign_to {
  shared_node &sn_;
  Assign_to(shared_node &sn) : sn_(sn) {}
  void operator()(const shared_node &new_node) { sn_ = new_node; }
};
```

Thus, valid nodes are assigned to the smart pointer inside the iterator's internal

`Variant`. Increment is even simpler:

```
Rope_iter &operator++()
{
  assert(seg_ != 0);
  seg_ = rope_->move(*seg_, i_, Assign_to(elem_.node_), true);
  return *this;
}
```

Decrement is a bit more complicated, since it has to handle decrements starting at the one-past-the-end state, similar to how the constructor handles starting at the beginning:

```
Rope_iter &operator--()
{
  if (seg_ == 0) {
    assert(!rope_->empty());
    seg_ = &rope_->last();
    i_ = seg_->post();
  }
  seg_ = rope_->move(*seg_, i_, Assign_to(elem_.node_), false);
  return *this;
}
```

In addition to iterating over ropes, Filter iterators also provide two special cases: iterating over a singleton sequence, and iterating over an empty sequence. These special cases forego the need for a dynamically allocated rope and thus offer optimizations like the one in `range_from_rope` described on pg. 89. To implement this, we can reuse the space used by `seg_`, since it is no longer required. Specifically, we replace `seg_` with a union member containing a segment pointer:

```
union Rope_iter_union {
  Segment *seg_; // normal case
  struct {
    bool single_ : 8; // singleton or empty sequence?
    bool end_ : 8; // if singleton sequence, at beginning or end?
  } s;
} u;
```

Here, bitfields are used so that the two `bool` members fit into the single word used by `seg_`.

Since C++ unions are not discriminated, we need some way to tell which union member to use. We find this discriminant in `rope_`, which we set to `0` when there is no rope. For the empty sequence, this is all the state needed. However, for the singleton sequence, we need somewhere to store the single element. For this, we can use `elem_` (further supporting the earlier decision to use in-place storage).

The constructors that initialize iterators for these two cases simply set `rope_` to `0` and set the members of `u.s` accordingly. For the singleton constructor, the single element must be passed to the constructor for both the begin and end iterators. Additionally, the increment and decrement operations are modified to toggle `u.s.end_` when `rope_` is `0`.

With this new conditional state, one operation that becomes significantly more complicated is equality:

```
bool operator==(const Rope_iter &rhs) const {
  return rope_ ? (u.seg_ == rhs.u.seg_ && (i_ == rhs.i_ || u.seg_ == 0))
               : (u.s.single_ == rhs.u.s.single_ &&
                  (!u.s.single_ || u.s.end_ == rhs.u.s.end_));
}
```

Since `Rope_iter` is an implementation template, it is brought to the public interface through a set of typedefs:

```
namespace filter {
  namespace ns_ctx {
    typedef detail::Rope_iter<detail::Ns_policy> Iter;
    typedef Range<Iter> Range;
  }
  namespace uda_ctx {
    typedef detail::Rope_iter<detail::Uda_policy> Iter;
    typedef Range<Iter> Range;
  }
  ...
}
```

Here, `X_policy` contains a typedef declaring `Variant_t` to mean `X_ctx::Variant`.

Finally, we return to the implementation of the `range_from_rope` template func-

tion, mentioned on pg. 89 and used throughout the implementation of Filter nodes:

```
template <class Iter>
inline Range<Iter> range_from_rope(Rope &local_range)
{
  if (local_range.empty())
    return Range<Iter>(); // empty-range optimization

  Rope *new_range = new Rope(local_range.env()); // safe: no throw below
  new_range->steal(local_range);
  return Range<Iter>(Iter(*new_range, false), Iter(*new_range, true));
}
```

Thus, `range_from_rope` abstracts the creation of iterators in order to use the above empty-range constructor when possible.

4. Segment implementation

The last piece of range iteration to describe is the implementation of segments. As with iterators, there are many variations on segments differing only superficially in the types used, so we start with two general segment templates:

```
template <class Policy, class Ptr> class Single;
template <class Policy, class Ptr> class Seq;
```

The `Policy` parameter here has the same meaning, and will receive the same arguments, as the `Policy` parameter in `Rope_iter`. The second parameter, `Ptr`, is used to abstract the type of the underlying pointer-like object given to the segment. Some common segments are:

```
Seq<Ns_policy, const ipr::Sequence<ipr::Decl> *> // namespace members
Seq<Uda_policy, const ipr::Sequence<ipr::Decl> *> // class/union members
Single<Expr_policy, const ipr::Expr *> // sub-expression
Single<Type_policy, const ipr::Type *> // part of compound type
```

However, IPR types are not the only arguments passed to `Single` and `Seq`. To see the concepts (in the Standard C++ use of the word) required of `Ptr`, we can look at the implementations of `Single` and `Seq`, shown in Figures 15 and 16, respectively. At the heart of both classes' `move` implementations is a call to `Policy::discern`. With this,

each context's policy can ensure that the right `discern` function is called to interpret the raw IPR. In `Single`, this raw IPR is acquired by dereferencing `ptr_` (of type `Ptr`), and in `Seq`, it comes from dereferencing and then subscripting `seq_` (of type `Ptr`). Thus, these two operations are the valid use patterns required of `Ptr` in `Single` and `Seq`, respectively.

Looking at the type "`const ipr::Sequence<ipr::Decl> *`", it clearly satisfies the concept requirements of `Ptr` for `Seq`. However, Filter uses several other types to transform the sequence of raw IPR nodes fed to `discern`. To see a first example, consider calling `child_expr` on the `ns_ctx::Variant` containing:

```
enum E {
  A = 1,
  B = 5,
  C = 10
};
```

The returned `expr_ctx::Range` should clearly contain the expressions "`1`", "`5`", and "`10`". However, there is no `ipr::Sequence<Expr>` in the IPR corresponding to this list. However, there is an `ipr::Sequence<Enumerator>`, thus we can translate this into a sequence of `ipr::Expr` nodes with the following model of `Ptr`:

```
struct Enum_init_seq {
  const ipr::Sequence<ipr::Enumerator> &seq_;
  Enum_init_seq(const ipr::Sequence<ipr::Enumerator> &s) : seq_(s) {}
  const Enum_init_seq &operator*() { return *this; }
  const ipr::Expr &operator[](int i) { return seq_[i].initializer(); }
};
```

Thus, the "`(*seq_)[i]`" expression in `Seq::move` will yield the expressions "`1`", "`5`", and "`10`" instead of the enumerators "`A`", "`B`", and "`C`". `Enum_init_seq` can then be plugged into `Seq` by the Filter `Enum` class as follows:

```
void Enum::append_expr(Rope &r) const {
  const ipr::Sequence<ipr::Enumerator> &s = ipr().members();
  r.append(new Seq<Expr_policy, Enum_init_seq>(Enum_init_seq(s),
                                               s.size()));
}
```

```
template <class Policy, class Ptr>
class Single : public Segment {
    Ptr ptr_;
  public:
    Single(const Ptr &ptr) : Segment(1), ptr_(ptr) {}

    shared_node move(Env_ct &env, int &i, bool forward) const
    {
      assert(i >= this->pre() && i <= this->post());
      i += (forward ? 1 : -1);
      assert(i >= this->pre() && i <= this->post());
      if (i == 0)
        return Policy::discern(env, *ptr_);
      return shared_node();
    }
};
```

Fig. 15. The implementation of the `Single` segment

```
template <class Policy, class Ptr>
class Seq : public Segment {
    Ptr seq_;
  public:
    Seq(const Ptr &seq, int end)
     : Segment(end), seq_(seq) {}

    shared_node move(Env_ct &env, int &i, bool forward) const
    {
      int inc = forward ? 1 : -1;
      assert(i >= this->pre() && i <= this->post());
      i += inc;
      assert(i >= this->pre() && i <= this->post());
      for (; i > this->pre() && i < this->post(); i += inc)
        if (shared_node n = Policy::discern(env, (*seq_)[i]))
          return n;
      return shared_node();
    }
};
```

Fig. 16. The implementation of the `Seq` segment

```
class Args_with_defs_seq {
  const ipr::Sequence<ipr::Expr> &actuals_;
  int num_actuals_;
  const ipr::Sequence<ipr::Parm> &parms_;
public:
  ...
  const ipr::Expr &operator[](int i) const
  {
    if (i < num_actuals_)
      return actuals_[i];
    return parms_[i].default_value();
  }
};
```

Fig. 17. A model of `Ptr` that merges actual and default arguments

The transformation done by `Enum_init_seq` is fairly simple. A more interesting transformation, shown in Figure 17, is needed to merge default and actual arguments to support the lowering described on pg. 69. Transformations are also used to extract lists of types from lists of parameters, and lists of variables and blocks from lists of catch statements.

In some cases, however, segments cannot be generated by `Single` or `Seq` and require a custom segment type. One example are segments which represent sequences that are lazily generated from dispatch tables (as described on pg. 89). The other example is segments representing statements. Recall, from pg. 71, that Filter linearizes nested chains of `ipr::Labeled_stmts`. Implementing efficient bidirectional iteration over such statements, while not penalizing the average case, requires an unexpectedly complex solution.

The basic issue is how to map the integer index passed to `move` to a sequence of nested labeled statements. Normally, the integer can be translated into the index of an `ipr::Sequence`, but in this case, we need to handle the possible stack of statements nested inside each element of the `ipr::Sequence`. One solution would be to gener-

alize the iterator state passed to `move` so that, for the special case of iteration over statements, the segment would have additional state to work with. This direction was pursued initially, however, each approach penalized the average case performance and memory usage of `func_ctx::Iter` and `any_ctx::Iter`.

The solution to the problem used by the Filter library is rather involved and would not contribute much to the overall discussion, so only a rough sketch is given. The basic ideas are: first, have the statements' *segment* store an auxiliary data structure allowing any statement to be described by two indices; second, pack these two integers into the single integer given to `move` with bitwise operations. Thus, for a K-bit word, use L bits to store the `ipr::Sequence<ipr::Stmt>` index and K-L bits to store the index into the list of `ipr::Labeled_stmt` nested underneath. Assuming a maximum label-nesting depth is 511, on a 32-bit architecture this leaves 23 bits for the `ipr::Sequence` index. The auxiliary data structure is only created if a label is actually found during iteration, otherwise, the segment just keeps a null pointer indicating the absence of any auxiliary data. Thus, for the average case, there is no per-iterator space overhead and only a slight performance overhead for statement iteration due to the additional bitwise operations and logic.

5. Range optimization

We describe one last optimization of the range implementation that significantly reduces the amount of dynamic allocation and dynamic dispatch. Recall that the function `range_from_rope` delays the dynamic allocation of a `Rope` until it is known whether the `Range` will be empty or not. Experimentally, this has a big payoff, avoiding 1/3 to 1/2 of all dynamic rope allocations. While dynamic allocation is inexpensive using the techniques described on pg. 86, at the micro-scale compared in Chapter IV, it can still introduce a factor overhead.

To take the optimization further, recall from pg. 97 that singleton ranges can also be constructed without ropes. However, with the current design, by the time it is known that a range will only contain a single element, a segment has already been dynamically allocated. Thus, the solution is to again delay dynamic allocation until the last possible moment.

The first step is to abstract rope and segment creation behind a factory called, for brevity, the `Ropery`. The `Ropery` has the following interface:

```
class Ropery {
 public:
  // normal cases: add a single IPR node, or a sequence of them
  template <class Policy> void pt(const typename Policy::Ipr_t &);
  template <class Policy>
    void seq(const ipr::Sequence<typename Policy::Ipr_t> &,
             unsigned int seq_sz);

  // special case: add a custom segment
  void custom(Segment *);

  // done adding segments, create Range now
  template <class Policy> Range<Rope_iter<Policy>> finish();
};
```

The way this class works is as follows. A `Ropery` is like a construction bay for one `Range`. To add children to the `Range`, nodes call `pt` or `seq` and pass IPR nodes. For all the special cases already outlined, the segments can be created manually and passed through `custom`. When no more segments need to be added, `finish` is called to produce the `Range` to return to the user.

From the nodes' perspective, appending children via `Ropery` is a little easier than what was done on pg. 93. However, the `Ropery` is now in a position to easily do several effective optimizations. First, in its private data, the `Ropery` can store the last IPR node added. When `finish` is called, if only one IPR node has been added, `Ropery` can use the `Range` singleton constructor. This even works if the node calls `seq` passing an `ipr::Sequence` of size 1 or calls `pt` and then passes an empty `ipr::`

`Sequence`. In experiments, more than half of the `Range` objects created require no dynamically allocated `Rope` or `Sequence` objects by this small addition.

Going farther, the `Ropery` also coalesces individual calls to `pt` and `seq` into a single segment instead of allocating segments for each. To do this, `Ropery` keeps a fixed-size buffer of the K most recent IPR nodes passed via `pt` or `seq` in an array member variable. When the buffer overflows or `finish` is called, `Ropery` puts the contents of this buffer into a dynamically allocated segment. Thus, the two children of the `Mem_call` expression "`a->foo(x)`" will be placed into a single segment, instead of two separate segments, even though the receiver and arguments are added by two separate calls to `pt` and `seq`, respectively. In this way, `Ropery` further decreases the number of segments created.

CHAPTER IV

COMPARISON

Based on the introduction and description of the Filter library in Chapters II and III, this chapter presents an evaluation of the Filter library by comparing solutions to several test problems using both Filter and traditional Visitor-based designs. These tests measure the effects of both Filter lowering (pg. 59) and the Filter's interface—namely the choice of type switching (pg. 81) and the inclusion of polymorphic child-functions (pg. 32).

From the results described in this chapter, we can see that, for tiny problems that require little work outside the Filter library, the Filter library incurs a 2-4x performance penalty when compared to the analogous Visitor-based solution. In the worst case, where Filter's lowering computations are unnecessary, this overhead can be as high as 9x. However, for larger, less synthetic problems, this overhead is shown to drop below 2x. Finally, compared to an existing whole-program graph traversal library in Pivot, custom traversal with the Filter library shows up to two orders of magnitude speedup.

The chapter starts by describing the method of evaluation in §A, and then, in §B, goes through the six tests conducted in detail. Finally, §C pulls the test results together to make some overall conclusions.

A. Method

This chapter compares two approaches to program analysis in Pivot: the way supported by the Filter library and the traditional approach. The comparison takes into account both performance, measured on a variety of inputs, and code complexity, measured by Logical Lines of Code (LLOC). While LLOC is generally an inaccurate

measure of any software engineering quality, in the examples shown, the discrepancy between the solutions still makes a clear point. The rest of this section goes into detail about the approaches compared and how the measurements were made.

1. Characterization of the traditional approach

We now examine what is meant by "traditional approach." First, the traditional approach uses only the public IPR interface and pre-existing libraries in the Pivot framework. This means that, at the moment, the only traversals available are graph-based breadth- and depth-first visitors. These traversals are slow and, lacking any tree structure to provide a traversal context, of limited utility. Hence, only one test is included in the evaluation that requires whole-program traversal.

Other than the IPR public interface and graph traversal, the test cases use four small library utilities, which we introduce here. The first utility, `Noop_visitor`, is a simple class which derives `ipr::Visitor`, turning it from an abstract to concrete base class by overriding *all* `visit` functions with empty bodies:

```
struct Noop_visitor : ipr::Visitor {
  void visit(const ipr::Node &) {}
  void visit(const ipr::Annotation &) {}
  ... 154 more overrides doing the same
};
```

The second utility, `Visitor_to_overload`, is useful for building visitors based on overload resolution instead of type equality. For example, to write a visitor that looks at all `Classic` operations, one might try to write:

```
struct Classic_visitor : Noop_visitor {
  void visit(const ipr::Classic &c) { found_classic(c); }
};
```

Unfortunately, when visiting any derived `Classic` node, like `Plus`, the overload resolution inside `Plus::accept` will call the `Plus` overload of `visit`. One solution to

this problem would be to design a "percolate visitor" which, by default, recursively forwards calls to `visit(const X &)` to `visit(const B &)`, where `B` is the immediate base class of `X`. Thus, if `Classic_visitor` was derived from this percolate visitor, visiting `Plus` would percolate the call up until it reached the intended `visit(const ipr::Classic &)` overload.

The problem with this "pure" Visitor pattern solution is that each call to a less-derived `visit` overload requires a virtual dispatch. The `Visitor_to_overload` template class achieves the same effect without any additional virtual dispatch:

```
template <class Action>
struct Visitor_to_overload {
  Action act;
  Visitor_to_overload() : act() {}
  Visitor_to_overload(Action a) : act(a) {}
  virtual visit(const ipr::Node &n) { act(n); }
  virtual visit(const ipr::Annotation &n) { act(n); }
  ... 154 more overrides doing the same
};
```

Now, the `Action` type argument can use the C++ overload resolution rules, which include derived-to-base conversion:

```
struct Classic_action {
  void operator()(const ipr::Classic &c) { found_classic(c); }
  void operator()(const ipr::Node &) {}
};

const ipr::Expr &e = ...
Visitor_to_overload<Classic_action> vis;
e.accept(vis);
```

Notice here that `Classic_action` is not an `ipr::Visitor` and thus must provide an `ipr::Node` overload to catch all the non-`Classic` cases. While `Visitor_to_overload` is difficult to understand at first, it is an extremely useful utility.

The last two utilities simulate the C++ `typeid` and `dynamic_cast` operators by building IPR visitors to achieve the same effect:

```
template <class To> const To *ipr_dynamic_cast(const ipr::Node &);
```

```
template <class To> const To *ipr_typeid(const ipr::Node &);
```

In both cases, the `To` parameter is restricted to IPR public interface types. The first function asks whether the given `Node` derives `To`. The second function asks whether the given `Node` has `To` as its most-derived public interface. Both implementations require only a single visitation to resolve the query and thus outperform `dynamic_cast`. The disadvantage of using `ipr_dynamic_cast`, however, is that it uses `Visitor_to_overload` , which generates a lot of code through template instantiation, and can noticeably slow down compilation compared to `ipr_typeid`.

The second characteristic of the "traditional approach" is the use of the Visitor pattern to perform type discovery. The Visitor pattern is introduced and contrasted with the type-switch style of the Filter library on pg. 81. A key property of visitors mentioned on pg. 81 is their use of callbacks to safely control the type casting that must occur during type discovery. Thus, when the traditional approach is applied to problems, we try to maintain this property. However, *strict* adherence to this principle can lead to obfuscated code. For example, consider the following use of a visitor (through `ipr_dynamic_cast`):

```
const ipr::Expr &e = ...
if (const ipr::Id_expr *i = ipr_typeid<ipr::Id_expr>(e))
  examine(*i);
```

Although a visitor is used, this can be seen as breaking the above principle. A pure version takes the form:

```
struct Custom_id_expr_visitor : Noop_visitor {
  void operator()(const ipr::Id_expr &i) { examine(i); }
};

const ipr::Expr &e = ...
e.accept(Custom_id_expr_visitor());
```

Notice how the simple logic of the original text has been muddied. Furthermore, what would be a simple linear set of tests with the first style becomes an increasingly

nested call stack with the second. Since no reasonable engineer would write more than a few such tests the second way before reverting to the first, the traditional approach presented in this chapter goes the impure route in practical cases where a simple use of `ipr_typeid` or `ipr_dynamic_cast` will suffice.

Another avenue not explored in this chapter is the "visitor combinator" approach [79]. While this style can lead to highly reusable visitors and traversals, it does by using significantly more virtual dispatch and dynamic allocation. Instead, the visitors presented in this chapter attempt to minimize the amount of virtual dispatch required and exclusively place visitors on the stack. In this way, it is easier to observe the overhead of the Filter's lightweight design (pg. 74) without worrying whether the Filter's positive relative performance was due to its merits or the high price of reusability imposed by visitor combinators.

2. Performance test setup

All performance tests were conducted on a Pentium 4, 2.4 GHz with 512 MB RAM. The operating system distribution used was CentOS 4 running Linux kernel 2.6.18. The reported results are the average of three runs. All tests were run without any other user applications running to reduce background noise. The resulting variance is low compared to the measured difference between test cases.

All tests were built with GCC 4.1.2 using full optimization (`-O3`). Performance measurements were made using the `librt` library function `clock_gettime`. Each test run is wrapped in a function that takes an IPR node as input. This function is executed between 10K to 1M times per measurement to amortize the cost of measurement and background noise. To avoid synthetic speedups due to training of the processor's branch predictor, each iteration tests the function over a different IPR node. The resulting test harness takes the form (modulo IPR type):

```
template <void (*F)(const ipr::Expr &)>
void test(Result &r, int iters, const vector<const ipr::Expr *> inputs)
{
  ... begin measurement
  for (int i = 0; i != iters; ++i)
    for (int j = 0; j != inputs.size(); ++j)
      F(r, *inputs[j]);
  ... end measurement
}
```

Here, the `Result` value `r` holds data computed by `F` that is printed to the console after `test` returns. This way, the optimizer cannot decide to remove the call to `F`. For all tests, the overhead of the loops in `test` is small compared to the cost of each call to `F`.

3. Measuring lines of code

To measure Logical Lines of Code (LLOC), the source code for each function (excluding the test harness mentioned) was hand counted using the following criteria:

- Whitespace and comments are not counted.
- Lines containing only braces are not counted.
- Function signatures and class declarations are counted.
- The declaration and body of trivial member functions may share a line.
- Statements spanning multiple lines still count for multiple lines.

The third criteria may not seem "logical" enough for LLOC, however, signatures and declarations in C++ play an important role in program semantics through overloading, overriding, and conversion. The final criteria seems to reward long lines, however, all code is formatted for 80-character columns, so the line cost of a complex expression cannot be avoided.

B. Tests

This section presents each test case and discusses the results. Briefly, the test cases are:

- uda-single (§1) : Given an `ipr::Decl` found in a class, identify the non-static member variables and functions.
- expr-single (§2) : Given an `ipr::Expr` found in a runtime expression, identify whether it is a call to a given free or member function.
- whole (§3) : Find all class definitions in the program.
- expr-tree1 (§4) : Given an `ipr::Expr` found in a runtime expression, identify *all* calls to a given free or member function found anywhere in the tree rooted at the expression.
- expr-tree2 (also §4) : Given an `ipr::Expr` found in a runtime expression, identify *all* calls to the built-in `operator+` found anywhere in the tree rooted at the expression.
- layout (§5) : Given two user-defined aggregates, are they "layout compatible," in the sense of Standard C++?

For each of these tests, there is a question of: "What should the test do once it finds what it is looking for?" Technically, some operation is needed to prevent the optimizer from determining that the computation is not used. Since this operation is included in the measurement time, an expensive operation (like printing to standard output) will amortize the cost of the mechanism being tested, namely the Filter library. Hence, the overhead could be arbitrarily reduced by introducing an arbitrarily large operation. To prevent this effect, a cheap operation is chosen: the increment of a counter. Thus, it is important to keep in mind that the performance differences

shown are purely in the mechanism and do not indicate the overall effect on a program using the Filter library.

1. uda-single

The problem statement for the uda-single test is:

> Given an `ipr::Decl` found in a class, identify the non-static member variables and functions.

To report the results, a counter is kept for each case and incremented each time a match is found. These counters are stored in a `Results` structure used by all solutions:

```
struct Results {
  int vars, funcs;
};
```

This test is a good starting point because it does not involve much work and so, when comparing the Filter solution to the traditional solution, the test mostly measures the overhead of the staple Filter operations: creating variants with `discern` and querying their contents. First though, we look at the baseline traditional solution. Here, `traditional_uda_single_test` is the function fed into the test harness described on pg. 113:

```
struct Uda_decl_visitor : Noop_visitor
{
  Results &r;
  Uda_decl_visitor(Results &r) : r(r) {}

  void visit(const ipr::Field &) { ++r.vars; }
  void visit(const ipr::Bitfield &) { ++r.vars; }

  void visit(const ipr::Fundecl &fd)
  {
    if (ipr_typeid<ipr::Class>(fd.membership()) != 0 && // member of a class
        (fd.specifiers() & ipr::Decl::Static) == 0) // not declared ‘static’
      ++r.funcs;
  }
};
```

Table II. uda-single LLOC results

	Filter	Traditional
LLOC	5	12

```
void traditional_uda_single_test(Results &r, const ipr::Decl &decl)
{
  Uda_decl_visitor vis(r);
  decl.accept(vis);
}
```

In this code, we can see that, for function declarations, a little extra work is required after uncovering the `ipr::Fundecl` to understand its full meaning. Specifically, we need to distinguish static member functions and free functions (possible through `friend` declarations). Also, we have to remember the two ways member variables can be expressed in the IPR. These two details are lowered away by the Filter library, allowing the following solution:

```
void filter_uda_single_test(Results &r, const ipr::Decl &d)
{
  if (maybe<uda_ctx::Variant> mv = uda_ctx::discern(*g_env, d))
    switch (mv->which()) {
      case uda_ctx::mem_var_e: ++r.vars; break;
      case uda_ctx::mem_func_e: ++r.funcs; break;
    }
}
```

Here, `g_env` is a global variable pointing to the Filter's environment. The reason for the `maybe` type is described on pg. 73.

Applying the rules for measuring LLOC described on pg. 114, we get the results shown in Table II. Looking beyond the LLOC difference, we can see how, by not requiring the callback style implied by the Visitor pattern, the Filter allows a more concise expression of the solution. This theme will be more evident in later tests.

To better frame the performance results, we consider two more solutions to the problem which provide reference points for performance. The first uses an implementation detail of the Pivot to achieve maximal performance:

```
void category_uda_single_test(Results &r, const ipr::Decl &d)
{
  switch (d.category) {
    case ipr::field_cat: case ipr::bitfield_cat:
      ++r.vars;
      break;
    case ipr::fundecl_cat:
    {
      const ipr::Fundecl &fd = static_cast<const ipr::Fundecl &>(d);
      if (fd.membership().category == ipr::class_cat &&
          (fd.specifiers() & ipr::Decl::Static) == 0)
        ++r.funcs;
      break;
    }
  }
}
```

Without going into Pivot category codes, we can see that they are similar in spirit to the Filter `which` codes. By switching on category codes, this solution uses the least possible indirection to get to the result.

On the other end of the performance spectrum, we can use the built-in C++ `dynamic_cast`:

```
void dynamic_cast_uda_single_test(Results &r, const ipr::Decl &d)
{
  if (dynamic_cast<const ipr::Field *>(&d) != 0 ||
      dynamic_cast<const ipr::Bitfield *>(&d) != 0)
    ++r.vars;
  else if (const ipr::Fundecl *fd = dynamic_cast<const ipr::Fundecl *>(&d))
    if (dynamic_cast<const ipr::Class *>(&fd->membership()) != 0 &&
        (fd->specifiers() & ipr::Decl::Static) == 0)
      ++r.funcs;
}
```

This is a useful example because it illustrates the high speed of the other three solutions compared to the general case represented by `dynamic_cast`.

Table III. uda-single performance results

	Filter	Traditional	Ratio	Category	`dynamic_cast`
All	.93s	.26s	3.57x	.090s	2.92s
Some	1.12s	.32s	3.5x	.11s	4.70s
None	.86s	.23s	3.74x	.062s	3.59s

To measure the performance of all four solutions, three input sets are used, named "All," "Some," and "None." These sets represent a list of `ipr::Decl` nodes where, respectively, all, some, and none of the nodes are non-static member functions or member variables. The appendix, pg. 174, lists the input file with the three input sets enclosed in classes with the same name.

Running 1M iterations over these input sets yields the results shown in Table III. On first glance, we can see that the Filter solution is 3-4x slower than the traditional solution. However, the traditional solution itself is 2-4x slower than the category solution. This illustrates that, at the microscopic level of this test, tiny performance differences are magnified into factors, not percent. Lastly, considering that the Filter is 3-5x faster than `dynamic_cast`, we can see that the Filter is still a very efficient mechanism for type discovery.

However, it still remains to identify the source of the 3-4x slowdown with the Filter compared to the traditional solution, especially since, externally, the Filter solution looks like the efficient category solution. In the Visitor pattern, a visitation costs 2 virtual dispatches (one for `accept`, one for `visit`). Thus, the traditional solution will use either 2 or 6 virtual dispatches, the latter only for the member function case. Considering now the Filter solution, based on the implementation details given in Chapter III, at a minimum the Filter requires:

- a function call to enter the library code,
- a category switch to uncover the IPR type,
- a pool allocation to allocate the internal node of the variant (pg. 86), and
- the switch shown in the user code.

Clearly, the pool allocation will involve some computation to pick the pool, extract a free chunk of memory, and adjust the free list. However, since the performance "quanta" here is single virtual dispatch, even the expense of extra function calls is significant since a virtual dispatch is only 20-25% slower than a statically resolved call [80]. Taken together, it is easy to see how these factors can add up to 6-8 virtual dispatches.

Another performance impediment of the Filter is that it fully discerns the type of a given node, even if these results are not used. For example, when `uda_ctx ::discern` encounters an `ipr::Typedecl`, it must check the `Typedecl`'s `initializer` (after calling `has_initializer`) to determine whether the returned variant holds a `class_e`, `union_e`, or `enum_e`. Of course, the distinction is not needed in this test, so these virtual dispatches are entirely unnecessary. This overhead can be seen in the comparatively worse ratio for the Empty input.

2. expr-single

The problem statement for the expr-single test is:

> Given an `ipr::Expr` found in a runtime expression, identify whether it is a call to a given free or member function.

Thus, this test maintains the same basic "`discern` and test" model as uda-single, but involves greater lowering to solve the problem. To report the results, a counter is kept

for each case. These counters, as well as pointers to the target functions, are stored in a `Results` structure used by both solutions:

