SIMPLIFYING THE ANALYSIS OF C++ PROGRAMS

A Dissertation
by
YURIY SOLODKYY

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

DOCTOR OF PHILOSOPHY

Chair of Committee,	Bjarne Stroustrup
Committee Members,	Jaakko Järvi
	Lawrence Rauchwerger
	Sergiy Butenko
Head of Department,	Duncan M. (Hank) Walker

August 2013

Major Subject: Computer Science and Engineering

ABSTRACT

Based on our experience of working with different C++ front ends, this thesis identifies numerous problems that complicate the analysis of C++ programs along the entire spectrum of analysis applications. We utilize library, language, and tool extensions to address these problems and offer solutions to many of them. In particular, we present efficient, expressive and non-intrusive means of dealing with abstract syntax trees of a program, which together render the visitor design pattern obsolete. We further extend C++ with open multi-methods to deal with the broader expression problem. Finally, we offer two techniques, one based on refining the type system of a language and the other on abstract interpretation, both of which allow developers to statically ensure or verify various run-time properties of their programs without having to deal with the full language semantics or even the abstract syntax tree of a program. Together, the solutions presented in this thesis make ensuring properties of interest about C++ programs available to average language users.

ACKNOWLEDGEMENTS

> Knowledge is in the end based on acknowledgement.
>
> Ludwig Wittgenstein

This Ph.D. thesis would not have been possible without direct or indirect support of many individuals who I was fortunate enough to meet through life and be friends with. I hope I did not forget anyone, as these were the people who taught me values, shown new horizons and inspired me with their encouragements.

First and foremost, I would like to express all my gratitude to my dissertation advisor and mentor, Bjarne Stroustrup, for believing in me despite my initial failures, and for creating an excellent research environment in the Programming Techniques, Tools and Languages Group that stimulates creative thinking, fosters new ideas and promotes collaboration. I would also like to thank him for creating C++, which became my hobby, my research, my bread and butter and my lingua franca to understanding other programming languages.

I am very grateful to Jaakko Järvi, who has been my unofficial supervisor all this time not only in computer science, but also in squash and writing. I would also like to thank Gabriel Dos Reis for forcing me to take his classes, which always were a source of great knowledge and research ideas. I am greatly indebted to Lawrence Rauchwerger for teaching me to listen and encouraging me to pursue my dreams. And I would like to thank Sergiy Butenko for helping me never feel homesick as well as being my emergency contact on all the forms, though probably without knowing it.

I am particularly thankful for all the discussions, collaborations and exchange of ideas we had with Peter Pirkelbauer, Andrew Sutton, Abe Skolnik, Luke Wagner, Jason Wilkins, Damian Dechev, Jacob Smith, Yue Li, Stephen Cook, John Freeman, Michael Lopez, Xiaolong Tang, Gabriel Foust, Olga & Roger Pearce, Mani Zandifar, Troy McMahon, Ioannis Papadopoulos, Nathan Thomas, Timmie Smith and many others in the Parasol Lab.

My doctoral studies would not have been possible without support and endorsement from my Master thesis advisor Volodymyr Protsenko, as well as professors Yuriy Belov and Mykola Glybovets. I am also greatly indebted to Roman Pyrtko, Khrystyna Lavriv, Igor Panchuk and Yaroslav Vasylenko who taught me true values during times when no one cared about those.

I would like to express my gratitude to all the friends who encouraged me through the years, no matter how long I did not answer their emails or return their phone calls: Alexander Boyko, Natasha Gulayeva, Roman Lototskyy, Iryna & Anna Khapko, Konstantyn Volyanskyy, Svyatoslav Trukhanov, Irina Shatruk, Jorge Dorribo Camba, Karen Triff, Zhanna Nepiyushchikh, Olga & Anatoliy Gashev, Mike & Elena Kolomiets, Ian Murray, Amineh Kamranzadeh, Gregory Berkolaiko and many, many others who always reminded me that there is more to life than grad school.

I would like to reserve my special thanks to Vahideh whose patience, support and encouragement were the driving force behind completing this thesis. Finally, I am forever grateful to my parents, Iryna and Myroslav, as well as my brother Eugene for their endless encouragement and love. This work is dedicated to them.

NOMENCLATURE

ABI	Application Binary Interface
ACM	Association for Computing Machinery
ADT	Algebraic Data Type
AST	Abstract Syntax Tree
DAG	Directed Acyclic Graph
DLL	Dynamically Linked Library
DSL	Domain-Specific Language
EDG	Edison Design Group
GADT	Generalized Algebraic Data Type
GCC	GNU Compiler Collection
I/O	Input/Output
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IPR	Internal Program Representation
ISO	International Organization for Standardization
LOC	Lines Of Code
LLVM	Low Level Virtual Machine
MPL	Boost Metaprogramming Library
OMD	Open-Method Description
OOP	Object-Oriented Programming
RTTI	Run-Time Type Information
SAIL	SAIL Abstract Interpretation Library
SELL	Semantically Enhanced Library Language
STL	Standard Template Library
XML	eXtensible Markup Language
XPR	eXternal Program Representation
XTL	eXtensible Typing Library

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. INTRODUCTION

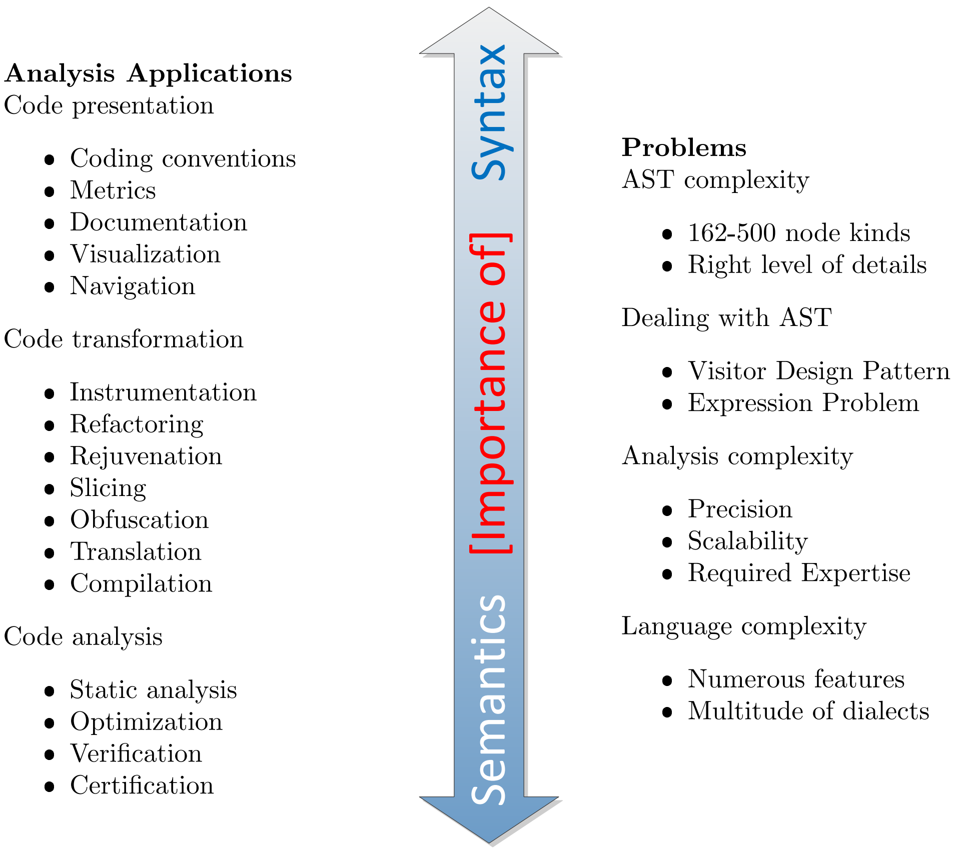
> Everything should be made as simple as possible, but no simpler.
>
> Albert Einstein

C++ has grown into a language in which the power of its abstraction mechanisms is a weakness of its analysis. Writing a tool capable of assuring a property that falls out of scope of the standard type checking performed by a compiler is a task comparable to writing the compiler itself. Availability of commercial and open-source front ends for C++ does not solve the problem for an average language user, because one has to get a very deep understanding of the language semantics as well as make oneself familiar with a concrete representation of a program's *Abstract Syntax Tree* (*AST*). What is more, the analysis of the abstract syntax tree is often written in a different language due to the alleged lack of expressive facilities for dealing with them in C++. Allegation is false; we believe that C++ has sufficiently general facilities capable of making ensuring properties of interest about a program expressive, efficient and available to non-experts. We demonstrate that with proper abstractions the complexity of the language's semantics can be hidden from the user, making the task of ensuring certain guarantees about a program not more difficult than writing the program itself.

1.1 Motivation

Software has become so integrated into our daily lives we often do not even realize its presence in tools and gadgets surrounding us. Simple toys and food processors that seem entirely mechanical; cameras and cell phones that became more powerful and functional than personal computers just a decade ago; street lights and cashier machines augmented with computer vision; cars, trains and airplanes — all today depend to various extents on embedded chips that run software of various degrees of complexity. As the software became ubiquitous, the guarantees of its quality, safety and compliance with the standards became increasingly important in the process of software development. For example, code developed in large teams often has to conform to certain coding conventions or standards like JSF++ [159], MISRA C++ [184] or CERT C++ [231]; its quality assurance may require the code to be instrumented with additional instructions before automated testing; finally, some code deployed on airplanes has to be proven correct by formal means to assure absence of run-time errors. While all these applications require to different extents the ability to analyze and modify the program's structure automatically, they differ in what aspects of the input program they find important. The left side of Figure 1.1 lists some common program-analysis tasks along the syntax-semantics spectrum depending on the kind of details that are important to the analysis.

The analyses at the top of the spectrum are more interested in syntactic information than semantic. They prefer the original source code with comments preserved and macros not expanded; the exact spacing used to indent the code and the exact positioning of various program elements in the line; they also easily tolerate erroneous and incomplete programs. As we go down the list, the semantic information becomes important too since the analyses may need to know the exact entity a given name refers to or the result of evaluating a compile-time meta-function. The syntactic information is still required, though less detailed. Towards the end of the list, the need for semantic information clearly dominates, as we need



Analysis Applications
Code presentation

- Coding conventions
- Metrics
- Documentation
- Visualization
- Navigation

Code transformation

- Instrumentation
- Refactoring
- Rejuvenation
- Slicing
- Obfuscation
- Translation
- Compilation

Code analysis

- Static analysis
- Optimization
- Verification
- Certification

Problems
AST complexity

- 162-500 node kinds
- Right level of details

Dealing with AST

- Visitor Design Pattern
- Expression Problem

Analysis complexity

- Precision
- Scalability
- Required Expertise

Language complexity

- Numerous features
- Multitude of dialects

Figure 1.1: Analysis applications and problems with them

to reason about possible and impossible run-time behaviors, the well-formedness of various constructs etc. Analyses at this level typically assume that the program has already passed the type checker and thus exclude erroneous and incomplete translation units. It is worth noting that the spectrum above greatly coincides with the kind of applications that are typically implemented in the front end, middle end and back end of a language processor.

1.2 Problem Description

The need for various kinds of information creates different kinds of problems at each level of the spectrum of Figure 1.1, shown there on the right. We follow the same direction to identify the most common ones.

1.2.1 AST Complexity

With availability of commercial and open-source front ends for C++, pure parsing of the language is not a major issue anymore. Modern front ends disambiguate a lot of information present in the AST; however, one still has to consider a large multitude of possible cases that have to be taken into account in every context. To give an example, the EDG front end for C++ [84] has more than 500 different AST-node kinds. Not all of them have to be taken into account in a given context, but it means that someone who wants to write the simplest analysis of a C++ program will have to be aware of these 500 cases and take them into account at various places of his analyzer. Pivot infrastructure [79] tries to leverage the amount of cases by abstracting away certain differences through generalization, but even Pivot still has 162 different node kinds in its AST.

Another challenge is the level of detail required of the AST for the analysis. Should the AST contain the information from before or after preprocessing? Should the comments be preserved and what entities should they be related to? Should any implicit information be made explicit: e.g. name resolution or implicit casts? Rose [219], for example, keeps information about macro expansion in its AST, while Pivot only deals with programs after preprocessing. The EDG front end allows one to choose what kind of information should be kept in the AST, making their AST essentially parameterized over the set of features needed.

1.2.2 Dealing with AST

Dealing with an AST of a program is prone to the well-known *expression problem* (§2.6), which in an object-oriented language is typically addressed with the *visitor design pattern* [101]. The solution has more disadvantages than advantages, discussed in details in §2.7, which often turn people to functional languages for their analysis application. Table 1.1 lists several static analysis tools that target C or C++ among other languages, and it is easy to see that OCaml was a language of choice in many cases for this kind of application.

Cuoq et al discuss the choice of OCaml in their experience report of implementing Frama-C [71]. The number one reason they list for adoption of OCaml instead of C++ is its expressiveness. Among other reasons, they list modularity, garbage collection and availability of a C parser. Authors also indicate that while it is usually harder to find people who can program in a functional language [183, 190] than in C++, they did not have such a problem as their pool of candidates consisted entirely of PhDs. The two works, to which they refer indicate similar reasons for the choice of OCaml [183] and Haskell [190], even though the applications do not deal with program analysis.

Having an analysis tool written in a different language can sometimes be problematic. Our prior industry experience suggests that developers working predominantly with a single language are reluctant to learn a different language just to help them ensure various properties of their programs. While the language a framework is written in is typically not exposed to its users, the language in which customizations are made is, and the latter is usually either the same as or related to the former.

1.2.3 Analysis Complexity

It is impractical to require a compiler of a general-purpose programming language to perform an elaborate static analysis because:

- static analysis is generally undecidable, making it hard to define what can and cannot be reliably detected
- it would put additional burden on compiler vendors
- it would increase the complexity of a compiler
- it would increase compilation time
- the errors detected might be of general interest only to certain application domains (e.g. multi-threading or numerical computations)

The availability of commercial and open-source static analysis tools solves the problem only partially:

Name	Targets	Written in	Based on	Customizations	References
Astree	C	OCaml	Abstract Interpretation		[24, 25]
BLAST	C	OCaml	Model Checking		[21, 121]
CIL	C	OCaml		Library	[191]
Clang Static Analyzer	C,Objective C	C++	Symbolic execution	GCC attributes	
Coccinelle	C	OCaml,Python	Computational Tree Logic, Model Checking	SmPL	[37, 200]
Coverity	C,C++,C#,Java	C++(EDG)	Abstract Interpretation		[19]
Cppcheck	C,C++	C++	Pattern Matching		
cpplint	C++	Python	Rule based heuristics		
GrammaTech CodeSonar	C,C++		Symbolic execution	Annotations	
Frama-C	C	OCaml		plug-ins	[71]
Parallel Studio XE	C,C++,Fortran				
Klocwork Insight	C,C++,C#,Java			XPath-like	
Lint	C	C		Comments	[140]
LDRA Testbed	C,C++,Java,Ada				
Monoidics INFER	C,C++		Separation Logic		[42]
Parasoft C/C++test	C,C++		Rule based		[150]
PC-Lint/FlexeLint	C,C++			Comments	
Polyspace	C,C++,Ada		Abstract Interpretation		[75]
Goanna	C,C++		Model Checking, Abstract Interpretation		[93]
Saturn	C	OCaml,C++	Boolean satisfiability	Calypso	[276]
SLAM	C	OCaml	Predicate abstraction, Model checking, Symbolic reasoning, Iterative refinement		[14]
Splint	C	C	Heuristics	Comments	[89, 90]

Table 1.1: Static Analysis Tools

- The multitude of language dialects often results in the inability of a tool to handle some of them.
- The tools are often too general in that they are written to work on any program in the language. Developers, on the other hand, are interested in proving the absence of bugs in a given program. The important difference is that it is always possible to write an analyzer that will prove absence of certain kinds of bugs in a given program, while it is impossible to do so for every program in the language [67, §7.1].
- The level of customization that the tool provides is crucial for balance between analysis time and the precision of its results. Learning to use customizations effectively can be time-consuming and is thus subject to time justification.
- The kind of analysis required is often very specific to the project, team or company. The design of such analysis may require expert knowledge of program analysis, language semantics and problem domain.
- The one-time-use nature of such tools often results in their high price.
- Most of the tools we are aware of perform the analysis of laid-out code and fail to take the semantics of user-defined abstractions into account.

1.2.4 Language Complexity

Finally, the design of any static analysis is tightly coupled with language's semantics. There have been several attempts to formalize the semantics of C [26,193,203] as well as of C++ [194,266]. Unfortunately, none of them is universally accepted as they are largely incomplete, lag behind the current standard and are too complex to understand, never mind reason about. To combat the complexity, several authors concentrated on formalizing smaller, orthogonal parts of the language's semantics: e.g. multiple inheritance [215,268], templates [226], object layout [211], construction and destruction [212] etc. Nevertheless, anybody who would like to reason about real-world C++ programs is left to a 400-pages-long standard describing the semantics of the language in English [131]. The document is regularly found to contain internal inconsistencies, leaves a lot of behavior unspecified, compiler- or platform-specific, yet it is the best description of the C++ semantics there is. It does not describe the state of any given C++ implementation, as compiler vendors implement largely incomparable subsets of the language augmented with numerous non-standard extensions. Anyone trying to reason about C++ programs thus has to deal not only with the language, but also with a multitude of its dialects. Both require significant expert knowledge regardless of the simplicity of analysis we would like to perform.

1.2.5 Challenges and Opportunities

We believe that assurance of additional guarantees about a program should be in the direct control of developers creating the program. They are familiar with the functional requirements of the problem domain, the constraints and limitations of the development environment as well as with the non-functional requirements specific to the project (e.g. subset of the language used, coding style etc.) and thus would be able to express them better. Assuring that their program satisfies those requirements would be not less important to them than assuring that their program is free from memory leaks or out-of-bound array accesses. To facilitate that, we believe that the analysis should be relatively easily implementable in the language itself. We cannot require all users of C++ to have a PhD in order for them to be able to design from scratch a static analysis they need; however, we can provide them with the simple building blocks and means of composing them into precise and scalable static analyses.

The research challenges of our work include discovering problem areas that make analysis of C++

programs difficult, consideration and evaluation of numerous alternatives, their integration into the language and interaction with the existing language facilities. The opportunities lay in lowering the threshold for the amount of work required to perform an analysis of a C++ program as well as new interesting use cases that might shape the design of relevant language features in the future.

1.3 Results

The thesis offers several approaches to simplifying the analysis of C++ programs, the core of which concentrates on simplifying dealing with Abstract Syntaxt Trees. Without going into the details of each approach, we summarize capabilities of several existing approaches alongside our approaches in Table 1.2. For this, we listed several desirable properties that applications dealing with abstract syntax trees benefit from and marked whether a given approach fully (+), partially (∗) or doesn't (−) support that property. Empty entries indicate that the property is not directly applicable. The details of each approach are discussed in the listed section.

	Extensibility of Functions	Extensibility of Data	Type Safe	Multiple Inheritance	Relational	Nesting	Retroactive	Local Reasoning	No Control Inversion	Redundancy Checking	Completeness Checking	Speed in Cycles	Discussed In
Discriminated Union	+	−	−	−	−	−	−	+	+	−	∗	39	§2.4.1.3
Tag Switch	+	∗	−	−	−	−	−	+	+	−	−	45	§5.3.1
Type Testing	+	+	+	+	−	−	+	+	+	−	−	577	§5.3.2
Polymorphic Exception	+	+	+	+	−	−	−	+	+	∗	∗	17000	§5.4
Virtual Functions	−	+	+	+	−	−	−	−	−		−	29	§2.3.2
Visitor Design Pattern	+	∗	+	+	∗	−	−	+	−		−	55	§2.7
Open Multi-methods	+	+	+	+	+	−	+	−	−		+	38	§4
Open Type Switch	+	+	+	+	+	−	+	+	+	∗	−	62	§5
Open Pattern Matching	+	+	+	+	+	+	+	+	+	∗	−	70	§6

Table 1.2: Approaches to Dealing with ASTs

It is easy to see that our solutions to dealing with Abstract Syntax Trees (separated at the bottom of the table) support the most properties among other solutions, and collectively cover them all. The approaches are also reasonably efficient in comparison to others.

1.4 Contributions

This dissertation addresses the complexity of analysis of C++ programs with several techniques that, when used together, significantly simplify the task, making it available to non-experts in the language semantics:

- We employ the Internal Program Representation of Pivot to decrease the incidental complexity of C++ abstract syntax trees with generic programming (§3.1.1).

- We implement open multi-methods for C++, which directly addresses the expression problem (§2.6) in the object-oriented world (Chapter 4).
- We develop a type switching solution that is more expressive and efficient than its object-oriented alternative – the visitor design pattern (Chapter 5).
- We develop an open pattern-matching facility for C++ that helps further in decomposing abstract syntax trees with user-defined patterns (Chapter 6).
- We offer a library solution to refinement of the C++ type system with domain-specific typing capabilities (Chapter 7).
- We offer a framework for static analysis of C++ programs based on abstract interpretation that does not require the user to deal with the abstract syntax trees of a program at all (Chapter 8).

Most of the proposed solutions have already been published in peer-reviewed conferences and journals. In particular:

- We successfully experimented with refining type systems with domain-specific abstractions [235, 236].
- We implemented open multi-methods for C++ [208, 209]
- We implemented open type switch for C++ [232]
- We implemented a pattern-matching solution for C++ comparable in expressiveness to functional languages [233, 234].

The following work has also been mostly done, though not published yet:

- We implemented a working prototype of our abstract interpretation framework. It can handle non-relational abstract domains and provides a wide variety of ready-to-use domains for dealing with scalars, arrays, classes, pointers and references etc.

We expect to finish work on the framework and submit it for publication in the nearest future.

1.5 Overview of the Thesis

Each chapter begins with its own introduction in which the subject of the chapter is informally presented and the contributions are summarized. It than typically covers some chapter-specific background information as well as implementation details, followed by evaluation of results and a discussion of advantages and limitations.

In Chapter 2 we introduce a common notation, a few basic definitions, and some terminology and preliminary material that are used later in other chapters.

In Chapter 3 we briefly summarize the work related to different parts of the thesis. This does not include the background information specific to each chapter, which is discussed in the chapter itself.

In Chapter 4 we introduce open multi-methods for C++, which generalize both overload resolution and run-time dispatch. We demonstrate their utility in various application domains, but concentrate on their use in program analysis applications, where they help address the expression problem. We then discuss their efficient implementation for C++, paying special attention to addressing the general multiple inheritance of C++ as well as dynamic linking.

In Chapter 5 we introduce open type switch, which allows for efficient case analysis on hierarchical extensible data types. The construct makes the use of the visitor design pattern in C++, as well as multiple dispatch workarounds based on it, obsolete by providing a better syntax and comparable performance.

Its implementation is again discussed in the context of fully general multiple inheritance of C++ and presence of dynamic linking.

In Chapter 6 we generalize the *Match* statement of the previous chapter to allow for functional-style pattern matching. We introduce the notion of open patterns where all the patterns in the system are user-definable first class citizens. We discuss its expressivity in the context of applications dealing with Abstract Syntax Trees.

Chapter 7 builds a framework for refining the C++ type system with domain-specific subtyping rules and shows how those can be used to ensure properties of interest about a program within a library solution. We demonstrate the approach with modeling two distinct domain-specific type systems in C++.

Chapter 8 builds on all the facilities presented thus far to create a simple static analysis framework that does not require dealing with the abstract syntax tree of a program at all. The framework is a work in progress that establishes its feasibility, usefulness and ease of use. The chapter describes its current state and future directions.

Chapter 9 concludes by summarizing the work presented in the thesis and provides some insights for possible future directions.

2. PRELIMINARIES[1]

> Do not laugh at notations; invent them, they are powerful.
>
> Richard P. Feynman

In this chapter we introduce some relevant notions used in the rest of the dissertation, discuss differences in the programming paradigms that give rise to them as well as explain some techniques used later. The definitions specific to C++ are taken verbatim from the standard [131]. We provide some comments to make them more available to researchers in other programming languages, but otherwise assume some familiarity with the language.

2.1 Mathematical Notation

We use capital letters to denote sets and lowercase letters to denote their elements. Sets $\mathbb{B}$, $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ represent the set of booleans, natural numbers, integers and reals, respectively.

A *unit set* is a set with exactly one element in it. We will use the notation $\{s\} = S$ to indicate (or check) that set S is a unit set, whose only element is s. A *power set* of a set S is the set $\wp(S)$ of all subsets of S, including the *empty set* $\varnothing$ and S itself.

A *Cartesian product* of sets X and Y is a set: $X \times Y = \{\langle x, y\rangle \mid x \in X \wedge y \in Y\}$ Given an element $\overline{x}$ of an n-argument Cartesian product $\overline{x} = \langle x_1, \cdots, x_n\rangle \in X_1 \times \cdots \times X_n$ we use the notation x_i to refer to its i^{th} component, while we put a line over a variable (e.g. $\overline{x}$) to indicate that its value belongs to a Cartesian product of sets.

R is a *relation between* X and Y when it is a subset of their Cartesian product: $R \subseteq X \times Y$; it is a *relation over* X, when $R \subseteq X \times X$. For binary relations, we will write xRy as a short for $\langle x, y\rangle \in R$ and say that x is in relation R with y.

Some binary relations may have the following properties:

- *Reflexivity*: $\forall x \in X, xRx$
- *Irreflexivity*: $\forall x \in X, \neg xRx$
- *Symmetry*: $\forall x, y \in X, xRy \implies yRx$
- *Antisymmetry*: $\forall x, y \in X, x \neq y \wedge xRy \implies \neg yRx$
- *Asymmetry*: $\forall x, y \in X, xRy \implies \neg yRx$
- *Transitivity*: $\forall x, y, z \in X, xRy \wedge yRz \implies xRz$

Relations that satisfy corresponding properties are called *reflexive*, *irreflexive*, *symmetric*, *antisymmetric*, *asymmetric*, and *transitive* relations, respectively. We will also use several *operators on binary relations.* If R is a binary relation over X, then each of the following is a binary relation over X:

- *Reflexive Closure*: $R^{=} = \{\langle x, x\rangle \mid x \in X\} \cup R$, where $R \subseteq X \times X$.
- *Reflexive Reduction*: $R^{\neq} = R \setminus \{\langle x, x\rangle \mid x \in X\}$, where $R \subseteq X \times X$.
- *Transitive Closure*: R^{+} is the smallest (with respect to $\subseteq$) transitive relation over X containing R.
- *Transitive Reduction*:R^{-} is a minimal (with respect to $\subseteq$) relation having the same transitive closure as R.

[1] Part of this chapter is reprinted with permission from "Open and Efficient Type Switch for C++" by Yuriy Solodkyy, Gabriel Dos Reis, Bjarne Stroustrup, 2012. Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications, Pages 963-982, Copyright 2012 by ACM.

- *Reflexive Transitive Closure*: $R^* = (R^+)^=$

A reflexive and transitive relation is called a *preorder*. A preorder that is antisymmetric is called a *partial order*. A set P with a partial order $\sqsubseteq$ on it is called a *partially ordered set* or a *poset*, written as $\langle P, \sqsubseteq \rangle$. For elements $a, b \in P$, $a = b \equiv a \sqsubseteq b \wedge b \sqsubseteq a$, while $a \sqsubset b \equiv a \sqsubseteq b \wedge a \neq b$ ($\sqsubset \equiv \sqsubseteq^{\neq}$ is called *strict order*). When $a \sqsubseteq b$ or $b \sqsubseteq a$, then a and b are said to be *comparable*, otherwise they are *incomparable*. A partial order in which every pair of elements is comparable is called a *total order*.

Given a poset $\langle P, \sqsubseteq \rangle$, and a subset $S \subseteq P$, an element $u \in P$ is called an *upper bound* for S if $\forall s \in S . s \sqsubseteq u$. An element $u \in P$ is the *least upper bound* or *supremum* for S (written as $\sqcup S$ or $\sup S$) if it is an upper bound and any other upper bound $u' \in P$ is greater than it: $u \sqsubseteq u'$. An element $s \in S$ is the *greatest element* (the *largest element*) of S if s is an upper bound for S. An element $m \in S$ is said to be the *maximal element* of S if $\nexists\ s \in S, m \sqsubseteq s$. A given subset $S \subseteq P$ may not have maximal elements at all or have several incomparable maximal elements, in which case it will have no greatest element. When it does have a greatest element, the greatest element is unique and is the only maximal element of the subset S, typically referred to as *maximum*. Maximum is effectively the supremum of S that is contained in the set S. When the largest element of the poset P itself exists, it is called *top* and is written $\top$ ($\top \equiv \sqcup P \in P$). The dual notions of *lower bound*, *greatest lower bound* or *infimum* ($\sqcap S$ or $\inf S$), *least element* (the *smallest element*), *minimal element*, *minimum* and the *bottom* ($\bot \equiv \sqcap \varnothing \in P$) can be defined similarly.

Given two partially ordered sets, one can define the following partial orders on the Cartesian product of these sets:

- *Lexicographical order* $\langle a, b \rangle \sqsubseteq \langle c, d \rangle \iff a \sqsubset c \vee (a = c \wedge b \sqsubseteq d)$.
- *Product order* $\langle a, b \rangle \sqsubseteq \langle c, d \rangle \iff a \sqsubseteq c \wedge b \sqsubseteq d$.
- *Reflexive closure of the direct product of strict orders* $\langle a, b \rangle \sqsubseteq \langle c, d \rangle \iff (a \sqsubset c \wedge b \sqsubset d) \vee (a = c \wedge b = d)$.

The orders are listed by increasing strength – the set of pairs of each subsequent order is included in the set of pairs of the preceding one. All three orders can be similarly extended for the Cartesian product of n sets.

2.2 Substitutability

Data abstraction is invaluable in software design in that it allows the separation of concerns of the developer and user of a data-structure. It enables a multitude of possible implementations to interact with a multitude of possible uses. As such, it relies on some form of substitutability being present in a language, which allows the use of a value of a type implementing the requirements in place of a formal parameter of a type stating them. A particular form of such substitutability is given through the following *substitution property*:

> If for each object s of type S there is an object t of type T such that for all programs $\mathbb{P}$ defined in terms of T, the behavior of $\mathbb{P}$ is unchanged when s is substituted for t, then S is a *subtype* of T (written as $S \mathrel{<:} T$).

This particular form is known as (strong) *behavioral* subtyping and was first introduced by Barbara Liskov [155]. Different programming languages adopted different weaker forms of *subtyping*, making sure they nevertheless adhere to the following *Liskov substitution principle*:

> In a given program $\mathbb{P}$, if type S is a subtype of type T, then objects of type S may be substituted for objects of type T without altering any of the desirable properties of that program (correctness, algorithmic complexity, etc.).

Most of the subtyping relations seen in programming languages form a preorder on types. A subtyping scheme is called *nominative* when only types declared in a certain manner (usually by explicitly stating their relation to each other) can form a subtyping relation. A subtyping scheme is called *structural* when the structure of two types determines whether they are in a subtyping relation. Single public inheritance in C++ is an example of a nominative subtyping scheme, while function template argument deduction is an example of a structural subtyping scheme. Existing implementations of subtyping are either *inclusive*, in which a value of a subtype type is also a valid value of a supertype type, or *coercive*, in which a value of a subtype type can be implicitly converted into a value of a supertype type.

2.3 Classes, Objects and Subobjects

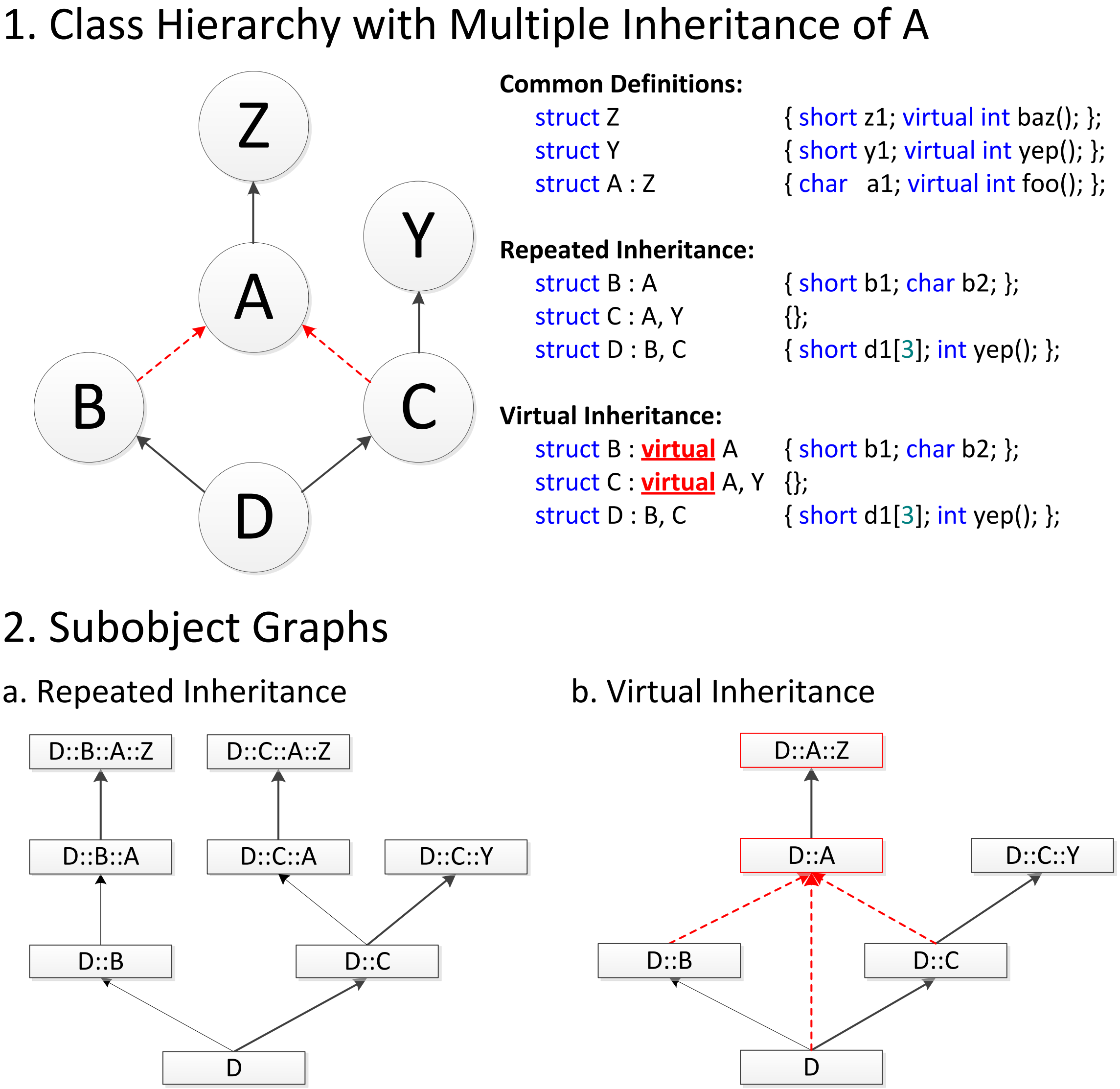
We assume a program $\mathbb{P}$ is represented by its symbol table, which can be queried for kinds of various entities in the program (e.g. variables, functions, types, templates, etc.) and information specific to the entities of a given kind (e.g. inheritance relations between classes, a specific entity that corresponds to a given use of an overloaded name, etc.). All subsequent definitions are thus implicitly parameterized over a given program $\mathbb{P}$.

The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An *object* is a region of storage that is created by a definition, by a new-expression or by the implementation when needed [131, §1.8]. An object has a *type*. The terms *object type*, *most-derived type* and the *dynamic type* of an object all refer to the type with which the object was created. The *static type* of an expression is the type of that expression as given by the static semantics, i.e. the typing rules of expressions. Object types have alignment requirements, which place restrictions on the addresses at which an object of that type may be allocated. An *alignment* is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated.

Informally, a *class* is a user-defined data type that describes the structure and the behavior of a set of objects. A definition of a class may involve many entities like base classes, data members, member functions, access permissions to those, friends etc. In this thesis, we will primarily be dealing with relations between objects and their classes, and will thus focus on those.

A class definition may include a list of base classes, each of which is called a *direct base class* for the class being defined. A class B is a *base class* of a class D if it is a direct base class of D or a direct base class of one of D's base classes. A class is an *indirect base class* of another if it is a base class but not a direct base class. A class is said to be (directly or indirectly) *derived* from its (direct or indirect) base classes. Unless redeclared in the derived class, members of a base class are also considered members of the derived class. The base class members in this case are said to be *inherited* by the derived class, while the relation between the derived and the base class is called *inheritance*.

C++ supports two kinds of inheritance: *non-virtual* [86] (also known as *replicated* [215] or *repeated* [268]) inheritance and *virtual* [86] (or *shared* [268]) inheritance. The difference between the two only arises in situations where a class indirectly inherits from the same base class via more than one path in the hierarchy. A class B is a *direct repeated base class* of D if B is mentioned in the list of base classes of D without the **virtual** keyword ($D \prec_R B$). Similarly, a class B is a *direct shared base class* of D if B is mentioned in the list of base classes of D with the **virtual** keyword ($D \prec_S B$). The union $\prec_R \cup \prec_S$ of these relations is the *direct base class* relation defined above, while their reflexive transitive closure $\texttt{<:} \equiv \preceq^* \equiv (\prec_R \cup \prec_S)^*$ defines the nominative *subtyping* relation on the set $\mathbb{C}$ of all classes of program $\mathbb{P}$. It is easy to see that $\texttt{<:}$ is antisymmetric, which together with reflexivity and transitivity makes it a partial order on $\mathbb{C}$. In general, a *class hierarchy* is a partially ordered set $(H, \texttt{<:})$ where H is a potentially open



1. Class Hierarchy with Multiple Inheritance of A

Common Definitions:

```
struct Z          { short z1; virtual int baz(); };
struct Y          { short y1; virtual int yep(); };
struct A : Z      { char  a1; virtual int foo(); };
```

Repeated Inheritance:

```
struct B : A      { short b1; char b2; };
struct C : A, Y   {};
struct D : B, C   { short d1[3]; int yep(); };
```

Virtual Inheritance:

```
struct B : virtual A     { short b1; char b2; };
struct C : virtual A, Y  {};
struct D : B, C          { short d1[3]; int yep(); };
```

2. Subobject Graphs

a. Repeated Inheritance

b. Virtual Inheritance

Figure 2.1: Multiple Inheritance in C++

set of classes and <: is a reflexive, transitive and antisymmetric *subclassing* relation on H.

2.3.1 Subobjects

Objects can contain other objects, called subobjects. A *subobject* can be a *member subobject*, a *base class subobject*, or an *array element*. An object that is not a subobject of any other object is called a *complete object*.

Consider the simple class hierarchy in Figure 2.1(1). Class D indirectly inherits from class A through its B and C base classes. In this case, the user may opt to keep distinct subobjects of class A (repeated inheritance) or a shared one (virtual inheritance) by specifying how B and C inherit from A. The kind of inheritance is thus not a property of a given class, but a property of an inheritance relation between classes and it is possible to mix the two. C++'s notion of multiple inheritance is fundamentally about subobjects, not just types. Virtual inheritance is about sharing the base-class subobjects, whereas non-virtual inheritance reflects distinction in base-class subobjects from distinct class inheritance paths [86].

A class hierarchy gives rise to a *subobject graph*, where a given class node may be replicated when inherited repeatedly or left shared when inherited virtually. The edges in such a graph represent *subobject containment* and indicate whether such containment is shared or exclusive. Every class C in the class hierarchy will have its own subobject graph representing the subobjects of an object of dynamic type C. Figure 2.1(2) shows subobject graphs for class D obtained for the class hierarchy in Figure 2.1(1) under repeated (a) and virtual (b) inheritance of class A by classes B and C. Shared containment is indicated with dashed arrows, exclusive containment – with solid ones.

We will use the term *object descriptor* to mean either pointer or reference to an object, which we will use interchangeably when not explicitly specified. An object descriptor of static type A referencing an object of dynamic type C can be understood as any C::$*$::A-node in the subobject graph of C. Rosie and Friedman [215] call A an *effective type* of object, while the node in the subobject graph representing it – its *effective subobject*.

A *cast* in such a model can be understood as a change from one effective subobject to another. We use the terms *source subobject* and *target subobject* to refer to the argument and result of the cast, respectively (effective subobjects before and after the cast). Their static types will be referred to as the *source type* and the *target type* of the cast, respectively. C++ distinguishes three kinds of casts: upcasts, downcasts, and crosscasts.

An *upcast* is a cast from a derived class to one of its bases. When the base class is *unambiguous* (unique according to the C++ name lookup rules [131, §3.4]), such casts are implicit and require no additional annotations. When the base class is *ambiguous*, cast failure is manifested statically in the form of a compile-time error. For example, this is the case with casting D to A under repeated multiple inheritance of A, in which case the user needs to explicitly cast the object to B or C first in order to indicate the desired subobject and resolve the ambiguity. In some cases, however, introduction of such an explicit cast is not possible, e.g. in implicit conversions generated by the compiler to implement covariant return types (§4.3.4), cross casts or conversions in generic code. This does not mean that in such cases we violate the Liskov substitution principle: the classes are still in a subtyping relation, but an implicit conversion is not available.

A *downcast* is a cast from a base class to one of its derived classes. The cast has to determine at run-time whether the source subobject is contained by a subobject of the target type in the dynamic type's subobject graph. Failure of such a cast is manifested dynamically at run-time.

A *crosscast* is a cast between classes that are not necessarily related by inheritance except by sharing a common derived class (subclass). According to the C++ semantics, such a cast is defined to be a composition of upcast to the target type and downcast to the dynamic type. While the downcast to the dynamic type is always guaranteed to succeed regardless of the source subobject, the upcast to the target type may be ambiguous, in which case the cast will fail at runtime. A cast from Y to B inside an object of dynamic type D in Figure 2.1(2a,2b) is an example of a successful crosscast. A similar cast from Y to A inside D under the repeated inheritance in Figure 2.1(2a) will fail because of the ambiguous upcast from D to A.

An interesting artifact of these distinctions can be seen in an example of casting a subobject of type Z to a subobject of type A in Figure 2.1(2a). The subobject D::B::A::Z will be successfully cast to D::B::A, while the D::C::A::Z will be successfully cast to D::C::A. These casts do not involve downcasting to D followed by an upcast to A, which would be ambiguous, but instead take the dynamic type of a larger subobject (D::B or D::C) that the source subobject is contained in into account in order to resolve the ambiguity. A similar cast from Y to A will fail; should Y have also been non-virtually derived from Z, the cast from D::C::Y::Z to A would have failed. This shows that the distinction between crosscast and downcast is not based solely on the presence of a subtyping relation between the source and target types, but also on the actual position of the source subobject in the dynamic type's subobject graph.

The notion of subobject has been formalized before [211, 215, 268]. We follow here the presentation of Ramamanandro et al [211]. A *base class subobject* of a given *complete object* is represented by a pair $\sigma = \langle h, l \rangle$ with $h \in \{\mathsf{Repeated}, \mathsf{Shared}\}$ representing the kind of inheritance (single inheritance is Repeated with one base class) and l representing the path in a non-virtual inheritance graph. A judgment of the form $\mathbb{P} \vdash C \prec \sigma \succ A$ states that in a program $\mathbb{P}$, σ designates a subobject of static type A within an object of type C. Omitting the context $\mathbb{P}$:

ϵ indicates an empty path, but we will generally omit it in writing when it can be understood from the context. In the case of repeated inheritance in Figure 2.1(1), an object of dynamic type D will have the following Repeated subobjects: D::C::Y, D::B::A::Z, D::C::A::Z, D::B::A, D::C::A, D::B, D::C, D. Similarly, in the case of virtual inheritance in the same example, an object of dynamic type D will have the following Repeated subobjects: D::C::Y, D::B, D::C, D as well as the following Shared subobjects: D::A::Z, D::Z, D::A. See Figure 2.1 for illustration.

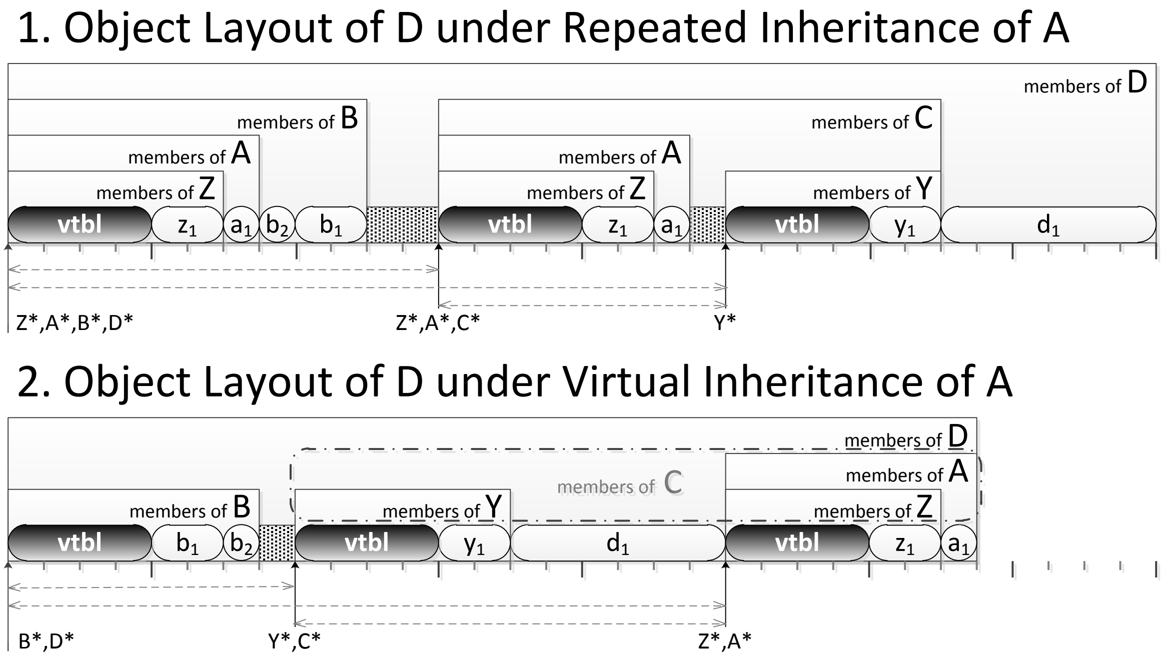
It is easy to show by structural induction on the above definition that $C \prec \sigma \succ A \implies \sigma = \langle h, C :: l_1 \rangle \wedge \sigma = \langle h, l_2 :: A :: \epsilon \rangle$, which simply means that any path to a subobject of static type A within an object of dynamic type C starts with C and ends with A. This observation shows that $\sigma_\perp = \langle \mathsf{Shared}, \epsilon \rangle$ does not represent a valid subobject. We call it an *invalid subobject* in order not to be confused with an *empty subobject*, which refers to a subobject without any data members. If $\Sigma_\mathbb{P}$ is the domain of all subobjects in the program $\mathbb{P}$ extended with $\sigma_\perp$, then a *cast* operation can be understood as a function $\delta : \Sigma_\mathbb{P} \to \Sigma_\mathbb{P}$. We use $\sigma_\perp$ to indicate an impossibility of a cast. The fact that δ is defined on subobjects as opposed to actual run-time values reflects the non-coercive nature of the operation, i.e. the underlying value remains the same. Any implementation of such a function must thus satisfy the following condition:

$$C \prec \sigma_1 \succ A \wedge \delta(\sigma_1) = \sigma_2 \implies C \prec \sigma_2 \succ B$$

i.e. the dynamic type of the value does not change during casting, only the way we reference it does. A is the *source type* and σ_1 is the *source subobject* of the cast, while B is the *target type* and σ_2 is the *target subobject* of it. The type C is the *dynamic type* of the value being casted. The C++ semantics states more requirements to the implementation of δ: e.g. $\delta(\sigma_\perp) = \sigma_\perp$ etc. but their precise modeling is out of scope of this discussion. We would only like to point out here that since the result of the cast does not depend on the actual value and only on the source subobject and the target type, we can memoize the outcome of a cast on one instance in order to apply its results to another.

2.3.2 Virtual Table Pointers

Some objects are *polymorphic*; for such objects, the implementation generates information that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the expressions used to access them. In particular, a class that declares or inherits a *virtual function* is called a *polymorphic class*, while objects belonging to it are called *polymorphic objects*.


1. Object Layout of D under Repeated Inheritance of A

2. Object Layout of D under Virtual Inheritance of A

Figure 2.2: Object Layout under Multiple Inheritance

A call to a *virtual function* is bound dynamically at run-time and is based on the object's most-derived type. The C++ standard [131] does not prescribe any specific implementation technique for *virtual function dispatch*. However, in practice, all C++ compilers use a strategy based on so-called *virtual function table*s (or *vtables* for short) for efficient dispatch. The *vtable* is part of the reification of a polymorphic class type. C++ compilers embed a pointer to a vtable (*vtbl-pointer* for short) in every object of polymorphic class type.

Figure 2.2 shows a typical object layout generated by a C++ compiler for class D from Figure 2.1(1) under repeated (1) and virtual (2) inheritance of A. The layouts represent an encoding of the corresponding subobject graphs from Figures 2.1(2a) and 2.1(2b) respectively.

Due to the extensibility of classes, the layout decisions for classes must be made independently of their derived classes – a property of the C++ object model that we will refer to as *layout independence*. In turn, the layout of derived classes must conform to the layout of their base classes relative to the offset of the base class within the derived one. For example, the layout of A in C is the same as the layout of A in B and is simply the layout of A. The layout of each subobject must additionally satisfy the alignment requirements of its object type, which is why the compiler might introduce *padding* between the members. Virtual base classes do not contribute to the fixed layout because they are looked up indirectly at run-time; however, they are not exempt from layout independence, since their lookup rules are agnostic of the concrete dynamic type.

The layout independence requirement results in the subobject graph of a derived class necessarily containing the subobject graph of its base class, i.e. the subobject graph of a base class is always homomorphic to the subobject graph of its any derived class. We say that the derived class subobject *directly extends* subobjects of all its direct base classes; it also *extends* subobjects of all its indirect base classes. We will indicate the *direct extension* relation with $<_1$, and its reflexive transitive closure

– the *extends* relation – with <. The *extends* relation mimics the subclassing relation on the level of subobjects.

Under non-virtual inheritance, members of the base class are typically laid out before the members of derived class, resulting in the base class being at the same offset as the derived class itself. In our example, the offset of A in B under regular (non-virtual) inheritance of A is 0. Under multiple inheritance, different base classes might be at different offsets in the derived class, which is why pointers of a given static type may be pointing only to certain subobjects in it. These positions are marked in the picture with vertical arrows decorated with the set of pointer types whose values may point into that position. Run-time conversions between such pointers represent casts between subobjects of the same dynamic type and may require adjustments to this-pointer (shown with dashed arrows) for type safety.

The vtbl-pointer that compiler embeds into every object of a polymorphic class type simply becomes an additional data member of that class (also subject to layout independence). CFront, the first C++ compiler, puts the vtbl-pointer at the end of an object. The so-called "common vendor C++ *ABI* (Application Binary Interface)" [55] requires the vtbl-pointer to be at offset 0 of an object.[1] We do not have access to the unpublished Microsoft ABI, but we have experimental evidence that in most cases, though not all, their C++ compiler also puts the vtbl-pointer at the start of an object. The exact offset of the vtbl-pointer within the (sub)object is not important: due to layout independence, every (sub)object of a polymorphic type S will have a vtbl-pointer at a predefined offset. That offset may be different for different static types S, in which case the compiler will know at which offset in type S the vtbl-pointer is located, but it will be the same within any subobject of a static type S. For a library implementation we assume the presence of a function **template** ⟨**typename** S⟩intptr_t vtbl(**const** S*s); that returns the address of the virtual table corresponding to the subobject pointed to by s. Such a function can be trivially implemented for the common vendor C++ ABI, where the vtbl-pointer is always at offset 0:

```
template ⟨typename S⟩ std::intptr_t vtbl(const S* s) {
    static_assert(std::is_polymorphic⟨S⟩::value, "error");
    return *reinterpret_cast⟨const std::intptr_t*⟩(s);
}
```

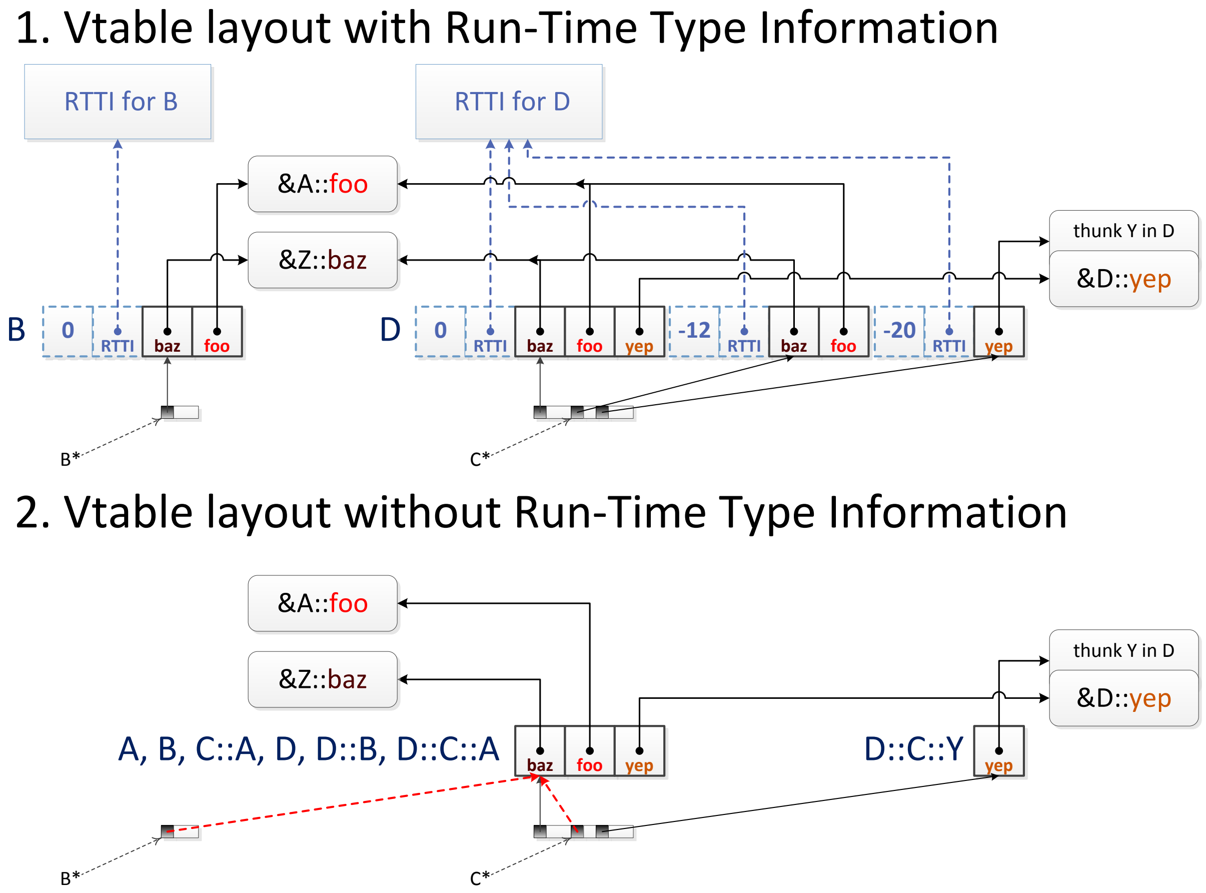
It can also be trivially overridden for any class that does not have its vtbl-pointer at offset 0, as demonstrated by the following example that will reproduce this scenario in Microsoft Visual C++:

```
struct Polymorph { virtual ~Polymorph(); };
struct NonPolymorph { int data; };
struct Mixed : NonPolymorph, Polymorph { };
inline std::intptr_t vtbl(const Mixed* p) {
    return vtbl(static_cast⟨const Polymorph*⟩(p));
}
```

Each of the vtbl data members shown in Figure 2.2 holds a vtbl-pointer referencing a group of virtual methods known in the object's static type. Figure 2.3(1) shows a typical layout of virtual function tables together with objects it points to for classes B and D.

Entries in the vtable to the right of the address pointed to by a vtbl-pointer represent pointers to functions, while entries to the left of it represent various additional fields like a pointer to a class' type information, offset to top, offsets to virtual base classes, etc. In many implementations, *this-pointer adjustments* required to dispatch the call properly were stored in the vtable along with function pointers.

[1] The following compilers are known to comply with the C++ ABI: GCC (3.x and up); Clang and LLVM-G++; Linux versions of Intel and HP compilers, and compilers from ARM. See http://morpher.com/documentation/articles/abi/ for details.


1. Vtable layout with Run-Time Type Information

2. Vtable layout without Run-Time Type Information

Figure 2.3: VTable layout with and without RTTI

Today most implementations prefer to use *thunk*s or *trampolines* – additional entry points to a function that adjust the this-pointer before transferring control to the function, which was shown to be more efficient [82]. Thunks in general may only be needed when a virtual function is overridden. In such cases, the overridden function may be called via a pointer to a base class or a pointer to a derived class, which may not be at the same offset in the actual object.

Generation of the *Run-Time Type Information* (or *RTTI* for short) can typically be disabled with a compiler switch and the Figure 2.3(2) shows the same vtable layouts once RTTI has been disabled. Since neither baz nor foo were overridden, the prefix of the vtable for the C subobject in D is the same as the vtable for its B subobject, the A subobject of C, or the entire vtable of A and B classes. Such a layout, for example, is produced by Microsoft Visual C++ 11 when the command-line option /GR– is specified. The Visual C++ compiler has been known to unify code identical on the binary level, which in some cases may result in sharing of the same vtable between unrelated classes (e.g. when virtual functions are empty).

2.4 Algebraic Data Types

In languages like ML and Haskell, an *Algebraic Data Type* (*ADT*) is a data type each of whose values is picked from a disjoint sum of (possibly recursive) data types, called *variants*. Each of the variants is marked with a unique symbolic constant called a *constructor*, while the set of all constructors of a given type is called a *signature*. Constructors provide a convenient way of creating a value of their variant type as well as a way of discriminating the variant type from an algebraic data type through pattern matching (§2.5).

Algebraic data types are particularly well-suited for representation of abstract syntax trees. Consider for example a simple *language of expressions*:

$$exp ::= val \mid exp + exp \mid exp - exp \mid exp * exp \mid exp/exp$$

An OCaml data type describing a term in this expression language can be declared as following:

```
type expr = Value  of int
          | Plus   of expr * expr
          | Minus  of expr * expr
          | Times  of expr * expr
          | Divide of expr * expr
          ;;
```

The set of values described by such an algebraic data type is defined inductively as the least set closed under the constructor functions of its variants. Algebraic data types draw their name from the practice of using case distinction in mathematical function definitions and proofs that involve *algebraic terms*.

Algebraic data types can be parameterized, as demonstrated by the following Haskell code that defines a binary tree parameterized on key type k and data type d stored in the nodes:

```
data Tree k d = Node k d (Tree k d) (Tree k d) | Leaf
```

2.4.1 Algebraic Data Types in C++

C++ does not have direct support of algebraic data types, but they can usually be emulated in a number of ways. Consider an ML data type of the form:

```
datatype DT = $C_1$ of {$L_{11} : T_{11}, ..., L_{1m} : T_{1m}$}
            | ...
            | $C_k$ of {$L_{k1} : T_{k1}, ..., L_{kn} : T_{kn}$}
```

There are at least 3 different ways to represent it in C++. Following Emir, we will refer to them as *encodings* [87]:

- Polymorphic Base Class (or *polymorphic encoding* for short)
- Tagged Class (or *tagged encoding* for short)
- Discriminated Union (or *union encoding* for short)

2.4.1.1 Polymorphic Base Class Encoding

In this encoding the user declares a polymorphic base class DT that will be extended by classes representing the variants. The base class might declare several virtual functions that will be overridden by its derived classes, for example accept as used by the Visitor Design Pattern (§2.7).

```
class DT { virtual ~DT{} };
class $C_1$ : public DT {$T_{11} L_{11}; ...T_{1m} L_{1m};$}
...
class $C_k$ : public DT {$T_{k1} L_{k1}; ...T_{kn} L_{kn};$}
```

To uncover the actual variant of such an algebraic data type, a user might use **dynamic_cast** to query one of the k expected run-time types (an approach used by Rose [219]) or she might employ a visitor design pattern devised for this algebraic data type (an approach used by Pivot [81] and Phoenix [172]). The most attractive feature of this approach is that it is truly open as we can extend classes arbitrarily at will (leaving the orthogonal issues of visitors aside).

For example, the algebraic data type from §2.4 describing a term in the expression language will be encoded as:

```
struct Expr { virtual ~Expr() {} };
struct Value  : Expr { int value; };
struct Plus   : Expr { Expr* e1; Expr* e2; }; ...
```

2.4.1.2 Tagged Class Encoding

This encoding is similar to the *polymorphic encoding* in that we use derived classes to encode the variants. The main difference is that the user designates a member in the base class, whose value will uniquely determine the most derived class a given object is an instance of. Constructors of each variant C_i are responsible for properly initializing the dedicated member with a unique value c_i associated with that variant. Clang [51] among others uses this approach.

```
class DT { enum kinds {$c_1, ..., c_k$} m_kind; };
class $C_1$ : public DT {$T_{11}L_{11}; ...T_{1m}L_{1m};$}
...
class $C_k$ : public DT {$T_{k1}L_{k1}; ...T_{kn}L_{kn};$}
```

In such a scenario, the user might use a simple **switch** statement to uncover the type of the variant combined with a **static_cast** to cast properly the pointer or reference to an object. People might prefer this encoding to the one above for performance reasons as it is possible to avoid virtual dispatch altogether. Note, however, that once we allow for extensions and do not limit ourselves to encoding algebraic data types only, it also has a significant drawback in comparison to the previous approach: we can easily check that a given object belongs to the most derived class, but we cannot say much about whether it belongs to one of its base classes. A visitor design pattern can be implemented to take care of this problem, but the control inversion that comes along with it will certainly diminish the convenience of having just a switch statement. Besides, the forwarding overhead might lose some of the performance benefits originally gained by putting a dedicated member into the base class.

The algebraic data type from §2.4 will be encoded as the following:

```
struct Expr { enum Tag {V,P,M,T,D} tag; Expr(Tag t) : tag(t) {} };
struct Value : Expr { int value; Value() : Expr(V), value() {} }; ...
```

2.4.1.3 Discriminated Union Encoding

This encoding is popular in projects that are either implemented in C or originated from C before coming to C++. It involves a type that contains a union of its possible variants, discriminated with a dedicated value stored as a part of the structure. The approach is used by the EDG [84] and many other front ends.

```
struct DT
{
    enum kinds {$c_1, ..., c_k$} m_kind;
    union {
        struct $C_1$ {$T_{11}L_{11}; ...T_{1m}L_{1m};$} $C_1$;
        ...
        struct $C_k$ {$T_{k1}L_{k1}; ...T_{kn}L_{kn};$} $C_k$;
    };
};
```

As before, the user can use a switch statement to identify the variant c_i and then access its members via the C_i union member. This approach is truly closed, as we cannot add new variants to the underlying union without modifying the class definition.

The encoding of the algebraic data type from §2.4 is now:

```
struct Expr
{
    enum Tag {V,P,M,T,D} tag;
    union {
        struct { int value; };              // V
        struct { Expr* e1; Expr* e2; };  // P,M,T,D
    };
};
```

2.4.2 Hierarchical Extensible Data Types

Classes differ from algebraic data types in two important ways. Firstly, they are *extensible*, for new variants can be added later by inheriting from the base class. Secondly, they are *hierarchical* and thus typically *non-disjoint* since variants can be inherited from other variants and form a subtyping relation between themselves [109]. In contrast, variants in conventional algebraic data types are *disjoint* and *closed*. Closedness means that once we have listed all the variants a given algebraic data type may have, we cannot extend it with new variants without modifying its definition. Disjointedness means that a value of an algebraic data type belongs to exactly one of its variants.

Some functional languages, e.g. ML2000 [9] and its predecessor, Moby, were experimenting with *hierarchical extensible sum types*, which are closer to object-oriented classes then algebraic data types are, but, interestingly, they provided neither traditional nor efficient facilities for performing case analysis on them.

2.5 Pattern Matching

Pattern matching has been closely related to *algebraic data types* and *equational reasoning* since the early days of functional programming. A simple evaluator of expressions in the *exp* language from §2.4 can be implemented as the following OCaml code:

```
let rec eval e =
  match e with
            Value  v       → v
          | Plus   (a, b) → (eval a) + (eval b)
          | Minus  (a, b) → (eval a) − (eval b)
          | Times  (a, b) → (eval a) * (eval b)
          | Divide (a, b) → (eval a) / (eval b)
          ;;
```

The solution is intuitive to the extent that even a person who never used pattern matching before can quickly figure out what the code does. It is also elegant, as it closely resembles case analysis in structural induction, as often used in math. The fact that cases are listed and analyzed in a single place in a program, commonly referred to as *local reasoning*, greatly contributes to both the elegance and the intuitiveness of the solution.

Closedness of algebraic data types is particularly useful for reasoning about programs by case analysis and allows the compiler to perform an automatic *exhaustiveness* check – a test of whether a given *match statement* covers all possible cases. Similar reasoning about programs involving extensible data types is more involved as we are dealing with a potentially open set of variants. *Completeness* checking in such a scenario reduces to checking for the presence of a case that handles the static type of the subject. Absence of such a case, however, does not necessarily imply incompleteness, only potential incompleteness, as the answer will depend on the actual set of variants available at run-time.

A related notion of *redundancy* checking arises from the tradition of using the *first-fit* strategy in pattern matching. It warns the user of any *case clause* inside a match statement that will never be entered because of a preceding one being more general. Object-oriented languages, especially C++, typically prefer the *best-fit* strategy (e.g. for overload resolution (§4.3.2) and class template specialization (§2.8)) because it is not prone to errors where the semantics of a statement might change depending on the ordering of preceding definitions. The notable exception in C++ semantics that prefers the *first-fit* strategy is the ordering of the **catch** handlers of a **try**-block. Similar to functional languages, the C++ compiler will perform *redundancy* checking on catch handlers and issue a warning that lists the redundant cases.