```
struct Results {
  const ipr::Fundecl const *free_func, const *mem_func;
  int free_calls, mem_calls;
};
```

We first consider the Filter solution, which is only slightly more complicated than the solution for uda-single. This code makes use of the `Call` and `Mem_call` node types, which are described on pg. 63:

```
void filter_expr_single_test(Results &r, const ipr::Expr &e)
{
  v = expr_ctx::discern(*g_env, e);
  switch (v.which()) {
    case expr_ctx::call_e:
      if (v.get<Call>().which() == Call::glo_func_e &&
          r.free_func == &v.get<Call>().glo_func().ipr().master())
        ++r.free_calls;
      break;
    case expr_ctx::mem_call_e:
      if (v.get<Mem_call>().which() == Mem_call::mem_func_e &&
          r.mem_func == &v.get<Mem_call>().mem_func().ipr().master())
        ++r.mem_calls;
      break;
  }
}
```

Notice that this code takes advantage of the close connection between Filter nodes and IPR nodes to use the `master` field of `ipr::Decl` and the identity property of IPR declarations (pg. 42) to efficiently finish the job.

The traditional solution requires more code in order to check all the places where function calls can be represented in the IPR:

```
// test whether the given Decl is one of the given functions
inline void process_decl(Results &r, const ipr::Decl &d)
{
  // don't need to check for 'Fundecl' first, since using node identity
  if (r.free_func == &d)
    ++r.free_calls;
```

```
  else if (r.mem_func == &d)
    ++r.mem_calls;
}

// visitor for the 'Call::function' field that looks for matching calls
struct Func_expr_visitor : Noop_visitor
{
  Results &r;
  Func_expr_visitor(Results &r) : r(r) {}

  void visit(const ipr::Id_expr &i)
  {
    // found a call of the form: 'foo(...)'
    process_decl(r, i.resolution());
  }

  // reused for 'Dot::member' and 'Arrow::member'
  void process_member(const ipr::Expr &e)
  {
    // check the 'x' in 'a->x(...)' to see if it is a member function
    if (const ipr::Id_expr *i = ipr_typeid<ipr::Id_expr>(e))
      process_decl(r, i->resolution());
  }
  void visit(const ipr::Dot &n) { process_member(n.member()); }
  void visit(const ipr::Arrow &n) { process_member(n.member()); }
};

// look for matching calls (called by Visitor_to_overload)
struct Expr_action
{
  Results &r;
  Expr_action(Results &r) : r(r) {}

  void operator()(const ipr::Node &) {}

  void operator()(const ipr::Classic &c)
  {
    // check for call to overloaded operator. if not, ignore
    if (c.has_impl_decl())
      process_decl(r, c.impl_decl().master());
  }

  // 'Call' is a special 'Classic'
  void operator()(const ipr::Call &c)
  {
    // don't forget about overloaded operator()
    if (c.has_impl_decl())
      process_decl(r, c.impl_decl());
```

Table IV. expr-single LLOC results

	Filter	Traditional
LLOC	13	34

```
    // strip possible parentheses around function name
    const ipr::Expr *func = &c.function();
    while (const ipr::Paren_expr *p = ipr_typeid<ipr::Paren_expr>(*func))
      func = &p->expr();

    // visit function expression, looking for calls
    Func_expr_visitor vis(r);
    func->accept(vis);
  }
};

void traditional_expr_single_test(Results &r, const ipr::Expr &expr)
{
  Expr_action act(r);
  Visitor_to_overload<Expr_action> vis(act);
  expr.accept(vis);
}
```

Here, we can see that the two-level case analysis of `ipr::Call` nodes translates into two visitors with two recursive calls. Also, we can see the importance of having a structure like `Results` for visitors: `Results` collects all the data that needs to be passed around into one object which can be passed by a single reference.

Counting the LLOC for each solutions produces the results in Table IV. We can attribute most of the extra code of the traditional solution to the manual lowering required to solve the problem.

Moving on to performance, again three input files are used which have the same characteristics as with uda-single: "All" contains only matches, "Some" contains about half matches, and "None" contains no matches. These sets of expressions are listed in the appendix, pg. 175. Running 1M iterations over these inputs yields the

Table V. expr-single performance results

	Filter	Traditional	Ratio
All	2.14s	.81s	2.64x
Some	1.88s	.65s	2.89x
None	1.55s	.36s	4.3x

results shown in Table V.

Compared to the uda-single results, these results are more polarized. We consider first the "None" case, whose ratio of 4.3 is worse than any of the uda-single ratios. Based on the explanation of runtime-expression lowering given on pg. 63, more computation is needed in `expr_ctx::discern` than `uda_ctx::discern` to determine which Filter node type to return. This does not count as overhead as long as this computation was required to solve the problem at hand. By definition, the "None" input contains only expressions that are not relevant to the problem, therefore this lowering computation is mostly wasted. This effect was already observed, to a lesser degree, in uda-single.

Considering the first two inputs, however, the ratio is better than it was for uda-single. The reason is that, with more *essential* computation required for each node, the library overhead described on pg. 116 is less significant. Actually, the ratios would be better, but there are a few cases where the library must waste computation. To see why, consider the `process_decl` function, which is part of the traditional solution. In `process_decl`, it is not necessary to determine whether the given `ipr::Decl` is a [non-]static member function, free function, or even an `ipr::Fundecl`, since we can simply test pointer equality and be done. However, `expr_ctx::discern` needs to make this determination to decide whether to return a `Call` or `Mem_call` node.

3. whole

The problem statement for the "whole" test is:

> Find all class definitions in the program.

The `Results` type, in this case, is simply an `int` that keeps track of the number of definitions found.

Recall, from pg. 32, that classes can appear deeply nested in many parts of a C++ program. Thus, while this problem statement is simple, its solution requires a significant amount of work that is best done by a reusable library. The Filter solution to this problem uses polymorphic child-functions. Using the `Traverse` utility built on pg. 37, we can write the following solution:

```
void filter_whole_test(int &r, const ipr::Namespace &ns)
{
  Prune p = Prune::type_expr(); // prune types and runtime expressions
  for (Traverse t(expr_ctx::discern(*g_env, ns), p); !t.done(); t.step())
    if (t->which() == uda_ctx::class_e && t->get<Class>().has_def())
      ++r;
}
```

Note that `Traverse` is not a large library-defined traversal, but a tiny (<20 LLOC) user-defined class. In fact, a recursive version that does not maintain a context-stack can be re-implemented in 5 LLOC:

```
void filter_rec_whole_test(int &ct, any_ctx::Variant v)
{
  if (v.which() == any_ctx::class_e && v.get<Class>().has_def())
    ++ct;
  for (any_ctx::Range r = v.child_any(Prune::type_expr()); \
       !r.empty(); \
       ++r.first)
    filter_rec_whole_test(ct, *r.first);
}
```

In both cases, we are using the `Prune` parameter of `child_any` to ignore runtime expressions and types, since class definitions cannot possibly occur in these contexts.

Moving on to the traditional solution, we have a problem because, in general, the IPR forms a graph. While there is a simple tree embedded in this graph (viz. the one traversed by Filter), there is no simple way to project this tree from the IPR's interface and currently no tree-based traversal algorithms in the Pivot library. There is, however, a family of graph-based traversals which can be used to visit nodes via depth- and breadth-first search. These graph-based traversals use a Visitor-based design and thus fit the traditional approach. Specifically, to perform a traversal, the user derives a class from the library class `DFSVisitor` and overrides virtual functions in the base class:

```
struct Graph_visitor : DFSVisitor
{
  int &r;
  Graph_visitor(int &r) : r(r) {}

  bool on_discover_node(const ipr::Node &n) // called by DFSVisitor
  {
    if (ipr::ipr_typeid<ipr::Class>(n) != 0)
      ++r;
    return true;
  }
};

void traditional_whole_test(int &r, const ipr::Namespace &ns)
{
  Graph_visitor vis(r);
  ns.accept(vis);
}
```

Here, `on_discover_node` is called for every node found during traversal. There is also an analogous `on_discover_edge` called for every edge. With the booleans returned from both these functions, `DFSVisitor` offers the ability to prune parts of the graph. However, without any coherent entity like Filter's contexts, it is highly non-trivial to achieve pruning equivalent to the Filter solution. The reason is that the graph traversal can be visualized as a flood fill of nodes along edges, so if a single edge into an otherwise-pruned component is left un-pruned, the entire component will be

Table VI. whole LLOC results

	Filter	Traditional
LLOC	5	10

traversed.

Comparing the two solutions on LLOC, we get the results shown in Table VI. Without any extra lowering performed by the traditional solution, the syntactic overhead here is primarily due to the inversion of control imposed by the Visitor pattern.

Two test input files, named "Small" and "Medium," are used to test the solutions. As the names suggest, they differ primarily in size. Both files attempt to capture "ordinary code" by employing a wide mix of language constructs. The Small input is 13 LOC and the Medium input is 81 LOC. Each test performs 10,000 iterations over the input. In order to isolate the effects of pruning, which cannot be done effectively in the traditional solution, Filter is measured with and without pruning. Additionally, since `on_discover_edge` is not used, it is made non-virtual and inline.

The results from these tests are shown in Table VII. The drastic performance differences can be attributed to the fact that, with or without pruning, there are many nodes Filter completely hides in the underlying IPR. Such nodes include `ipr::Region`, `ipr::Scope`, and all `ipr::Name`-derived classes. Together, these nodes make up a significant fraction of all IPR nodes. Thus, even compared to the un-pruned Filter, the traditional solution is visiting more nodes.

It is important not to read too much into this test about the traditional approach since `DFSVisitor` is not representative of a normal Visitor-based traversal. As explained on pg. 146, most frameworks include a collection of tree traversal algorithms which would perform better against the Filter solution. However, this test does

Table VII. whole performance results

	Filter (pruning)	Filter (no pruning)	Traditional
Small	.06s	.27s	13.80s
Medium	.33s	1.52s	54.11s

make a few clear recommendations for any future design of Visitor-based traversals in Pivot. First, the concept of "context" provides a simple and effective way to prune traversals. Second, for performance, at least some traversals should be designed not to span the IPR graph, focusing on a subset of nodes needed for a target use case.

4. expr-tree1 and expr-tree2

Even after the test on pg. 125, the question of Filter iterator performance (the focus of pg. 88) remains open due to the weak upper bound provided by the `DFSVisitor`. This section gives a much clearer picture by building a custom tree-traversal visitor for runtime expressions that is able to exactly achieve the pruning and node elision of the Filter library. Once the high-performance of the traditional solution has been regained, the relative performance of Filter depends on whether the lowering it performs is necessary. To better illustrate this, this section presents two tests:

> expr-tree1: Given an `ipr::Expr` found in a runtime expression, identify *all* calls to a given free or member function found anywhere in the tree rooted at the expression.
>
> expr-tree2: Given an `ipr::Expr` found in a runtime expression, identify *all* calls to the built-in `operator+` found anywhere in the tree rooted at the expression.

The idea is that expr-tree1 will perform the expr-single test, which requires the lowering done by Filter, on each node during traversal while expr-tree2 can be solved with a simple IPR type query by the traditional approach.

For the expr-tree1 test, the `Results` type will be the same as for the expr-single test. For the expr-tree2 test, the `Results` type will simply by an `int` counter. With these types, and reusing the test function from expr-single, the Filter solution to expr-tree1 can be written:

```
void filter_expr_tree1_test(Results &results, expr_ctx::Variant v)
{
  filter_expr_single_test(results, v);
  for (expr_ctx::Range r = v.child_expr(); !r.empty(); ++r.first)
    filter_expr_tree1_test(results, *r.first);
}
```

Using an iterative strategy, the solution to expr-tree2 can be written:

```
std::vector<expr_ctx::Variant> st; // global to reduce new/delete calls

void filter_expr_tree2_test(Results &results, expr_ctx::Variant v)
{
  st.push_back(v);
  while (!st.empty()) {
    if (st.back().which() == expr_ctx::call_e &&
        st.back().get<Call>().which() == Call::builtin_e &&
        st.back().get<Call>().builtin() == ipr::plus_cat)
      ++results;
    expr_ctx::Range r = st.back().child_expr();
    st.pop_back();
    for (; !r.empty(); ++r.first)
      st.push_back(*r.first);
  }
}
```

The reason for choosing different strategies for the two solutions is two-fold: first, it demonstrates that the Filter library is conducive to both strategies (as opposed to the traditional approach, which requires recursion); second, the iterative approach is measurably faster for expr-tree2.

To build the traditional solution, we first build a reusable expression traversal. Using generative programming techniques, we take advantage of the `Unary`, `Binary`, and `Ternary` classes derived by all IPR runtime expressions. The type arguments of these classes describe the types of the operands and thus provide us with all the information needed to stay inside the runtime expression context.

```
// meta-function: generates visitor calls if T is an Expr or Expr_list
template <class T> struct visit_if_expr {
  static void args(T, ipr::Visitor &) {} // base case: no call
};
template <> struct visit_if_expr<const ipr::Expr &> {
  static void args(const ipr::Expr &e,ipr::Visitor &v) {e.accept(v);}
};
template <> struct visit_if_expr<const ipr::Expr_list &> {
  static void args(const ipr::Expr_list &e,ipr::Visitor &v) {v.visit(e);}
};

// called by Visitor_to_overload. visit current node with 'node_vis',
// then recurse on children by calling the same Visitor_to_overload that
// called us (given by 'outer')
struct Visit_and_recurse_action
{
  ipr::Visitor &node_vis;
  ipr::Visitor *outer;
  Visit_and_recurse_action(ipr::Visitor &v) : node_vis(v), outer(0) {}
  void set_outer(ipr::Visitor &v) { outer = &v; }

  void operator()(const ipr::Node &) {} // base case: do nothing

  template <class C, class T>
  void operator()(const ipr::Unary<C, T> &e)
  {
    e.accept(node_vis);
    visit_if_expr<T>::args(e.operand(), *outer);
  }

  template <class C, class T1, class T2>
  void operator()(const ipr::Binary<C, T1, T2> &e)
  {
    e.accept(node_vis);
    visit_if_expr<T1>::args(e.first(), *outer);
    visit_if_expr<T2>::args(e.second(), *outer);
  }

  template <class C, class T1, class T2, class T3>
```

```
  void operator()(const ipr::Ternary<C, T1, T2, T3> &e)
  {
    e.accept(node_vis);
    visit_if_expr<T1>::args(e.first(), *outer);
    visit_if_expr<T2>::args(e.second(), *outer);
    visit_if_expr<T3>::args(e.third(), *outer);
  }

  // otherwise, nullary expr
  void operator()(const ipr::Expr &e) { e.accept(node_vis); }

  // special case: list of Expr
  void operator()(const ipr::Expr_list &e)
  {
    node_vis.visit(e);
    const ipr::Sequence<ipr::Expr> &seq = e.elements();
    for (int i = 0, sz = seq.size(); i != sz; ++i)
      seq[i].accept(*outer);
  }
};

void visit_expr_tree(const ipr::Expr &expr, ipr::Visitor &v)
{
  Visit_and_recurse_action act(v);
  Visitor_to_overload<Visit_and_recurse_action> vis(act);
  vis.act.set_outer(vis);
  expr.accept(vis);
}
```

The technique shown here allows a very concise way to visit the children of the hundreds of concrete IPR expression types by using the unique inheritance scheme of the IPR (viz. the use of arity classes). In a way, the combined use of `Visitor_to_overload` and arity classes is similar to the polymorphic child-functions of the Filter library. The technique can also be applied to work on `ipr::Type` and `ipr::Name` in the same way since they also make regular use of the arity classes. However, the technique does not [currently] apply to statements and declarations.

Using `visit_expr_tree`, we can write the traditional solution to expr-tree1:

```
struct Call_expr_single
{
  Results &r;
  Examine_action(Results &r) : r(r) {}
```

```
  void operator()(const ipr::Node &) {}
  void operator()(const ipr::Expr &e) { \
    traditional_expr_single_test(r, e); \
  }
};

void traditional_test(Results &r, const ipr::Expr &expr)
{
  Examine_action act(r);
  Visitor_to_overload<Examine_action> vis(act);
  visit_expr_tree(expr, vis);
}
```

Similarly, the solution to expr-tree2 can be written:

```
struct Find_builtin_plus_action
{
  int &r;
  Find_builtin_plus_action(int &r) : r(r) {}

  void operator()(const ipr::Node &) {}

  void operator()(const ipr::Plus &p)
  {
    if (!p.has_impl_decl())
      ++ct_;
  }
};

void traditional_test(int &r, const ipr::Expr &expr)
{
  Find_builtin_plus_action act(r);
  Visitor_to_overload<Find_builtin_plus_action> vis(act);
  visit_expr_tree(expr, vis);
}
```

In applying the LLOC measure, the implementation of `visit_expr_tree` is included because it was specially crafted for the present purpose of traversing runtime expressions; if the problem was changed to search a function body for runtime expressions, a different custom traversal visitor would need to be written, and so on. The resulting LLOC counts are shown in Table VIII. As these numbers clearly indicate, aside from lowering, the second major facility of the Filter library is allowing precise, *ad hoc* traversal. We say *ad hoc* because Filter does not fix a traversal strategy,

Table VIII. expr-tree LLOC results

	Filter	Traditional
expr-tree1	4	81
expr-tree2	12	49

Table IX. expr-tree1 performance results

	Filter	Traditional	Ratio
All	6.14s	2.56s	2.40x
Some	7.45s	2.60s	2.87x
None	6.20s	2.15s	2.88x

providing instead the necessary information to perform traversal.

Moving on to performance, for expr-tree1, three inputs named "All," "Some," and "None" were used, with the same meaning as in earlier tests. Each input consists of a single expression tree, listed on pg. 176. The results of running 1M iterations are shown in Table IX. Notice here that the "None" input does not produce worse performance as it did in previous tests. The reason for this is that the lowering required for `Call`, `Mem_call`, and `Member` is the only expensive lowering done by Filter for runtime expressions. In their absence, there is little computation to waste.

To test expr-tree2, three different input expressions are used, each of a different size and containing different expression constructors. These are named "Small," "Medium," and "Large" and listed on pg. 176. The results of running 1M iterations are shown in Table X. As expected, the ratios are much worse than the previous tests. Specifically, this is due to the fact that the only work required of the traditional solu-

Table X. expr-tree2 performance results

	Filter	Traditional	Ratio
Small	1.99s	.22s	9.05x
Medium	6.39s	1.16s	5.51x
Large	8.87s	1.51s	5.87x

tion, at each node in the traversal, is a single visitation (done by `Visitor_to_overload`). The Filter library, on the other hand, is doing the same lowering that it did for expr-tree1 and expr-single since the library has no way of knowing the limited way in which the result will be used.

An important conclusion to draw from these two tests and the whole-program traversal test on pg. 125 is that the performance impact of precise traversal is much greater than that of using the Filter library. Thus, if the concise way in which Filter allows traversal to be expressed leads the user to write more precise traversals, we can view Filter as providing an overall speedup, even in situations like expr-tree2.

5. layout

The problem statement for the final test is:

> Given two user-defined aggregates, are they "layout compatible," in the sense of Standard C++?

This test gives more insight into the use of Filter for doing the type of recursive case analysis for which it was designed. While uda-single and expr-single gave tiny examples of case analysis, their primary goal was to measure the performance of variants. This test provides a more realistic scenario.

"Layout compatibility" is described in the C++ standard [32] by the following clauses:

- 3.9.11 : If two types `T1` and `T2` are the same type, then `T1` and `T2` are *layout-compatible* types.
- 7.2.8 : Two enumeration types are layout compatible if they have the same *underlying type*.
- 9.2.15 : Two POD-struct types are layout compatible if they have the same number of non-static data members, and corresponding non-static data members (in order) have layout-compatible types.
- 9.2.16 : Two POD-union types are layout compatible if they have the same number of non-static data members, and corresponding non-static data members (in any order) have layout-compatible types.

To keep the implementations relatively simple, we will take a few shortcuts: to sidestep the issue of underlying type, clause 7.2.8 is ignored (so enumerations are only compatible if they are identical); because POD-checking is a separate analysis problem altogether, the POD requirement of 9.2.15 and 9.2.16 is ignored; and the "in any order" flexibility of unions is ignored, allowing unions and structs to be checked in the same way.

To solve this problem, two types of checks are needed. The first applies clause 9.2.15 to recursively check user-defined aggregates for compatibility. The second takes two non-UDA types and checks that they are identical. Thus, when comparing `S1` and `S2` in the following code:

```
struct A {};
struct B {}
struct S1 { A a; int *x; };
struct S2 { B b; int *y; };
```

the compatibility check is applied to the types of `a` and `b`, and the equality check is applied to the types of `x` and `y`.

Using the type unification performed by the IPR, a first attempt might be to use C++ pointer equality to test type equality. Unfortunately, any types involving non-type expressions, like `Id_expr`, will not be unified, even though they may be equal. For example, in:

```
struct X { A *x1; };
struct Y { A *y1; };
```

the variables `x1` and `y1` have equal types, but have different `ipr::Type` nodes because of the `Id_expr` used to describe the use of the name of a user-defined type.

Type equality is implemented in the Filter and traditional solutions by the functions:

```
bool filter_equal(type_ctx::Variant, type_ctx::Variant);
bool traditional_equal(const ipr::Type &, const ipr::Type &);
```

The difference between the implementations of these functions is not as illustrative as the compatibility checks, thus we skip their presentation here and refer to the appendix, pg. 176, for the listing.

Using the equality check, the Filter compatibility check can be implemented as follows:

```
// skip declarations that are not member variables
inline void skip(uda_ctx::Range &r)
{
  while (!r.empty() && r.first->which() != uda_ctx::mem_var_e)
    ++r.first;
}

bool filter_compatible(Uda u1, Uda u2)
{
  uda_ctx::Range r1 = u1.def_members(), r2 = u2.def_members();

  // iterate over uda, skipping everything except member variables
  for (skip(r1), skip(r2);
       !r1.empty() && !r2.empty();
```

```
      ++r1.first, skip(r1), ++r2.first, skip(r2))
  {
    // bitfield checks
    Mem_var mv1 = r1.first->get<Mem_var>(), \
            mv2 = r2.first->get<Mem_var>();
    if (mv1.is_bitfield() != mv2.is_bitfield() ||
        mv1.is_bitfield() && !same_literal(mv1.precision(), \
                                           mv2.precision()))
      return false;

    // get types
    type_ctx::Variant v1 = mv1.type(), v2 = mv2.type();
    if (v1.which() != v2.which())
      return false;

    if (v1.which() != type_ctx::udt_use_e) {
      // use equality for non-udt
      if (!filter_equal(v1, v2))
        return false;
    }
    else {
      // use equality for enums, compatibility for udas
      Udt_use udt1 = v1.get<Udt_use>(), udt2 = v2.get<Udt_use>();
      if (udt1.which() != udt2.which())
        return false;
      if (&udt1.udt_ipr() != &udt2.udt_ipr() &&
          (udt1.which() == Udt_use::enum_e ||
           !filter_compatible(udt1.uda(), udt2.uda())))
        return false;
    }
  }

  // require equal number of member variables
  return r1.empty() && r2.empty();
}
```

This code follows directly from the problem definition. The loop iterates simultaneously over the members of the given UDA, resting only on pairs of member variables. In the special case that the members are bitfields, they must be the same width (checked by a function `same_literal` that is already implemented as part of type equality). Next, both members' types are checked, using compatibility for UDAs and equality otherwise. If the loop completes successfully and no member variables

remain in either list, the UDAs are compatible.

We now consider the traditional solution, which will be presented in parts. Using a top-down order, we start with `traditional_compatible` and the utility classes/functions it requires:

```
// visitor for uncovering Fields and Bitfields, used in 'skip'
struct Is_member : Noop_visitor {
  bool is_member;
  Is_member() : is_member(false) {}
  void visit(const ipr::Field &) { is_member = true; }
  void visit(const ipr::Bitfield &) { is_member = true; }
};

// skip non-Field/Bitfield Decls
inline void skip(const ipr::Sequence<ipr::Decl> &seq, int &i, int sz)
{
  for (; i != sz; ++i) {
    Is_member vis;
    seq[i].accept(vis);
    if (vis.is_member)
      return;
  }
}

// pull out the 'members' sequence from a Class or Union
struct Get_members_visitor : Noop_visitor {
  const ipr::Sequence<ipr::Decl> *members;
  Get_members_visitor() : members(0) {}
  void visit(const ipr::Class &c) { members = &c.members(); }
  void visit(const ipr::Union &u) { members = &u.members(); }
};

bool traditional_compatible(const ipr::Udt &t1, const ipr::Udt &t2)
{
  // compatible if the same IPR node
  if (&t1 == &t2)
    return true;

  // pull the members out of user-defined aggregates
  Get_members_visitor gm1, gm2;
  t1.accept(gm1);
  t2.accept(gm2);
  if (!gm1.members || !gm2.members)
    return false;
```

```
  // iterate through member variables
  const ipr::Sequence<ipr::Decl> &seq1 = *gm1.members, \
                                 &seq2 = *gm2.members;
  int sz1 = seq1.size(), sz2 = seq2.size();
  int i1 = 0, i2 = 0;
  for (skip(seq1, i1, sz1), skip(seq2, i2, sz2);
       i1 != sz1 && i2 != sz2;
       ++i1, skip(seq1, i1, sz1), ++i2, skip(seq2, i2, sz2))
  {
    // pull out type of Field/Bitfield, check precisions of Bitfields
    Check_bitfield_and_get_type decl_vis(seq2[i2]);
    seq1[i1].accept(decl_vis);
    if (!decl_vis.compatible)
      return false;
    const ipr::Type &type1 = *decl_vis.type1, &type2 = *decl_vis.type2;

    // check compatibility of member variables' types
    Compatible_action1 act(type2);
    Visitor_to_overload<Compatible_action1> type_vis(act);
    type1.accept(type_vis);
    if (!type_vis.act.compatible)
      return false;
  }

  // no hanging members
  return i1 == sz1 && i2 == sz2;
}
```

From this code, we can see the same high-level organization as the Filter solution. Without the `Uda` common base, a visitor is required to pull out the `members` sequence. Also, without `Field` and `Bitfield` being lowered to the same node, `skip` requires a visitor. In the body of the loop, the work has been split into two visitors: one to pull out types and check bitfields, and one to handle member compatibility.

The `Check_bitfield_and_get_type` visitor is implemented as follows:

```
struct Check_bitfield_and_get_type : Noop_visitor
{
  bool compatible;
  const ipr::Decl &decl2;
  const ipr::Type *type1, *type2;

  Check_bitfield_and_get_type(const ipr::Decl &d2)
```

```
    : compatible(false), decl2(d2), type1(0), type2(0)
  {}

  void visit(const ipr::Field &n)
  {
    type1 = &n.type();
    if (const ipr::Field *f = ipr_typeid<ipr::Field>(decl2)) {
      type2 = &f->type();
      compatible = true;
    }
  }

  void visit(const ipr::Bitfield &n)
  {
    type1 = &n.type();
    if (const ipr::Bitfield *bf = ipr_typeid<ipr::Bitfield>(decl2)) {
      type2 = &bf->type();
      compatible = same_literal(n.precision(), bf->precision());
    }
  }
};
```

Notice that, to discover the full types of both declarations, we need nested visitation. The outer visitor discovers the first type and then, from within this context, invokes the inner visitor. Fortunately, for the problem at hand, the second visitation can be done using only `ipr_typeid`. However, this illustrates the general problem that the Visitor pattern makes it difficult to "be in two places at the same time." This effect becomes more pronounced in the implementation of `Compatible_action1` (used in `traditional_compatible` above):

```
// Called by Visitor_to_overload: uncovers the first type
struct Compatible_action1
{
  bool compatible;
  const ipr::Type &type2;

  Compatible_action1(const ipr::Type &t2):compatible(false),type2(t2) {}

  void operator()(const ipr::Udt &n)
  {
    Compatible_action2 act(n);
    Visitor_to_overload<Compatible_action2> vis(act);
    type2.accept(vis);
```

```
    compatible = vis.act.compatible;
  }

  void operator()(const ipr::As_type &n)
  {
    if (const ipr::Udt *u = reduce_to_udt(n)) {
      Compatible_action2 act(*u);
      Visitor_to_overload<Compatible_action2> vis(act);
      type2.accept(vis);
      compatible = vis.act.compatible;
    }
    else
      // fall back on type equality
      compatible = visitor_equal(n, type2);
  }

  void operator()(const ipr::Type &n)
  {
    compatible = visitor_equal(n, type2);
  }

  void operator()(const ipr::Node &) { assert(false); }
};

// Called by Visitor_to_overload: uncovers the second type
struct Compatible_action2
{
  bool compatible;
  const ipr::Udt &type1;

  Compatible_action2(const ipr::Udt &t1):compatible(false),type1(t1) {}

  void operator()(const ipr::Udt &type2)
  {
    compatible = traditional_compatible(type1, type2);
  }

  void operator()(const ipr::As_type &n)
  {
    if (const ipr::Udt *type2 = reduce_to_udt(n))
      compatible = traditional_compatible(type1, *type2);
    else
      // fall back on type equality
      compatible = traditional_equal(type1, n);
  }

  void operator()(const ipr::Type &n)
  {
    compatible = traditional_equal(type1, n);
```

Table XI. layout LLOC results

	Filter	Traditional
LLOC	69	188

```
  }
  void operator()(const ipr::Node &) {}
};
```

This code uses the well-known "double dispatch" technique to simulate multi-methods [77]. The simulation, however, makes the logic, represented by the above pair of classes, difficult to understand. The basic idea is that, if both `type1` and `type2` are `Udt`s (or in the case of `As_type`, can be converted to `Udt`s), recursively check compatibility. Otherwise, use type equality.

To complete the traditional solution, we also need an implementation for the function `reduce_to_udt`, used above. The code to do this mainly deals with lowering `As_type`, so we skip the implementation here and refer to the appendix, pg. 176.

Combining the LLOC for checking type equality and the code shown here, the final results are shown in Table XI. Looking at the two solutions, part of this overhead can be attributed to the lowering done by the Filter library. The most significant lowering done in the traditional solution involves the `As_type` node, requiring both the `reduce_to_udt` and `reduce_to_td` functions. However, a large part of the overhead in the traditional solution results not from lowering, but the forced-recursion of the Visitor pattern.

As an additional informal comparison: the Filter solution took around 2 hours to write and debug, while the traditional solution took a full day (even after writing the Filter solution). Thus, the extra LLOC are in no way mindless boilerplate. In fact,

Table XII. layout performance results

	Filter	Traditional	Ratio
Deep	4.16s	2.35s	1.77x
Shallow	1.03s	.47s	2.19x

as one might expect from an attempted reading, the profusion of recursion requires significantly more mental effort than Filter's "first-order" solution. Together, both the LLOC and informal complexity differences between the two solutions make a strong supporting argument for the design discussion on pg. 81.

Moving on to performance, two input sets of pairs of classes were used: "Deep" and "Shallow." The Deep input set was distinguished by requiring several levels of member examination to make a positive or negative determination. These input sets are listed on pg. 182. The results for running 100K iterations for each input set are shown in Table XII. Compared to previous tests, this test shows the least Filter overhead. This follows from the general observation that, when measuring the overhead of a mechanism, as the problem size increases, the overhead of the mechanism decreases.

C. Summary

Taking into account the test results presented in this chapter, we can now ask the question: do Filter's benefits outweigh its costs? The benefits are the code reduction seen in every test and the performance improvements compared to an imprecise traversal (pg. 125). The costs are the performance overhead shown in every test, except "whole," and the mental overhead of learning an extra library. Notice that, because of the lightweight design described on pg. 74, this cost/benefit analysis does

not need to consider whether Filter lowers too much (since the IPR is still present), whether the extra lowered data will exhaust memory too quickly (since there is no persistent data structure), or whether the upfront cost of lowering is justified by its use (since lowering is incremental).

We start by considering the performance cost. As already established at the end of pg. 128, if the Filter's polymorphic child-functions allow the user to write a more precise traversal, the resulting speedup dwarfs the library overhead. In particular, Filter contexts provide a simple, yet effective, way to prune the program tree that is difficult to emulate on a per-edge basis. Even without traversal, the performance cost decreases as the complexity of the user's analysis grows and the work done outside Filter amortizes the overhead inside Filter. This effect can be seen in the progression of tests going from uda-single to expr-single to layout.

Moving to the software engineering benefits of the Filter library, we can see that Filter solutions to the test problems are consistently 2-4x shorter. Part of this reduction is primarily due to the lowering done by Filter and could be equivalently achieved by a library of utility functions which could be used within the traditional solution. However, in all cases, the code reduction is also related to the mandatory use of callbacks. Also, when traversal is required (viz. expr-tree and layout), Filter's polymorphic child-functions represent a distinct code reduction.

The differences in the code were more than just the LLOC formally measured. For each test, the Filter solution was more "structured," in the sense of the ideals of Structured Programming expounded by Dijkstra in [81]:

> In vague terms we may state the desirability that the structure of the program text reflects the structure of the computation. Or, in other terms, "What can we do to shorten the conceptual gap between the static pro-

> gram text (spread out in "text space") and the corresponding computations (evolving in time)?"

In fact, from the perspective described in this quote, the Visitor pattern, applied to the problem of heterogeneous tree traversal, may be viewed as an anti-pattern. But how can this be the case? After all, Visitor is a Design Pattern. The first point to remember is that the Visitor pattern is not the solution to the problem "how to discover an object's type in a heterogeneous data structure," but rather "how to non-intrusively add operations to a class hierarchy" [41]. While it is possible to force the latter perspective, as shown by the traditional solutions in this chapter, it is hardly the natural one. Thus, that the Visitor pattern does not produce a clean solution is a simple consequence of using a design pattern when its problem statement does not naturally describe the problem at hand.

Taking these arguments into account, we can see that as analyses using Filter become large and complex, the relative costs drop and the benefits increase, and thus Filter should definitely be used when it applies. As a counter-example, the characteristic application which should *not* use Filter would be small and have high performance requirements. In this case, however, even the traditional approach imposes unnecessary overhead; the analysis writer should consider the category-/switch-based approach demonstrated on pg. 116.

CHAPTER V

RELATED WORK

The general problem addressed by the Filter library is how to access information in trees of heterogeneous nodes. Since heterogeneous trees appear in many situations, especially when dealing with languages and compilation, there are many other libraries, language features, and tools that explore the same design space as the Filter library. Instead of listing them independently, we will consider two orthogonal sub-problems and describe the related work in terms of these sub-problems.

A. Traversal

For simple tree structures, it is not much trouble for the user to write code that directly works with the tree. However, when the tree has many different types of nodes, the user can end up writing a lot of "boilerplate" code that only serves the purpose of ensuring that an "interesting" computation reaches all nodes in the tree. A nice example of this phenomena is given in [60] as the motivation for the "Scrap Your Boilerplate" library.

A common reaction to this problem is to factor out this boilerplate into a reusable library that allows the user to write *only* the interesting code. In object-oriented programming, this solution often takes the form of the Visitor [41] pattern and a reusable traversal algorithm. For example, if the goal is to collect all the functions in a program, Elsa [38] (based on the Elkhound parser [82]) will allow the user to write:

```
struct CollectVisitor : public ASTVisitor {
  vector<Function *> funcs;
  virtual bool visit(Function *f) { funcs.push_back(f); }
};

vector<Function *> findFuncs(TranslationUnit *unit) {
  CollectVisitor v;
```

```
  unit->traverse(v);
  return v.funcs;
}
```

Here, `traverse` is supplied by the library and calls `visit` on every `Function`-typed node in the tree. This basic strategy is used by the Eclipse [39], Elsa [38], JJTraveler [83], Rose [40], SableCC [37] projects, and others.

On the functional programming side, early work was done not so much to avoid boilerplate code, but to allow more structured reasoning about algorithms [84,85]. For example, [86] describes the cata-, ana-, hylo-, and para-morphism families of recursive algorithms, each supporting algebraic equations that can be used for calculational reasoning about the program. For these frameworks, the reduction in boilerplate is just a "perk" of casting your algorithm in terms of one of given morphisms.

More recent work in functional programming has focused less on the algebraic properties and more on writing less code. Examples include Strafunksi [87], the "Scrap Your Boilerplate" (SYB) approach [60,88,89], and Generic Haskell [90]. These approaches place less restrictions on the data types traversed and allow more *ad hoc* specification of operations to be performed on each node. For example, SYB allows us to write code analogous to the Visitor-based code above:

```
collectFunc :: Function -> [Function]
collectFunc f = [f]

findFuncs :: Program -> [Function]
findFuncs p = (everything (++) (mkQ [] collectFunc)) p
```

Here, `everything` is a library-defined reduction that applies its second argument to every node in the tree and collects the results with its first argument (`++` is concatenation). Additionally, `mkQ` is a library-defined function, analogous to the `ASTVisitor` base, that takes an argument interested in a single type and produces a query over *any* type. Reusable traversals that transform the tree are also available.

While these traversal algorithms are useful, by the nature of both the object-oriented and functional solutions, the user is locked into a specific traversal strategy. This strategy includes: the order of visitation, the context available when the "interesting" code is called, which nodes are visited or ignored, and when the traversal terminates. For example, in the cases of Elsa, this lock-in means that the user is out of luck if they require the traversal context [91]. Other libraries try to cover more cases by offering several versions of the traversal algorithm. For example, Rose provides the user with two traversal orders and a mechanism, similar to attribute grammars, for passing information from parent to child or child to parent. Eclipse allows the user to prune sub-trees by returning `false` from `visit`. Lastly, SYB allows the user to traverse two trees simultaneously with the traversals in `Data.Generics.Twins`.

Ultimately, however, it is difficult to satisfy all the traversal needs of the user in this manner, and a more sophisticated approach is needed. Hence, newer libraries in both the object-oriented and functional worlds have worked to provide the user composable primitives which span the entire design space of traversals. For object-oriented approaches, this effort has taken the form of *visitor combinators* [79]. JJTraveler, which is part of the ASF+SDF Meta-Environment project [92], supplies tens of such combinators. For example, a top-down traversal can written:

```
public class TopDown implements Visitor {
  private Visitor impl;
  public TopDown(Visitor v) {
    impl = new Sequence(v, new All(this));
  }
  public void visit(Visitable x) throws VisitFailure {
    impl.visit(x);
  }
}
```

Thus, visitors are assembled using composition, not inheritance. To put the new `TopDown` combinator to work, we can write:

```
public class CollectFunc extends NodeVisitor {
  ArrayList funcs = new ArrayList;
  public void visit(Function f) {
    funcs.add(f);
  }
  public static ArrayList findFuncs(Node root) {
    Visitor v = new TopDown(new CollectFunc);
    root.accept(v);
    return v.funcs;
  }
}
```

In the functional world, the combinatorial style of traversal, sometimes called "strategic programming" [93, 94], is supported by the higher-order features of the language. Building on previous classifications of traversal strategies, Ren and Erwig [95] describe a highly-parameterized library, *Reclib*, built on top of SYB. One common thread in both the object-oriented and functional approaches is the inspiration by the work done in term-rewriting [96, 97], particularly the traversal strategies available in Stratego/XT [98].

While this direction of development strives to build up a general library of traversals, another direction has been to burrow down to find the underlying primitives that can be used to *implement* any traversal. Here, again, both object-oriented and functional styles have arrived at the same basic idea: provide a reusable mechanism to "get to" the children of a node, without knowing the parent node's full type. The meaning of "get to" is different for the two styles. In object-oriented programming, this means returning a list of references to the children. For example, the `AnyVisitor` class is used to implement visitor combinators [79]:

```
class AnyVisitable {
  void accept_Any(AnyVisitor);
  int nrOfKids();
  AnyVisitable[] getKid(int);
}
```

Using this interface, the `All` combinator can be written:

```
class All implements Visitor {
  Visitor v;
  public All(Visitor v) { this.v = v; }
  public void visit(AnyVisitable x) {
    for (int i = 0; i < x.nrOfKids(); i++) {
      x.getKid(i).accept(this.v);
    }
  }
}
```

It is easy to see how several different traversals can be implemented with this functionality. In fact, similar interfaces can be found as undocumented functionality of Rose and Eclipse, which is presumably used to implement the public traversals described earlier.

In functional programming, the analogous enumeration of children is done with what Lämmel and Peyton Jones [60] call "the non-recursive map trick" (although they explain that the technique goes back to Meijer *et al.* [86]). Here, instead of providing a mapping