The patterns that work with algebraic data types we have seen so far are generally called *tree patterns* or *constructor patterns*. Their analog in object-oriented languages is often referred to as a *type pattern* since it may involve type testing and type casting. Special cases of these patterns are *list patterns* and *tuple patterns*. The former lets one split a list into a sequence of elements in its beginning and a tail with the help of the *list* constructor : and the *empty list* constructor [] e.g. [x:y:rest]. The latter does the same with tuples using the *tuple* constructor (,,. . .,) e.g. ([x:xs],'b',(1,2.0),"hi",True).

Pattern matching is not used solely with algebraic data types and can be applied equally well to built-in types. The following Haskell code defines the factorial function in the form of equations:

```
factorial 0 = 1
factorial n = n *factorial (n−1)
```

Here 0 in the left hand side of the first *equation* is an example of a *value pattern* (also known as a *constant pattern*) that will only match when the actual argument passed to the function factorial is 0. The *variable pattern* n (also referred to as an *identifier pattern*) in the left hand side of the second equation will match any value, *binding* the variable n to that value in the right hand side of the equation. Similar to the variable pattern, the *wildcard pattern* _ will match any value, neither binding it to a variable nor even obtaining it. Value patterns, variable patterns and wildcard patterns are generally called *primitive patterns*. Patterns like variable and wildcard patterns that never fail to match are called *irrefutable*, in contrast to *refutable* patterns like value patterns, which may fail to match.

In Haskell 98 [141] the above definition of factorial could also be written as:

```
factorial 0 = 1
factorial (n+1) = (n+1) *factorial n
```

The (n+1) pattern in the left-hand side of the equation defining the factorial is an example of an *n+k pattern* and is an equation in itself: $n + 1 = v$. According to its informal semantics, "Matching an $n + k$ pattern (where n is a variable and k is a positive integer literal) against a value v succeeds if $v \geq k$, resulting in the binding of n to $v - k$, and fails otherwise" [205]. n+k patterns were introduced into Haskell to let users express inductive functions on natural numbers in much the same way as functions defined through case analysis on algebraic data types. Besides succinct notation, such a language feature could facilitate automatic proof of termination of such functions by a compiler. Unfortunately, Peano numbers, used as an analogy to algebraic data type representation of natural numbers, are not always the best abstraction for representing other mathematical functions. This, together with the numerous ways of defining the semantics of generalized n+k patterns, were some of the reasons why the feature was never generalized in Haskell to other kinds of expressions, even though there were plenty of known applications. Moreover, numerous debates over the semantics and usefulness of the feature resulted in n+k patterns being removed from the language altogether in the Haskell 2010 standard [85]. The generalization of n+k patterns, called *application patterns*, has been studied by Nikolaas N. Oosterhof in his Master's thesis [199]. Application patterns essentially treat n+k patterns as equations, while matching against them attempts to solve or validate the equation.

While n+k patterns were something very few languages had, another common feature of many programming languages with pattern matching is pattern guards. A *guard* is a predicate attached to a pattern that may make use of the variables bound in it. The result of its evaluation will determine whether the case clause and the body associated with it will be *accepted* or *rejected*. The following OCaml code for the *exp* language from Section 6.1 defines the rules for factorizing expressions $e_1e_2 + e_1e_3$ into $e_1(e_2 + e_3)$ and $e_1e_2 + e_3e_2$ into $(e_1 + e_3)e_2$ with the help of guards spelled out after the keyword when:

```
let factorize  e =
    match e with
      Plus(Times(e1,e2), Times(e3,e4)) when e1 = e3 → Times(e1, Plus(e2,e4))
    | Plus(Times(e1,e2), Times(e3,e4)) when e2 = e4 → Times(Plus(e1,e3), e4)
    | e → e
    ;;
```

One may wonder why we could not write the above case clause as Plus(Times(e,e_2), Times(e,e_4)) to avoid the guard. Patterns that permit use of the same variable in them multiple times are called *equivalence patterns*; the requirement of the absence of such patterns in a language is called *linearity*. Neither OCaml nor Haskell support such patterns, while Miranda [254] and Tom's pattern-matching extension to C, Java and Eiffel [186] support *non-linear patterns*.

The example above illustrates yet another common pattern-matching facility – *nesting of patterns*. In general, a constructor pattern composed of a linear vector of (distinct) variables is called a *simple pattern*. The same pattern composed not only of variables is called a *nested pattern*. Using nested patterns, with a simple expression in the case clause we could define a predicate that tests the top-level expression to be tagged with a Plus constructor, while both of its arguments to be marked with a Times constructor, binding their arguments (or potentially pattern matching further) respectively. Note that the visitor design pattern does not provide this level of flexibility and each of the nested tests might have required a new visitor to be written. Nesting of patterns like the one above is typically where users resort to *type tests* and *type casts* that in case of C++ can be combined into a single call to **dynamic_cast**.

Related to nested patterns are *as-patterns* that help one take a value apart while still maintaining its integrity. The following rule could have been a part of a hypothetical rewriting system in OCaml similar to the one above. Its intention is to rewrite expressions of the form $\frac{e_1/e_2}{e_3/e_4}$ into $\frac{e_1}{e_2}\frac{e_4}{e_3} \wedge e_2 \neq 0 \wedge e_3 \neq 0 \wedge e_4 \neq 0$.

```
    | Divide(Divide(_,e2) as x, Divide(e3,e4)) → Times(x, Divide(e4, e3))
```

We introduced a name "x" as a synonym of the result of matching the entire sub-expression Divide(_,e_2) in order to refer to it without recomposing in the right-hand side of the case clause. We omitted the conjunction of relevant non-zero checks for brevity, one can see that we will need access to e_2 in it however.

Decomposing algebraic data types through pattern matching has an important drawback that was originally spotted by Wadler [264]: they expose the concrete representation of an abstract data type, which conflicts with the principle of *data abstraction*. To overcome the problem, he proposed the notion of *views* that are implicitly applied during pattern matching and represent conversions between different representations. As an example, imagine polar and Cartesian representations of complex numbers. A user might choose polar representation as a concrete representation for the abstract data type complex, treating the Cartesian representation as a view, or vice versa:[2]

```
complex ::= Pole real real
view complex ::= Cart real real
  in  (Pole r t) = Cart (r * cos t) (r * sin t)
  out (Cart x y) = Pole (sqrt (x^2 + y^2)) (atan2 x y)
```

The operations then might be implemented in whatever representation is the most suitable, while the compiler will implicitly convert representation if needed:

```
  add  (Cart x1 y1) (Cart x2 y2) = Cart (x1 + x2) (y1 + y2)
  mult (Pole r1 t1) (Pole r2 t2) = Pole (r1 * r2) (t1 + t2)
```

The idea of views was later adopted in various forms in several languages: Haskell [40], Standard ML [197], Scala (in the form of *extractors* [87]) and F♯ (under the name of *active patterns* [249]).

Logic programming languages like Prolog take pattern matching to an even greater level. The main difference between pattern matching in logic languages and functional languages is that functional pattern matching is a "one-way" matching where patterns are matched against values, possibly binding some variables in the pattern along the way. Pattern matching in logic programming is "two-way" matching based on *unification* where patterns can be matched against other patterns, possibly binding some variables in both patterns and potentially leaving some variables *unbound* or *partially bound* – i.e. bound to patterns. A hypothetical example of such functionality can be matching a pattern Plus(x,Times(x,1)) against another pattern Plus(Divide(y,2),z), which will result in binding x to a Divide(y,2) and z to Times(Divide(y,2),1) with y left unbound, leaving both x and z effectively a pattern.

2.6 Expression Problem

Conventional algebraic data types, as found in most functional languages, allow for easy addition of new functions on existing data types. However, they fall short in extending data types themselves (e.g. with new constructors), which requires modifying the source code. Object-oriented languages, on the other hand, make data type extension trivial through inheritance, but the addition of new functions operating on these classes typically requires changes to the class definition. This dilemma was first discussed by Cook [58] and then accentuated by Wadler [265] under the name *expression problem.* Quoting Wadler:

> "The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts)."

Zenger and Odersky later refined the expression problem in the context of independently extensible solutions [281] as a challenge to find an implementation technique that satisfies the following requirements:

- *Extensibility in both dimensions*: It should be possible to add new data variants, while adapting the existing operations accordingly. It should also be possible to introduce new functions.
- *Strong static type safety*: It should be impossible to apply a function to a data variant, which it cannot handle.
- *No modification or duplication*: Existing code should neither be modified nor duplicated.

[2] We use the syntax from Wadler's original paper for this example

- *Separate compilation*: Neither data type extensions nor addition of new functions should require re-typechecking the original data type or existing functions. No safety checks should be deferred until link or runtime.
- *Independent extensibility*: It should be possible to combine independently developed extensions so that they can be used jointly.

While these requirements were formulated for extensible data types with disjoint variants, object-oriented languages primarily deal with hierarchical data types. We thus found it important to state explicitly an additional requirement based on the Liskov substitution principle [155]:

- *Substitutability*: Operations expressed on more general data variants should be applicable to ones that are more specific (the latter being in a subtyping relation with the former).

Numerous solutions have been proposed to dealing with the expression problem in both functional [104, 161] and object-oriented camps [116, 149, 201, 280], but very few have made their way into one of the mainstream languages. We refer the reader to Zenger and Odersky's original manuscript for a discussion of the approaches [281]. Interestingly, most of the discussed object-oriented solutions focused on the visitor design pattern [101] and its extensions, which even today seem to be the most commonly used approach to dealing with the expression problem in object-oriented languages.

2.7 Visitor Design Pattern

The *visitor design pattern* [101] was devised to solve the problem of extending existing classes with new functions in object-oriented languages. Consider the polymorphic encoding of the expression-term's ADT given in §2.4.1.1 and imagine that we would like to implement an evaluation function for a term in the expression language. A typical object-oriented approach would be to introduce a virtual function **virtual int** eval() **const** = 0; inside the abstract base class Expr, which will be implemented correspondingly in all the derived classes.

```
struct Expr { virtual int eval() const = 0; };
struct Value  : Expr { ... int eval() const { return value; } };
struct Plus   : Expr { ... int eval() const { return e1.eval() + e2.eval(); } }; ...
```

This works well as long as we know all the required operations on the abstract class in advance. Unfortunately, this is very difficult to achieve in reality as the code evolves, especially in a production environment. To put this in context, imagine that after the above interface with evaluation functionality has been deployed, we decided that we need similar functionality that pretty-prints a term or saves it in XML format [277]. Adding new virtual function implies modifying the base class and creating a versioning problem with the code that has already been deployed using the old interface.

To alleviate this problem, the *visitor design pattern* separates the *commonality* of all such future member functions from their *specifics*. The former deals with identifying the most-specific derived class of the receiver object known to the system at the time the base class was designed. The latter provides implementation of the required functionality once the most-specific derived class has been identified. The interaction between the two is encoded in the protocol that fixes a *visitation interface* enumerating all known derived classes on one side and a dispatching mechanism that guarantees selecting the most-specific case with respect to the dynamic type of the receiver in the visitation interface. An implementation of this protocol for our Expr example might look like the following:

```
// Forward declaration of known derived classes
struct Value; struct Plus; ... struct Divide;

// Visitation interface
```

```
struct ExprVisitor {
    virtual void visit (const Value&) = 0;
    virtual void visit (const Plus&)  = 0;
    ...  // One virtual function per each known derived class
    virtual void visit (const Divide&) = 0;
};
// Abstract base and known derived classes
struct Expr          { virtual void accept(ExprVisitor& ) const = 0; };
struct Value : Expr {      ... void accept(ExprVisitor& v) const { v.visit (*this); } };
struct Plus  : Expr {      ... void accept(ExprVisitor& v) const { v.visit (*this); } };
```

Note that even though implementations of accept member functions in all derived classes are syntactically identical, a different visit is called. We rely here on the overload resolution mechanism of C++ to pick the most specialized visit member function applicable to the static type of ***this**.

A user can now implement new functions by overriding ExprVisitor's functions. For example, our eval function can now be implemented as following:

```
int eval(const Expr* e)
{
  struct EvalVisitor : ExprVisitor {
    int rslt ;
    void visit (const Value& e) { rslt = e.value; }
    void visit (const Plus&   e) { rslt = eval(e.e1) + eval(e.e2); }
    void visit (const Minus& e) { rslt = eval(e.e1) − eval(e.e2); }
    void visit (const Times& e) { rslt = eval(e.e1) * eval(e.e2); }
    void visit (const Divide& e) { rslt = eval(e.e1) / eval(e.e2); }
  } v;
  e→accept(v);
  return v.rslt ;
}
```

Any new function that we would like to add to Expr can now be implemented in much the same way, without the need to change the base interface. This flexibility does not come free, though, and we would like to point out some pros and cons of this solution.

The most important advantage of the visitor design pattern is the *possibility to add new operations* to the class hierarchy without the need to change the interface. Its second most-quoted advantage is *speed*: the overhead of two virtual function calls incurred by the visitor design pattern is often negligible on modern architectures. Yet another advantage that often remains unnoticed is that the above solution achieves extensibility of functions with *library-only means* by using facilities already present in the language. Nevertheless, there are quite a few disadvantages.

The solution is *intrusive* since we had to inject syntactically the same definition of the accept method into every class participating in visitation. It is also *specific to hierarchy*, as we had to declare a visitation interface specific to the base class. There is a significant amount of both reusable and non-reusable *boilerplate code* that has to be written, which increases in the first case with the number of variants and in the second case with every argument that has to be saved inside the visitor to be available during the callback. Even once all the boilerplate has been written and the visitation interface has been fixed we are still left with some annoyances incurred by the pattern. One of them is the necessity to work with the *control inversion* that visitors put in place. Because of it, we have to save any local state and any arguments that some of the visit callbacks might need from the calling environment. Similarly, we have to save the result of the visitation, as we cannot assume that all the visitors that will potentially be implemented on a given hierarchy will use the same result type. Using visitors in a generic algorithm requires even more precautions.

More importantly, visitors *hinder extensibility* of the class hierarchy: new classes added to the hierarchy after the visitation interface has been fixed will be treated as their most derived base class present in the interface. A solution to this problem has been proposed in the form of *Extensible Visitors with Default Cases* [280, §4.2]; however, the solution, after remapping it onto C++, has problems of its own. The visitation interface hierarchy can easily be grown linearly (adding new cases for the new variants each time), but independent extensions by different authorities require a developer's intervention to unify them all before they can be used together. This may not be feasible in environments that use dynamic linking. Besides, to avoid writing even more boilerplate code in new visitors, the solution would require usage of virtual inheritance, which typically has an overhead of extra memory dereferencing. Additionally, on top of the two virtual function calls already present in the visitor pattern, the solution will incur two additional virtual calls and a dynamic cast for each level of visitor extension. Additional double dispatch is incurred by the forwarding of default handling from a base visitor to a derived one, while the dynamic cast is required for safety and can be replaced with a static cast when the visitation interface is guaranteed to be grown linearly (extended by one authority only). Yet another virtual call is required to be able to forward computations to subcomponents on tree-like structures to the most-derived visitor. This last function lets one avoid the necessity of using the heap to allocate a temporary visitor through the *factory design pattern* [101] used in the *Extensible Visitor* solution originally proposed by Krishnamurti, Felleisen and Friedman [149].

2.8 C++ Templates

A *template* in C++ defines a family of classes or functions or an alias for a family of types. The family can be parameterized with a number of arguments, each of which can be of the following three forms: *type parameter*, *non-type parameter* and *template parameter*. Template type parameters are indicated with keywords **typename** or **class** with no semantic difference between them. Template non-type parameters are indicated with their type, which can include preceding type parameters, a limited subset of built-in types and their derivatives. Template template parameters are indicated with a template declaration signature: e.g. **template** ⟨**template**⟨**typename**⟩**class** P⟩. . ..

A *class template* defines the layout and operations for an unbounded set of related types. The following example demonstrates a class template Array parameterized over a type parameter T and a non-type parameter N:

```
template ⟨class T, int N⟩
struct Array {
    typedef T element_type; // member type
    T& operator[](int);       // member function
    template ⟨class U⟩        // member template
      bool operator==(const Array⟨U,N⟩&);
    T elements[N];            // data member
};
```

A *partial specialization* of a class template provides an alternative definition of the template that is used instead of the *primary class template* definition when the arguments in a specialization match those given in the partial specialization. Each class template partial specialization is a distinct template and its definition is not required to follow the primary definition in any way. The following example provides two partial specializations of the above class template: one for pointer types and one for the second parameter being 0.

```
template ⟨class T, int  N⟩ struct Array⟨T∗,N⟩ {...};
template ⟨class T⟩         struct Array⟨T ,0⟩ {...};
```

A partial specialization *matches* a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list. Multiple matching partial specializations are compared according to a predefined partial order [131, temp.class.order] and a single best match is chosen or an ambiguity error is reported.

A *function template* defines an unbounded set of related functions. A function template can be overloaded with other function templates and with normal (non-template) functions.

```
template ⟨class T, int  N⟩ T   sum(const Array⟨T   , N⟩&);
template ⟨         int  N⟩ int sum(const Array⟨bool,N⟩&);
                           int sum(const Array⟨bool,0⟩&);
```

Template arguments of a function template can be explicitly specified or, when possible, implicitly deduced from a call. When function template is overloaded, the use of a function template specialization might be ambiguous, in which case a similar partial ordering of overloaded function template declarations is used to find out which of the applicable ones is the most specialized [131, temp.func.order].

An *alias template* defines a synonym for a family of types that can be understood as a type function that takes a number of template arguments and maps them to a new type. For example, the following definitions introduce a type function Point that maps any type T to a type Array⟨T, 3⟩, as well as a type function PointElementType that will map the argument type of Point type function to the element type of the underlying implementation type Array⟨T, 3⟩.

```
template ⟨class T⟩ using Point = Array⟨T, 3⟩;
template ⟨class T⟩ using PointElementType = Point⟨T⟩::element_type;
```

2.8.1 Template Metaprogramming

Interestingly enough, C++ has a pure functional sublanguage in it that has a striking similarity to ML and Haskell. The sublanguage in question is based on the template facilities of C++, can be used for compile-time *metaprogramming*, and has been shown to be Turing complete [259]. By *metaprogram* here, we understand a program that can analyze and/or manipulate another program or its part, including itself. The Haskell definition of factorial we saw earlier can be rewritten as the following metafunction to be evaluated at compilation time:

```
template ⟨int N⟩ struct factorial     { enum { result = N∗factorial⟨N−1⟩::result  };  };
template ⟨⟩      struct factorial ⟨0⟩ { enum { result = 1 }; };
```

One can easily see similarity with equational definitions in Haskell, with the exception that more specific cases (specialization for 0) have to follow the general definition in C++. The main difference between the Haskell definition and its C++ counterpart is that the former describes computations on *run-time values*, while the latter deals with *compile-time values*.

The similarity between templates in C++ and patterns in functional languages becomes even more obvious once we note that templates essentially represent *code patterns* – code recipes for generating data structures and algorithms parameterized by a variety of possible arguments. While functional patterns concentrate on recipes for concise expression of predicates, C++ templates are more general and allow for expressing arbitrary type-safe code patterns, which in turn makes them somewhat more verbose. Nevertheless, we demonstrate in Chapter 6 that templates can be used to build succinct, first-class functional patterns in C++.

It turns out, the template sublanguage of C++ is so powerful, we can easily encode the *exp* language of §2.4 in it. We start with encoding variants:

```
template ⟨int N⟩                class value  {};
template ⟨class E1, class E2⟩ class plus   {};
template ⟨class E1, class E2⟩ class minus  {};
template ⟨class E1, class E2⟩ class times  {};
template ⟨class E1, class E2⟩ class divide {};
// Compile–time encoding of term 3*4+2
typedef plus⟨times⟨value⟨3⟩,value⟨4⟩⟩, value⟨2⟩⟩ e3x4p2;
```

A value N is represented as an instantiation of a class template value parameterized with a non-type parameter of type **int**, e.g. value⟨5⟩. Similarly, the addition of two terms is encoded with an instantiation of a class template plus with type arguments that correspond to an encoding of the subterms. Type e_3x4p2 can then be viewed as a compile-time encoding of term $3 * 4 + 2$ in the language. A metafunction equivalent of the eval function can be defined as following:

```
template ⟨class E⟩
  struct eval; // Undefined
template ⟨size_t N⟩
  struct eval⟨       value⟨N⟩⟩ { enum { res = N } };
template ⟨class E1, class E2⟩
  struct eval⟨  plus⟨E1,E2⟩⟩ { enum { res = eval⟨E1⟩::res + eval⟨E2⟩::res } };
template ⟨class E1, class E2⟩
  struct eval⟨ minus⟨E1,E2⟩⟩ { enum { res = eval⟨E1⟩::res − eval⟨E2⟩::res } };
template ⟨class E1, class E2⟩
  struct eval⟨ times⟨E1,E2⟩⟩ { enum { res = eval⟨E1⟩::res * eval⟨E2⟩::res } };
template ⟨class E1, class E2⟩
  struct eval⟨divide⟨E1,E2⟩⟩ { enum { res = eval⟨E1⟩::res / eval⟨E2⟩::res } };
constexpr int M = eval⟨e3x4p2⟩::res; // M ==14
```

Modern metaprogramming practice in C++ will tell us better ways of encoding the *exp* language and the eval metafunction; however, here we were striving for clarity of presentation and opted for an implementation that requires a minimum of prior knowledge.

2.8.2 Expression Templates

The ability to express the *exp* language at compilation time seems like a nice but useless trick. We introduced it first, because it gives rise to a somewhat more elaborate, but extremely powerful technique, used in real-world production code. The technique allows one to capture the structure of expressions in a C++ program and then give them an alternative semantics: e.g. evaluate them lazily, fix or change the order of evaluation, parallelize, optimize etc. The technique was independently invented by Todd Veldhuizen [257] and David Vandevoorde [256] and is generally known in the C++ community by the name *expression templates* coined by Veldhuizen.

The compile-time encoding of the *exp* language we saw earlier has an important property: it encodes the structure of an expression in a type. Explicit description of such expressions is very tedious, but in many cases it can be implicitly created through function and operator overloading:

```
template ⟨class T⟩
struct value {
    value(const T& t) : m_value(t) {}
    T m_value;
};
template ⟨class T⟩
```

```
struct variable {
    variable() : m_var() {}
    T m_var;
};
template ⟨class E1, class E2⟩
struct plus {
    plus(const E1& e1, const E2& e2) : m_e1(e1), m_e2(e2) {}
    const E1m_e1; const E2m_e2;
};
// ...definitions of other expressions
template ⟨class T⟩
    value⟨T⟩ val(const T& t) { return value⟨T⟩(t); }
template ⟨class E1, class E2⟩
    plus⟨E1,E2⟩ operator+(const E1& e1, const E2& e2)
    { return plus⟨E1,E2⟩(e1,e2); }
```

With this, one can now capture various expressions as follows:

```
variable⟨int⟩ v;
auto x = v + val(3);
```

The type of the variable x – plus⟨variable⟨**int**⟩,value⟨**int**⟩⟩ – captures the structure of the expression, while the values inside of it represent various subexpressions the expression was composed with. Such an expression can be arbitrarily but finitely nested. Note that the value 3 is not added to the value of the variable v here, but the expression v+3 is recorded, while the meaning of this expression can be different in different contexts. A general observation is that only the shape of the expression becomes fixed at compilation time, whereas the values of the variables involved in it can be changed arbitrarily at run time, allowing for *lazy evaluation* of the expression later. The polymorphic function eval, below, implements just that:

```
template ⟨class T⟩
    T eval(const value⟨T⟩& e) { return e.m_value; }
template ⟨class T⟩
    T eval(const variable ⟨T⟩& e) { return e.m_value; }
template ⟨class E1, class  E2⟩
    auto eval(const plus⟨E1,E2⟩& e)
        → decltype(eval(e.m_e1) + eval(e.m_e2))
          { return eval (e. m_e1) + eval(e.m_e2); }
```

Note again how this implementation of eval resembles equations in Haskell that decompose an algebraic data type. The similarities are so striking that there were attempts to use Haskell as a pseudocode language for template metaprogramming in C++ [173]. A key observation in this analogy is that C++ function template overloading is similar to case analysis performed by defining equations of Haskell functions. Variables introduced via the template clause of each equation serve as *variable patterns*, while the names of actual templates describing arguments serve as *variant constructors*. An important difference between the two is that Haskell's equations use the *first-fit* strategy, making the order of equations important, while C++ uses the *best-fit* strategy based on partial ordering of function template specializations, thus making the order irrelevant.

We would like to point out that this time eval is a function template and thus produces executable code. eval from §2.8.1 was a class template that was only used for compile-time computations. Both use the structure of the type to drive the compile-time recursion: in the class template it is used to fold the result, whereas in the function template it is used to generate code that corresponds to evaluation of an expression. It is important to understand that each instantiation of the eval function template is not a (run-time) recursive function, but rather a function that calls a completely different function. The call to eval(x), for example results in the following call tree:

```
eval⟨plus⟨variable⟨int⟩,value⟨int⟩⟩⟩(x)
    eval⟨variable⟨int⟩⟩(x.m_e1)
        x.m_e1.m_var
    eval⟨value⟨int⟩⟩(x.m_e2)
        x.m_e2.m_value
```

Because the bodies of the functions eval⟨variable⟨**int**⟩⟩ and eval⟨value⟨**int**⟩⟩ are small, they are both likely to be inlined inside the call to eval⟨plus⟨variable⟨**int**⟩,value⟨**int**⟩⟩⟩, which will still be small enough to get inlined, resulting in the entire call to eval(x) being partially evaluated to x.m_e_1.m_var + x.m_e_2.m_value.

Expression templates can be easily analyzed and transformed at compile time as long as the information of interest can be expressed in terms of the types involved and not the values they store. For instance, in the above example, it is very easy to specialize eval (or any other operation, e.g. collect) for an expression of the form $c_1 * x + c_2$ where c_i are some (not known) constant values and x is any variable. Specializing for a concrete instance of that expression $2 * x + 3$ will be much harder, because in our encoding both values (2 and 3) are represented with the same type value⟨**int**⟩ and thus are indistinguishable at compilation time. To alleviate this, we could have devised a template that allocates a dedicated type for each constant, making its value part of the type:

```
template ⟨class T, T t⟩ struct constant {};

template ⟨class T, T t⟩
  T eval(const constant⟨T,t⟩& e) { return t; }
template ⟨class T, class E⟩
  constant⟨T,0⟩ eval (const times⟨constant⟨T,0⟩, E⟩&)
  { return constant⟨T,0⟩(); }
template ⟨class E, class T⟩
  constant⟨T,0⟩ eval (const times⟨E,constant⟨T,0⟩⟩&)
  { return constant⟨T,0⟩(); }
```

Here the first equation for eval describes the necessary general case for handling expressions of type constant⟨T,t⟩, while the other two are redundant cases that can be seen as an optimization detecting expressions of the form $e * 0$ and $0 * e$ for any arbitrary expression e and returning 0 without actually computing e.

Unfortunately, a similar pattern to detect expressions of the form $x - x$ for any variable x cannot be expressed because expression templates are blind to object identity and can only see their types. This means that expression templates of the form $x - y$ are indistinguishable at compilation time from expressions of the form $x - x$ because their types are identical.

2.8.3 Concepts

Concept is the C++ community's long-established term for a set of requirements for template parameters. Concepts were not included in C++11, but techniques for emulating them with enable_if [135] have been in use for a while [225]. Here we present briefly the notation for *template constraints* – a lighter version of the concepts proposal [248].

A *constexpr function* is a syntactically restricted function that can be evaluated at compilation time [131, dcl.constexpr(3)]. Currently **constexpr** functions are restricted to a single return statement that may not have any side effects. A *constraint* is an unconstrained **constexpr** function template that takes no arguments and returns a **bool**. The proposal for template constraints extends the syntax accepted inside **constexpr** functions to express certain syntactic and semantic requirements. Consider, for example, the definition of Equality_comparable constraint:

```
template⟨class T⟩
constexpr bool Equality_comparable() {
    return requires (T a, T b) {
        bool = {a ==b};
        bool = {a ≠b};
    };
}
```

The *requires clause* introduces a set of syntactic constraints that the type T must satisfy. For convenience, it may introduce some formal parameters to avoid using declval⟨T⟩ [131, declval]. Here is an incomplete list of some commonly-used syntactic requirements:

```
e;              // e is a valid expression
T = {e};        // decltype(e) is convertible to T
T == {e};       // decltype(e) is the same as T
typename T::X;  // X is an associated type of T
```

which now lets us interpret the above definition. A type T that satisfies the Equality_comparable constraint must have an equality == and inequality ≠ operators defined on it. The result of these operators should be convertible to **bool**.

An algorithm or a data type definition may then put constraints on its arguments as following:

```
template ⟨Equality_comparable T⟩ class point;
template ⟨Equality_comparable T⟩
  bool lexicographic_array_compare(const T (&a)[N], const T (&b)[N]);
```

Constraint violations are diagnosed at the point of use, just like type errors. Unlike template instantiation errors of C++98, which were reported in terms of implementation details, constraint errors are expressed in terms of unsatisfied requirements and better represent programmer's intent. We refer the reader to the original "Concepts Lite" proposal [248] for an in-depth discussion of template constraints.

3. RELATED WORK[1]

> When you want to do something differently from the rest of the world, it's a good idea to look into whether the rest of the world knows something you don't.
>
> unknown programmer

In this chapter, we look at other work that is related to the topic of this dissertation. We start with an overview of various frameworks that support parsing of C++. We then look at various library-based approaches as well as approaches based on a preprocessor or a tool that help one ensure properties of interest.

For simplicity, we refer to approaches that can be implemented as a library within a language as *internal*, whereas we refer to approaches that require a preprocessor, a tool or modifications to the compiler as *external*.

3.1 C++ Parsers and Front Ends

Parsing C++ code is notoriously difficult: the language cannot be described with an LL or LR grammar of any lookahead size, and syntax analysis depends on semantic disambiguation [133, 270]. Because of this, many industrial-strength C++ front ends like Clang [51], GCC [106] and EDG [84] do not use parser generators, but instead rely on manually-handcrafted parsers. Today they are the typical choices for parsing C++ code due to their high conformance to the standard. Unfortunately, GCC is not a good choice when the code being parsed uses language extensions from other compilers, while EDG is free only for academic research.

ROSE is a source-to-source transformation framework for C++, developed at Lawrence Livermore National Laboratory [219]. Its strength is the preservation of syntactic information about the original program. They achieve that by keeping information about macro expansion as well as ensuring that only the affected program entities are printed after transformation, while the rest of the code is copied verbatim from the original source.

Elsa is a C++ parser that was built to demonstrate the effectiveness of the Elkhound parser generator [170]. Elsa's description of almost the entire C++ grammar (with the exception of template template arguments) was done in about 3500 lines of code. Its parser and AST are designed to be easily extensible by adding more grammar rules. Unfortunately, Elsa only does partial semantic analysis of the program, and it does not provide a preprocessor and thus does not keep links back to the original source code. Besides, its user community is rather small and major developments on the project stopped a few years ago.

Modern compiler frameworks such as LLVM [151], Phoenix [172] and GCC [106] provide an extensible architecture that can be used to build tools for language analysis, optimization, reverse engineering etc.

[1] Parts of this chapter are reprinted with permission from "Open and Efficient Type Switch for C++" by Yuriy Solodkyy, Gabriel Dos Reis, Bjarne Stroustrup, 2012. Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications, Pages 963-982, Copyright 2012 by ACM; as well as "Design and Evaluation of C++ Open Multi-Methods" by Peter Pirkelbauer, Yuriy Solodkyy, Bjarne Stroustrup, 2010. Science of Computer Programming, Volume 75, Issue 7, Pages 638-667, Copyright 2009 by Elsevier B.V.; and also "Extending Type Systems in a Library: Type-Safe XML Processing in C++" by Yuriy Solodkyy, Jaakko Järvi, 2011. Science of Computer Programming, Volume 76, Issue 4, Pages 290-306, Copyright 2010 by Elsevier B.V.

Typically, these frameworks try to leverage development of compiler back ends for different programming languages, which makes their program representation be much closer to the level of machine instructions than to the level of the original source code. The highest level of representation in Phoenix, for example, will already have data types as well as code laid out in a particular way, losing a significant amount of information about the user's original intent: e.g. a matrix multiplication operator inlined by the front end will look like a spaghetti of conditional and unconditional jumps, making it impossible to recognize the original intent. The strength of these tools lies in the multitude of readily available low-level optimizations in them and in their ease of targeting multiple platforms.

3.1.1 Pivot

To simplify abstract syntax trees, the Pivot infrastructure [81] developed by my colleagues at Texas A&M University uses generic programming extensively to minimize the number of different node kinds. It has 162 different node kinds in its *Internal Program Representation* (*IPR*) [78], all of which fall into only 5 basic kinds that can be treated generically in many contexts. Pivot uses unification of nodes to minimize the size of the actual AST. IPR is a compact (through the use of node unification) and generic (through its conceptualization of AST node kinds) representation of the entire C++ language, and is capable of handling erroneous and incomplete programs. Abstractions used by IPR allow it to represent not only existing language features, but also the features of the new C++11 standard [131] as well as language extensions used by various compilers. The representation stays as close as possible to the original source code, thus maintaining not only semantically important information, but also information on the exact syntax used to represent program's entity. This puts the framework into the unique position of being capable of writing both semantic analyzers and syntax checkers, as well as working with uninstantiated templates and annotations provided inside comments. Pivot does not provide its own C++ front end; it relies on our modified versions of existing front ends in order to generate IPR.

3.2 Internal Approaches

Several libraries taking the role of an extended type checker have been proposed before. For example, C++ libraries for tracking physical units are presented in [15, 35]. Such libraries are generally possible because C++ templates define a Turing-complete functional sub-language [259] that allows arbitrary computations on constants, types and templates to be performed at compile time. Such *template metaprograms* [258] have found uses in various C++ libraries (see e.g. Boost.type_traits [162], and numerous other libraries in the Boost library collection [30]). To help this style of metaprogramming, a number of techniques have been discovered: e.g. to selectively enable or disable functions in an overload set [135,136], detect the presence of certain class-members, concept-check type parameters [225] etc. Template metaprogramming, however, remained a relatively ad-hoc activity until the introduction of the Boost Metaprogramming Library (MPL) [1,119]. MPL provides a solid foundation for metaprogramming in C++, defining essentially a little programming language and a supporting library for defining *metafunctions*; in MPL metafunctions are functions from types to types. MPL allows one to define higher-order metafunctions, lambda metafunctions etc., and provides a host of data structures and algorithms for storing and manipulating types.

Veldhuizen and Gannon proposed the idea of *Active Libraries* [260], which take an active role in generating and optimizing code as well as interacting with programming tools. Stroustrup proposed the idea of *SELL: Semantically Enhanced Library Languages* [245, 246]. The difference between SELL and Active Libraries approach lays mainly in the accents: while Active Libraries rely only on library extensions (although they advocate for introducing the language features necessary for building such extensions), the SELL approach admits extensions through tools and thus eliminates requiring changes to the language by offloading the necessary support to a tool.

Reusing existing high-level library components for many different applications can simplify the development of a large system, but is known to result in poor performance of the resulting applications. Besides, heavy use of active libraries often significantly slows down compilation times. To overcome these deficiencies, Kennedy et al propose the idea of *Telescoping Languages* [145]. Under their approach, end users can still develop programs in high-level domain-specific programming systems, while optimal performance is achieved through a language preprocessor, generated for a particular component library through exhaustive analysis of it. The generation of such a language processor is automatic, though it may take a significant amount of time due to exhaustive analysis of the library. Nevertheless, once the preprocessor is produced, it can efficiently translate and optimize the code using the knowledge gathered during the generation phase.

3.3 External Approaches

The main difficulty in developing language tools that fall into this category for C++ is the complexity of the language as well as absence of a standardized front end, which is available in many other languages. Besides, the proliferation of compiler extensions and language dialects as well as differences in support of the C++ standard [129] has made the creation of such a standardized front end even more difficult.

Astrée [25] is a static program analyzer that proves absence of run time errors in programs written in C. It can analyze C programs that do not use dynamic memory allocation and recursion, which is the case for most embedded applications. The analyzer is capable of detecting the use of undefined behavior, the violation of an implementation-defined behavior, and the violation of assertions. It is also sound for floating point computations, as it considers all possible rounding errors. Written in OCaml, it scales up to industrial-sized programs and can be run in parallel.

PolySpace Verifier [169] is a commercial general-purpose static analyzer for embedded applications based on abstract interpretation. It can handle code in C, C++ and Ada, scales to industrial-sized programs and can be customized for specific platforms and operating environments. It incorporates many known abstract domains into the analysis as well as integrates visual feedback about the results into an IDE. A comparative study of various static analyzers used to detect buffer overflow vulnerabilities was conducted by Zitser et al. [283] and clearly shows an advantage in precision of the analysis done by PolySpace Verifier.

Coverity is another industrial-strength bug-finding tool based on abstract interpretation that works with C, C++, Java and C# [19]. The developers of the tool take extra care to not only find potential problems in code, but also track and suppress them, present in an understandable form to developer etc.

3.4 Multiple Dispatch

Programming languages can support multi-methods through either built-in facilities, preprocessors, or library extensions. Naturally, tighter language integration enjoys a much broader design space for type checking, ambiguity handling, and optimizations when compared to libraries. In this section, we will first review both library and non-library approaches to multiple dispatch in C++ and then give a brief overview of multi-methods in other languages.

Cmm [229] is a preprocessor-based implementation for an open multi-methods C++ extension. It takes a translation unit and generates C++ dispatch code from it. Cmm is available in two versions: one uses RTTI to recover the dynamic type of objects to identify the best overrider, and the other achieves constant-time dispatch by relying on a virtual function overridden in each class. Dispatch ambiguities are signaled by throwing runtime exceptions. Cmm allows dynamically linked libraries to register and unregister their open-methods at load and unload time. In addition to open-method dispatch, Cmm also provides call-site virtual dispatch. Call site virtual dispatch delays the binding to regular overloaded functions if one of their actual arguments is preceded by the **virtual** keyword – a syntax proposed in [243] after an earlier idea by Doug Lea (see [243, §13.8]).

```
void foo(A&);
void foo(B&); // B derives from A

// call site virtual dispatch
foo(virtual x); // which foo gets invoked depends on the dynamic type of x
```

Cmm does not provide special support for repeated inheritance, and therefore its dispatch technique does not entirely conform to virtual function semantics.

DoubleCpp [20] is another preprocessor-based approach for multi-methods dispatching on two virtual parameters. It essentially translates these multi-methods into the visitor pattern. For doing so, DoubleCpp requires access to the files containing the class definitions in order to add the appropriate accept and visit methods. DoubleCpp, unlike other visitor-based approaches, reports potential ambiguities.

The accessory functions papers [94, 272] allow open-method dispatch based on a single virtual argument and discuss ideas to extend the mechanism for multiple dispatch. The compilation model they describe, like our approach, uses cooperation between the compiler and linker to perform ambiguity resolution and dispatch-table generation. However, the accessory functions are integrated into the regular v-tables of their receiver types, which requires the linker to not only generate the dispatch table but also to recompute and resolve the v-table index of any other virtual member function. Neither paper provides a detailed discussion of the intricacies when multiple inheritance is involved.

Loki [4], based on Alexandrescu's template programming library with the same name, provides several different dispatchers that balance between speed, flexibility, and code verbosity. Currently, it only supports multi-methods with two arguments, except for the constant-time dispatcher, which allows more arguments. The static dispatcher provides call resolution based on overload resolution rules, but requires manual *linearization* of the class hierarchy (§5.3.1) in order to uncover the most derived type of an object first. All other dispatchers do not consider hierarchical relations and effectively require explicit resolution of all possible cases.

In [202], Panizzi and Pastorelli describe their open-method implementation for OOLANG, a language with a C++-like object model that was developed for the Apemille SPMD supercomputer. The paper gives special attention to the handling of covariant return types. OOLANG's system differs from our implementation in the handling of repeated inheritance. Classes that repeatedly inherit from a base class must define an overrider for each open-method that uses the base class as a type for a virtual parameter. Furthermore, OOLANG does not use covariant return type information for ambiguity resolution.

Besides the approaches mentioned in §4.4, languages can provide multi-method abstractions through a library (e.g. Python [216]). Chambers and Chen [49] present an alternative implementation technique based on a lookup DAG. Their work generalizes multiple dispatch to be a subset of predicate-based dispatch.

In order to estimate how often multiple dispatch is used in practice, Muschevici et al [188] studied programs that utilize dynamic dispatch. The article introduces a language-independent model for describing multiple dispatch, and defines six metrics on generic functions (i.e. in C++, an open-method family or a virtual function and its overriders) that measure aspects such as the number of arguments used for dynamic dispatch, the number of overriders in a multi-method family, etc. Using these metrics, the article analyzes nine applications—mostly compilers—written in six different languages: CLOS, Dylan, Cecil, MultiJava, Diesel [48], and Nice [29]. Their results show that 13%-32% of generic functions utilize the dynamic type of a single argument, while 2.7%-6.5% of them utilize the dynamic type of multiple arguments. The remaining 65%-93% generic functions have a single concrete method, and therefore are not considered to use the dynamic types of their arguments. In addition, the study reports that 2%-20% of generic functions had two concrete function implementations, and 3%-6% had three. The numbers decreases rapidly for functions with more concrete overriders.

3.5 Type Switching

Lookup caches have long been used to reduce the overhead of dynamically-bound message passing in Smalltalk [255]. *Inline caching* improves on that by noting that the type of the receiver at a given call site rarely varies, so the call instruction can be speculatively modified to jump directly to a previously-looked-up method [76]. In this case, the method must ensure that the type of the receiver has not changed and redirect the call to generic lookup otherwise. The effects of inline caching on modern architectures can be seen through hardware call-target prediction in §5.6.1.

Polymorphic Inline Caches [123] generalize the idea of inline caches further by building a decision tree in the method prologue that caches all lookup results. The main difference of this approach from our work is that it requires code generation at run time, while we do not require recompilation, relinking or any computations in the case of dynamic linking. The reason for this is the difference in the initial setting: they map an arbitrary number of receiver types to an arbitrary number of implementations, while we map an arbitrary number of receivers to a fixed number of jump targets. This lets us generate code at compile time that incorporates both the initial execution and the memoized one.

Extensible Visitors with Defaults [280, §4.2] offer a solution to the extensibility problem of visitors. The visitation interface hierarchy can easily be grown linearly, but independent extensions by different programmers require manual coordination. In addition to the double dispatch, the solution incurs two additional virtual calls and a dynamic cast for each level of visitor extension. The solution is simpler with virtual inheritance, which adds even more indirections.

Löh and Hinze proposed to extend Haskell's type system with open data types and open functions [161]. The solution allows top-level data types and functions to be marked as open with concrete variants and overloads defined anywhere in the program. The semantics of open extension is given by transformation into a single module, which assumes a whole-program view and thus is not an open solution by our definition. Besides, open data types are extensible but not hierarchical, which avoids the problems discussed here.

Kennedy et al [144] considered encoding generalized algebraic data types in C# [206] using the visitor design pattern. That translation made essential use of generic methods, the equivalent of "virtual function template" (as they would be called if such functionality existed in C++) to handle some of the open set aspects of GADTs.

Polymorphic variants in OCaml [104] allow the addition of new variants while defining subtyping on them. The subtyping, however, is not defined between the variants, but between combinations of them. This maintains disjointness between values from different variants and makes an important distinction between *extensible sum types* like polymorphic variants and *extensible hierarchical sum types* like classes. Our memoization device can be used to implement pattern matching on polymorphic variants as well.

Tom is a pattern-matching compiler that can be used together with Java, C or Eiffel to bring a common pattern matching and term rewriting syntax into the languages [186]. In comparison to our approach, Tom has much bigger goals: the combination of pattern matching, term rewriting and strategies turns Tom into a fully-fledged tree-transformation language. Its type patterns and %match statement can be used as a type switch, but their implementation is based on decision trees and an instanceof-like predicate.

Pattern matching in Scala [195] also supports type switching through type patterns. The language supports extensible and hierarchical data types, but their handling in a type switching constructs varies. Sealed classes are handled with an efficient switch over all tags, but extensible classes are handled with a combination of an InstanceOf operator and a decision tree [87].

3.6 Pattern Matching

Language support for pattern matching was first introduced for string manipulation in COMIT [279], which subsequently inspired similar primitives in SNOBOL [92]. SNOBOL4 had string patterns as first-class data types providing operations of concatenation and alternation [108]. The first reference to modern pattern-matching constructs as seen in functional languages is usually attributed to Burstall's work on structural induction [38]. Pattern matching was further developed by the functional programming community, most notably Hope [39], Miranda [254], ML [177] and Haskell [141]. In the context of object-oriented programming, pattern matching has been first explored in Pizza [196] and Scala [87,195]. The idea of first-class patterns dates back at least to Tullsen's proposal to add them to Haskell [253]. The calculus of such patterns has been studied in detail by Jay [137,138].

There are two main approaches to compiling pattern-matching code: the first is based on *backtracking automata* and was introduced by Augustsson [11], the second is based on *decision trees* and was first described by Cardelli [43], though he attributes the technique to Dave MacQueen and Gilles Kahn in their implementation of the Hope compiler [39]. The backtracking approach usually generates smaller code [152], whereas the decision tree approach produces faster code by ensuring that each primitive test is only performed once [165].

MatchO was one of the first attempts to implement pattern matching without extending the host language [261]. While the library essentially added first-class patterns to Java, the amount of syntactic and run-time overhead was overwhelming. The author offers to provide customized syntax to patterns by invoking an inline parser, while to improve performance the author looks into combining visitors and visitor combinators with pattern matching. This resembles our use of an efficient type switch to uncover the dynamic type, combined with slower but more flexible pattern matching.

Functional C# was a similar approach, bringing pattern matching to C# as a library [204]. The approach uses lambda expressions and chaining of method calls to create a structure that is then evaluated at run time for the first successful match. The approach supports a form of active patterns, simple n+k patterns, list and tuple patterns as well as type patterns (without structural decomposition). Unfortunately, the solution scales very poorly for match statements with more than two case clauses as it is essentially based on sequential type tests. Besides, the approach is ill-suited for tests involving nesting of patterns.

Both *MatchO* and *Functional C#* were object-oriented solutions. In the functional community, Rhiger explored a similar idea of introducing pattern matching into Haskell as a library [213]. He uses functions to encode patterns and pattern combinators, which allows him to detect pattern misapplication errors at compile time through the Haskell type system. Unfortunately, the purity of a functional language prevents him from expressing simply variable patterns (more specifically, the bindings they introduce) in a library setting, which then affects the overall elegance of the solution. In particular, it seems to be impossible in his approach to express equivalence patterns or even an equivalence combinator as the value of the bound variable is not available in the left-hand side of the matching expression, only in the right-hand side. The library is sufficiently expressive to express basic patterns (value, wildcard, predicate, etc.), boolean pattern combinators and set patterns. The author reports that his solution requires 2-4 times more reductions than Haskell's built-in pattern matching. He attributes the overhead to currying and continuation passing and hypothesizes that it should be eliminated by standard partial evaluation techniques.

Racket has a very powerful macro system that allows it to express open pattern matching in the language entirely as a library [251]. The work builds on earlier attempts to bring pattern matching to Scheme, extending it and making it more efficient [274]. The solution is remarkable in that unlike most of the library approaches to open pattern matching, it does not rely on naïve backtracking and, in fact, encodes the optimized algorithm based on backtracking automata [11,152]. The solution is easy to extend with new kinds of patterns. Its only disadvantage is the fact that the untyped nature of Racket makes any static checking of patterns hard, if not impossible.

Grace is another programming language that provides a library solution to pattern matching through objects [124]. The language aims at teaching various programming paradigms and has an interesting peculiarity: it does not have pre-defined control structures. Instead, control structures are expressed directly in the language with the help of partial functions and lambda expressions. The **match** statement, for example, takes the form of a clausal definition **match**(e)case(c_1)...case(c_n), in which case blocks c_i are single-argument blocks that take pattern objects as input. Patterns are objects in Grace. The compiler provides some syntactic sugar to denote some common patterns, but otherwise, user-defined patterns must be created explicitly. They are composed and applied at run time using the naïve backtracking approach without any current provision for optimizations. This is a design choice, as Grace is concerned not with performance, but with the expressiveness of the language and ease of teaching different programming paradigms. Unlike many object-oriented languages, Grace does not use nominative subtyping and thus their case analysis is never hierarchical; instead, they use structural subtyping, which is more common in functional languages.

Thorn is a dynamically-typed scripting language that provides first-class patterns [27]. Pattern matching in Thorn, however, is not implemented in a library; instead, it is tightly integrated into the language and interacts seamlessly with other constructs. The language defines a handful of atomic patterns and pattern combinators to compose them, and, similar to Newspeak and Grace, uses the duality between partial functions and patterns to support user-defined patterns. The duality is particularly suitable for providing support to the structural decomposition of classes: along with a regular constructor, a class may provide a destructuring constructor, which allow the use of construction syntax in the pattern-matching context. As with the other solutions to first-class patterns we looked at, the pattern matching is currently performed in the naïve backtracking way. Unlike the library solutions, however, tighter integration into the language and the presence of built-in patterns and pattern combinators should enable at least some of the known optimization techniques mentioned above [152,165], remember that Thorn is an interpreter though.

Prop was a language extension that brought pattern matching and term rewriting into C++ [154]. The extension aimed at building high-performance compiler and language transformation systems. It did not offer first-class patterns, but supported most of the functional-style patterns and provided an optimizing compiler for both pattern matching and garbage-collected term rewriting. *App* was another pattern-matching extension to C++ [192]. It provided syntax for defining algebraic data types and pattern matching on them that was very close to Haskell. Unlike Prop, App concentrates on algebraic data type decomposition and only supports wildcard, variable, constructor and a zero pattern – a value pattern for **nullptr**. App also does not provide any optimizations and simply generates a cascade of type tests.

Tom is a pattern-matching compiler that took the ideas in Prop much further, relying on the fact that pattern-matching primitives and optimizations on them are language agnostic [186]. It brings a common pattern-matching and term-rewriting syntax into Java, C and Eiffel, but can be implemented on top of many other programming languages. Thanks to its distinct syntax, it is transparent to the semantics of the host language and is implemented as a preprocessor to that language. Similar to Prop and App, Tom neither supports first-class patterns, nor is open to new patterns. Tom's goals differ from ours in aiming to be a tree-transformation language similar to Stratego/XT, XDuce and others. Tom's approach is prone to the general problems of any preprocessor based solution [245, §4.3]. In particular, it is part of a dedicated tool chain. Our library approach avoids that and lets us employ C++ semantics within patterns: e.g. our patterns work directly on underlying user-defined data structures, avoiding abstraction penalties. The tight integration with the language semantics makes our patterns first-class citizens that can be composed and passed to other functions.

Matchete is a language extension to Java that brings together different flavors of pattern matching: functional-style patterns, Perl-style regular expressions, XPath expressions, Erlang's bit-level patterns etc. [122]. The extension does not try to make patterns first-class citizens, but instead concentrates on implementing existing best practices and their tight integration into Java. This also avoids the necessity of remaining within the boundaries of Java and allows them to improve significantly the syntax. The extension offers a wide variety of built-in patterns, but it does not try to optimize the matching and resorts to naïve backtracking.

OOMatch is another Java extension that brings pattern matching and multiple dispatch close together [214]. The approach generalizes multiple dispatch by offering to use patterns as multi-method arguments and then orders overriders based on the specificity of their arguments. This allows avoiding many of the ambiguities typical of symmetric multiple dispatch, but it does not eliminate them entirely. Similar to others, the approach only deals with a limited set of built-in patterns.

When a class hierarchy is fixed, one can design a pattern language that involves semantic notions represented by the hierarchy. Pirkelbauer devised a pattern language for Pivot [81] capable of representing various entities in a C++ program using syntax very close to C++ itself. The patterns were translated with a tool into a set of visitors implementing the underlying pattern-matching semantics efficiently [207]. Earlier, Cook et al used expression templates to implement a query language for Pivot's class hierarchy [57].

3.7 Abstract Interpretation

The theoretical background for the theory of abstract interpretation was given in [60, 65]. In [62] Cousot shows how the classical semantics of programs, modeled as transition systems, can be derived from one another by Galois connection based abstract interpretations. A very detailed calculational design of a generic abstract interpreter capable of handling different abstract domains is presented in [61].

Representation of source-to-source transformations by abstract interpretation has been presented in [69]. General theoretic background for the modular static analysis of object-oriented languages with abstract interpretation presented in detail in Logozzo's PhD thesis [160]. Patrick Cousot maintains an extensive survey of all the work that has been done on abstract interpretation [63].

In addition to great amount of work done on the theoretical aspects of abstract interpretation, a significant amount of research was focusing on developing abstract domains that can be used for various static analyses. In the subsequent sections, we give a brief overview of them.

3.7.1 Non-Relational Abstract Domains

The non-relational abstract domains approximate the set of values of a given program variable at each program point. In most cases, the abstract environment associated with each program point is typically represented as a map-lattice $\mathbf{Vars} \to \mathbf{D}$ from the set of program variables **Vars** to the domain **D** (*value domain*) that abstracts values of individual variables.

Intervals [64, 65] represent constraints on the upper or lower bound of a single variable, e.g. $(k_1 < x < k_2)$. Interval analysis is very popular due to its simplicity and efficiency: an interval abstraction for n variables requires $O(n)$ space, and all operations require $O(n)$ time in the worst case.

In traditional analysis, predicates are typically approximated by the boolean values true and false, where one typically means "certainly x" and the other means "x or don't know". *Kleene's three-valued logic* [147] adds a third distinct value to represent lack of knowledge. One can think of Kleene's three values as *false*, *true* and *unknown* ($\top$). Kleene's logic was completed in a *Belnap's four-valued logic* [18] by the addition of the fourth truth value, *overdefined*, denoted usually by $\bot$. Kleene's "unknown" value $\top$ represents absence of information about truth or falseness of a given predicate, while the Belnap's "overdefined" value $\top$ represents presence of conflicting information about a predicate. Belnap's 4 logical values can be represented by the interval abstraction as: $\bot = [\,]$, $false = [\mathsf{false}, \mathsf{false}]$, $true = [\mathsf{true}, \mathsf{true}]$, $\top = [\mathsf{false}, \mathsf{true}]$.

Sign abstract domain [61] approximates the sign $(+, 0, -)$ of a numeric expression. Its lattice varies in precision from a simple one containing incomparable elements $-, 0, +$ augmented with $\top$ and $\bot$, to more precise ones including combinations like non-negative, non-null, non-positive etc.

3.7.1.1 Classical Data Flow Analysis

The relationship between the classical data flow analysis and abstract interpretation of programs has been initially discussed in [65] using analysis of available expressions as an example, however it is applicable to many classical analyses. A related discussion that elaborates in details on constant propagation analysis can be found in [185].

Initialization abstract domain ensures that variables are initialized before they are read. Its value domain's lattice consists of only two elements $\bot, \top$.

A nontrivial expression in a program is *available* at a program point p, if whenever control reaches p, the value of the expression has previously been computed, and since the last computation of the expression, no argument of the expression has had its value changed. *Available expressions* [65, §5.3.2] abstract domain helps inferring all the available expressions in a program. Assuming $Expr_{\mathbb{P}}$ is the set of expressions occurring in program $\mathbb{P}$. Abstract environment for available expressions analysis is a reverse powerset lattice $\langle \wp(Expr_{\mathbb{P}}), \supseteq \rangle$ that can be represented as a map-lattice $Expr_{\mathbb{P}} \to \mathbb{B}$ with traditional *pointwise ordering* of booleans $\mathbb{B}$ ordered as $\mathbb{1} \sqsubseteq \mathbb{0}$.

An expression is *very busy* at program point p if no matter what path is taken from that program point, the expression must always be used before any of the variables occurring in it are redefined. Similar to available expressions, the abstract domain of *very busy expressions* is a reverse powerset lattice $\langle \wp(Expr_{\mathbb{P}}), \supseteq \rangle$ that can be represented as a map-lattice $Expr_{\mathbb{P}} \rightarrow \mathbb{B}$ with traditional pointwise ordering of booleans $\mathbb{B}$ ordered as $\mathbb{1} \sqsubseteq \mathbb{0}$. Unlike available expressions analysis, which is an example of forward analysis requiring forward system of equations, this analysis requires propagating information backwards through the program graph.

A variable is *live* at a program point p if its current value may be read during the remaining execution of the program. Liveness of variables is another global flow problem that requires backwards analysis. Abstract domain for *liveness* analysis is a powerset lattice $\langle \wp(Vars), \subseteq \rangle$ that can be represented as a map-lattice $\mathbf{Var} \rightarrow \mathbb{B}$ with ordering on booleans defined as $\mathbb{0} \sqsubseteq \mathbb{1}$.

Constant propagation is a classical analysis that tries to determine the values of expressions at compile time when possible. The *constants* abstract domain [185] used by this analysis is again a map-lattice $\mathbf{Vars} \rightarrow \mathbf{Constant}$, where the domain **Constant** contains incomparable values drawn from a given domain augmented with $\top$ and $\bot$. Mohnen describes them intuitively as "not constant due to conflict" and "not constant due to missing information".

3.7.2 Relational Symbolic Abstract Domains

The relational abstract domains approximate relations between program entities at possibly different program points.

A certain number of classical data flow analysis techniques are included in or generalized by the determination of *linear equality* relations among program variables. For example constant propagation can be understood as the discovery of very simple linear equality relations among variables such as $\{X = 1, Y = 5\}$. However the resolution of the more general problem of determining linear equality relations among variables allows the discovery of symbolic constants (such as X=N, Y=5*N+1). The same way common sub-expressions can be recognized which are not formally identical, but are semantically equivalent because of relationships among variables. Also, the loop invariant computations as well as loop induction variables (modified inside the loop by the same loop invariant quantity) can be determined on a basis, which is not purely syntactical. The problem of determining equality relationships between a linear combination of the variables of the program and a constant was solved by Karr [142]. His work allows for direct expression of a *linear equality* abstract domain.

The *polyhedron abstract domain* [70] (often called *convex polyhedra*) represents inequalities of the form $c_1x_1 + \cdots + c_nx_n \geq k$. Convex polyhedra are an efficient representation for conjunctions of linear inequality constraints. This abstraction is very popular due to its ability to express precise constraints. However, this precision comes with a very high complexity overhead.

The complexity of the general convex polyhedral domain has motivated the development of the *two-variables per linear inequality* abstract domain [227], which is capable of capturing linear inequality with at most 2 variables: $ax + by \leq c$. It tries to retain the expressiveness of linear inequalities with a lower complexity.

The *linear congruences* abstract domain [113, 180] captures linear combination of variables equal to given number modulo a fixed number.

Octagons abstract domain [178, 180, 182] is an efficient representation for a system of inequalities on the sum or difference of variable pairs, e.g. $(\pm x \pm y \leq k)$ and $(x \leq k)$. The implementation of octagons [179] is a based on difference bound matrices, a data structure used to represent constraints on differences of variables, as in $(x - y \leq k)$ and $(x \leq k)$. Efficiency is an advantage of this representation: the spatial cost for representing constraints on n variables is $O(n^2)$, while the temporal cost is between $O(n^2)$ and $O(n^3)$, depending on the operation.

The *octahedron* abstract domain [52] represents constraints of the form $(\pm x_j \pm \cdots \pm x_k \geq c)$, where x_i are numerical variables such that $x_i \geq 0$, while parameter c is a rational constant automatically discovered by the analysis.

The *decision tree* abstract domain [25, §6.2.4] relates boolean variables to expressions used to compute them so that proper inference can be made in conditions.

The abstract domain for *numerical stability* of floating-point variables [166] can be used to determine the stability of the numerical errors arising inside a loop in which floating-point computations are carried out. The analysis based on this domain allows one to determine which particular operation makes a loop unstable. A similar *roundoff error* abstract domain [112, 167] can be used to find the origin of roundoff errors in floating-point computations. Another relational abstract domains for the detection of *floating-point run-time errors* [181] uses linear forms with interval coefficients to take the non-linearity of rounding of IEEE 754-compliant floating-point numbers into account. The domain can be used to detect overflows or invalid operations.

The *ellipsoid* abstract domain [25, §6.2.3] abstracts expressions of the form $X^2 - aXY + bY^2 \leq k$ by $r(X, Y) = k$. These expressions are commonly found in applications dealing with device logic.

3.7.2.1 Temporal Abstract Domains

Several domains have been proposed to verify various temporal properties of programs. Abstract domain for termination analysis has been explored among others in [68] and in relationship to databases in [13]. A number of different abstract domains has been combined together to determine bounds on execution time of various parts of code [269].

Abstract interpretation has also been used to perform static analysis of *cache performance* [7]. It utilizes two abstract domains, each of the form $c^{\#} : L \to 2^M$ mapping cache lines to sets of memory blocks. One of the domains is then used to classify definite cache hits, while the other is used to classify definite misses.

4. OPEN MULTI-METHODS[1]

> Functions delay binding; data structures induce binding. Moral: Structure data late in the programming process.
>
> Alan J. Perlis

In this chapter, we present the rationale, design, implementation and evaluation of a C++ language extension called *open multi-methods*. It aims at providing support of multiple dispatch and open-class extensions in the language, both of which are used to address the expression problem (§2.6). The material presented here is based on published papers [208, 209]. The text has been reworked in part to unify terminology and mathematical definitions with the rest of this thesis, but mainly to clarify some details about the algorithm for dispatch table generation, which we have been asked about after the papers have been published.

This work was done in cooperation with Peter Pirkelbauer, Bjarne Stroustrup, and in very early stages – Nan Zhang, which is why it is important to give context to individual contributions of the authors. The work started as a term project for a class taught by Dr. Stroustrup in the Fall of 2005. Peter brought up the idea of looking into implementing multi-methods for C++ and was leading the project since. Most of the initial experimentation was done together through pair-programming, collaborative editing and continuous discussions, which allowed us to bounce the ideas back and forth quickly. While agreeing on the overall design, we often disagreed on various issues. Peter, for example, from the very beginning understood the importance of open-class extensions and was pushing for *open multi-methods*. I was favoring performance and control over extensibility of functions and was pushing for *closed multi-methods* (§4.6.3.1). Dr. Stroustrup's experience of designing other C++ features was always invaluable in making the right choices and breaking the ties. While we were modeling our dispatch semantics after overload resolution in C++, we observed that its type-safe implementation might require proliferation of dispatch tables per each call site seen. To avoid that, I suggested incorporating the covariance of return types into our ambiguity resolution mechanism, which allowed us to generate type safe, call site independent dispatch tables (§4.3.4). Dr. Stroustrup further suggested a policy for resolving late ambiguities, which are often unavoidable in the presence of dynamic linking (§4.4.1). While looking for alternative implementations, I also suggested mapping ideas of Gibbs and Stroustrup used to implement fast dynamic casting [107] to obtain an implementation of multiple dispatch based on Chinese reminders (§4.6.3.2). Even though we did pair programming while implementing the open multi-methods compiler, Peter was the most experienced with EDG front end and was the one behind the keyboard most of the time. While working on his dissertation, he further improved the implementation to deal with standard headers and real-world code. I later rejuvenated that implementation for current C++ compilers in order to be able to compare the performance of our open multi-methods extension with the performance of an open type switch (§5.6.6).

[1] Reprinted with permission from "Design and Evaluation of C++ Open Multi-Methods" by Peter Pirkelbauer, Yuriy Solodkyy, Bjarne Stroustrup, 2010. Science of Computer Programming, Volume 75, Issue 7, Pages 638-667, Copyright 2009 by Elsevier B.V.

4.1 Introduction

Runtime polymorphism is a fundamental concept of object-oriented programming (OOP), typically achieved by late binding of method invocations. "*Method*" is a common term for a function chosen through *runtime polymorphic dispatch*. Most object-oriented languages (e.g. C++ [244], Eiffel [171], Java [10], Simula [23], and Smalltalk [110]) use only a single parameter at runtime to determine the method to be invoked ("*single dispatch*"). This is a well-known problem for operations where the choice of a method depends on the types of two or more arguments ("*multiple dispatch*"). A technique called *double dispatch* was created to workaround this problem [128]. A separate problem is that dynamically dispatched functions have to be declared within class definitions. This is intrusive and often requires more foresight than class designers possess, complicating maintenance and limiting the extensibility of libraries. The visitor design pattern (§2.7) was created to address this problem. *Open multi-methods* provide an abstraction mechanism that solves both problems by separating operations from classes and enabling the choice of *dynamic* vs. *static* dispatch on a per-parameter basis.

When dynamically dispatched functions are declared within classes, such functions are often referred to as "*multi-methods*". When declared independently of the type on which they dispatch, such functions are often referred to as *open-class extensions*, *accessory functions* [272], *arbitrary multi-methods* [176], or "*open-methods*". Languages supporting multiple dispatch include CLOS [239], MultiJava [53, 175], Dylan [222], and Cecil [47]). We implemented and measured both multi-methods and open-methods. Since open-methods address a larger class of design problems than multi-methods and are not significantly more expensive in time or space, our discussion concentrates on open-methods.

Generalizing from single dispatch to open-methods raises the question how to resolve function invocations when no overrider provides an exact type match for the runtime-types of the arguments. *Symmetric dispatch* treats each argument alike but is subject to ambiguity conflicts. *Asymmetric dispatch* resolves conflicts by ordering the arguments based on some criteria (e.g., an argument list is considered left-to-right). Asymmetric dispatch semantics is simple and ambiguity free (if not necessarily unsurprising to the programmer), but it is not without criticism [46]. It differs radically from C++'s symmetric function overload resolution rules and does not catch ambiguities.

We derive our design goals for the open-method extension from the C++ design principles outlined in [243, §4]. For open-methods, this means the following: open-methods should address several specific problems, be more convenient to use than all workarounds (e.g. the visitor pattern), and outperform them in both time and space. They should neither prevent separate compilation of translation units nor increase the cost of ordinary virtual function calls. Open-methods should be orthogonal to exception handling in order to be considered suitable for hard real-time systems (e.g. [168]), and parallel to the virtual and overload resolution semantics.