```
children :: Node -> [Node]
```

each data type provides a function which applies a functional argument to each child. Then, from this 1-layer traversal, N-layer traversals (like `everything`) can be built easily. Commenting on the appropriateness of this design, Lämmel and Peyton Jones state:

> The beautiful thing about building a recursive traversal strategy out of non-recursive `gmapT` is that we can build many different strategies using a single definition of `gmapT`.

Despite offering this child enumeration functionality, all these libraries focus on the *traversal* as the primary (or only) public interface. (A notable exception is the Polaris [99] optimizing compiler, whose child enumeration for FORTRAN expressions

was the inspiration for Filter child enumeration.) This is a little surprising because, with child enumeration, the user can compose their own *ad hoc* traversals with very little boilerplate, which was one of the initial reasons for doing all this work in the first place! Based on this observation, the Filter library avoids the research problem of finding a "basis" for the space of traversals by just providing child enumeration directly and integrating the feature well with the rest of the library.

On the other hand, boilerplate reduction is not the only advantage of using library-defined traversals: since traversals have more structure than raw iterations/recursions, they can be more easily optimized and support simpler program reasoning [86]. Thus, there are advantages to a traversal-centric library design that are not present at all in the Filter library.

Filter does have one convenient property that is not immediately present in the approaches described above. As explained at the end of Chapter II and used throughout Chapter IV, Filter integrates case analysis in a way that is shown to provide simple and effective pruning of traversals. With other libraries' mechanisms for child enumeration, all context is lost in the queried list of children, which means there is no simple way to partition the children into lists of "do traverse" and "don't traverse."

To better see this point, we first more precisely describe the correspondence between the Filter and SYB libraries. The five non-Any context variants correspond to five mutually recursive Haskell algebraic data types. The Filter nodes that can appear in a given variant correspond to constructors for the corresponding data type. The Any-context variant then corresponds to the universe of types that a transformation/query built by `mkT`/`mkQ` operates over. The recursive invocations made by `gfoldl` for each child correspond to the nodes returned by `child_any`. Hence, `everywhere` and `everything` correspond to recursion over Any-context variants (using `child_any`), such

as done by `Traverse` in Figure 6.

From this correspondence, a natural question is: what corresponds to a traversal over *other* `X_ctx::Variant` types (using `child_X`)? This has been a key technique for efficiency and correctness in several Filter solutions shown thus far. For example, pg. 48 poses the analysis problem: find the uses of a given variable in the conditions of `if` statements in a given function. Assuming the following subset of Filter, translated into Haskell data types:

```
data Stmt = Block [Stmt] | If Expr Stmt Stmt | Loc_var Type String
data Expr = Call Expr [Expr] | New Type [Expr] | Name_expr String
data Type = Ptr Type | Array Type Expr | Udt_use String
```

a fairly direct mapping of the Filter solution shown on pg. 48 is the following SYB code:

```
findUse :: String -> Expr -> Bool
findUse name (Name_expr s) = (name = s)
findUse _ _ = False

findIf :: String -> Stmt -> Bool
findIf name (If cond _ _) = everything (||) (False `mkQ` (findUse name)) cond
findIf _ _ = False

varInIfCond :: String -> Block -> Bool
varInIfCond name body = everything (||) (False `mkQ` (findIf name)) body
```

However, while this code is technically correct, it will wastefully examine all the expressions and types in the body at least once (twice for expressions in `if` conditions).

One solution to this waste would be to use a pruning traversal, like `everythingBut` . However, this is a *subtractive* method which requires that we enumerate all the nodes we do *not* want to visit. What we would rather do is indicate, additively, the nodes we *do* want to visit. Thus, experience with the Filter library suggests a new addition to the `Data.Generics.Schemes` module:

```
only :: Term a => (r -> r -> r) -> (b -> r) -> r -> a -> r
```

In this signature, the domain of the second argument (`b` above) corresponds to the

choice of `X_ctx::Variant`. This function can be implemented in terms of the same `gfoldl` and `cast` methods used by the other traversal combinators (here, using the simpler `gmapQ` derived from `gfoldl`):

```
only k f r x = case cast x of
                 Just b -> foldl k (f b) (gmapQ (only k f r) x)
                 Nothing -> r
```

Analogous changes can be made for the transformation and monadic traversal versions. With these traversals, the SYB user can express the same type of traversal pruning as the Filter user.

Applying this perspective to the object-oriented libraries, we can see that the `Visitor` base class central to the Visitor pattern effectively forces all nodes to use a single recursive algebraic data type. In Filter terms, this means that library users must deal exclusively with `any_ctx::Variant` during traversal. One approximation of Filter variants/Haskell data types for the purpose of pruning is inheritance. Inheritance allows, for example, the `AnyVisitable` user to use `instanceof`/`dynamic_cast` in the same way as `cast` in `only`. However, whereas Filter variants and Haskell data types represent a set of possibilities, base classes represent commonality in signatures and implementation, which is unlikely to coincide with the former notion. Hence, there does not appear to be a simple way to modify Visitor-based libraries to achieve context-based pruning analogous to the Filter library.

B. Type discovery

Since the nodes in a program are heterogeneous, a common task is to discern the actual type of a node found through a parent or a traversal. For example, if you start with a node of type `If`, representing the `if` statement in a programming language:

```
class If {
  public:
```

```
    const Expr &condition() const;
    const Stmt &then_branch() const;
    const Stmt &else_branch() const;
};
```

after calling `If::condition`, it is necessary to find out the *specific* type of `Expr` that was returned. The same problem appears in functional programming with algebraic-data types. For example, when examining the first argument of the `If` constructor, shown in the following Haskell code:

```
data Stmt = If Expr Stmt Stmt | Do ... | Seq ... | ...
data Expr = Test ... | Call ... | ...
```

we have to ask: "is this `Expr` a `Test`, `Call`, or one of the other choices?"

There are two main ways to answer this question. In the *callback style*, the user provides a set of functions, one for each node type of interest, and one of these functions is called if that node is found. Notable examples of the callback style are the Visitor pattern [41] and the transformations in SYB.

Here is an example of the Visitor pattern being used to discern the type of a `Stmt` node in Pivot:

```
class Stmt_visitor : ipr::Visitor {
  void visit(const ipr::For &n) { ... use f }
  void visit(const ipr::If_then &n) { ... use n }
  ...
};

namespace ipr {
  class For : ipr::Stmt {
    void accept(ipr::Visitor &v) { v.visit(*this); }
    ...
  };
  ... the same for every other ipr::Stmt
}

void take_stmt(const ipr::Stmt &s) {
  Stmt_visitor v;
  s.accept(v);
}
```

Tracing the order of execution, we can see the inversion of control characteristic of the callback style. We start in the user code, `take_stmt`, then go into the library code, `ipr ::For::accept` (or whatever the most-derived type of `s` is), and finally return to the user code, `Stmt_visitor::visit`. The callback style is widely used in source analysis frameworks, including Eclipse, Elsa, Rose, SableCC, and parts of the ASF+SDF Meta-Environment project.

In the second style, which we call the *type-switch* style, the programmer writes something that looks like a `switch` statement over types. For example in Simula 67, one of the earliest examples of a type-switch, we can write:

```
stmt :- ...
INSPECT stmt
  WHEN For DO ...
  WHEN If_then DO ...
  ...
  OTHERWISE ...
...
```

Following the execution through this statement, we can see that there is no inversion of control from user-code to library-code and back again, as in the previous example. Instead, control flow jumps from `Inspect` to the appropriate `When X do Y` or `Otherwise` clause, and then continues after.

Although the syntax varies, type-switches are present in many languages. One example is the `typecase` operation in Common Lisp [100] and Scala [101]. When derived classes (in OO languages) are translated to data constructors of Algebraic Data Types in functional languages, type-switching is just a special case of the more general *pattern-matching* facilities, such as the `match` operation in ML [52] and `case` operation in Haskell [102]. If not present as a single operation, type-switching can be emulated with dynamic type tests, such as `dynamic_cast` in C++ [32] and `instanceof` in Java [103].

The Filter library uses this type-switching style, but it has a disadvantage compared to some of the above techniques. When the Filter user writes:

```
func_ctx::Variant v = ...
switch (v.which()) {
  case func_ctx::loop_e:
    ... v.get<Loop>() ...
}
```

there is a chance to make a typing error, for example calling `get<If>` when `v.which` is not `if_e`. This error is caught by both statements like `INSPECT` and `match`, by nature of being language features that affect the typing rules. The Visitor pattern also catches this type of error, which we suspect is one of the reasons behind its popularity.

C. Iterators

The design of iterators in popular C++ libraries, notably the Standard Template Library (STL) [32] and Boost.Iterator library [33], was the primary inspiration for the lightweight design of the Filter library. Ignoring the fact that iterators present a homogeneous sequence to the user while Filter nodes present a heterogeneous tree, Table I (pg. 22) establishes a metaphor between Filter nodes and iterators to illustrate the way Filter nodes can be created and destroyed as needed. The metaphor goes beyond this, though.

For most STL iterators, the iterator's data consists of a single pointer into the iterated container. However, some iterators need more sophisticated mechanisms: `std ::deque` iterators store 4 pointers which are used to view the underlying sequence-of-segments-of-elements as a sequence-of-elements; input stream iterators store a stream pointer and the last object extracted from the stream in order to split the effectful operation of stream extraction into an effectful increment and pure dereference. Similarly, Filter nodes generally hold two pointers: one to the underlying IPR node and

another to the `Env` structure. Occasionally, extra state is used to cache the result of queries to the IPR. Like STL iterators, though, this state is kept simple and small enough that construction, copying, and destruction is fast.

The Boost.Iterator library provides iterators that significantly modify the user's view of the underlying data. For example, `filter_iterator` allows elements of an underlying sequence to be selectively ignored by a user-specified function object passed to `filter_iterator` on construction. As another example, `transform_iterator` applies a user-specified unary function object to the elements of an underlying sequence to present a new view of the underlying sequence. Similarly, the Filter applies the lowering logic in its `discern` functions (pg. 73) to transform the IPR's interface into the interface of the Filter nodes.

CHAPTER VI

CONCLUSION

This chapter concludes the thesis by revisiting the four design goals described in the Introduction, summarizing how each was realized in the Filter library, and discussing future work.

A. Design goals revisited

In §A of the Introduction, two functional and two stylistic goals were given for the Filter library. We now reconsider these goals, how they have been realized in the Filter library, and the results.

The first functional goal stated was to simplify case analysis and traversal. This goal has been realized in the Filter library by context-specific variants (pg. 22) and polymorphic child-functions (pg. 32), respectively. When applied in the tutorial, the case analysis support helped us remember to consider all the corner cases of the language and, in both the tutorial and comparison, the traversal support allowed us to scrap a lot of boilerplate (pg. 37). Finally, the integration of these two features was shown to provide an effective way to prune traversals (pg. 48, pg. 125, pg. 146).

The second functional goal stated was to perform lowering on the IPR. This goal has been realized by the Filter node types and the lowering algorithm (pg. 59) that produces these nodes. Although each type of lowering performed is unimpressive, the comparison demonstrated a significant overall code reduction (pg. 120). Additionally, this lowering simplified the tutorial due to the absence of, for example, the `ipr::As_type` node (pg. 59).

The first stylistic goal stated was to provide a lightweight view of the IPR. This goal has been realized by the choice of value semantics for Filter types (pg. 22) and

fast implementations for the associated operations (pg. 74, pg. 88). By keeping the Filter close to the IPR, the user is able to easily slip back and forth between the two, as used in the tutorial (pg. 29) and comparison (pg. 134). Unfortunately, when combined with the functional goal of lowering, this choice results in a factor of performance loss for some raw operations (pg. 116). However, this overhead drops off as the amount of "real work" in the analysis increases (pg. 134).

The second stylistic goal stated was to avoid imposing any inversion of control on user code. This has been realized by the use of the *type-switch* method of type discovery instead of the Visitor pattern (pg. 81, pg. 153). Comparison tests (pg. 128, pg. 134) showed that this method allows Filter user code to be written in a more readable manner. This point is made again in the context of Structured Programming (pg. 143).

B. Future work

There are several ways in which this work could be extended in the future to better serve the writer of high-level semantic analyses. In this final section, three promising directions are considered.

The first direction would be to build a traversal library, in the spirit of those described on pg. 146, to provide the performance and reasoning benefits mentioned. Using the child enumeration facilities of the Filter library, such a library could be written with little boilerplate and thus be able to focus on the algorithmic design of the traversals, not the intricacies of C++.

Another direction for extending Filter would be to add new contexts, i.e., new Filter namespaces containing a set of `Variant`, `Members`, `Iter`, and `Range` types, as well any new Filter node types needed. One idea for a new context would be the context of

constant expressions. These are expressions that appear, for example, in the bounds of arrays, the precision of bitfields, the non-type arguments of template identifiers, and the initializers of inline global static constants. This context would consist of the subset of runtime expressions that may be used in these cases and would assist the user in writing analyses like constant propagation and dependency analysis.

A second context to consider adding would be that of *uninstantiated templates*. This would in fact require adding *four* new contexts (`templ_uda_ctx`, `templ_func_ctx` , `templ_expr_ctx`, and `templ_type_ctx`) that mimicked the instantiated versions but allowed the additional possibility of unresolved syntax. This would allow the Filter user to treat templates like concrete entities, rather than families of instantiations. For example, an analysis could be performed once for a template instead of once for every instantiation, yielding better performance. Alternatively, a verification analysis for generic algorithms would also need to analyze uninstantiated bodies.

The last direction for extension would be to support *program transformation* for the same class of high-level semantic analyses as targeted by Filter. Currently, program transformation is supported in Pivot through a low-level `impl` interface to the IPR. A transformation library based on Filter would allow the user to specify intended transformations directly in terms of Filter types. Thus, the user would be able to seamlessly analyze and transform programs in terms of the lowered view presented by the Filter library.

REFERENCES

[1] F. P. Brooks, Jr., *The Mythical Man-month (Anniversary Edition).* Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] H. Sutter and A. Alexandrescu, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series).* Boston, MA: Addison-Wesley Professional, 2004.

[3] "Joint strike fighter, air vehicle, C++ coding standard," Lockheed Martin, Tech. Rep. 2RDU00001 Rev C, December 2005.

[4] G. Kroah-Hartman, "Kernel korner: proper Linux kernel coding style," *Linux J.*, vol. 2002, no. 99, p. 7, 2002.

[5] I. F. Darwin, *Checking C Programs with Lint.* Sebastopol, CA: O'Reilly & Associates, Inc., 1986.

[6] D. Gregor and S. Schupp, "STLlint: lifting static checking from languages to libraries," *Softw. Pract. Exper.*, vol. 36, no. 3, pp. 225–254, 2006.

[7] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, 2007.

[8] L. Cardelli, "Typeful programming," Digital Systems Research Center, Palo Alto, CA, Tech. Rep. 45, May 1989.

[9] J. C. Reynolds, "Syntactic control of interference," in *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* New York: ACM, 1978, pp. 39–46.

[10] J.-P. Talpin and P. Jouvelot, "The type and effect discipline," *Inf. Comput.*, vol. 111, no. 2, pp. 245–296, 1994.

[11] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security.* Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[12] G. Morrisett, "Programming with effects in Coq," in *MPC '08: Proceedings of the 9th International Conference on Mathematics of Program Construction.* Berlin: Springer-Verlag, 2008, pp. 28–28.

[13] D. Dechev, N. Rouquette, P. Pirkelbauer, and B. Stroustrup, "Verification and semantic parallelization of goal-driven autonomous software," in *Autonomics '08: Proceedings of 2nd International Conference on Autonomic Computing and Communication Systems.* Turin, Italy: ACM, 2008.

[14] "Cscope home page," http://cscope.sourceforge.net, Mar. 2009.

[15] "Doxygen," http://www.doxygen.org, Mar. 2009.

[16] "LXR / the Linux cross reference," http://lxr.linux.no, Mar. 2009.

[17] G. D. Parrington, "A stub generation system for C++," The University of Newcastle upon Tyne, Tech. Rep., 1995.

[18] I. Kiselev, *Aspect-Oriented Programming with AspectJ.* Indianapolis, IN: Sams Publishing, 2002.

[19] D. B. Loveman, "Program improvement by source to source transformation," in *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages.* New York: ACM, 1976, pp. 140–152.

[20] R. Metzger and Z. Wen, *Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization.* Cambridge, MA: MIT Press, 2000.

[21] R. V. Bennett, A. C. Murray, B. Franke, and N. Topham, "Combining source-to-source transformations and processor instruction set extensions for the automated design-space exploration of embedded systems," *SIGPLAN Not.*, vol. 42, no. 7, pp. 83–92, 2007.

[22] M. J. Wolfe, *High Performance Compilers for Parallel Computing.* Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[23] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* San Francisco, CA: Morgan Kaufmann Publishers Inc., 2002.

[24] S. Rus, M. Pennings, and L. Rauchwerger, "Sensitivity analysis for automatic parallelization on multi-cores," in *ICS '07: Proceedings of the 21st Annual International Conference on Supercomputing.* New York: ACM, 2007, pp. 263–273.

[25] "The jargon file," http://www.catb.org/~esr/jargon/html/C/crufty.html, Mar. 2009.

[26] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code.* Boston, MA: Addison-Wesley Professional, June 1999.

[27] M. B. Siff, "Techniques for Software Renovation," Ph.D. dissertation, University of Wisconsin at Madison, 1999.

[28] M. Harsu, "Re-engineering Legacy Software through Language Conversion," Ph.D. dissertation, University of Tampere, 2000.

[29] "Edison Design Group," http://www.edg.com, Mar. 2009.

[30] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA: MIT Press, 2002.

[31] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[32] International Organization for Standardization, *ISO/IEC 14882:2003: Programming Languages: C++*, 2nd ed. Geneva, Switzerland: International Organization for Standardization, Oct. 2003.

[33] *Boost Iterator Library*, C++ Boost, Mar. 2009, http://www.boost.org/libs/iterator.

[34] R. E. Johnson and B. Foote, "Designing reusable classes," *Object-Oriented Programming*, vol. 1, no. 2, pp. 22–35, 1988.

[35] R. E. Sweet, "The Mesa programming environment," in *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*. New York: ACM, 1985, pp. 216–229.

[36] J. Vlissides, "Protection, part I: the Hollywood principle," *C++ Report*, Feb. 1996.

[37] E. M. Gagnon and L. J. Hendren, "Sablecc, an object-oriented compiler framework," in *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*. Washington, DC: IEEE Computer Society, 1998, p. 140.

[38] "Elkhound and Elsa," http://www.cs.berkeley.edu/~smcpeak/elkhound, Mar. 2009.

[39] "Eclipse Java development tools (JDT)," http://www.eclipse.org/jdt, Mar. 2009.

[40] D. J. Quinlan, M. Schordan, Q. Yi, and B. R. de Supinski, "Semantic-driven parallelization of loops operating on user-defined containers," in *LCPC '03: Languages and Compilers for Parallel Computing, 16th International Workshop*. Berlin: Springer-Verlag, Oct. 2003, pp. 524–538.

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. New York: Addison-Wesley Publishing Co., 1995.

[42] E. W. Dijkstra, "Some questions," Nov. 1974, circulated privately. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD463.PDF

[43] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[44] "Phoenix compiler infrastructure," http://research.microsoft.com/phoenix, Mar. 2009.

[45] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai, and S. Chan, "A compiler framework for speculative analysis and optimizations," in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. New York: ACM, 2003, pp. 289–299.

[46] "Gnu compiler collection (GCC) internals," http://gcc.gnu.org/onlinedocs/gccint, Mar. 2009.

[47] D. J. Pearce, P. H. Kelly, and C. Hankin, "Efficient field-sensitive pointer analysis of C," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, p. 4, 2007.

[48] D. Gregor, M. Marcus, T. Witt, and A. Lumsdaine, "Foundational concepts for the C++0x standard library," ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming language C++, Tech. Rep. N2677=08-0187, Jun. 2008. [Online]. Available: www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2677.pdf

[49] "GCC, the gnu compiler collection," http://gcc.gnu.org, Mar. 2009.

[50] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz — open source graph drawing tools," in *Graph Drawing, 9th International Symposium, GD 2001*. Berlin: Springer, Sep. 2001.

[51] G. Birtwhistle, O. Dahl, B. Myhrhaug, and K. Nygaard, *Simula Begin*. London: Chartwell-Bratt Ltd, 1979.

[52] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*. Cambridge, MA: MIT Press, 1990.

[53] R. I. Bull, A. Trevors, A. J. Malton, and M. W. Godfrey, "Semantic grep: Regular expressions + relational abstraction," in *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*. Washington, DC: IEEE Computer Society, 2002, p. 267.

[54] H. A. Müller and K. Klashinsky, "Rigi-a system for programming-in-the-large," in *ICSE '88: Proceedings of the 10th International Conference on Software Engineering*. Los Alamitos, CA: IEEE Computer Society Press, 1988, pp. 80–86.

[55] M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Mueller, "On integrating visualization techniques for effective software exploration," in *INFOVIS '97: Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97)*. Washington, DC: IEEE Computer Society, 1997, p. 38.

[56] S. Robitaille, R. Schauer, and R. K. Keller, "Bridging program comprehension tools by design navigation," in *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*. Washington, DC: IEEE Computer Society, 2000, p. 22.

[57] *Understand for C++: User Guide and Reference Manual*, Scientific Toolworks, Inc, Mar. 2009, http://www.scitools.com.

[58] *User guide*, Imagix, http://www.imagix.com/support/docs.html.

[59] M. Abadi and L. Cardelli, *A Theory of Objects*. Secaucus, NJ: Springer-Verlag Inc., 1996.

[60] R. Lämmel and S. P. Jones, "Scrap your boilerplate: a practical design pattern for generic programming," *ACM SIGPLAN Notices*, vol. 38, no. 3, pp. 26–37, Mar. 2003.

[61] B. Stroustrup, "A rationale for semantically enhanced library languages," in *Proceedings of the First International Workshop on Library-Centric Software Design (LCSD '05)*, 2006, as Technical Report 06-12 of Rensselaer Polytechnic Institute, Computer Science Department.

[62] S. Guyer and C. Lin, "Broadway: A compiler for exploiting the domain-specific semantics of software libraries," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 342–357, Feb. 2005.

[63] G. D. Reis and B. Stroustrup, "Specifying C++ concepts," in *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York: ACM Press, 2006, pp. 295–308.

[64] A. Stepanov and P. McJones, *Elements of Programming*. New York: Addison-Wesley Professional, 2009.

[65] J. B. Kam and J. D. Ullman, "Global data flow analysis and iterative algorithms," *J. ACM*, vol. 23, no. 1, pp. 158–171, 1976.

[66] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, CA: ACM Press, New York, 1977, pp. 238–252.

[67] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 1997.

[68] P. Pirkelbauer, S. Parent, M. Marcus, and B. Stroustrup, "Runtime concepts for the C++ standard template library," in *SAC '08: Proceedings of the 2008 ACM Symposium on Applied Computing*. New York: ACM, 2008, pp. 171–177.

[69] T. L. Veldhuizen, "Expression templates," *C++ Report*, vol. 7, no. 5, pp. 26–31, Jun. 1995.

[70] B. Stroustrup, *The C++ Programming Language (Third Edition and Special Edition)*. New York: Addison-Wesley Publishing Co., 1997.

[71] *Boost Pool Library*, C++ Boost, Mar. 2009, http://www.boost.org/libs/pool.

[72] H. Sutter, *More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[73] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr., "Lock-free reference counting," in *PODC '01: Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*. New York: ACM, 2001, pp. 190–199.

[74] J. Bonwick, "The slab allocator: an object-caching kernel memory allocator," in *USTC'94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference*. Berkeley, CA: USENIX Association, 1994, pp. 6–6.

[75] J. Bonwick and J. Adams, "Magazines and vmem: Extending the slab allocator to many cpus and arbitrary resources," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA: USENIX Association, 2001, pp. 15–33.

[76] B. K. Rosen, "High-level data flow analysis," *Commun. ACM*, vol. 20, no. 10, pp. 712–724, 1977.

[77] P. Pirkelbauer, Y. Solodkyy, and B. Stroustrup, "Open multi-methods for C++," in *GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering*. New York: ACM, 2007, pp. 123–134.

[78] H. E. Hinnant, B. Stroustrup, and B. Kozicki, "A brief introduction to rvalue references," ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming language C++, Tech. Rep. N2027=06-0097, 2006.

[79] J. Visser, "Visitor combination and traversal control," in *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. New York: ACM, 2001, pp. 270–282.

[80] "Technical report on C++ performance," ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Tech. Rep. N1487=03-0070, 2003.

[81] E. W. Dijkstra, "Notes on structured programming," in *Structured Programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. London, UK: Academic Press Ltd., 1972, ch. 1.

[82] S. G. McPeak, "Elkhound: A fast, practical GLR parser generator," University of California at Berkeley, Berkeley, Tech. Rep., 2003.

[83] A. van Deursen and J. Visser, "Source model analysis using the JJTraveler visitor combinator framework," *Softw., Pract. Exper.*, vol. 34, no. 14, pp. 1345–1379, 2004.

[84] R. S. Bird, "An introduction to the theory of lists," in *Logic of Programming and Calculi of Discrete Design*, M. Broy, Ed. New York: Springer-Verlag, 1987, pp. 3–42.

[85] L. Meertens, "Algorithmics — towards programming as a mathematical activity," in *Mathematics and Computer Science*, ser. CWI Monographs Volume 1, J. de Bakker, M. Hazewinkel, and J. Lenstra, Eds. Amsterdam: North-Holland Publishing Company, 1986, pp. 289–334.

[86] E. Meijer, M. M. Fokkinga, and R. Paterson, "Functional programming with

bananas, lenses, envelopes and barbed wire," in *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. London, UK: Springer-Verlag, 1991, pp. 124–144.

[87] R. Lämmel and J. Visser, "A Strafunski Application Letter," in *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, ser. LNCS, V. Dahl and P. Wadler, Eds., vol. 2562. Berlin: Springer-Verlag, Jan. 2003, pp. 357–375.

[88] R. Lämmel and S. P. Jones, "Scrap more boilerplate: reflection, zips, and generalised casts," in *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*. New York: ACM Press, 2004, pp. 244–255.

[89] ——, "Scrap your boilerplate with class: extensible generic functions," in *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*. New York: ACM Press, Sep. 2005, pp. 204–215.

[90] D. Clarke and A. Löh, "Generic Haskell, specifically," in *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*. Deventer, The Netherlands: Kluwer, B.V., 2003, pp. 21–47.

[91] "cc.ast: The abstract syntax tree description," http://www.cs.berkeley.edu/~smcpeak/elkhound/sources/elsa/doc/cc.ast.html, Mar. 2009.

[92] M. G. J. van den Brand, A. v. Deursen, J. Heering, H. A. d. Jong, M. d. Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser, "The ASF+SDF meta-environment: a component-based language development environment," in *CC '01: Proceedings of the 10th International Conference on Compiler Construction*. London, UK: Springer-Verlag, 2001, pp. 365–370.

[93] R. Lämmel, E. Visser, and J. Visser, "The Essence of Strategic Programming," Oct.15 2002, 18 p.; Draft; Available at http://www.cwi.nl/~ralf.

[94] R. Lämmel and J. Visser, "Design Patterns for Functional Strategic Programming," in *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*. Pittsburgh: ACM Press, Oct.5 2002, 14 pages.

[95] D. Ren and M. Erwig, "A generic recursion toolbox for Haskell or: scrap your boilerplate systematically," in *Haskell '06: Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*. New York: ACM, 2006, pp. 13–24.

[96] M. G. J. van den Brand, P. Klint, and J. J. Vinju, "Term rewriting with traversal functions," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 2, pp. 152–190, 2003.

[97] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser, "Program transformation with scoped dynamic rewrite rules," *Fundam. Inf.*, vol. 69, no. 1-2, pp. 123–178, 2005.

[98] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/XT 0.17. A language and toolset for program transformation," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 52–70, June 2008.

[99] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, "Parallel programming with Polaris," *Computer*, vol. 29, no. 12, pp. 78–82, 1996.

[100] G. L. Steele, Jr., *Common LISP: the Language (2nd ed.)*. Newton, MA: Digital Press, 1990.

[101] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "The Scala language specification version 2.7," Jan. 2009, www.scala-lang.org/docu/files/ScalaReference.pdf.

[102] S. P. Jones, Ed., *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, September 2002. [Online]. Available: http://haskell.org/definition/haskell98-report.pdf

[103] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java^{TM} Language Specification*, 3rd ed. Reading, MA: Addison-Wesley Professional, 2005.

APPENDIX A

LISTINGS

The appendix contains the input files and code listings referred to by Chapter IV.

A. uda-single

The declarations in each class form the input sets to the uda-single test described on pg. 116.