In application to our running example of the expression language from §2.4, open multi-methods allow us to express non-intrusively the evaluator of an *exp*-term as following:

```
int eval(virtual const Expr&) { throw std::invalid_argument("eval"); }
int eval(const Value& x) { return x.value; }
int eval(const Plus&   x) { return eval(x.e1) + eval(x.e2); }
int eval(const Minus& x) { return eval(x.e1) − eval(x.e2); }
int eval(const Times& x) { return eval(x.e1) * eval(x.e2); }
int eval(const Divide& x) { return eval(x.e1) / eval(x.e2); }
```

Additionally, the function that compares two terms for equality can now be simply expressed as:

```
bool operator==(virtual const Expr&, virtual const Expr&) { return false; }
bool operator==(const Value& a, const Value& b) { return a.value  == b.value;  }
bool operator==(const Plus&  a,  const Plus&   b) { return a.e1==b.e1 && a.e2==b.e2); }
bool operator==(const Minus& a, const Minus& b) { return a.e1==b.e1 && a.e2==b.e2); }
bool operator==(const Times& a, const Times& b) { return a.e1==b.e1 && a.e2==b.e2); }
bool operator==(const Divide& a, const Divide& b) { return a.e1==b.e1 && a.e2==b.e2); }
```

4.1.1 Summary

The contributions of this chapter include:

- A design of open-methods that is consistent with C++ call-resolution semantics.
- An efficient implementation and performance data to support its practicality.
- A first known consideration of repeated and virtual inheritance for multi-methods.
- A novel idea of harnessing covariance of the return type for ambiguity resolution.
- A discussion of handling open-methods in the presence of dynamic linking.

4.2 Application Domains

Whether open-methods address a sufficient range of problems to be a worthwhile language extension is a popular question. We think they do, but like all style questions, it is not a question that can in general be settled without examples and data. This is why in the context of this chapter we start with presenting examples that we consider characteristic for larger classes of problems and that would benefit significantly. We then explain fundamentals that drive the design, provide the details of our implementation and compare its performance to alternative solutions. In what follows, we mark examples with 1 when they primarily demonstrate multiple dispatch and with 2 when they demonstrate open-class extensions.

4.2.1 Shape Intersection[1]

An intersect operation is a classic example of multi-method usage [243, §13.8]. For a hierarchy of shapes, intersect() decides if two shapes intersect. Handling all different combinations of shapes (including those added later by library users) can be quite a challenge. Worse, a programmer needs specific knowledge of a pair of shapes to use the most specific and efficient algorithm.

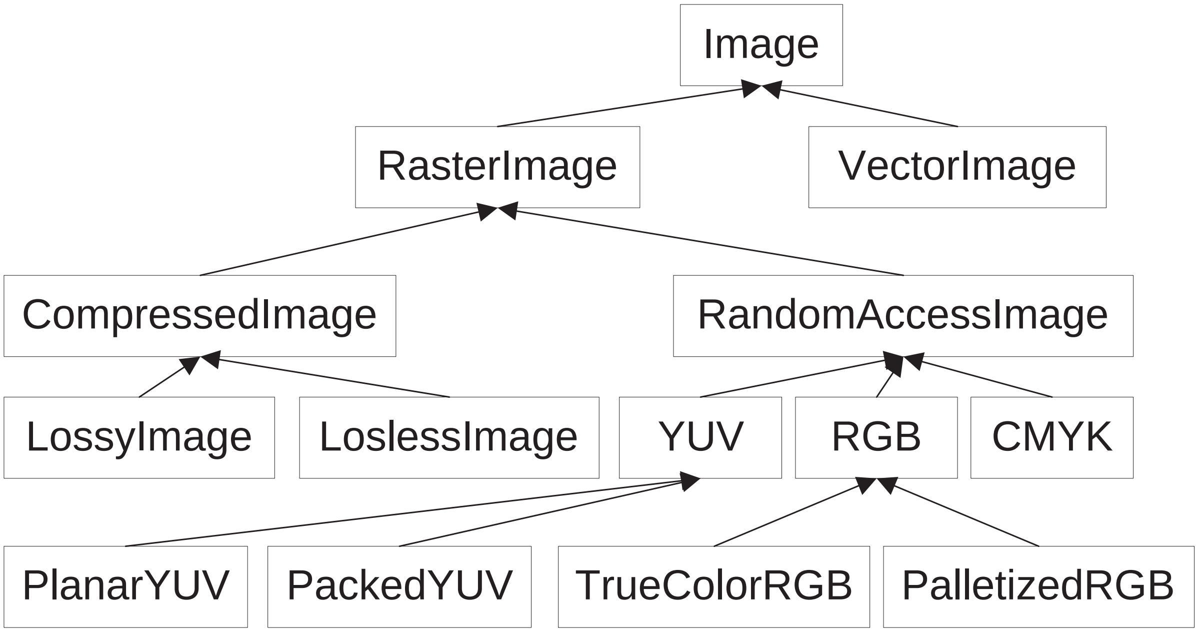
Using the multi-method syntax from [243, §13.8], with **virtual** indicating runtime dispatch, we can write:

```
bool intersect (virtual  const Shape&,      virtual  const Shape&); // open−method
bool intersect (virtual  const Rectangle&, virtual  const Circle &); // overrider
```

We note that for some shapes, such as rectangles and lines, the cost of double dispatch can exceed the cost of the intersect algorithm itself.

4.2.2 Data Format Conversion[1][2]

Consider an image format library, written for domains such as image processing or web browsing. Conversion between different representations is not among the core concerns of an image class and a designer of a format typically cannot know all other formats. Designing a class hierarchy that takes aspects like this into account is hard, particularly when these aspects depend on polymorphic behavior. In this case, generic handling of formats by converting them to and from a common representation in general gives unacceptable performance, degradation in image quality, loss of information, etc. An optimal conversion between different formats requires knowledge of exact source and destination types, therefore it is desirable to have open-class extensions in the language, like open-methods. Here is the top of a realistic image format hierarchy:



A host of concrete image formats such as RGB24, JPEG, and planar YUY2 will be represented by further derivations. The optimal conversion algorithm must be chosen based on a source-target pair of formats [132, 275]. In §4.8, we present an implementation of this example, here we simply demonstrate with a call:

```
bool convert(virtual const image& src, virtual image& dst);
RGB24 bmp("image.bmp");
JPEG jpeg;
convert(bmp,jpeg);
```

4.2.3 Type Conversions in Scripting Languages[12]

Similar to §4.2.2, this example demonstrates the benefits of open-methods in the context of type conversions. Languages used for scripting are often dynamically typed and a value may often be converted to other types depending on use. For example, variable x initialized as string can be used in contexts where integers or even dates are expected, while variable y initialized as integer can perfectly be used inside the catenation of strings. In such cases, an interpreter will try to convert actual values to the type required in the context according to some conversion rules. A typical implementation of such conversion will use either nested switch statements or a table of pointers to appropriate conversion routines. None of these approaches is extensible or easy to maintain. However, multi-methods provide a natural mechanism for such implementations:

```
class ScriptableObject  { virtual  ~ScriptableObject (); };
class String   : ScriptableObject  {};
class Number : ScriptableObject  {};
class Integer  : Number            {};
bool convert(virtual  const ScriptableObject & src,  virtual  ScriptableObject & tgt);
bool convert(virtual  const Number&           src ,  virtual  String &            tgt );
bool convert(virtual  const String &           src ,  virtual  Number&            tgt );
```

```
// ... etc.
```

4.2.4 Dealing with an AST[2]

We saw that dealing with abstract syntax trees in an object-oriented language is the subject of the expression problem (§2.6) and is typically addressed with a visitor design pattern (§2.7). Open-methods provide a non-intrusive alternative for dealing with ASTs, which we demonstrated in §4.1 on the example of the expression language from §2.4.

In §4.8, we further compare our experiences in implementing a compiler pass with open-methods and visitors.

4.2.5 Binary Method Problem[1]

Often times we have a two-argument method, whose meaning is trivial to define when both arguments are of the same type, but not so obvious in cases when arguments are of different, though related through inheritance types. Such methods are characteristic to many logical and arithmetic operations, and have been rigorously studied by Bruce et al. [36]. They define a *binary method* of some object of type τ as a method that has an argument of the same type τ. Binary methods pose a typing problem and among different solutions to the problem, authors propose to use multi-methods.

In the multi-methods setting, binary methods simply become multi-methods with two arguments of the same type. Consider for example an equality comparison of two objects:

```
class Point { double x, y; };
bool equal(const Point& a, const Point& b) { return a.x ==b.x && a.y ==b.y; }
class ColorPoint : Point { Color c; };
```

When a class ColorPoint derives from Point and adds a color property, the question arises on how equal should be defined: should it just compare the coordinates or should it also compare the color properties of both arguments? The second option is only viable if both arguments are of type ColorPoint. If the argument types differ, we can choose either to return false or to compare the coordinates only. Depending on the problem domain, both choices can be acceptable. Here we simply note that multi-methods are an ideal solution for the implementation of the latter policy where the comparison of Point with ColorPoint only compares coordinates:

```
bool equal(virtual const Point&      a, virtual const Point&      b);
bool equal(virtual const ColorPoint& a, virtual const ColorPoint& b);
```

4.2.6 Selection of an Optimal Algorithm Based on Dynamic Properties of Objects[1]

Often, we can use dynamic types to choose a better algorithm for an operation than would be possible using only static information. Using open-methods, we can use the dynamic type information to select more efficient algorithms at runtime without added complexity or particular foresight. Consider a matrix library providing algorithms optimized for matrices with specific dynamic properties. Storing these dynamic properties as object attributes is not easily extensible and is error prone in practice. Letting the compiler track them using open-methods for dispatch (run-time algorithm selection) is simpler. For instance, the result of $A{*}A^T$ is a symmetric matrix – if such a matrix appears somewhere in computations, we may consider a broader set of algorithms when the result is used in other computations.

```
class Matrix { virtual ~Matrix(); };
class Symatrix  : Matrix   {}; // symmetric matrix
class DiagMatrix : Symatrix {}; // diagonal matrix
```

```
Matrix&     operator+(virtual const Matrix&    , virtual  const Matrix&    );
Symatrix& operator+(virtual const Symatrix& , virtual  const Symatrix& );
DiagMatrix& operator+(virtual const DiagMatrix& , virtual const DiagMatrix&);
```

Depending on the runtime type of the arguments, the most specific addition algorithm is selected at runtime and the most specific result type returned. The static result type would still be Matrix& when the static type of an argument is a Matrix& since we cannot draw a more precise conclusion about the dynamic type of the result (see §4.3.4 and §4.4.2 for details). However, since the operator is selected according to the dynamic type, the optimal algorithm will be used for the result when it is part of a larger expression.

Other interesting properties to exploit include whether the matrix is upper/lower triangular, diagonal, unitary, non-singular, or symmetric/Hermitian positive definite. Physical representations of those matrices may also take advantage of the knowledge about the structure of a particular matrix and use less space for storing the matrix.

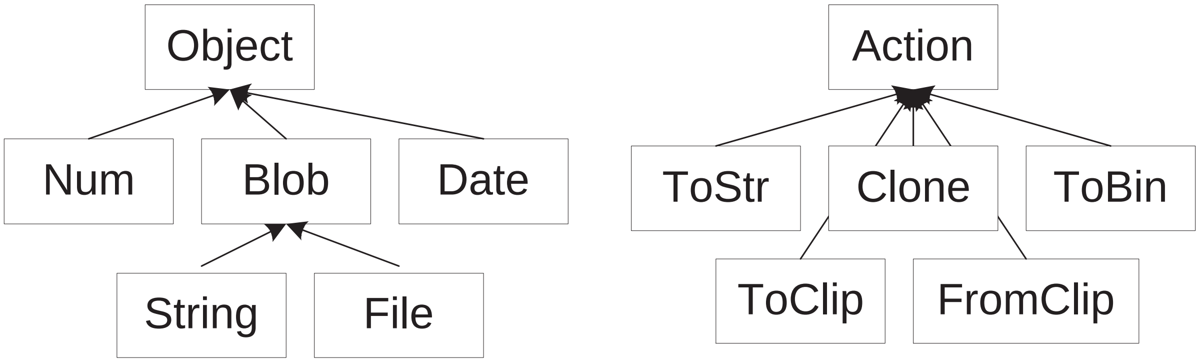
The polymorphic nature of the multiple dispatch requires the result to be returned by either reference or pointer to avoid slicing. Since the reference must refer to a dynamically allocated object, this creates a lifetime problem for that object. Common approaches to such problems include relying on a garbage collector and using a proxy to manage lifetime. An efficient proxy is easy to write:

```
// A memory−managing proxy class (note lowercase name).
class matrix
{
    // pointer to the actual polymorphic Matrix
    std::unique_ptr⟨Matrix⟩ the_matrix;

    matrix(Matrix& actual) : the_matrix(&actual) {}
};

matrix operator+(const matrix& a, const matrix& b)
{
    // Forward operator+ to the actual open−method and attach the result to the proxy.
    return matrix(∗a.the_matrix + ∗b.the_matrix);
}
```

The unique_ptr is a simple and efficient (not-reference counting) “smart” pointer that is part of the C++0x standard [17] and has been widely available and used for years [16].

4.2.7 Action Systems[1]

Dynamically typed languages, such as Smalltalk [110], Ruby [250] or Python [216], can dispatch polymorphic calls on classes not bound by inheritance. As long as a method with a given name exists in the class, it will be called, otherwise an exceptional action is taken. Such *duck typing* semantics is often desirable in statically-typed object-oriented languages (possibly with restriction of objects being derived from a certain base class). To achieve this, we may represent methods (here called actions) as objects and then apply a given action to a given set of parameters.



The above figure shows a class hierarchy with objects and actions. Note that the same action applied to a different object may have a completely different meaning.

```
Object& execute(virtual const      Action& act, virtual  Object& obj);
String & execute(virtual const   ToString& act, virtual  Number& obj);
  File & execute(virtual const SaveToFile& act, virtual    Blob& obj);
// ... etc.
```

Action objects resemble function objects in C++ in a way. The main difference between actions and C++ function objects is that actions participate in call dispatching on equal bases with other arguments, while function objects invariably define the scope for call dispatching. Simply put, this means that in case of action objects, other arguments of a call can affect the choice of a call's target at runtime (symmetric behavior), while with function objects they cannot (asymmetric behavior).

4.2.8 Extending Classes with Operations[2]

Once defined, the object-oriented way to extend a class's functionality is to derive a new class and introduce the new behavior there. However, this technique only succeeds if the programmer has control over the source code that instantiates objects (for example, if the code has been designed to use a factory). Consider a system framework that responds to various events. The events may require logging in different logs and formats. While it is feasible to provide a common interface for different kinds of logs, it is rather difficult to foresee all possible formats in which logging can be done. Open-methods eliminate the need to modify class declarations directly, and improve the support for separation of concerns.

```
struct Log            {}; // Interface to different logs
struct FileLog  : Log {}; // Logs to File
struct EventLog : Log {}; // Logs to OS event log
struct DebugLog : Log {}; // Logs to debug output

struct Event { virtual ~Event(); }; // Interface for various types of events
struct Access     : Event  {};
struct FileAccess : Access {};
struct DirAccess  : Access {};
struct DBAccess   : Access {};

// Specializations of how to log various types of events in text format
void log_as_text   (virtual    Access& evt, Log& log, int priority );
void log_as_text   (virtual DirAccess& evt, Log& log, int priority );

// Specializations of how to log various types of events in XML format
void log_as_xml    (virtual    Access& evt, Log& log, int priority );
void log_as_xml    (virtual DirAccess& evt, Log& log, int priority );

// Specializations of how to log various types of events in binary format
void log_as_binary (virtual    Access& evt, Log& log, int priority );
void log_as_binary (virtual DirAccess& evt, Log& log, int priority );
// etc. for any other formats that may be required in the future.
```

4.2.9 Open-Methods Programming

The benefits noted in the examples stem from fundamental advantages offered by open-methods. They allow us to approximate widely accepted programming principles better than more conventional language constructs and allow us to see conventional solutions, such as visitors, as workaround techniques. For examples like the ones presented above, open-methods simply express the design more directly. From a programmer's point of view open-methods

- *are non-intrusive*: We can provide runtime dispatch (a virtual function) for objects of a class without modifying the definition of that class or its derived classes. Using open-methods implies less nonessential coupling than conventional alternatives.
- *provide order independence for arguments*: The rules for an argument are independent of whether it is the first, second, third, or whatever argument. The first argument of a member function (the this pointer) is special only in that it has notational support. The dynamic/static resolution choice has become independent of the choice of argument order.
- *improve ambiguity detection*: Ambiguous calls are detected in a way that cannot be done with only a per-argument check (as for conventional multiple dynamic dispatch) or only a per-translation-unit check (as for conventional static checking). Using open-methods there is simply more information for the compiler and linker to use.
- *provide multiple dynamic dispatch*: We can directly select a function based on multiple dynamic types; no workarounds are required.
- *improve performance*: faster than workarounds when more than one dynamic type is involved with no memory overhead compared to popular workaround techniques (see §4.7 and §5.6.6).

From a language design point of view, open-methods make the rules for overriding and overloading more orthogonal. This simplifies language learning, programming, reasoning about programs, and maintenance.

Potential objections to the use of open-methods include that:

- the set of operations on objects of a class are not defined within the class. However, that is true as soon as you allow any freestanding function and is essential for conventional mathematical notation and programming styles based on that. Information hiding is not affected.
- the first argument is not fundamentally different so open-methods do not obey the "send a message to an object" ("object-oriented") model of programming. We consider that model unrealistically restrictive for many application domains, such as classic math [243].
- the set of overriders for a virtual function is not found within a specific set of classes (the set of classes derived from the class that introduced the virtual function). On the other hand, we never have to define a new derived class just to be able to override.
- open-methods are open; that is, they do not provide a closed set of overloading candidates for a given function name. However, we consider that a good feature in that it allows for non-intrusive extension. For C++, the decision not to syntactically distinguish overriders or overloaded functions was taken in 1983 and cannot be changed now [243, §11.2.4].

Obviously, we consider open-methods a significant net gain compared to alternatives, but the final proof (as far as proofs are possible when it comes to the value of programming language features) will have to wait for the application of open-methods in several large real-world programs.

4.3 Definition of Open Methods

An *open-method* is a family of dynamically dispatched functions, where the callee depends on the dynamic type of one or more arguments. ISO C++ supports compile-time (static) function overloading on an arbitrary number of arguments and runtime (dynamic) dispatch on a single argument. The two mechanisms are orthogonal and complementary. We define open-methods to generalize both, so our language extension must unify their semantics. Our dynamic call resolution mechanism is modeled after the overload resolution rules of C++. The ideal is to give the same result as static resolution would have given had we known all types at compile time. To achieve this, we treat the set of overriders as a *viable set* of functions and choose the single most specific method for the actual combination of types.

We derive our terminology from virtual functions: a function declared virtual in a base class (super class) can be overridden in a derived class (sub class):

Definition 1. *An* open-method *is a freestanding function with one or more parameters declared virtual.*

Definition 2. *An open-method f_2* overrides *an open-method f_1 if it has the same name, the same number of parameters, covariant virtual parameter types, invariant non-virtual parameter types, and a possibly covariant return type. In such case, we say that f_2 is an* overrider *of f_1.*

Definition 3. *An open-method that does not override another open-method is called a* base-method*.*

Definition 4. *A base-method and all the open-methods that override it form an* open-method family*.*

While this is not strictly necessary, for practical reasons we require that a base-method should be declared before any of its overriders. This parallels other C++ rules and greatly simplifies compilation. This restriction does not prevent us from declaring different overriders in different translation units. For every overrider and base-method pair, the compiler checks if the exception specifications and covariant return type (if present) comply with the semantics defined for virtual functions.

Definition 5. *A* dispatch table *(DT) maps the type-tuple of the base-method's virtual parameters to actual overriders that will be called for that type-tuple.*

The following example demonstrates a simple class hierarchy and an open-method defined on it:

```
struct A { virtual ~A(); } a;
struct B : A {} b;

void print(virtual A&, virtual A&); // (1) base–method
void print(virtual B&, virtual A&); // (2) overrider
void print(virtual B&, virtual B&); // (3) overrider
```

Here, both (2) and (3) are overriders of (1), allowing us to resolve calls involving every combination of A's and B's. For example, a call `print(a,b)` will involve a conversion of `b` to an `A&` and invoke (1). This is exactly what both static overload resolution and double dispatch would have done.

To introduce the role of multiple inheritance, we can add to that example:

```
struct X { virtual ~X(); };
struct Y : X, A {};

void print(virtual X&, virtual X&); // (4) base–method
void print(virtual Y&, virtual Y&); // (5) overrider
```

Here (4) defines a new open-method print on the class hierarchy rooted in X. Y inherits from both A and X, and according to our definition (5) overrides both (4) and (1).

We note that whether it would be better to require an overrider to be explicitly specified as such is an orthogonal decision. Here we simply follow the C++ tradition set up by virtual functions to do this implicitly.

4.3.1 Type Checking and Call Resolution of Open-Methods

Type checking and resolving calls to open-methods involves three stages: compile time, link time, and runtime.

- *Overload resolution at compile time*: the goal of overload resolution is to find a unique open-method in the overload set visible at the call site, through which the call can be (but not necessarily will be) dispatched. The open-method determines the necessary casts of the arguments, and the return type expected at the call site.
- *Ambiguity resolution at link time*: the pre-linker aggregates all overriders of a given open-method family, checks them for return type consistency, performs ambiguity resolution, and builds the dispatch tables.
- *Dynamic dispatch at runtime*: the dispatch mechanism looks up the entry in the dispatch table that contains the most specific overrider for the dynamic types of the arguments and invokes that overrider.

This three-stage approach parallels the resolution to the equivalent modular-checking problem for template calls using concepts in C++0x [114]. Further, the use of open-methods (as opposed to ordinary virtual functions and multi-methods) can be seen as adding a runtime dimension to generic programming [12].

4.3.2 Overload Resolution

The purpose of *overload resolution* in the context of open multi-methods is to identify an open-method that the compiler will use for type checking and inferring the result type expected from the call. In general, the C++ overload resolution rules [129] remain unchanged: the *viable set* includes both open-methods and regular functions and the compiler treats them equally. Once a unique best match is found, the call can be type checked against it. For the dispatch, any of its base-methods can be chosen. Which one is selected is irrelevant, as any further overrider would likewise override all base-methods.

Consider the following example:

```
struct X { virtual ~X(); };
struct Y { virtual ~Y(); };
struct Z { virtual ~Z(); };

void foo(virtual X&, virtual Y&); // (1) base–method
void foo(virtual Y&, virtual Y&); // (2) base–method
void foo(virtual Y&, virtual Z&); // (3) base–method

struct XY : X, Y {} xy;
struct YZ : Y, Z {} yz;

void foo(virtual XY&, virtual Y&); // (4) overrides 1 and 2
void foo(virtual Y&, virtual YZ&); // (5) overrides 2 and 3
```

A call foo(xy,yz) is ambiguous according to the standard overload resolution rules as overriders 4 and 5 are equally good matches. To resolve this ambiguity, a user may explicitly cast some or all of the arguments to make the call unambiguous according to the overload resolution rules: e.g. calling foo(xy,**static_cast**⟨Y&⟩(yz)) will uniquely select 4 as a base-method for the call. Alternatively, a user may introduce a new overrider **void** foo(**virtual** XY&, **virtual** YZ&), which will become a unique best match for the call.

4.3.3 Ambiguity Resolution

Once we are in the ambiguity resolution phase done by the prelinker, we assume that the overload resolution phase has selected a unique best match for type checking of each open-method call site (otherwise, it would have reported a compile-time error). At this phase, we have information about all available overriders of a particular open-method family, and we only report ambiguities that prevent us from building a complete dispatch table.

Our ideal for resolving open-method calls combines the ideals for virtual functions and overloading:

- virtual functions: the same function is called regardless of the static types of the arguments at the call site.
- *overloading*: a call is considered unambiguous if (and only if) every parameter is at least as good a match for the actual argument as the equivalent parameter of every other candidate function and that it has at least one parameter that is a better match than the equivalent parameter of every other candidate function.

This implies that a call of a single-argument open-method is resolved equivalently to a virtual function call, and the rules we describe do closely approximate this ideal. As mentioned, the static resolution is done exactly according to the usual C++ rules. The dynamic resolution is presented as the algorithm for generating dispatch tables in §4.3.5. Before looking at that algorithm, we present some key motivating examples.

4.3.3.1 Single Inheritance

In object models supporting single inheritance, ambiguities can only occur with open-methods taking at least two virtual parameters. Such ambiguities can only be introduced by new overriders, not by extending the class hierarchy. They can be resolved by introducing a new overrider. Open-methods with one dynamic argument are identical to virtual functions and are always ambiguity free. Thus, open-methods provide an unsurprising mechanism for expressing non-intrusive ("external") polymorphism. This eliminates the need to complicate a class hierarchy just to support the later addition of additional "methods" in the form of visitors.

4.3.3.2 Repeated Inheritance

Consider the repeated inheritance case shown in Figure 2.1 together with the following set of open-methods visible at a call site to foo(d1,d2), where d1 and d2 are of type D&:

```
void foo(virtual A&, virtual A&);
void foo(virtual B&, virtual B&);
void foo(virtual B&, virtual C&);
void foo(virtual C&, virtual B&);
void foo(virtual C&, virtual C&);
```

Even though, overriders for all possible combinations of B and C (the base classes of D) are declared, the call with two arguments of type D is rejected at compile time. The problem in this case, is that foo(A&,A&) will be chosen by overload resolution as the *base-method* for the call, and both arguments will have to be implicitly cast to A& before dispatch. Unfortunately, there are multiple subobjects of type A inside D, which makes an implicit conversion from D to A ambiguous (see §2.3.1).

To resolve that conflict, a user can either add an overrider foo(D&,D&) visible at the call site or explicitly cast arguments to either the B or C subobject. Making an overrider for foo(D&,D&) available at the call site eliminates the need to choose a subobject as it guarantees that independently of the *effective subobject*, the call would always be dispatched to the same overrider.

If the conflict is resolved by casting, a question remains on how the linker should resolve a call with two arguments of the dynamic type D? Because of the casting, only one subobject (either B or C) will be effective at run-time on each argument. Thus, we can fill our dispatch table for each effective subobject independently. With it, for each combination of types, there is a unique *best match* according to the usual C++ rules. Here is the dispatch table built for this case, where we mark the positions of actually provided overriders in bold font:

	A	B	C	D::B	D::C
A	$\boldsymbol{\langle A, A\rangle}$	$\langle A, A\rangle$	$\langle A, A\rangle$	$\langle A, A\rangle$	$\langle A, A\rangle$
B	$\langle A, A\rangle$	$\boldsymbol{\langle B, B\rangle}$	$\boldsymbol{\langle B, C\rangle}$	$\langle B, B\rangle$	$\langle B, C\rangle$
C	$\langle A, A\rangle$	$\boldsymbol{\langle C, B\rangle}$	$\boldsymbol{\langle C, C\rangle}$	$\langle C, B\rangle$	$\langle C, C\rangle$
D::B	$\langle A, A\rangle$	$\langle B, B\rangle$	$\langle B, C\rangle$	$\langle B, B\rangle$	$\langle B, C\rangle$
D::C	$\langle A, A\rangle$	$\langle C, B\rangle$	$\langle C, C\rangle$	$\langle C, B\rangle$	$\langle C, C\rangle$

Since the base-method is foo(A&,A&) and the arguments will have to be cast to A before dispatch, technically, we should have listed D::B::A and D::C::A subobjects in the dispatch table. We mention the subobjects D::B and D::C instead, since they are the largest subobjects of D for which the choice of A subobject is *unambiguous.* This resolution exactly matches our ideals.

Analog to single inheritance, extending a class hierarchy using repeated inheritance cannot introduce ambiguities. *Ambiguous* subobjects are determined at compile time and reported as errors.

4.3.3.3 Virtual Inheritance

Consider now the virtual inheritance class hierarchy shown in Figure 2.1 together with the set of open-methods from §4.3.3.2: In contrast to repeated inheritance, a D now has only one A subobject, shared by B, C, and D. This causes a problem for calls requiring conversions, such as foo(b,d); is that D to be considered a B or a C? There is not enough information to resolve such a call. Note that the problem can arise in such a way that we cannot catch it at compile time, because D's definition could be in a different translation unit:

```
C& rc = d;
foo(b,rc); // Actually called as: foo((A&)b,(A&)rc);
B& rb = d;
foo(b,rb); // Actually called as: foo((A&)b,(A&)rb);
```

Using static type information to resolve either call would violate the fundamental rule for virtual function calls: use runtime type information to ensure that the same overrider is called from every point of a class hierarchy. At runtime, the dispatch mechanism will (only) know that we are calling foo with a B and a D. It is not known whether (or when) to consider that D a B or a C. Remember that in all these cases foo(A&,A&) is still the base-method for the call and thus the arguments are cast to A&, which is, under the virtual inheritance, a *shared subobject.* Based on this reasoning (embodied in the algorithm in §4.3.5) we must generate this dispatch table:

	A	B	C	D::A
A	$\boldsymbol{\langle A, A\rangle}$	$\langle A, A\rangle$	$\langle A, A\rangle$	$\langle A, A\rangle$
B	$\langle A, A\rangle$	$\boldsymbol{\langle B, B\rangle}$	$\boldsymbol{\langle B, C\rangle}$	$\langle ?, ?\rangle$
C	$\langle A, A\rangle$	$\boldsymbol{\langle C, B\rangle}$	$\boldsymbol{\langle C, C\rangle}$	$\langle ?, ?\rangle$
D::A	$\langle A, A\rangle$	$\langle ?, ?\rangle$	$\langle ?, ?\rangle$	$\langle ?, ?\rangle$

We cannot detect the ambiguities marked with $\langle ?, ?\rangle$ at compile time, but we can catch them at link time when the entire set of classes and overriders is known.

4.3.4 Covariant Return Types

Covariance of the return type in virtual functions refers to the possibility of having a different result type in the overrider than the one mentioned in the original virtual function. For type safety, the result type of the overrider is required to be covariant (i.e. more specialized) to the result type of the original virtual function.

Covariant return types are a useful element of C++. If anything, they appear to be more useful for operations with multiple arguments than for single argument functions. Presence of covariant return types complicates the use of workaround techniques (§4.8.2).

As an example of relevance of covariant return type use, consider our matrix example from §4.2.6. We know that the sum of two symmetric matrices necessarily results in a symmetric matrix, while the sum of two diagonal matrices is necessarily diagonal. We thus would like to express this fact through the type system:

```
Matrix&     operator+(virtual Matrix&     , virtual  Matrix&    );
Symatrix& operator+(virtual Symatrix& ,  virtual  Symatrix& );
DiagMatrix& operator+(virtual DiagMatrix& , virtual DiagMatrix&);
```

In single dispatch, *covariance* of a return type means change in the same direction as the change of the receiver's type: from less to more specialized. Consequently, covariance of return type for open-methods means change in the same direction as the change of argument types between base-method (*bm*) and its overrider (*or*). Liskov's substitution principle [155] guarantees that any call type-checked based on *bm* can use *or*'s covariant result without compromising type safety (§2.2).

In single dispatch, covariant return types can be implemented by adding a new virtual function in the derived class that introduces covariant return. In such scenario, calls type-checked against the base class are dispatched through thunks, which will upcast the covariant return type to the expected base class. Calls type-checked against the derived class will be dispatched through the new v-table entry. The same idea is applicable to open-methods, where overriders introducing a covariant return type, also constitute a new base-method.

In our matrix example, the return type of each overrider changes covariantly with respect to the changes in the argument types, which validates the necessity of generalizing the covariant return rules for open-methods. Doing so turns out to be useful because covariant return types help eliminate what would otherwise have been ambiguities.

Consider the class hierarchies $A \leftarrow B \leftarrow D$ and $R1 \leftarrow R2 \leftarrow R3$ together with this set of open-methods:

```
R1* foo(virtual A&, virtual A&);
R2* foo(virtual A&, virtual B&);
R3* foo(virtual B&, virtual A&);
```

A call foo(b,b) appears to be ambiguous and the rules outlined so far would indeed make it an error. However, choosing R2*foo(A&,B&) would throw away information compared to using R3*foo(B&,A&):

an R3 can be used wherever an R2 can, but R2 cannot be used wherever an R3 can. Therefore, we prefer a function with a more derived return type and for this example get the following dispatch table:

	A	B	D
A	$\boldsymbol{\langle A, A\rangle}^{R1}$	$\boldsymbol{\langle A, B\rangle}^{R2}$	$\langle A, B\rangle^{R2}$
B	$\boldsymbol{\langle B, A\rangle}^{R3}$	$\langle B, A\rangle^{R3}$	$\langle B, A\rangle^{R3}$
D	$\langle B, A\rangle^{R3}$	$\langle B, A\rangle^{R3}$	$\langle B, A\rangle^{R3}$

At first glance, this may look useful, but ad hoc. However, a closer look reveals that one of the choices is simply not type safe: a call to foo(b,b), type-checked against R3*foo(B&,A&) at compile time, would expect a pointer to an object of type R3 (or any of its sub-classes) returned, which R2 is not. This is why R2*foo(A&,B&) cannot be used for dispatching such a call. On the other hand, the same call type-checked against R2*foo(A&,B&) elsewhere is expecting a pointer to R2 (or any of its sub-classes) returned from the call, and hence would readily accept R3. This is why selecting R3*foo(B&,A&) is the only viable choice here, which consequently resolves the ambiguity.

From a pure implementational point of view, an open-method with a return type that differs from its base-method becomes a new base-method and requires its own dispatch table (or equivalent implementation technique). The fundamental reason is the need to adjust a this-pointer in the return type in calls. Obviously, the resolutions for this new base-method must be consistent with the resolution for its base-method (or we violate the fundamental rule for virtual functions). However, since R2*foo(A&,B&) will not be part of R3*foo(B&,A&)'s dispatch table, the only consistent resolution is the one we chose.

If the return types of two overriders are siblings, then there is an ambiguity in the type-tuple that is a meet of the parameter-type tuples. Consider for example that $R3$ derives directly from $R1$ instead of $R2$, then none of the existing overriders can be used for $\langle B, B\rangle$ tuple as its return type on one hand has to be a subtype of $R2$ and on the other a subtype of $R3$. To resolve this ambiguity, the user will have to provide explicitly an overrider for $\langle B, B\rangle$, which must have the return type derived from both $R2$ and $R3$.

Using the covariant return type for ambiguity resolution also allows the programmer to specify preference of one overrider over another when asymmetric dispatch semantics is desired. The support of covariant return types thus not only improves static type information, but also enhances our ambiguity resolution mechanism. We are unaware of any other multi-method proposal using a similar technique.

4.3.5 Algorithm for Dispatch Table Generation

During generation of dispatch table, we have to deal with both classes and subobjects of those classes. While in general we use capital letters to indicate sets, for consistency with examples we will use variables A, C, R to range over classes, indicating the variables referring to sets of classes explicitly. Variable A will typically refer to classes representing the *static type* of an object, while variable C will refer to its *dynamic type*. We use variables x, y to range over subobjects of these classes and X, Y to range over sets of such subobjects.

Given a program $\mathbb{P}$ and a subclassing relation $<:$ defined on its classes (§2.3), consider a multi-method $Rf(A_1, \cdots, A_k)$ with k virtual arguments. Class A_i is the root of the class hierarchy of the i^{th} argument. The set $F^f = \{R_j f_j(C_{j1}, \cdots, C_{jk}) \mid j \in [0, m-1]\}$ is the set of actual overriders the user provided, where we assume $f_0 \equiv f$ to be the base-method.

$A_i^{<:} = \{C \mid C <: A_i\}$ is the class hierarchy of the i^{th} argument – the set of classes in subclassing relation with A_i. $X_i = \{x \mid x \in C \prec \sigma \succ A_i\}$ is a set of all subobjects of static type A_i in the class hierarchy $A_i^{<:}$

of the i^{th} argument. In the simple case of single inheritance, X_i will only contain a single subobject per each class C derived from A_i, however, in the case of multiple inheritance it may contain several different subobjects of the same static type A_i per each class C derived from A_i. Elements of X_i thus identify not only the dynamic type C of an object of static type A_i, but also a particular derivation path from C to A_i and whether it involved repeated or virtual inheritance (see §2.3.1 for details). We assume presence of trivial functions $\zeta_i : X_i \to A_i^{<:}$ that take a subobject and return an object type it belongs to: i.e. $\zeta_i(C \prec \sigma \succ A_i) = C$.

For a given multi-method f, we distinguish an *argument-type tuple* from an *argument tuple*.

An *argument-type tuple* is an element $\overline{c}$ of the Carthesian product $A^f = A_1^{<:} \times \cdots \times A_k^{<:}$. Subset $O^f = \{\langle C_{j1}, \cdots, C_{jk}\rangle \mid j \in [0, m-1]\} \subseteq A^f$ is the set of argument-type tuples of the overriders $f_j \in F^f$. Bijection $\phi : O^f \leftrightarrow F^f$ maps argument tuples of overriders to overriders themselves, while function $\rho : O^f \to R^{<:}$ maps an argument tuple to result type returned by the corresponding overrider.

An *argument tuple* is an element $\overline{x}$ of the Carthesian product $X^f = X_1 \times \cdots \times X_k$ – the set of all possible argument tuples of f. Set $Y^f \subseteq X^f$ is the set of argument tuples, on which the user defined overriders f_j for f. There exists an obvious surjection $\zeta : Y^f \to O^f$ where given argument tuple is mapped to an argument-type tuple by keeping only the dynamic type of subobject: $\zeta(\langle y_1, \cdots, y_k\rangle) = \langle \zeta_1(y_1), \cdots, \zeta_k(y_k)\rangle$. We have $|Y^f| \geq |O^f| = |F^f| = m$ because $Y^f \xrightarrow{\zeta} O^f \overset{\phi}{\leftrightarrow} F^f$.

Intuitively, X^f is the set of all cells in a dispatch table as seen in examples leading to this section, while Y^f is the subset of cells that were marked in bold – argument combinations on which the user defined overriders. Note that since ζ is a surjection, a single overrider's argument-type tuple may have been an image of several argument tuples, for example due to repeated multiple inheritance. We demonstrate with an example for multi-method on repeated inheritance discussed in §4.3.3.2:

$$\begin{aligned}
\mathsf{A}^{<:} &= \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\} \\
\mathsf{A}^{\mathsf{foo}} &= \mathsf{A}^{<:} \times \mathsf{A}^{<:} \\
O^{\mathsf{foo}} &= \{\langle \mathsf{A}, \mathsf{A}\rangle, \langle \mathsf{B}, \mathsf{B}\rangle, \langle \mathsf{B}, \mathsf{C}\rangle, \langle \mathsf{C}, \mathsf{B}\rangle, \langle \mathsf{C}, \mathsf{C}\rangle\} \\
X_{1,2} &= \{\mathsf{A}, \mathsf{B::A}, \mathsf{C::A}, \mathsf{D::B::A}, \mathsf{D::C::A}\} \\
X^{\mathsf{foo}} &= X_1 \times X_2 \\
Y^{\mathsf{foo}} &= \{\langle \mathsf{A}, \mathsf{A}\rangle, \langle \mathsf{B::A}, \mathsf{B::A}\rangle, \langle \mathsf{B::A}, \mathsf{C::A}\rangle, \langle \mathsf{C::A}, \mathsf{B::A}\rangle, \langle \mathsf{C::A}, \mathsf{C::A}\rangle\}
\end{aligned}$$

Should the user have also provided an overrider foo(D&,D&), the set Y^{foo} would have been extended with four tuples corresponding to combinations of subobjects D::B::A and D::C::A of D in both parameters. Note that for brevity in the examples we were only mentioning the largest subobjects that uniquely identify the subobject of static type A. This means that C in column or row header should have been spelled as C::A etc.

We now define a function β that returns the *base subobject* in a given object:

$$\begin{aligned}
\beta(\sigma_\perp) &= \sigma_\perp \\
\beta(\mathsf{A} \prec (\mathsf{Repeated}, \mathsf{A}{::}\epsilon) \succ \mathsf{A}) &= \sigma_\perp \\
\beta(\mathsf{C} \prec (h, \mathsf{C}{::}\mathsf{B}{::}l) \succ \mathsf{A}) &= \mathsf{B} \prec (\mathsf{Repeated}, \mathsf{B}{::}l) \succ \mathsf{A}
\end{aligned}$$

The first line tells that base subobject of an *invalid subobject* is an invalid subobject. The second line indicates that the base subobject of a complete object is also an invalid subobject. Note that A in this case may have had base classes, however we are not considering those as we only look at A subobject of a complete object A. The third line tells that independently of whether we are looking at shared or *repeated subobject* of an object C, the result is going to be a repeated subobject of an object B, which is a direct base class of C.

As an example, consider subobjects of static type A in the class hierarchy from Figure 2.1. In case of repeated inheritance we get: $\beta(\mathsf{D::B::A}) = \mathsf{B::A}$, $\beta(\mathsf{D::C::A}) = \mathsf{C::A}$, $\beta(\mathsf{C::A}) = \mathsf{A}$, $\beta(\mathsf{B::A}) = \mathsf{A}$, $\beta(\mathsf{A}) = \sigma_\perp$, while in case of virtual inheritance we get: $\beta(\mathsf{D::A}) = \mathsf{A}$, $\beta(\mathsf{B::A}) = \mathsf{A}$, $\beta(\mathsf{C::A}) = \mathsf{A}$, $\beta(\mathsf{A}) = \sigma_\perp$.

Recollecting the definitions from §2.3.2, we can now more formally state that a subobject x *directly extends* subobject y (written as $x \lessdot_1 y$) when $\beta(x) = y \wedge y \neq \sigma_\perp$. Reflexive transitive closure $\lessdot \;\equiv\; \lessdot_1^*$ of *directly extends* relation is called *extends* relation. While extends relation mimics subclassing relation on the level of subobjects, it should not be confused with *subobject containment* relation [268, §3.4]. *Extends* relates subobjects of the same static type within *different* object types, while *containment* relates subobjects of different static types within *the same* object type. For example: $\mathsf{D::B::A} \lessdot \mathsf{B::A}$ (extends), but $\mathsf{D::B::A} \sqsubseteq \mathsf{D::B}$ (containment). Informally, direct extension drops the beginning of the subobject path, while direct containment drops its end.

For convenience, we also define β_i on argument tuples:

$$\beta_i(\langle x_1, \cdots, x_k \rangle) \;\equiv\; \langle x_1, \cdots, \beta(x_i), \cdots, x_k \rangle, \beta(x_i) \neq \sigma_\perp.$$

With it, we define function $\boldsymbol{\beta}$ that maps an argument tuple to a set of immediate predecessor tuples:

$$\boldsymbol{\beta}(\overline{x}) \;\equiv\; \{\beta_i(\overline{x}) \mid \overline{x} \in X^f, \beta_i(\overline{x}) \text{ exists}, i = 1, k\}.$$

Even though we model our run-time dispatch after overload resolution, the situation is significantly simpler. While virtual arguments can be of pointer or reference types, possibly qualified, etc., only the class type of an argument can be changed in an overrider – the structure of the argument type will have to remain the same. This makes the choice of an overrider for a given combination of arguments significantly simpler as we do not have to model the entire set of rules used for general overload resolution [131, §over.match.best]. The relevant bit of the rules states that: "if class B is derived directly or indirectly from class A and class C is derived directly or indirectly from B, then conversion of C to B is better than conversion of C to A". This is already adequately modeled with the subclassing relation <:. The choice of the best viable function treats multiple arguments independently, which is why we use *product order* to extend partial order <: to Cartesian product of argument types.

A *dispatch table* DT_f is a mapping $DT_f : X^f \to O^f$ that maps all possible argument tuples to the argument-type tuples of overriders used to handle the call on such argument. For a given argument tuple $x \in X^f$, dispatch table entry $DT_f[x]$ is defined as following:

$$DT_f[x] = \begin{cases} \zeta(x) & x \in Y^f \\ a & \{a\} = M = \min_{y \lessdot x} DT_f[y] \\ a & \{a\} = \arg\min_{c \in M} \rho(c) \\ \text{Ambiguity} & \text{otherwise} \end{cases}$$

The first case indicates that if argument tuple is covered by one of the overriders provided by the user, we simply return the argument-type tuple corresponding to this argument tuple. The second case computes a set M of minimal (with respect to `<:`) argument-type tuples, representing dispatch table entries of all the subobjects that x extends. If M is a unit set, we have the least element, which represents the most specific overrider for the argument. When M is not a unit-set, we look at the return types $\rho(c)$ of all the argument-type tuples $c \in M$ and see if there is a single such argument-type tuple, whose return type is the smallest (with respect to `<:`). If that is the case, we have an overrider whose covariant return type is more specialized than the covariant return types of other, otherwise ambiguous, overriders.

The above recursion exhibits optimal substructure and has overlapping sub-problems, which lets us use dynamic programming [59] to create an efficient algorithm for generation of the dispatch table, as shown in Algorithm 1.

Both `<:` and $\lessdot$ are overloaded in the algorithm by extending their respective partial orders to the product order (§2.1). We iterate over argument tuples in the reverse *topological* order (with respect to $\lessdot$) in order to visit the least specialized arguments first. This ensures that during computation of a dispatch table entry, all the entries it depends on have already been computed. Note, that the empty set of immediate predecessors is only possible on the argument tuple that starts the open-method hierarchy, where we by definition would always have an overrider – the base-method. We then maintain a set M of incomparable minimal (with respect to `<:`) argument-type tuples, reflecting the most specific overriders chosen for direct base subobjects. We check that return types differ before we check which one is more specialized because `<:` is reflexive and we may remove an incomparable c, with the same $\rho(c)$. We do not do a similar check on c itself because even if removed, it will be added again as an entry that is not dominated.

To analyze the performance of the algorithm, we first note that comparison of two type-tuples from X^f can be done in time $O(k)$. If $n = \max(|A_i^{<:}|, i = 1, k)$ and $v = \max(v_i, i = 1, k)$ (where v_i is a maximum number of times A_i is used as non-virtual base class in any class of hierarchy $A_i^{<:}$) then $|X^f| <= (n * v)^k$ and the number of edges for topological sort is less then $k * (n * v)^k$. Therefore the complexity of topological sorting X^f is $O(k * n^k)$. The inner for-loop has complexity $O(k^2 * n^k)$ so the overall complexity is $O(n^k)$ since k is a constant defining the amount of virtual arguments. This means that the algorithm is linear in the size of the dispatch table.

4.3.6 Alternative Dispatch Semantics

While our goal is to unify virtual function dispatch and overload resolution into open-methods semantics, this is not always possible. Consider for example the repeated inheritance class hierarchy from §4.3.3 with a virtual function added:

```
struct A        { virtual void foo(); }; // virtual function
struct B : A    {};
struct C : A    { virtual void foo(); }; // virtual function
struct D : B, C {};

void bar(A&);               // overloaded function
void bar(C&);               // overloaded function

void foobar(virtual A&); // open−method
void foobar(virtual C&); // open−method

D d;
B& db = d;                  // B part of D
C& dc = d;                  // C part of D

// (runtime) Virtual Member Function Semantics:
db.foo();                   // calls A::foo
dc.foo();                   // calls C::foo
```

Algorithm 1 Dispatch Table Generation

```
Require: $Y^f \subseteq X^f$
Require: $\min_{x \in X^f} x \in Y^f$ // base-method must also be overrider
  $L \leftarrow \text{reverse}(\text{top\_sort}_{\lessdot}(X^f))$ // topologically sort according to $\lessdot$, reverse to have the least specific first
  for all $x \in L$ do
    if $x \in Y^f$ then // argument tuple is covered by overrider
      $DT_f[x] \leftarrow \zeta(x)$ // use argument-type tuple of overrider
    else
      $X \leftarrow \boldsymbol{\beta}(x)$ // argument tuples of immediate predecessors
      $M \leftarrow \varnothing$ // minimal argument-type tuples
      while $X \neq \varnothing$ do
        $\exists y \in X$
        $X \leftarrow X \setminus \{y\}$
        dominated $\leftarrow$ false
        for all $c \in M$ do
          if $DT_f[y] <: c$ then
            $M \leftarrow M \setminus \{c\}$
          else if $c <: DT_f[y]$ then
            dominated $\leftarrow$ true
            break
          else if $\rho(DT_f[y]) \neq \rho(c)$ then
            if $\rho(DT_f[y]) <: \rho(c)$ then
              $M \leftarrow M \setminus \{c\}$
            else if $\rho(c) <: \rho(DT_f[y])$ then
              dominated $\leftarrow$ true
              break
            end if
          end if
        end for
        if $\neg$ dominated then // not less specific than those in $M$
          $M \leftarrow M \cup \{DT_f[y]\}$
        end if
      end while
      if $|M| = 1$ then // $M$ is a unit set
        $DT_f[x] \leftarrow a$, where $M = \{a\}$
      else
        Ambiguity Error: No best overrider in $M$ for handling $x$
      end if
    end if
  end for
```

```
d.foo();                // error : ambiguous

// (compile time) Overload Resolution Semantics:
bar(db);                // calls bar(A&)
bar(dc);                // calls bar(C&)
bar(d);                 // calls bar(C&) (why not ambiguous?)

// (runtime) open-method Semantics:
foobar(db);             // calls foobar(A&)
foobar(dc);             // calls foobar(C&)
foobar(d);              // error : ambiguous
```

Virtual dispatch semantics and overload resolution semantics go different ways in this case. Since the two language features are not entirely orthogonal, we had to decide which semantics to follow.

From a technical point of view, both semantics can be implemented for open multi-methods. The reason we decided not to model the semantics after overload resolution in this case is that the resulting cross-casting behavior could have been surprising to the user due to the implicit switching of different subobjects. On the other hand, the difference between the ordinary virtual function (foo) call and the ordinary overloaded resolution for (bar) in this case is odd and depends on obscure rules that may be more historical than fundamental. Calls to the open-method foobar follow the virtual function resolution. This is why our open-method semantics strictly corresponds to virtual member function semantics in ISO C++ but does not entirely reflect overload resolution semantics. The reason is that less information is available for compile time resolution than for link-time or runtime resolution. For example, the resolution of **static_cast** and **dynamic_cast** can differ even given identical arguments: **dynamic_cast** can use more information than **static_cast**.

Due to our decision to model the semantics after virtual dispatch, we require covariance of the return type on overriders, while had we modeled after overload resolution, we could have only required convertibility of return types.

4.4 Design Choices and Decisions

Type-safety and ambiguities have always been a major concern to systems with multiple open dispatch. One of the first widely known languages to support open-methods was CLOS [239]. CLOS linearizes the class hierarchy and uses asymmetric dispatch semantics to avoid ambiguity. Snyder [230] and Chambers [46, 47] observe that silent ambiguity resolution makes errors in programs hard to spot. Therefore, Cecil uses symmetric dispatch semantics and dispenses with object hierarchy linearization in order to expose these errors at compile time. Recent studies [6, 100, 174, 176] explore the trade-offs between multi-methods and modular type checking in languages with neither a total order of classes nor asymmetric dispatch semantics. In particular, Millstein and Chambers discuss a number of models that embrace or restrict the expressive power of the language to different degrees. The described models range from globally type-checked programs to modularly type-checked units. We will briefly discuss our evaluation of these approaches in the context of C++ later in this section.

This work aims for maximal flexibility and relies on a global type-checking approach for open-methods [2]. We motivate this approach with the goal not only to support object-oriented programming but also to enhance the support for functional and generic programming styles in C++.

The cost of the global type-checking approach is that some ambiguities can be detected late – in particular at the load time of *Dynamically Linked Libraries* (*DLL*). DLLs are almost universally used with C++, thus a design for open-methods that does not allow for DLLs is largely theoretical. We do not currently have an implementation supporting dynamic linking, but we outline a design addressing the major issues in such a scenario.

Our guiding principle is to support the use cases described in §4.2 with language features that are guaranteed to be type-safe in every scenario. The idea is to report errors as long as we can assume that ambiguities can be resolved by programmers. Only when it is too late for that, we have to use type-safe resolution mechanisms. This section discusses the design decisions we have made based on three language aspects: ambiguity resolution, covariant return types, and pure (abstract) open-methods.

4.4.1 Late Ambiguities

Late ambiguities are ambiguities that are detected at a stage in the build process when programmer intervention is no longer feasible. They can occur, for example, when classes use virtual inheritance while some definitions necessary to declare a resolving overrider cannot be accessed. Consider the example given in §4.3.3.3. Examples for late ambiguities include:

- the class D was defined as a local class, since the class name would be local to the function scope.
- the class D was defined in an implementation file of a library, but the class definition was not exported in a header file.
- a library defined classes A, B, and C as well as implemented, but did not export, an open-method foo. The definition of D results in a late ambiguity.

In all cases, a programmer could not declare a resolving overrider.

A second source of late ambiguities is when independently developed libraries define conflicting overriders, but the definition of one of the involved classes is not available. Consider, the single inheritance hierarchy of §4.3.3 with an open-method foo(A,A). A library defines, but does not export B and an overrider for foo(B, A), while another library defines C and an overrider for foo(A,C). A call foo(b,c) is ambiguous but cannot be resolved, because the definition of B is not available. Ambiguities that emerge from the use of dynamically linked libraries are always late.

Resolution mechanism for late ambiguities: If there is no unique best match for a possible type tuple, we choose an overrider from all best matches. Interestingly, any overrider will result in a correct program provided the rest of the program is correct and in principle we could even pick a random overrider from the set of best matches. Nevertheless, the choice is deterministic, but remains unspecified.

Not to specify which overrider we choose among type-safe candidates, keeps the resolution mechanism symmetric, as no candidate is preferred. The use of a deterministic choice is not strictly necessary, but it allows for reproducibility – always the same method will be selected from a set of candidates.

Consider the following example of image format conversion. For a discussion of the problem and an implementation see §4.2.2 and §4.8.1 respectively. The following code shows a common header file and two independently developed libraries that support additional image formats.

```
// Common header: ImageLibrary.h
struct Image {
  virtual ~Image();
  // ...
};
struct TiffImage : Image { ... };

void convert(virtual const      Image& from, virtual      Image& to) { ... }
void convert(virtual const TiffImage& from, virtual      Image& to) { ... }
void convert(virtual const      Image& from, virtual TiffImage& to) { ... }

// DLL−Jpeg supporting JPEG images
#include ”ImageLibrary.h”
struct JpegImage : Image { ... };
void convert(virtual const      Image& from, virtual JpegImage& to) { ... }
void convert(virtual const JpegImage& from, virtual      Image& to) { ... }
void convert(virtual const TiffImage& from, virtual JpegImage& to) { ... }
void convert(virtual const JpegImage& from, virtual TiffImage& to) { ... }

// DLL−Png supporting PNG images
#include ”ImageLibrary.h”
struct PngImage : Image { ... };
```

```
void convert(virtual const  PngImage& from, virtual     Image& to) { ... }
void convert(virtual const     Image& from, virtual  PngImage& to) { ... }
void convert(virtual const TiffImage& from, virtual  PngImage& to) { ... }
void convert(virtual const  PngImage& from, virtual TiffImage& to) { ... }
```

The header file of an image library framework defines two classes (Image, TiffImage), and a base-method convert together with two overriders that implement conversions from TiffImage to a general Image and vice versa. A library (DLL-Jpeg) derives a new type JpegImage from Image and introduces new overriders for convert that handle all possible combinations of known image formats. Likewise, another library (DLL-Png) derives a new class PngImage from Image and introduces similar overriders. Now a call to convert a JpegImage into a PngImage is ambiguous. Libraries DLL-Jpeg and DLL-Png could stem from different vendors that do not know about each other. In systems that use dynamically linked libraries, such problems are hard to predict and design for.

Note that, since the class definitions of the respective other library were not available when DLL-Png and DLL-Jpeg were implemented, neither developer could possibly provide resolving overriders. The question thus arises to which convert should a call convert(JpegImage, PngImage) resolve?

Any overrider (including base-method) has to assume that a dynamic type resolving to Image is an unknown derived type. Consequently, each convert must be written so that it manipulates its arguments of types Image polymorphically (for example, by using virtual functions). This implies that as long as convert's code does not make more assumptions about its arguments than the interface defined in the base-class guarantees, any overrider can be chosen.

Alternative techniques to handle or prevent (late) ambiguities include asymmetric choice, preventive elimination of overriders that could be prone to symmetry, or exceptions that signal an error:

- *System specified choice:* Other systems with open-methods use a specified policy to resolve ambiguities. This involves preferred treatment of overriders that are more specialized on a specified argument (e.g. CLOS [239]) and class hierarchy linearization (CLOS, Dylan [222]). Making the resolution explicit, breaks symmetric dispatch, as programmers can write code that exploits the specification.
- *Limit extensibility:* In [174, 176], Millstein and Chambers discuss limitations to the type systems that prevent late ambiguities. Their system M disallows virtual inheritance across modules. Moreover, open-methods have a specified argument position. Adding overriders across module boundaries is permitted only when the type in that argument is covariant and the type is defined in the same module. MultiJava [53] is based on system M. In practice, these limitations have been found to be overly restrictive (Relaxed MultiJava [54, 175] and C++ concepts [134]). In addition, requiring C++ code to comply with the provided inheritance restrictions is not an option.

  In [6], Allen et al. develop a different set of restrictions for modular type checking of multiple dispatch for Fortress [5]. Instead of restricting multiple inheritance across modules, the notion of a meet function resolves ambiguities that originate from virtual inheritance. Moreover, their set of restrictions is sensitive to whether a function is a multi-method (defined in class) or an open-method (freestanding function). Overriding open-methods across module boundaries is not possible. Like in System M, overriding multi-methods is tied to a single distinguished argument position (the self argument) and the module of the type definition.

  Systems that require overriders defined in another module to override a specific argument position with a covariant type defined in that module are unable to handle bidirectional image conversion well. Assuming that the first argument is special, DLL-Jpeg could not provide overriders for conversions to JpegImage.

- *User specified choice:* Parasitic methods as implemented in Java [33] (an implementation for Smalltalk also exists [95]) add an object-oriented flavor to multi-methods and make them an integral part of classes. Multi-methods can be inherited from a base class and overridden (or shadowed) in the derived class. Parasitic methods give the receiver precedence over other arguments. The encapsulation guarantees that a compiler can check for multiple argument ambiguities. Virtual inheritance ambiguities are implicitly resolved by users, as the resolution is sensitive to the order of multi-method declarations within the class definition.

  In [100], Frost and Millstein unify encapsulated multiple dispatch with predicate dispatch. They replace the dependence on textual order with first match semantics, where later predicates implicitly exclude earlier predicates.

  The global checking presented in this chapter resolves less virtual inheritance ambiguities silently than an encapsulated approach would. Moreover, the use of encapsulation requires control over the construction of the receiver object. Even if that can be handled by using a factory approach, this would be unable to solve and only recast the ambiguity illustrated by the conversion example: Which converter class takes precedence, the one defined by DLL-Jpeg or the one defined by DLL-Png?
- *Glue-methods:* Relaxed Multi-Java [175] resolves ambiguity conflicts by introducing glue methods (to glue DLL-Jpeg and DLL-Png) that the system-integrator provides. This is a viable solution for software developers integrating several libraries, but it is not a feasible scenario for end-user applications, as dynamically linked modules can be loaded into the process without the direct request of a developer. This is the case for various component object models where applications may request an object by name from the system. The operating system will locate and load the module in which the object resides.
- *Throw an exception:* Some implementations (e.g. Cmm [229]) throw an exception at dispatch time when an ambiguity is encountered. We disagree with this approach because each candidate alone is a type-safe choice and should be able to handle the requested operation. Moreover, this approach forces programmers to consider open-method calls as a potential source for exceptions, while their choice of how to handle this exception is limited and likely will result in program termination.
- *Program termination:* Instead of waiting until runtime, the application can terminate (or fail to link) when ambiguous overriders are detected. We argue analogously to the exception case that termination is an inadequate response for a choice among type-safe operations.

4.4.2 Consistency of Covariant Return Types

Before we go into a detailed discussion, we would like to point out that the focus of this section is on the consistency of covariant return types among overriders available at runtime. The use of covariant return type for ambiguity resolution is orthogonal to the problems discussed here and is discussed in detail in §4.3.4.

Different DLLs can specify conflicting covariant return types. Consider a two-class hierarchy $A \leftarrow B$ and another two-class hierarchy $R1 \leftarrow R2$. The base-method R1 foo(**virtual** A&, **virtual** A&) is defined in a header visible by two dynamically linked modules D_1 and D_2 that do not know anything about each other. Module D_1 introduces overrider R2 foo(A&, B&) and module D_2 introduces overrider R1 foo(B&, B&). Each of the dynamically linked modules perfectly type checks and links with foo() resolved through the following dispatch tables:

D_1	A	B
A	$\boldsymbol{\langle A, A\rangle}^{R1}$	$\boldsymbol{\langle A, B\rangle}^{R2}$
B	$\langle A, A\rangle^{R1}$	$\langle A, B\rangle^{R2}$

D_2	A	B
A	$\boldsymbol{\langle A, A\rangle}^{R1}$	$\langle A, A\rangle^{R1}$
B	$\langle A, A\rangle^{R1}$	$\boldsymbol{\langle B, B\rangle}^{R1}$

When both libraries are linked together, we get the dilemma of how to resolve a call with both arguments of type B. On one side foo(B&,B&) from D_2 is more specialized, but on the other side foo(A&,B&) from D_1 imposes the additional requirement that the return type of whatever is called for $\langle B, B\rangle$ should be a subtype of R2, which R1 is not. Such scenario would be rejected at compile/link time, however at load time we do not have this option anymore.

Keeping all dispatch tables of a particular open-method consistent on the overrider that will be called for a particular combination of types will force us to choose between suboptimal and type unsafe alternatives. What is worse is that there may not be a unique type-safe alternative.

Imagine for example that a module D_3 introduces an overrider R3 foo(B&, A&) where $R1 \leftarrow R3$, so $R2$ and $R3$ are siblings. When D_1 and D_3 are loaded together, neither R2 foo(A&, B&) nor R3 foo(B&, A&) can be used to resolve a call with both arguments of type B – both alternatives are type unsafe for the other overrider.

To deal with this subtlety, we propose for the DLL case to weaken the requirement that the same overrider should be called for the same tuple of dynamic types regardless of the static types used at the call site. We require that the same overrider be used only if it is type-safe for the caller. Strictly speaking R1 foo(B&,B&) is not an overrider of R2 foo(A&, B&) as defined in §4.3, because its return type is not changing covariantly in respect to the types of arguments. Therefore, it cannot be considered for the dynamic resolution of calls made statically through the base-method R2 foo(A&, B&).

Taking the above into account, we propose that the dynamic linker fills in the dispatch table of every base-method independently. This results in:

D_1	A	B
A	$\boldsymbol{\langle A, A\rangle}^{R1}$	$\boldsymbol{\langle A, B\rangle}^{R2}$
B	$\boldsymbol{\langle B, A\rangle}^{R3}$	$\boldsymbol{\langle B, B\rangle}^{R1}$

D_2	A
A	$\boldsymbol{\langle A, B\rangle}^{R2}$
B	$\langle A, B\rangle^{R2}$

D_3	A	B
B	$\boldsymbol{\langle B, A\rangle}^{R3}$	$\langle B, A\rangle^{R3}$

It looks as if the dispatch table for the base-method R1 foo(A&,A&) now violates covariant consistency, but in reality it does not because all the return types in it are cast back through thunks to R1, which is the type statically expected at the call site.

As can be seen, this logic may result in different functions being called for the same type tuple depending on the base-methods seen at the call site. We note, however, that *the call is always made to the most specialized overrider that is type-safe for the caller.*

4.4.3 Pure Open-Methods

There are no abstract (pure virtual) open-methods; that is, every open-method must be defined. Consider a (dynamic) library D_1 that introduces a new class and a second (dynamic) library D_2 that defines a new abstract open-method. When both libraries are (dynamically) linked together the presence of an overrider for the class in D_1 cannot be guaranteed. The alternative would be runtime “method not defined” errors (reported as exceptions), but that solution would be inconsistent with the rest of C++ and would limit the use of open-methods in embedded systems.

4.5 Relation to Orthogonal Features

In this section, we discuss the relationship of open-methods to other language features.

4.5.1 Namespace

Virtual functions have a class scope and can only be overridden in the derived classes. Open-methods do not have such a scope by default, so the question arises when should an open-method be considered an overrider and when just a different open-method? Let us look at the following example:

```
namespace X
{
  class A {};
  void bar(virtual A&); // base−method
  class B : A {};
  void bar(virtual B&); // (1)
}
namespace Z
{
  void bar(virtual B&); // (2)
}
namespace Y
{
  class D : X::A {};
  void bar(virtual D&); // (3)
}
class C : X::A {};
void bar(virtual C&);   // (4)
```

In the presented implementation, an overrider has to be declared in the same namespace as its base-method (1). Open-methods with the same name and compatible parameter types, defined in different namespaces would not be considered overriders. The major benefit of this approach is that it is easy to understand and implement. Unfortunately, such semantics are not unifiable with overrider declarations of virtual function calls, where derived classes can be declared in a different namespace. **using** declarations present a potential work around to these limitations.

An alternative would be to let overriders be declared in any namespace (1,2,3,4). It is easy to understand, but defeats the purpose of namespaces that were introduced to better structure the code and avoid name-clashes among independently developed modules.

Another alternative may consider an open-method to be an overrider if its base-method is defined in the same scope or in the scope of their argument types and their base classes. In this scenario (1, 3, 4) would override; (2) would not. Among its advantages is that it closely resembles argument dependent lookup. It would also work for virtual functions. Its downside, however, is that it is harder to comprehend.

4.5.2 Access Privileges

Open-methods are generic freestanding functions, which do not have the access privileges of member functions. If an open-method needs access to non-public members of a class, that class must declare a particular open-method as a friend.