```
class All
{
  int x1;
  void foo() {}
  int y1 : 3;
  int y2 : 8;
  int x2;
  void bar() {}
  void baz() {}
  int x3;
};
class Some
{
  friend void foo();
  static int x;
  void bar() {}
  typedef int X;
  static void baz();
  union U {};
  int f : 8;
  enum E { E1, E2 };
  typedef float F;
  int y;
};
class None
{
  friend void foo();
  static int x;
  typedef int X;
  static void baz();
```

```
  union U {};
  enum E { E1, E2 };
  class D {};
};
```

B. expr-single

The expressions in each function form the input sets to the expr-single test described on pg. 120.

```
namespace N { void foo(); } // for use in tests
struct A { A operator+(A); }; // for use in tests

void all(A a)
{
  ::N::foo();
  (::N::foo)();
  N::foo();
  (N::foo)();
  a + a;
  a.operator+(a);
  a.::A::operator+(a);
  a.A::operator+(a);
}
void some(A a)
{
  N::foo();
  throw 1;
  dynamic_cast<A *>(&a);
  a + a;
  1 + 2;
  a.::A::operator+(a);
  new A;
  (a.*(&A::operator+))(a);
}
void none(A a)
{
  throw 1;
  dynamic_cast<A *>(&a);
  1 + 2;
  (a.*&A::operator+)(a);
  throw 1;
  dynamic_cast<A *>(&a);
  1 + 2;
```

```
  (a.*&A::operator+)(a);
}
```

C. expr-tree1

The three expressions in `foo` are the inputs to the expr-tree1 test described on pg. 128.

```
struct A { A operator+(A); }; // for use in tests
namespace N { A foo(A); } // for use in tests

void foo(A a)
{
  a.operator+(N::foo(a + ::N::foo(a)) + a.A::operator+(a.operator+(a)));
  (&N::foo(a) == &a) ? 1 : ((&(a + a) + 2) == (new A) ? sizeof(A) : 4);
  *((1 == 2) ? new int(1) : ((int *)4)) = (sizeof(A) > 1 ? 0 : throw "things");
}
```

D. expr-tree2

The three expressions in `foo` are the inputs to the expr-tree2 test described on pg. 128.

```
void foo(int i, int j, int k)
{
  (i + j) * k;
  ::a(i + j, sizeof(j), *(new int (1))), delete new double, a(1,2,3);
  (*(&i < &j ? &j : &k)) = (*(new int) = (i > k ? j + j : throw 5));
}
```

E. layout-compatible

1. Implementation

The following implements `filter_equal` from pg. 134:

```
// utility function to test whether two expressions are literals with the same
// exact value. does not perform constant-folding
inline bool same_literal(expr_ctx::Variant e1, expr_ctx::Variant e2)
{
  if (e1.which() != e2.which() || e1.which() != expr_ctx::literal_e)
    return false;
  return equal(e1.get<Literal>().string(), e2.get<Literal>().string());
}
```

```
// return whether the types are equal
inline bool filter_equal(type_ctx::Variant v1, type_ctx::Variant v2)
{
  // ipr::Type* implies equal types, but has false negatives
  if (v1.node().ipr() == v2.node().ipr() && v1.node().ipr() != 0)
    return true;

  // handle equality at the node
  if (v1.which() != v2.which())
    return false;
  switch (v1.which())
  {
    // definitely not equal because ipr::Type* equality test above failed
    case type_ctx::cpp_builtin_e:
    case type_ctx::non_cpp_builtin_e:
      return false;

    // use of UDT equal if ipr::Udt* equal
    case type_ctx::udt_use_e:
      if (v1.get<Udt_use>().which() != v2.get<Udt_use>().which())
        return false;
      return &v1.get<Udt_use>().udt_ipr() == &v2.get<Udt_use>().udt_ipr();

    // check equality at node, then 'break' to check equality at children
    case type_ctx::ptr_e:
    case type_ctx::ref_e:
    case type_ctx::ptr_to_member_e:
      break;
    case type_ctx::qualified_e:
      if (v1.get<Qualified>().ipr().qualifiers() !=
            v2.get<Qualified>().ipr().qualifiers())
        return false;
      break;
    case type_ctx::array_e:
      if (!v1.get<Array>().has_bound() || !v2.get<Array>().has_bound() ||
          !same_literal(v1.get<Array>().bound(), v2.get<Array>().bound()))
        return false;
      break;
    case type_ctx::func_t_e:
      if (v1.get<Func_t>().ellipsis() != v2.get<Func_t>().ellipsis())
        return false;
      break;
  }

  // test equality of children, same number of children
  type_ctx::Range r1 = v1.child_type(), r2 = v2.child_type();
```

```
  for (; !r1.empty() && !r2.empty(); ++r1.first, ++r2.first)
    if (!filter_equal(*r1.first, *r2.first))
      return false;
  if (!r1.empty() || !r2.empty())
    return false;

  return true;
}
```

The following implements `traditional_equal` and `reduce_to_udt` from pg. 134:

```
// utility function for checking that two expressions refer to the same Literal
inline bool same_literal(const ipr::Expr &e1, const ipr::Expr &e2)
{
  const ipr::Literal *l1 = ipr_typeid<ipr::Literal>(e1),
                     *l2 = ipr_typeid<ipr::Literal>(e2);
  if (!l1 || !l2)
    return false;
  return equal(l1->string(), l2->string());
}

// utility function for removing unneeded Scope_refs from As_type::expr
inline const ipr::Expr &strip_scope_ref(const ipr::Expr &in)
{
  const ipr::Expr *e = &in;
  while (const ipr::Scope_ref *s = ipr_typeid<ipr::Scope_ref>(*e))
    e = &s->member();
  return *e;
}

const ipr::Typedecl *reduce_to_td(const ipr::As_type &);

// visitor for Id_expr::resolution() that pulls out Typedecls
struct Get_typedecl_visitor : Noop_visitor
{
  const ipr::Typedecl *td;

  Get_typedecl_visitor() : td(0) {}

  void visit(const ipr::Typedecl &n) { td = &n; }

  // Alias may eventually resolve to Typedecl
  void visit(const ipr::Alias &n)
  {
    if (const ipr::As_type *a = ipr_typeid<ipr::As_type>(n.type()))
      td = reduce_to_td(*a);
  }
};

// if this As_type refers to the use of a Typedecl, return it
```

```
const ipr::Typedecl *reduce_to_td(const ipr::As_type &at)
{
  const ipr::Expr &e = strip_scope_ref(at.expr());
  if (const ipr::Id_expr *i = ipr_typeid<ipr::Id_expr>(e)) {
    Get_typedecl_visitor vis;
    i->resolution().accept(vis);
    return vis.td;
  }
  return 0;
}

const ipr::Udt *reduce_to_udt(const ipr::As_type &);

// visitor for Id_expr::resolution() that pulls out Udts
struct Get_uda_action
{
  const ipr::Udt *udt;

  Get_uda_action() : udt(0) {}

  void operator()(const ipr::Udt &n) { udt = &n; }

  // typedecl may be for a Udt
  void operator()(const ipr::Typedecl &n)
  {
    if (n.has_initializer()) {
      if (const ipr::Udt *u = ipr_dynamic_cast<ipr::Udt>(n.initializer()))
        udt = u;
    }
  }

  void operator()(const ipr::Alias &n)
  {
    if (const ipr::As_type *a = ipr_typeid<ipr::As_type>(n.type()))
      udt = reduce_to_udt(*a);
  }

  void operator()(const ipr::Node &) {}
};

// if this As_type refers to the use of a Udt, return it
const ipr::Udt *reduce_to_udt(const ipr::As_type &at)
{
  const ipr::Expr &e = strip_scope_ref(at.expr());
  if (const ipr::Id_expr *i = ipr_typeid<ipr::Id_expr>(e)) {
    Visitor_to_overload<Get_uda_action> vis;
    i->resolution().accept(vis);
    return vis.act.udt;
  }
```

```
  return 0;
}

bool traditional_equal(const ipr::Type &t1, const ipr::Type &t2);

struct Equal_types_check : Noop_visitor
{
  bool equal;
  const ipr::Type &type2;

  Equal_types_check(const ipr::Type &t2) : equal(false), type2(t2) {}

  // reused by visit(Product) and visit(Sum)
  bool equal_seq(const ipr::Sequence<ipr::Type> &seq1,
                 const ipr::Sequence<ipr::Type> &seq2)
  {
    int sz1 = seq1.size(), sz2 = seq2.size();
    if (sz1 != sz2)
      return false;
    for (int i = 0; i != sz1; ++i)
      if (!traditional_equal(seq1[i], seq2[i]))
        return false;
    return true;
  }

  // for each case below, compare equality at node and then check children:

  void visit(const ipr::Pointer &n)
  {
    if (const ipr::Pointer *p = ipr_typeid<ipr::Pointer>(type2))
      equal = traditional_equal(n.points_to(), p->points_to());
  }

  void visit(const ipr::Reference &n)
  {
    if (const ipr::Reference *r = ipr_typeid<ipr::Reference>(type2))
      equal = traditional_equal(n.refers_to(), r->refers_to());
  }

  void visit(const ipr::Array &n)
  {
    if (const ipr::Array *a = ipr_typeid<ipr::Array>(type2))
      equal = same_literal(n.bound(), a->bound()) &&
              traditional_equal(n.element_type(), a->element_type());

  }

  void visit(const ipr::Function &n)
  {
```

```
    if (const ipr::Function *f = ipr_typeid<ipr::Function>(type2))
      equal = traditional_equal(n.source(), f->source()) &&
              traditional_equal(n.target(), f->target()) &&
              traditional_equal(n.throws(), f->throws());
  }

  void visit(const ipr::Ptr_to_member &n)
  {
    if (const ipr::Ptr_to_member *p = ipr_typeid<ipr::Ptr_to_member>(type2))
      equal = traditional_equal(n.containing_type(), p->containing_type()) &&
              traditional_equal(n.member_type(), p->member_type());
  }

  void visit(const ipr::Qualified &n)
  {
    if (const ipr::Qualified *q = ipr_typeid<ipr::Qualified>(type2))
      equal = n.qualifiers() == q->qualifiers() &&
              traditional_equal(n.main_variant(), q->main_variant());
  }

  void visit(const ipr::Product &n)
  {
    if (const ipr::Product *p = ipr_typeid<ipr::Product>(type2))
      equal = equal_seq(n.elements(), p->elements());
  }

  void visit(const ipr::Sum &n)
  {
    if (const ipr::Sum *s = ipr_typeid<ipr::Sum>(type2))
      equal = equal_seq(n.elements(), s->elements());
  }

  void visit(const ipr::As_type &n)
  {
    // Since &n != &type2, only chance for equality is:
    // As_type -> Id_expr -> Typedecl, and the two Typedecls are equal
    if (const ipr::Typedecl *td1 = reduce_to_td(n))
      if (const ipr::As_type *a = ipr_typeid<ipr::As_type>(type2))
        if (const ipr::Typedecl *td2 = reduce_to_td(*a))
          equal = td1 == td2;
  }
};

// return whether two types are equal (identical)
bool traditional_equal(const ipr::Type &t1, const ipr::Type &t2)
{
  // ipr::Type* implies equal types, but has false negatives
  if (&t1 == &t2)
```

```
    return true;
  // test equality with structural recursion
  Equal_types_check vis(t2);
  t1.accept(vis);
  return vis.equal;
}
```

2. Inputs

The sets of pairs of classes in `Deep` and `Shallow` form the inputs to the layout-compatible test on pg. 134.

```
namespace N // contents used in tests below
{
  struct X { float x; };
  struct Y { float y; };
  struct U { Y y; X *p; };
  struct V { X x; X *px; };
  struct W { Y y; Y *py; };
}
namespace Deep
{
  struct A {
    int i;
    N::X x;
  };
  struct B {
    int j;
    N::Y y;
  };

  struct C {
    int *(*pf)(N::X *);
    N::V v;
  };
  struct D {
    int *(*pf)(N::X *);
    N::W w;
  };

  struct E {
    A pa;
    A *pb;
    N::U u;
```

```
  };
  struct F {
    B pa;
    A *pb;
    N::V v;
  };

  struct H
  {
    F f;
    int i;
  };
  struct I {
    A a;
    E e;
  };
}
namespace Shallow
{
  struct A {
    One::A a1;
    One::E e1;
    One::A a2;
    One::E e2;
  };
  struct B {
    One::A a1;
    One::E e1;
    One::A a2;
    One::E e2;
  };

  struct C {
    int i;
    float *j;
    double k;
  };
  struct D {
    int i;
    float *j;
    double k;
  };

  struct E {
    float f;
```

```
    int *i;
  };
  struct F {
    float g;
    float *j;
  };

  struct G {
    N::X *x;
  };
  struct H {
    N::Y *y;
  };
}
```

VITA

Name:	Luke A. Wagner
Address:	Department of Computer Science and Engineering Texas A&M University TAMU 3112 College Station, TX 77843
Email:	andhow@gmail.com

Education:	B.S., Computer Science, Texas A&M University, 2006 M.S., Computer Science, Texas A&M University, 2009
Internships:	National Instruments - Instrument driver group (2003) Microsoft - Systems management server (2005) Microsoft - Windows kernel (2006) IBM - AIX kernel (2007)
Employment:	Mozilla (2009 - Present)