4.5.3 Smart Pointers

In C++, programmers use smart pointers, such as auto_ptr (in current C++) as well as shared_ptr and weak_ptr (in Boost [30] and C++0x [16]) for resource management. The use of smart pointers together with open-methods is no different from their use with (virtual) member functions. For example:

```
struct A { virtual ~A(); };
struct B : A {};
void foo(virtual A&);

void bar(shared_ptr⟨A⟩ ptr) { foo(*ptr); }
```

Defining open-methods directly on smart pointers is not possible. In the following example, (1) yields an error, as ptr1 is neither a reference nor a pointer type. The declaration of (2) is an error, because shared_ptr is not a polymorphic object (it does not define any virtual function). Even when shared_ptr were polymorphic, the open-method declaration would be meaningless. A shared_ptr⟨B⟩ would not be in an inheritance relationship to shared_ptr⟨A⟩, thus the compiler would not recognize foo(**virtual** shared_ptr⟨B⟩&) as an overrider.

```
void foo(virtual  shared_ptr ⟨A⟩  ptr1); // (1) error
void foo(virtual  shared_ptr ⟨A⟩& ptr2); // (2) error
```

4.6 Implementation

We have implemented open-methods as described in §4.3 by modifying the EDG compiler front end [84]. This includes dispatch table generation and thunk generation for multiple inheritance and covariant return. To reduce the dispatch table size, we have also implemented the dispatch table compression techniques presented in [8]. Our current implementation does not support dynamically linked libraries and detection of late ambiguities.

4.6.1 Changes to Compiler and Linker

Our mechanism extends ideas presented in [94, 272] as to the compiler and linker model. We adopted the multi-method syntax proposed in [243], which in turn was inspired by an earlier idea by Doug Lea (see [243, §13.8]). One or more parameters of a non-static freestanding function can be specified to be **virtual**. Overloading functions based only on the virtual specifier is not allowed.

A *virtual argument* must be a reference or pointer to a polymorphic class (that is, a class containing at least one virtual function). For example:

```
struct A { virtual  ~A(); };

void print (virtual  A&);              // ok
void print (int, virtual  A*);         // ok
void print (int, virtual  const A&); // ok

void dump(virtual A);                  // compiler error
void dump(virtual int);                // compiler error
```

For each translation unit, the EDG compiler lowers the high-level abstractions in C++ to equivalent code in C. We added an implementation that lowers open-method calls to C according to the object-model presented in §4.6.2. In addition, the compiler puts out an *open-method description* (*OMD*) file that stores the data needed to generate the runtime data structure discussed in §4.6.2. This includes the names of all classes, their inheritance relationships, and the kind of inheritance. Open-methods are represented by name, return type, and their parameter list. Finally, the OMD-file also contains definitions of all user-defined types that appear in signatures of open-methods (both as virtual and regular parameters).

These definitions are necessary to generate class definitions for arguments to open-methods that are passed by value.

The pre-linker uses Coco/R [273] to parse the OMD-files. Then, the pre-linker synthesizes the OMD-data, associates all overriders with their base-methods, generates dispatch tables, issues link-errors for ambiguities, determines the indices necessary to access the open-method, and initializes the data structures described in §4.6.2.

When the call of an overrider requires adjustments of this-pointers (as is sometimes needed in multiple inheritance hierarchies), the pre-linker creates thunks and makes the dispatch table entries refer to them instead. During dispatch table synthesis, the linker will report errors for all argument combinations that do not have a unique best overrider. The output of the pre-linking stage is a C-source file containing the missing definitions. If the linker generates a library, the pre-linker also puts out a merged OMD-file.

4.6.2 Changes to Object Model

We augment the IA-64 C++ object model [55] by four elements to support constant time dispatching of open-methods. First, for each base-method there will be a dispatch table containing the function addresses. Second, the v-table of each subobject contains an additional pointer to the *open-method table* (*om-table*). Finally, the indices used for the open-method-table offsets are stored as global variables.

The figures 4.1 and 4.2 show the layout of objects, v-tables, om-tables and dispatch-tables for repeated and virtual inheritance. Our extensions to the object-model are shown with gray background. From left to right the elements in each diagram represent the object, v-table, om-table, and dispatch table(s) for the class hierarchy in §4.3.3. From top to the bottom, the objects are of type A, B, C, and D respectively.

An open-method can be declared after the declarations of the classes used in its virtual parameters. Therefore, the compiler cannot reserve v-table entries to store the data related to open-method dispatch immediately in a class's virtual function table. Hence, we always extend every v-table by one pointer referencing the om-table, which can be laid down later by the pre-linker.

The om-table reserves one position for each virtual parameter of each base-method, where objects of this type can be passed as arguments. This position stores an index into the corresponding dimension of the dispatch table. Since the size of the om-tables is not known at compile time, our technique relies on a literal for each open-method and virtual parameter position (called foo$_1$st, foo$_2$nd in figures 4.1 and 4.2 that determines the offset within the om-tables.

Note that these figures depict our actual implementation, where entries for first argument positions already resolve one dimension of the table lookup. Entries for all other argument positions store the byte offset within the table.

In the presence of multiple-inheritance, this-pointer shift might be required to pass the object correctly. In this case, we replace the address of the overrider by an address of a thunk that takes care of correctly adjusting this-pointer. As described in §4.3.3.2 in case of repeated inheritance, different bases can show different dispatch behavior depending on the subobject to which this-pointer refers. As a result, different bases may point to different om-tables. In case of virtual inheritance, the open-method dispatch entries are only stored through the types mentioned in the base-method. Hence, in the virtual inheritance case, all open-method calls are dispatched through the virtual base type.

4.6.3 Alternative Approaches

We considered a few other design alternatives and explored their trade-offs in extensibility and performance.

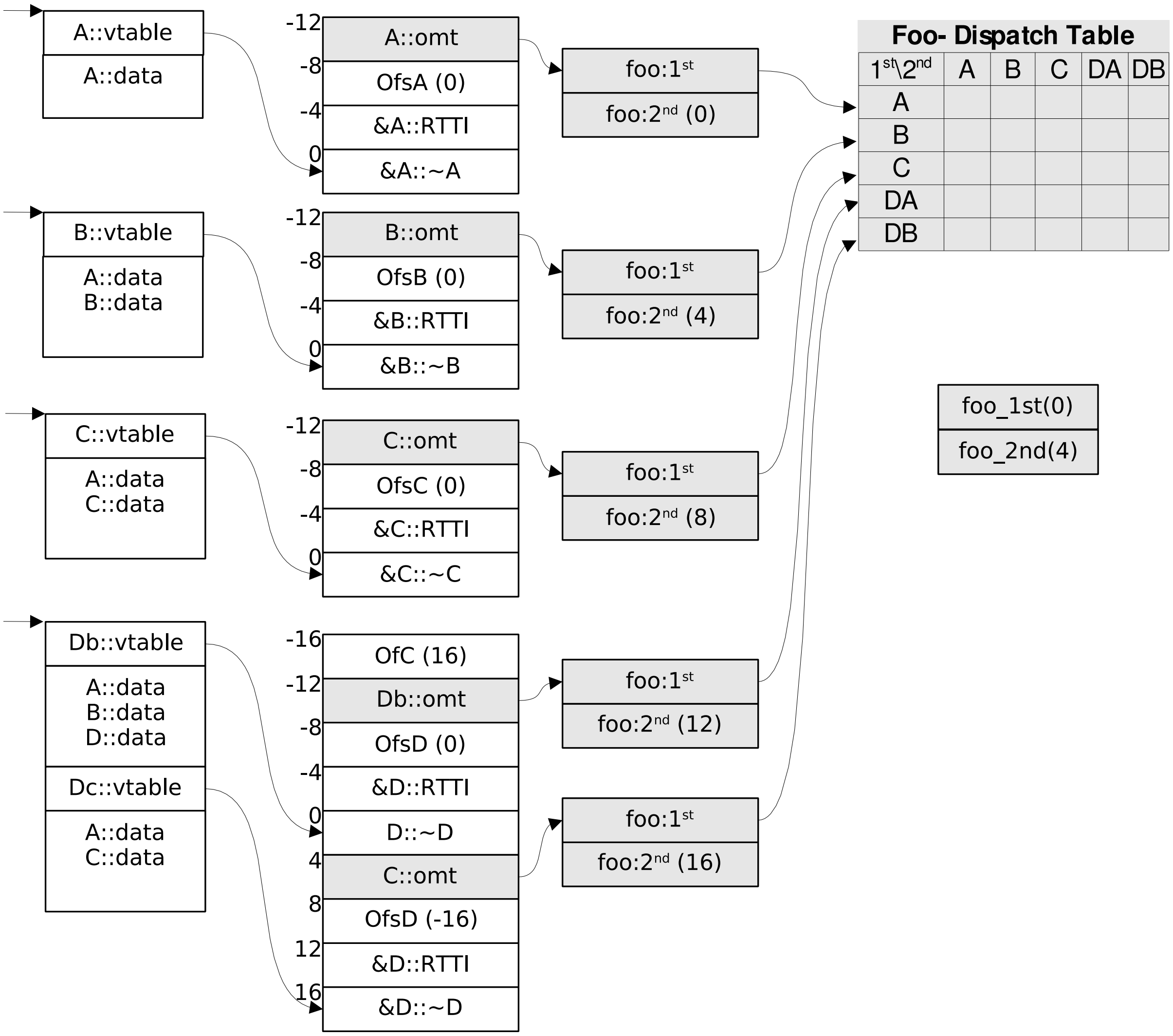


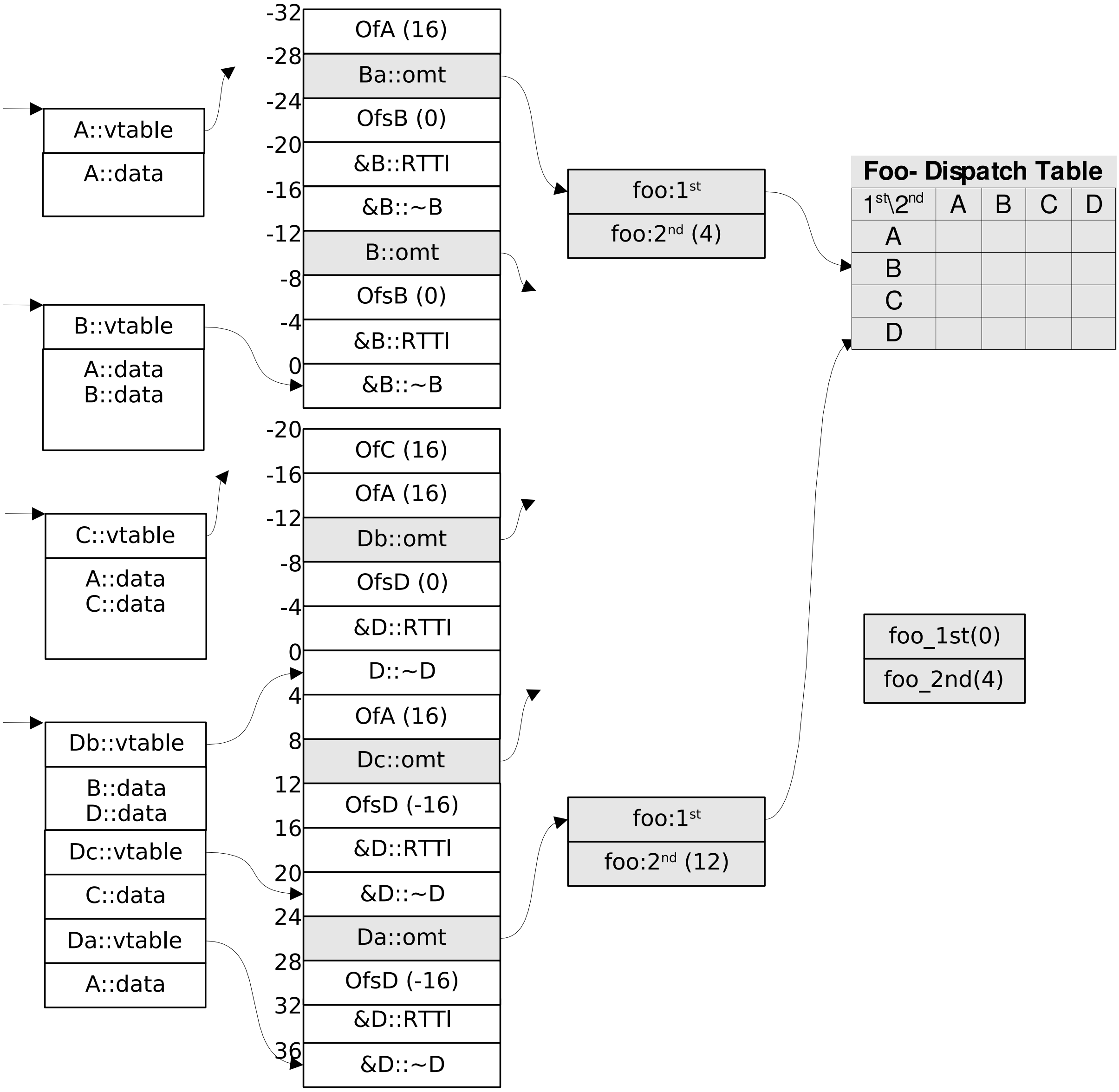
Figure 4.1: Object Model for repeated inheritance

4.6.3.1 Multi-Methods

Unlike open-methods, multi-methods require the base-method to be declared in the class definition of its virtual parameters. This allows the offset within the v-table be known at compile time, which saves two indirections per argument of a function call (one for the om-table, and one to read the index within the om-table). For a call with k virtual arguments, open-methods need $4k+1$, while multi-methods need only $2k+1$ memory references to dispatch a call. The downside of multi-methods is that existing classes cannot easily be extended with dynamically dispatched functions.

With the restriction of in-class declarations imposed by multi-methods it seems logical to declare a multi-method either as a member function or as a friend non-member function. Consider:

```
class Matrix
{
 // multi−method declaration as a non−member function
 friend Matrix& operator+(virtual const Matrix& lhs, virtual const Matrix& rhs);
```



1st\2nd	A	B	C	D
A				
B				
C				
D				

Figure 4.2: Object Model for virtual inheritance

```
  // equivalent declaration as a member function
  virtual Matrix& operator*(virtual const Matrix&);
};
```

We implemented only the non-member version of multi-methods. The member version can be implemented with exactly the same techniques. However, in many cases, it is harder to write code that uses the member version because an overrider must be a member of (only) one class – and the main rationale for multi-methods is to elegantly deal with combinations of classes. Even the non-member (friend) version is hard to use.

By requiring a declaration to be present in a class, we limit the polymorphic operations to those

Algorithm 2 Dispatching with Chinese Remainders

```
for all argument positions $i$ of a multi-method $f$ do
    $n_i = x_i \mod m_i$
end for
call $D[n_1, \cdots, n_k]$ with arguments provided
```

that the class designer thought of. That requires too much foresight of the class designer or leads to unstable classes (classes that keep having multi-methods added). Such problems are well-known in languages relying on member functions. Open-methods provide an abstraction mechanism that solves such problems by separating operations from classes.

4.6.3.2 Chinese Remainders

As we saw in §4.6.2, support of open-methods required an extra indirection via open-method table to get index of the class in appropriate argument position. This extra indirection was needed because open-methods are not bound to the class and as a result, we do not know how many of them a class may have, therefore we cannot reserve entries in the v-table for them. In this section, we present an "ideal" scheme for implementing open-methods, inspired by ideas presented in [107]. The proposed scheme circumvents the necessity for open-method tables by moving all the necessary information from the class to the dispatch table.

Suppose that for every multi-method f there is a function $I_f : T \times N \to N$ such that for any type $t \in T$ (where T is a domain of all types) and argument position $n \in N$ it returns index of type t in the n^{th} dimension of the f's dispatch table. If such function is reasonably fast (preferably constant time) and its range is small (preferably from zero to the maximum number of types that can be used in any argument position) then we can efficiently implement multiple dispatch by properly arranging rows and columns according to the indices returned by I_f. As in [107], we use the Chinese Remainder theorem [59] to generate function I_f.

Theorem 1. Chinese Remainder Theorem *Let $m_1, \cdots, m_k$ be integers with $\gcd(m_i, m_j) = 1$ whenever $i \neq j$. Let m be the product $m = m_1 m_2 \cdots m_k$. Let $a_1, \cdots, a_k$ be integers. Consider the system of congruences:*

$$\begin{cases} x \equiv a_1 (\text{mod } m_1) \\ x \equiv a_2 (\text{mod } m_2) \\ \cdots, \\ x \equiv a_k (\text{mod } m_k) \end{cases}$$

Then there exists exactly one $x \in Z_m$ satisfying this system.

Since we may have different class hierarchies in different argument positions, we have to consider each argument position separately. Assuming that there can be q different types $t_{i1}, t_{i2}, \cdots, t_{iq}$ in an argument position i, we may assign a different prime number $m_{ij} : j = 1, q$ to each of them and then according to the Chinese Remainder Theorem find a number x_i that satisfies the above equation. Storing x_i for each dimension (argument position) of dispatch table, we will come to the dispatching algorithm shown in listing 2. Since k is known at compile time, no actual iteration is required and the algorithm takes constant time.

In this scenario, every class (or more specifically every argument position i where this class may appear as virtual argument) will have a prime number m_i assigned to it, while the dispatch table will have a number x_i computed through Chinese Remainders, associated with each of its dimensions. The result of $x_i \mod m_i$ gives us the column within the appropriate dimension of dispatch table.

This dispatching technique has the nice property that it does not need any modifications of the v-table in order to introduce a new open-method on the class. Once allocated, prime numbers can be reused for any number of open-methods defined on the class regardless of the argument position in which a type is used. After dispatch table allocation, we simply have to compute the number x_i for each of the argument positions. Extending such a table, that may be required after introduction of a new class in the hierarchy, is also simple: allocate new rows and columns and recompute x_i taking prime numbers of newly added classes into account.

We demonstrate the approach with an example. Consider the following class hierarchy and an open-method foo defined on it:

```
                             // Assigned primes:
class A                 {}; //  2
class B : public A {}; //  5
class C : public A {}; //  3
class D : public B,      //
              public C {}; // 13 for  D/B and  7 for D/C
class E : public D {}; // 17 for  E/B and 19 for E/C

void foo(virtual  A&, virtual  A&);
void foo(virtual  B&, virtual  B&);
void foo(virtual  B&, virtual  C&);
void foo(virtual  C&, virtual  B&);
void foo(virtual  C&, virtual  C&);
void foo(virtual  E&, virtual  E&);
```

The following dispatch table is built:

	A^2	B^5	C^3	D::B^{13}	D::C^7	E::B^{11}	E::C^{17}	$x \equiv$	mod
A^2	$\boldsymbol{\langle A, A\rangle}$	$\langle A, A\rangle$	$\langle A, A\rangle$	$\langle A, A\rangle$	$\langle A, A\rangle$	$\langle A, A\rangle$	$\langle A, A\rangle$	0	2
B^5	$\langle A, A\rangle$	$\boldsymbol{\langle B, B\rangle}$	$\boldsymbol{\langle B, C\rangle}$	$\langle B, B\rangle$	$\langle B, C\rangle$	$\langle B, B\rangle$	$\langle B, C\rangle$	1	5
C^3	$\langle A, A\rangle$	$\boldsymbol{\langle C, B\rangle}$	$\boldsymbol{\langle C, C\rangle}$	$\langle C, B\rangle$	$\langle C, C\rangle$	$\langle C, B\rangle$	$\langle C, C\rangle$	2	3
D::B^{13}	$\langle A, A\rangle$	$\langle B, B\rangle$	$\langle B, C\rangle$	$\langle B, B\rangle$	$\langle B, C\rangle$	$\langle B, B\rangle$	$\langle B, C\rangle$	3	13
D::C^7	$\langle A, A\rangle$	$\langle C, B\rangle$	$\langle C, C\rangle$	$\langle C, B\rangle$	$\langle C, C\rangle$	$\langle C, B\rangle$	$\langle C, C\rangle$	4	7
E::B^{11}	$\langle A, A\rangle$	$\langle B, B\rangle$	$\langle B, C\rangle$	$\langle B, B\rangle$	$\langle B, C\rangle$	$\boldsymbol{\langle E, E\rangle}$	$\boldsymbol{\langle E, E\rangle}$	5	11
E::C^{17}	$\langle A, A\rangle$	$\langle C, B\rangle$	$\langle C, C\rangle$	$\langle C, B\rangle$	$\langle C, C\rangle$	$\boldsymbol{\langle E, E\rangle}$	$\boldsymbol{\langle E, E\rangle}$	6	17

$x \equiv$	0	1	2	3	4	5	6	$x =$
$\mod P(a_1)$	2	5	3	13	7	11	17	1062506

Dispatching a call will then look like $DT_{foo}[x_{foo} \mod P(a_1), x_{foo} \mod P(a_2)](a_1, a_2)$. Having pointers a_1 and a_2 to the actual arguments of a call, we can look up the v-tables of those arguments and the prime numbers $P(a_i)$ associated with their types. Suppose that the prime number $P(a_1)$ associated with the first argument is 11, while the prime number $P(a_2)$ associated with the second argument is 3. To get the row number inside the dispatch table associated with the first argument, we compute the remainder of dividing x_{foo} = 1062506 by 11, which is 5. Row number 5 corresponds to the B subobject of an object with dynamic type E. Similarly, we get the column associated with the type of the second argument through finding the remainder of dividing 1062506 by 3, which is 2. Column number 2 corresponds to type C, which means that the dynamic type of the second argument is C. The number 1062506 is associated with the dispatch table, through which the call is being dispatched. To find the overrider that will be handling the call, we simply look up an address of the function that is stored at the intersection of the fifth row and the second column, which is foo(B&,C&).

Despite its elegance, this approach is rather theoretical because it is hard to use for large class hierarchies. The reason is that we need to assign different prime numbers to each class and perform computations on numbers that are bound by the product of these primes. The product of only the first nine primes fits into a 32-bit integer and the first 15 primes into a 64-bit integer. Table compression techniques [8] or the use of minimal perfect hash functions [59] instead, can help overcome the problem.

The idea of using Chinese reminders for multiple dispatch seems to be quite natural. In particular, in response to the original publication [208] of the material presented in this chapter Gabor Greif sent us his unpublished notes on a similar use of Chinese Remainders for implementing multiple dispatch [115] in Dylan.

4.7 Results

In order to discuss time and space performance, we compare code generated by our C++ Open Method Compiler, described in §4.6, to a number of prototype implementations, the visitor pattern, Cmm, DoubleCpp, and the Loki library. The prototypes of our design alternatives were implemented in C to approximate the lowering of C++ code to C. They were initially developed to assess performance trade-offs of different approaches and workaround techniques and include open-methods (can be declared freely), multi-methods (have to be declared in class, thus the om-tables can be embedded into the v-table, saving two indirections per argument §4.6.3.1.), and a Chinese Remainder (§4.6.3) based implementation. These implementations layout the dispatch table for the concrete example described below as C data structures, and then dispatch calls through it.

We wrote 20 classes (representing shapes, etc.) that can intersect each other. Overall, this results in 400 combinations for binary dispatch functions. We implemented 40 specific intersect functions to which all of the 400 combinations are dispatched. In order to get a reliable timing of the function invocation, these 40 intersect functions only increment a counter. Since not all techniques we use support multiple inheritance, these 20 classes only use single inheritance. The actual test consists of a loop that randomly chooses 2 out of 32 objects and invokes the intersect method. We implemented a table-based random number generator that is simple and does not contain any floating-point calculations or integer-divisions. We ran the loop twice with the same random numbers: The first run allows implementations that build the dispatch data structure on the fly to warm up and load data/code into the cache. The second loop was timed. The clock-cycle based timer takes the time before and after the loop and we calculate the average number of clock-cycles per loop to compare the results.

4.7.1 Implementations

We tested the approaches on a Pentium D at 2.8 GHz running CentOS Linux, a Core 2 Duo running Mac OS X and an Intel Core i5 460M at 2.53 GHz running Windows 7 Professional. The code for the performance tests was compiled with G++ 4.1 (Linux), GCC 4.0.1 (OS X) and G++ 4.7.2 (Windows 7) with optimization level set to -O3. The C++ Open Method Compiler generates source code lowered to C, which was compiled with the corresponding GCC versions and linked to the pre-linker generated dispatch tables.

Using the Chinese Remainder approach, the number associated with the dispatch table grows exponentially with the number of types. Therefore, the test is limited to 8 types instead of 20 and the size of the executable is omitted.

For Loki, we only tested the static dispatcher because the others require manual handling of all possible cases. Using other dispatchers would have been closer to a scenario of a manually allocated array of functions through which calls are made. However, as we indicated before, the dual nature of multi-methods require them to provide both dynamic dispatch and automatic resolution mechanism.

4.7.2 Results and Interpretation

Our experimental results can be summarized in terms of execution time and program size:

Approach	**Size**	**Cycles/Loop**		
	Linux	Linux	Mac	Win
Virtual function	n/a	75	55	29
Multi-methods prototype	42 972	78	60	33
Open-methods prototype	40 636	82	63	38
C++ Open-method Compiler	42 504	82	64	56
Double Cpp	34 812	120	82	48
C++ Visitor	38 236	132	82	40
Chinese Remainders	n/a	175	103	57
Cmm (constant time)	155 344	415	239	107
Cmm	155 056	1 320	772	562
Loki Library	75 520	3 670	2 238	3167

Table 4.1: Comparison of Open Multi-Methods against other approaches to multiple dispatch

Executable size: To obtain a comparable size of the executable, we used the regular EDG frontend to generate C code for the alternative approaches. Then we compiled all intermediate C files with GCC, where optimizations were set to minimize code size. Moreover, we stripped off the symbols from the executables. The size of dispatch tables is mentioned as one of the major drawbacks of providing multi-methods as programming language feature [272]. However, our results reveal that the best achievable code size is roughly the same for visitors, prototyped multi-/open-method, and C++ Open-method Compiler implementations. With the visitor, each shape class has intersect methods for all 20 shapes of the hierarchy. A somewhat smarter approach would be to remove redundant intersect overriders. However, removing specific overriders is tedious and difficult to maintain, since the dispatch would be based on the static type information of the base class. Even an optimized approach would require as many v-table entries as there are in a dispatch table, simply because each type contains 20 intersect entries in the v-table. Multiplying this with the number of shapes, 20, results in 400, exactly the number of entries found in the dispatch table. We further discuss memory requirements of various alternatives of multiple dispatch in §5.6.6.

Execution time: The results for prototyped multi-methods, prototyped open-methods, and C++ Open-method Compiler are (as expected) roughly comparable to a single virtual function dispatch on all three platforms, hence, the better performance compared to the visitors is not surprising. However, the fact that multi-methods reduce the runtime by up to 38% in comparison to the reference implementation using the visitor is noteworthy. We conjecture this is an effect of the size of the class hierarchy and that the time to double dispatch depends on the number of overriders.

Significance of performance: The performance numbers come from experiments designed to highlight the cost of multiple dispatch: the functions invoked hardly do anything. Depending on the application, the improved performance may or may not be significant. For the image conversion example, gains in execution speed are negligible compared to time spent in the actual conversion algorithm. In other cases, such as the evaluation of expressions using user-defined arithmetic types, traversal of abstract syntax trees, and some of the most frequent shape intersect examples, the speed differences among the double dispatch approaches appear to be notable.

Contrary to much “popular wisdom”, our experiments revealed that for many applications the use of dispatch tables for open-methods and multi-methods actually reduce the program size compared to brute-force and work-around techniques. Under the assumption that the use of open-methods in C++ would be similar to Muschevici et al.’s results [188], discussed in §3.4, we conclude that the size of the dispatch table will remain small for most practical cases.

4.8 Experiences

In order to compare open-methods with double dispatch and the visitor pattern, we have implemented some of the examples from §4.2.

4.8.1 Image Format Conversion

The first example is image conversion. To meet the performance requirements typical for image processing applications, information about the exact source and destination formats is indispensable for an efficient conversion. With this information, we can call a routine geared for that specific pair of formats. Any attempt to work through a common base interface will significantly hinder performance, and should be avoided. This is why we use a fairly shallow class hierarchy to represent different image formats. Another interesting aspect of this example is that when the pair of formats is known statically, it is feasible to write a generic conversion algorithm that relies on some format traits. This is an approach taken by Adobe’s GIL library [31]. Therefore, the main goal in this example is to uncover the dynamic types of both arguments and pass on these uncovered arguments together with their static types to a set of overloaded template functions.

```
template ⟨class SrcImage, class DstImage⟩
bool generic_convert (const SrcImage& src, DstImage& dst);

typedef unsigned char color_component;

struct image {
  // member functions to access row buffer , width, height etc.
};

struct RGB : image { // abstract base of all RGB images
  struct color { color_component R, G, B, A; };
  virtual color get_color (int i , int j ) const              = 0;
  virtual void  set_color (int i , int j , const color & c) = 0;
};

struct RGB32 : RGB { ... }; // implements get_color, set_color
// ... Similar definitions for RGB24, RGB16, RGB15, RGB08

struct YUV : image { // abstract base of all YUV images
  struct color { color_component Y, U, V, A; };
  virtual color get_color (int i , int j ) const              = 0;
  virtual void  set_color (int i , int j , const color & c) = 0;
};
struct UYVY : YUV { ... }; // implements get_color, set_color
// ... Similar definitions for YUY2,Y41P,CLJR,YVU9,YV12,IYUV,I420,Y800 etc.
struct CMYK : image {
  struct color { color_component C, M, Y, K; };
  virtual color get_color (int i, int j) const           = 0;
  virtual void set_color (int i, int j, const color& c) = 0;
};
```

Open multi-methods to handle the cases can be listed separately from class definitions:

```
// Base open−method. Fails as we do not know anything about the formats
bool convert(virtual const image& src, virtual image& dst) { return false; }
// Slow polymorphic conversions
bool convert(virtual const RGB& src, virtual RGB& dst);
bool convert(virtual const RGB& src, virtual YUV& dst);
bool convert(virtual const YUV& src, virtual RGB& dst);
bool convert(virtual const YUV& src, virtual YUV& dst);
// Fast generic conversions, generated for each combination of types
bool convert(virtual const RGB32& src, virtual RGB32& dst) { return generic_convert(src, dst); }
bool convert(virtual const RGB32& src, virtual RGB24& dst) { return generic_convert(src, dst); }
bool convert(virtual const RGB32& src, virtual YUY2& dst) { return generic_convert(src, dst); }
bool convert(virtual const RGB32& src, virtual YVU9& dst) { return generic_convert(src, dst); }
bool convert(virtual const RGB32& src, virtual I420& dst) { return generic_convert(src, dst); }
```

In case of double dispatch, the code becomes cluttered with definitions to support the mechanism:

```
// Forward declare all classes that would participate in double dispatch
struct RGB32; struct RGB24; // ... others
struct image {
  // member functions to access row buffer, width, height etc.
  // Double dispatch support code
  virtual bool convert_to(image& dst) const = 0;
  virtual bool convert_from(const RGB32 & src) { return false; }
  virtual bool convert_from(const RGB24 & src) { return false; }
  virtual bool convert_from(const RGB16 & src) { return false; }
  // ... etc. for all other leaf image classes
};
struct RGB32 : RGB {
  virtual bool convert_to(image& dst) const    { return dst.convert_from(*this); }
  virtual bool convert_from(const RGB32 & src) { return generic_convert(src, *this); }
  virtual bool convert_from(const RGB24 & src) { return generic_convert(src, *this); }
  virtual bool convert_from(const RGB16 & src) { return generic_convert(src, *this); }
  // ... etc. for all other leaf image classes
};
```

The major disadvantage of the double dispatch approach [128] is that we have to foresee the whole hierarchy at the moment we are defining its root. This is necessary for declaring the interface for uncovering types. Once it is defined, we cannot extend it for newly created classes – they will all be treated as their closest ancestor in the hierarchy. Another problem with double dispatch is that its supportive structures clutter the code. This may be acceptable when double dispatch is needed for only one algorithm, but when several algorithms require it (e.g. we would also like to have a polymorphic **bool** compare(**virtual const** image& a, **virtual const** image& b)) then the code may quickly get out of hands. While this aspect of the double dispatch can be solved with the visitor pattern (§2.7) at the cost of two extra virtual calls, the open-method solution will remain cleaner as open-methods do not even need to be defined together with the class. We discuss the visitor pattern in greater detail in our second example.

The number of lines in the implementation with open methods was smaller, but overall, the number of lines in both implementations is growing as square of the number of classes in the hierarchy. We note that for open multi-method implementations this is a rather exceptional case, because the class hierarchy was shallow, while we were interested in uncovering all possible type combinations. For the double dispatch, this is rather typical case because the supportive definitions will have to be there anyway.

In the image conversion example, the main purpose of the open-methods is to discover the dynamic types of both arguments and then forward the call to an overloaded function. This raises a question of whether introduction of parameterized overriders would not make such definitions easier. In such a case, users can introduce only a base open method and a parameterized version of all the overriders. The compiler will then use the parameterized version to generate all the entries in the dispatch table:

```
// Base open−method. Fails as we do not know anything about the formats
bool convert(virtual const image& src, virtual image& dst) { return false; }
// Slow polymorphic conversions
bool convert(virtual const RGB& src, virtual RGB& dst);
bool convert(virtual const RGB& src, virtual YUV& dst);
bool convert(virtual const YUV& src, virtual RGB& dst);
bool convert(virtual const YUV& src, virtual YUV& dst);
// Fast generic conversions, generated for each combination of types
template ⟨class Source, class Destination⟩
bool convert(virtual const Source& src,virtual Destination& dst) {
  return generic_convert (src, dst);
}
```

This feature however, can be a subject of a separate work so we do not investigate it here.

4.8.2 AST Traversal

The second example discusses the use of open-methods to traverse ASTs. The key focus thereby is on extending classes with open dispatch rather than multiple dispatch. Open-methods essentially become virtual functions that can be added to a class after it has been defined. The examples in this section reflect our experience of writing an analysis pass for the Pivot source-to-source transformation infrastructure [246]. The Pivot uses the visitor pattern to type safely uncover the dynamic type of AST nodes. The Pivot consists of 162 classes, but in the ensuing discussion, we limit the AST hierarchy to only two of them, where one Expr is a base class for all kinds of expressions, and the other Unary is an implementation of unary expressions.

```
struct Expr { // ...
  virtual accept(Visitor& v) const { v.visit(*this); }
};
struct Unary : Expr { // ...
  accept(Visitor& v) const { v.visit(*this); }
};
struct Visitor {
  void visit(const Expr&) = 0;
  void visit(const Unary&) = 0;
};
```

Forwarding calls to base implementations: Currently, the Pivot has 162 node types. The Pivot provides a number of intermediate abstract base classes that factor commonalities (e.g. Expr, Type, Declaration, etc.) of the 162 node types. If the logic of the visitor can be implemented in terms of a single base class, the bodies of the more specific types will need to explicitly invoke the base implementation (compare to the implementation for Unary). Open-methods have this forwarding behavior by default.

```
struct SimpleVisitor : Visitor {
  virtual void visit (const Expr& e) { ... };
  virtual void visit (const Unary& u) { visit (static_cast⟨Expr&⟩(u)); } // calls visit (Expr&)
};
// All objects derived from Expr will be handled by simpleOpenMethod
void simpleOpenMethod(virtual const Expr&) {... }
void foo(Expr& e, Unary& u) {
  SimpleVisitor vis;
  e.accept(vis); // invokes SimpleVisitor :: visit (const Expr&)
  u.accept(vis); // invokes SimpleVisitor :: visit (const Unary&) first
  simpleOpenMethod(e); // invokes simpleOpenMethod(const Expr&)
  simpleOpenMethod(u); // invokes simpleOpenMethod(const Expr&)
}
```

Passing of arguments and results: The signatures of the visit and accept functions are determined when the visitor and the AST node classes are defined. Passing additional arguments to (or returning a value from) the visit functions requires intermediate storage as part of the visitor class. In the following example, inh and syn correspond to the input and result values of a function.

```
struct AnalysisPassVisitor : Visitor { // ...
  const InheritedAttr & inh; // data member for input parameter
  SynthesizedAttr *     syn; // data member for return value
  Visitor (const InheritedAttr & inherited) : inh(inherited), syn() {}
};
```

To avoid code duplication, it is useful to factor constructing the visitor and reading out the result into separate functions.

```
SynthesizedAttr * visit_foo (const Expr& e, const InheritedAttr & inh) {
  AnalysisVisitor v(inh); // construct the visitor and pass the context
  e.accept(v);
  return v.syn;           // read and return the result
}
```

With open-methods, additional arguments can easily be specified as part of their signatures.

```
SynthesizedAttr* analysisPass(virtual const Expr& e, const InheritedAttr& inh);
```

Covariant return type: Since the result requires intermediate storage, covariant return types cannot easily be implemented with the visitor pattern. Consider the following implementation of a visitor that creates and returns a copy of an AST node.

```
struct CloneExpr : Visitor {
  Expr* result; // data member for return value
  virtual void visit (const Expr& e) { result = new Expr(e); } // make a copy of an Expr object
  virtual void visit (const Unary& u) { result = new Unary(u); }// make a copy of a Unary object
};
Expr* clone(const Expr& e) { // analog of a base−method
  CloneExpr v;
  e.accept(v);
  return v.result ;
}
```

Cloning a unary expression loses some type information, because the cloned objects would be returned as Expr. An implementation that is able to return covariant types requires a different visitor implementation (or instantiation) with similar boilerplate code for each covariant return type. These repetitive definitions can be eliminated by using templates. The following example shows a cloning visitor that returns a Unary object.

```
struct CloneUnary : Visitor  {
  Unary* result ; // data member for covariant return value
  virtual void visit (const Expr& e) { assert (false ); } // Can never be called !
  virtual void visit (const Unary& u) { result = new Unary(u); }
};
Unary* clone(const Unary& u) { // analog of an overrider  with covariant return type
  CloneUnary v;
  u.accept(v);
  return v.result ;
}
```

The definition of open-methods with covariant return types is straightforward:

```
Expr& clone(virtual const Expr& a); // base−method
Unary& clone(virtual const Unary& u); // overrider with covariant return type
```

Similar to open-methods, a hand crafted technique for modeling covariant return type with visitors also creates additional "dispatch tables" for each overrider with covariant return type. Those are created in a form of v-tables for additional visitors. Interestingly enough, the overall size of such dispatch tables would be larger than those generated for open-methods, because visitors require v-table entries for visit methods that can never occur at runtime (see the assert in CloneUnary::visit(Expr&) above). This is not the case with open-methods, as overriders with covariant return type will operate on smaller class hierarchies for their arguments.

Passing open-methods as callbacks: Open-methods nevertheless have disadvantages sometimes in comparison with the visitor pattern. Consider a traversal mechanism that traverses an AST in certain order. The mechanism can accept either a visitor or an open-method for node visitation:

```
void evaluation_order_traversal  (const Expr& e, Visitor & v);
void evaluation_order_traversal  (const Expr& e, void(*fn)(const Expr&));
```

In case of a visitor, we can pass or accumulate some data during visitation. However, in case of open-method, we would need to accumulate data elsewhere.

```
struct CodeGenerationVisitor : Visitor { // ...
  std::vector<Instruction> instructions; // instruction stream inside visitor
  void visit(const Expr& e) { ...} // generate code for e
};
CodeGenerationVisitor v;
evaluation_order_traversal(root_node(),v);
```

Although our current implementation does not support taking the address of an open-method, we can simulate that behavior by wrapping the open-method call inside a thunk.

```
std::vector<Instruction> instructions; // global instruction stream
void generate_code(virtual const Expr& e) { instructions.push_back(...); }
void generate_code(virtual const Unary& e) { instructions.push_back(...); }
//...
void thunk_generate_code(const Expr& e) { generate_code(expr); }
evaluation_order_traversal(root(), &thunk_generate_code);
```

4.9 Discussion

In this chapter, we presented a novel approach to dispatching open multi-methods that is in line with the multiple inheritance semantics of the current C++ object model and the C++ overload resolution rules. This implies compile-time or link-time detection of ambiguities. By considering covariant return types in the ambiguity resolution, we reduce the number of potential conflicts. We have discussed an implementation based on modifications to the EDG compiler front end and have described a mechanism that supports the integration of several translation units. Our evaluation of different approaches to implementing open-methods in C++ shows that our approach is significantly better (in time and space) than current workarounds. Indeed, it is only 16% slower than single dispatch. Since the dispatch is constant time and does not rely on exceptions to signal ambiguities, it is applicable in embedded and hard real-time systems.

4.9.1 Directions for Future Work

There are several possible directions for future work.

4.9.1.1 Virtual Function Templates

Virtual function templates are a powerful abstraction mechanism not part of C++ (see [243, §15.9.3]). Generating v-tables for virtual function templates requires a whole-program view and C++ traditionally relies almost exclusively on separate compilation of translation units. Currently, the concrete semantics of templated virtual functions (and open-methods) remains an open topic.

4.9.1.2 Function Pointers to Open-Methods

Pointers to member functions in C++ preserve polymorphic behavior when they point to a virtual member function. To be in line with this semantics, pointers to open-methods should preserve dynamic dispatch too. This could be implemented by generating a thunk every time an address of an open-method is taken, and using the address of this thunk instead. Inside the function, the compiler simply generates a call to the appropriate open-method. Note that similar to single dispatch in C++, it will not be possible to take the address of a particular open-method overrider – the returned function will always dispatch dynamically.

4.9.1.3 Calling a Base Implementation

C++ provides syntax to call a particular base implementation of a virtual member function directly, avoiding dynamic dispatch. This is often used to call the function in the base class. To do this, C++ requires the user to use a fully qualified name of virtual member function: e.g. p→ MyClass::foo();. It is likely that similar functionality will be required for open-methods.

We propose to be able to fix the dynamic type of an argument at the point of open-method call to a particular unambiguous base class. This can either be done via fix_type⟨Base⟩(arg) or introduce a special syntax, such as: arg as Base. The concrete form is still under discussion. Under this approach users will be able to say foo(d1 as B, d2), which means that the runtime type of d1 is considered to be B rather than its actual runtime type. This effectively fixes the corresponding row or column in the dispatch table during the call. We note that the static type of d1 has to be unambiguously derived from B in order for such type fix to be applicable.

This approach differs from the one used in MultiJava [54], where users have a choice between resend and super calls. resend invokes a less specific implementation that the current overrider refines, providing it can be uniquely determined. In cases when the overrider that invokes resend refines multiple overriders, a compile time error is reported. A call to super dispatches to an implementation for a strict superclass of the receiver object.

4.9.2 Limitations

While open multi-methods provide the most general solution to the expression problem in the context of an object-oriented language, they have a few quirks that may inhibit their introduction into the language and future adoption.

Implementing something as a language feature gives language designers and implementers the advantage of providing a better syntax, proper type checking, improved diagnostics, seamless integration with orthogonal features and superior optimizations. Unfortunately, the history of introducing new features into the language shows they are not always eagerly adopted or even implemented, leading to their possible deprecation. To start from, there might be not enough need in the feature to justify the burden that will be put on language implementations – remember from §3.4 that studies suggest only 2.7%-6.5% of functions utilize multiple dispatch, though it is not clear how much use is there for the open-methods. Even when implemented, there is no guarantee the feature will actually be used. The use of the feature in the newly developed code might be avoided for the lack of proper education and expertise or simply common knowledge of feature's utility. Refactoring of the existing code to use the new feature may be subjected to backward compatibility issues, possibility of introducing new bugs during the transition, effect on other parts of the system, impact on performance etc. The C++ standardization committee is already very busy working on other features, so it may still take years before open multi-methods may even be considered, even though we submitted them to the committee in 2007. In this respect, being a library solution makes it an important advantage of the visitor design pattern, as it can be implemented today on existing compilers.

Modularity of open multi-methods is certainly an advantage in many cases, but in some cases, it may also be seen as inconvenience. Overriders can potentially be distributed across translation units, static and dynamic libraries etc., which inhibits local reasoning about code. True openness of multi-methods to both class and function extensions is important for dealing with the expression problem, however it effectively gets rid of the notion of exhaustiveness and redundancy checking, so useful in case analysis in other languages. Exhaustiveness checking at compile or link time reduces to checking the presence of base-method's implementation. Redundancy checking is replaced with ambiguity errors, which in many cases may be more difficult to resolve than simple reordering of redundant cases. Introduction of overriders whose only purpose is to resolve ambiguities may clutter the code and further complicate its understanding.

5. OPEN TYPE SWITCH[1]

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

Open multi-methods overcome all the deficiencies of the visitor design pattern (§2.7), but inadvertently introduce other quirks, which depending on application may or may not be important (§6.5.1). In this chapter we look at an alternative mechanism to dealing with the expression problem (§2.6), which, similarly, has its pros and cons in different kinds of applications. In particular, we develop an open and efficient type switch construct for classes that allows for local reasoning and a more direct expression of the program's logic than multi-methods and especially visitors do.

To simplify experimentation and gain realistic performance using production-quality compilers and tool chains, we implement our type switch construct as an ISO C++11 library, called *Mach7*[1]. This library-only implementation provides concise notation and superior performance, relying on only a few C++11 features, template meta-programming, and macros. It is somewhat slower than the solution provided by our open multi-methods (§4), but runs about as fast as OCaml and Haskell equivalents, and, depending on the usage scenario, compiler and underlying hardware, comes close or outperforms the handcrafted C++ code based on the *visitor design pattern*. The library is non-intrusive and, similarly to open multi-methods, circumvents most of the extensibility restrictions typical of the visitor design pattern (§2.7).

The content of this chapter is based on the published article [232] and represents the work done in collaboration with Dr. Gabriel Dos Reis and Dr. Stroustrup.

5.1 Introduction

Classic algebraic data types (§2.4) as seen in functional languages are closed and their variants are disjoint, which allows for efficient implementation of case analysis on such types. Data types in object-oriented languages are extensible and hierarchical (implying that variants are not necessarily disjoint). To be general, case analysis on such types must be *open*: allow for independent extensions, modular type checking, and dynamic linking. To be accepted for production code, its implementation must also equal or outperform all known workarounds. Existing approaches to case analysis on hierarchical extensible data types are either efficient or open, but not both. Truly open approaches rely on expensive class-membership tests combined with decision trees. Efficient approaches rely on sealing either the class hierarchy or the set of functions, which loses extensibility (§5.2.3).

Given the expression language from §2.4 we saw in §2.4.1 that in an object-oriented language we can easily introduce new variants, but introduction of new functions required modification of existing code – subject of the expression problem (§2.6). With *Mach7*, we offer an external introspection of objects with case analysis, similar to that performed with pattern matching on algebraic data types in functional

[1] Reprinted with permission from "Open and Efficient Type Switch for C++" by Yuriy Solodkyy, Gabriel Dos Reis, Bjarne Stroustrup, 2012. Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications, Pages 963-982, Copyright 2012 by ACM.

[1] The library is available at http://parasol.tamu.edu/mach7/

languages (§2.5):

```
int eval(const Expr& e)
{
  Match(e)
    Case(const  Value& x) return x.value;
    Case(const   Plus& x) return eval(x.e1) + eval(x.e2);
    Case(const  Minus& x) return eval(x.e1) - eval(x.e2);
    Case(const  Times& x) return eval(x.e1) * eval(x.e2);
    Case(const Divide& x) return eval(x.e1) / eval(x.e2);
  EndMatch
}
```

The ideas and the *Mach7* library were motivated by our unsatisfactory experiences working with various C++ front ends and program analysis frameworks [51,81,158,172]. The problem was not in the frameworks per se, but in the fact that we had to use the *visitor design pattern* [101] to inspect, traverse, and elaborate abstract syntax trees of target languages. We found visitors unsuitable to express application logic directly, surprisingly hard to teach students, and often slower than handcrafted workaround techniques. Instead, users were relying on dynamic casts in many places, often nested, thus preferring shorter, cleaner, and more direct code to visitors. The consequential performance penalty was usually not discovered until later, when it was hard to remedy.

5.1.1 Summary

The contributions of this chapter can be summarized as following:

- A technique for implementing open and efficient type switching on extensible hierarchical data types as seen in object-oriented languages.
- The technique delivers equivalent performance to that of closed algebraic data types, and outperforms the visitor design pattern for open class hierarchies.
- A library implementation provides the notational convenience of functional-language constructs for selecting among types.
- A new approach to type switching that combines subtype tests and type conversion, outperforming approaches that combine subtype tests (even constant-time ones) with decision trees, even over small class hierarchies.
- Complete integration with the existing C++ abstraction mechanisms and object model.
- A constant-time function that partitions a set of objects with the same static type into equivalence classes based on the inheritance path of the static type within the dynamic type.

In particular, our technique for efficient type switching:

- Is open by construction (§5.3.2), non-intrusive, and avoids the control inversion, as typical for visitors.
- Works in the presence of multiple inheritance, both repeated and virtual, as well as in generic code (§5.3.5).
- Comes close to and often significantly outperforms the workaround techniques used in practice, e.g. the visitor design pattern, without sacrificing extensibility (§5.6.1).
- Does not require any changes to the C++ object model or computations at link or load time.
- Can be used in object-oriented languages with object models similar to that of C++.
- Is implemented as a library written in ISO Standard C++11.

This is the first technique for handling type switching efficiently in the presence of the general multiple inheritance present in C++. Being a library, *Mach7* can be used with any ISO C++11 compiler (e.g., Microsoft, GNU, or Clang) without requiring any additional tools or preprocessors. It sets a new threshold for acceptable performance, brevity, clarity, and usefulness of a forthcoming compiler implementation of the open type switch in C++.

5.2 Background

In this section we define what type switch is, look at how it is commonly implemented in object-oriented languages and explain the problems associated with such implementations.

5.2.1 Type Switch

In general, a *type switch* or *typecase* is a multiway branch statement that distinguishes values based on their type. In a multi-paradigm programming language like C++, which supports parametric, ad-hoc, and subtyping polymorphisms, such a broad definition subsumes numerous different typecase constructs studied in the literature [109, 120, 263]. In this work, we only look at typecasing scenarios based on the class inheritance of C++, similar to those studied by Glew [109]. We use the term *type switch* instead of a broader *typecase* to stress the run-time nature of the type analysis similar to how regular **switch** statement of C++ performs case analysis of values at run time.

Given an object descriptor, called *subject*, of static type S referred to as the *subject type*, and a list of *target types* T_i associated with the branches, a type switch statement needs to identify a suitable clause m based on the dynamic type D <: S of the subject as well as a suitable conversion that coerces the subject to the target type T_m. Due to multiple inheritance, types T_i may not all directly derive from the static type S. However, the type of the applicable clause T_m will necessarily have to be a supertype of the subject's dynamic type D <: T_m. A hypothetical type switch statement, not currently supported by C++, may look as following:

switch (subject) { **case** T_1: s_1; ... **case** T_n: s_n; }

There is no need for an explicit *default* clause in our setting because it is semantically equivalent to a case clause guarded by the subject type: **case** S: s. The only semantic difference such a choice makes is in the treatment of null pointers. One may naively think that null pointers should be handled by the default clause. However, not distinguishing between invalid object and valid object of a known static but unknown dynamic type may lead to nasty run-time errors.

Similar control structures exist in many programming languages, e.g. *match* in Scala [195], *case* in Haskell [141] and ML [177], *typecase* in Modula-3 [44] and CLOS (as a macro), *tagcase* in CLU [156], *union case* in Algol 68, and date back to at least Simula's *Inspect* statement [72]. The statement can, in general, be given numerous plausible semantics:

- *first-fit* semantics will evaluate the first statement s_i such that T_i is a base class of D.
- *best-fit* semantics will evaluate the statement corresponding to the most-specialized base class T_i of D if it is unique (subject to ambiguity).
- *exact-fit* semantics will evaluate statement s_i if $T_i = D$.
- *all-fit* semantics will evaluate all statements s_i whose guard type T_i is a supertype of D (order of execution has to be defined).
- *any-fit* semantics might choose non-deterministically one of the statements enabled by the all-fit semantics.

The list is not exhaustive and depending on a language, any of them is a plausible choice. Functional languages, for example, often prefer first-fit semantics because it is similar to case analysis in mathematics. Object-oriented languages are typically inclined to best-fit semantics due to its similarity to overload resolution and run-time dispatch; however, some do opt for first-fit semantics to mimic the functional style: e.g. Scala [195]. Exact-fit semantics can often be seen in languages supporting discriminated union types (sum types): e.g. *variant records* in Pascal, Ada and Modula-2, *oneof* and *variant objects* in CLU, *unions* in C and C++, etc. All-fit and any-fit semantics might be seen in languages based on predicate dispatching [88] or guarded commands [77], where a predicate can be seen as a characteristic function of a type, while logical implication – as subtyping.

5.2.2 Open Type Switch

Type switch alone does not solve the expression problem in the context of an object-oriented language, for the existing code may have to be modified to consider new variants. Relying on a default clause is not an acceptable solution in this situation, because often the only reasonable default behavior is to raise an exception. Zenger and Odersky observed that defaults transform type errors that should manifest statically into runtime exceptions [281]. In our experience, newly added variants were more often extending an existing variant than creating an entirely disjoint one. In a compiler, for example, a new kind of type expression will typically extend a TypeExpression variant, while a new form of annotation will extend an Annotation variant, thus not extending the root ASTNode directly. Due to the substitutability requirement (§2.6,§2.2), this new variant will be treated as a variant it extends in all the existing code. The functions that will be affected by its addition and thus have to be modified will be limited to functions directly analyzing the variant it extends and not providing a default behavior.

To account for this subtlety of extensible hierarchical data types, we use a term *open type switch* to refer to a type switch that satisfies all the requirements of an *open solution to the expression problem* stated in §2.6 except for the *no modification or duplication* requirement. We loosen it to allow modification of functions for which the newly added variant becomes a disjoint (orthogonal) case not handled by a default clause. We believe that the loosened requirement allows us to express pragmatically interesting restrictions that developers are willing to live with. Furthermore, open type switch overcomes all the major shortcomings of the visitor design pattern:

- Case analysis with an open type switch is non-intrusive as it inspects the hierarchy externally and can be applied retroactively.
- New variants can be accounted for in the newly written code and will be seen as a base class or default in the existing code.
- The affected functions are limited to those for which the newly added variant is a disjoint case.
- The code avoids the control inversion and the need for boilerplate code that visitors introduce, and is thus a more direct expression of the intent.

5.2.3 Existing Approaches to Type Case Analysis

The closed nature of algebraic data types allows for their efficient implementation. The traditional compilation scheme assigns unique (and often small and sequential) tags to every variant of the algebraic data type and type switching is then implemented with a multi-way branch [238] (usually a jump table) over all the tags [11]. Dealing with extensible hierarchical data types makes this approach infeasible:

- *Extensibility* implies that the compiler may not know the exact set of all the derived classes until link-time (due to *separate compilation*) or even run-time (due to *dynamic linking*).

- *Substitutability* implies that we should be able to match tags of derived classes against case labels representing tags of base classes.
- The presence of *multiple inheritance* might require pointer adjustments that are not known at compile time (e.g. due to virtual base classes, ambiguous base classes or crosscasting).

There are two main approaches to implementing case analysis on extensible hierarchical data types discussed in the literature.

The first approach is based on either explicit or implicit sealing of the class hierarchy on which type switching can be performed. C++11, for example, allows the user to prohibit further derivation by specifying a class to be "final" [131], similar to Scala and Java. The compiler then may use the above tag allocation over all variants to implement type analysis [87, §4.3.2]. In some cases, the sealing may happen implicitly. For example, languages with both internal and external linkage may employ the fact that classes with internal linkage will not be externally accessible and are thus effectively sealed. While clearly efficient, the approach is not open as it avoids the question rather than answers it.

The broader problem with this approach is that techniques that rely on unique or sequential compile or link-time constants violate *independent extensibility* (§2.6) since without a centralized authority there is no guarantee same constant will not be chosen in a type-unsafe manner by independent extensions. Updating such constants at load time may be too costly even when possible. More often than not, however such updates may require code regeneration since decision trees, lookup tables etc. may have been generated by compiler for given values.

An important practical solution that follows this approach is the visitor design pattern [101]. The set of visit methods in a visitor's interface essentially seals the class hierarchy. Extensions have been proposed in the literature [280], but they have problems of their own, as discussed in §3.5.

The second approach employs type-inclusion tests combined with decision trees [43] to avoid unnecessary checks. Its efficiency is then entirely focused on the efficiency of type-inclusion tests [45, 56, 74, 83, 107, 148, 220, 262, 271, 282].

C++ has handled general dynamic casting since 1987, when multiple inheritance was added to the language [242]. Wirth later presented a technique that can be used to implement subtype tests by traversing a linked list of types [271]. His encoding required little space, but ran in time proportional to the distance between the two types in the class hierarchy. A trivial constant-time type-inclusion test can be implemented with a *binary matrix*, encoding the subtyping relation on the class hierarchy [74]. While efficient in time, it has quadratic space requirements, which makes it expensive for use on large class hierarchies. Cohen proposed the first space-efficient constant-time algorithm, but it can only deal with single inheritance [56]. *Hierarchical encoding* is another constant-time test that maps subtype queries into subset queries on bit-vectors [45, 148]. The approach can handle multiple inheritance, but the space and time required for a subtype test in this encoding increases with the size of the class hierarchy; also, Caseau's approach [45] is limited to class hierarchies that are lattices. Schubert's *relative numbering* [220] encodes each type with an interval $[l, r]$, effectively making type-inclusion tests isomorphic to a simple range checking. The encoding is optimal in space and time, but it is limited to single inheritance. *PQ-Encoding* of Zibin and Gil employs PQ-trees to improve further space and time efficiency of the constant-time type-inclusion testing [282]. While capable of handling type-inclusion queries on hierarchies with multiple inheritance, the approach makes the closed world assumption and can be costly for use with dynamic linking because it is not incremental. The approach of Gibbs and

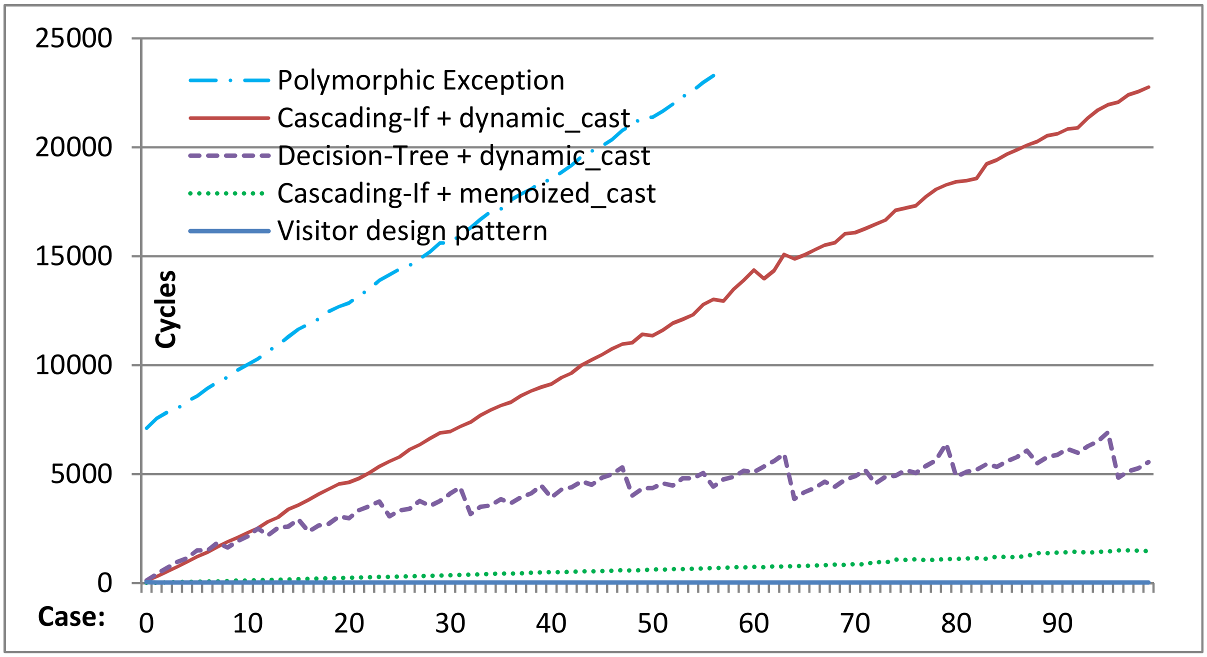


Figure 5.1: Type switching based on naïve techniques

Stroustrup [107] employs divisibility of numbers to obtain a constant-time type-inclusion test. The approach can handle multiple inheritance and was the first constant-time technique to addresses the problem of casts between subobjects. Unfortunately, the approach limits the size of the class hierarchies that can be encoded with this technique. Ducournau proposed a constant-time type-inclusion test based on the fact that, in an open solution, a class has a known number of base classes, and thus perfect hashes can be used to map them to this-pointer offsets typically used to implement subobject casts [83]. Unfortunately, the approach addresses only virtual multiple inheritance and (similarly to other approaches) relies on load-time computations. An excellent introduction to and detailed analysis of existing constant-time type-inclusion tests can be found in [262, 282].

With the exception of work by Gibbs and Stroustrup [107], all the approaches to efficient type-inclusion testing we found in the literature were based on the assumption that *the outcome of a subtyping test as well as the subsequent cast depend only on the target type and the dynamic type of the object.* Although that assumption is sound for subtyping tests and subtype casts for shared inheritance (including single), it does not reflect the relationship between subobjects in the general case of multiple inheritance as found in C++ (§2.3.1).

5.2.4 Source of Inefficiency

Our initial attempts to create a type switch construct were based on several naive approaches, which were extremely inefficient, but which we pursued nevertheless for other useful properties they offered. The approach based on dynamic cast (§5.3.2) was the only truly open solution to type switching; the approach based on exceptions (§5.4) was originally discussed in the functional community [9, 109] as a potential way of implementing type switching on hierarchical datatypes; experiments with memoization of the outcome of dynamic cast (§5.5) gave us the insight into the uniqueness of vtbl-pointers (§5.3.4), but were not sufficient by themselves; when possible we also tried to combine the solutions with a

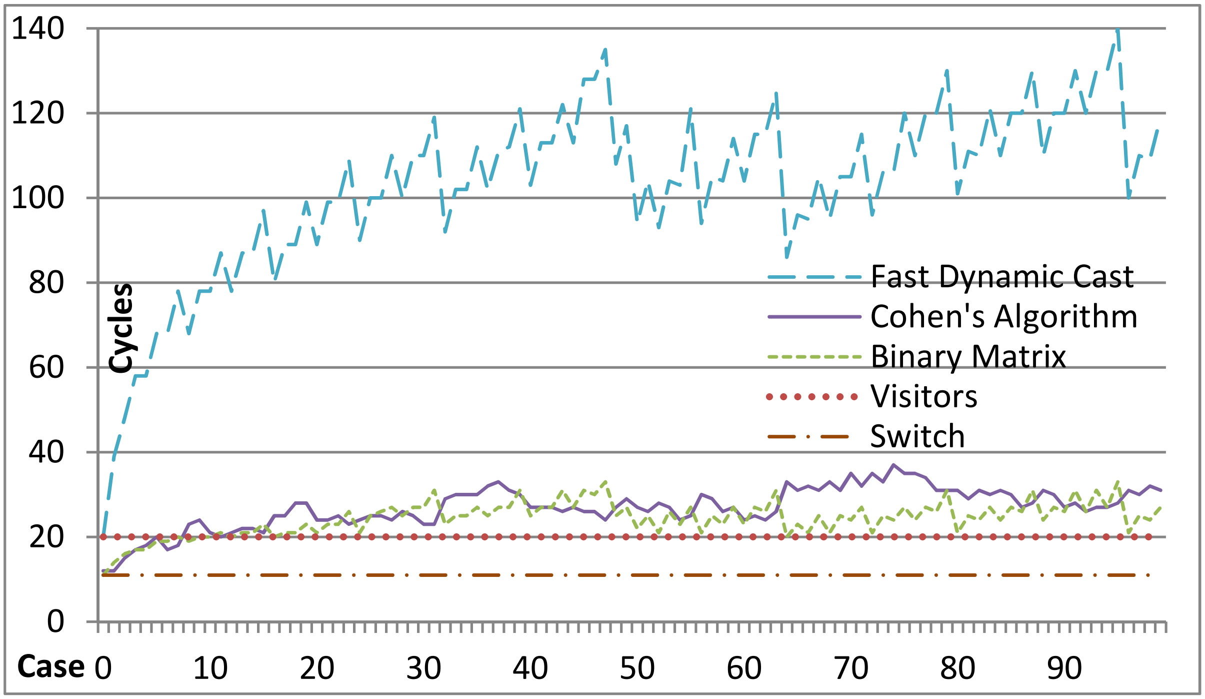


Figure 5.2: Type switch based on constant-time subtype tests

decision tree, that can be generated in a compiler solution, but none of the solutions could compare to the performance of a visitor design patterns for even small number of case clauses. Figure 5.1 summarizes these experiments.

While constant-time type-inclusion tests are invaluable in optimizing subtype tests in programming languages, their use in implementing a type switch is inferior to some workaround techniques. This may prevent wide adoption of a language implementation of such a feature due to its inferior performance. We implemented 3 constant-time type-inclusion tests: binary matrix [262], Cohen's algorithm [56], and fast dynamic cast [107] and combined them with a decision tree to implement a type switch on a class hierarchy ideally suited for such scenarios: a perfect binary tree with classes number $2i$ and $2i+1$ derived from a class number i. Our workaround techniques included the visitor design pattern and a switch on the sealed sequential set of tags.

The chart in Figure 5.2 shows the number of cycles (Y-axis) each technique took to recognize an object of the dynamic type i (X-axis). Despite known limitations, binary matrix and Cohen's algorithm are some of the fastest known type-inclusion tests for single inheritance [262]. It is nonetheless easy to see that the logarithmic cost associated with the decision tree very quickly surpasses the constant overhead of double dispatch (20 cycles) present in the visitor design pattern or the jump-table implementation of the switch on all tags (11 cycles). We expect the cost of techniques capable of handling multiple inheritance to be even higher, especially those addressing casting between subobjects (e.g. fast dynamic cast). The edgy shape of timing results reflects the shape of the class hierarchy used for this experiment.

5.3 Type Switching

While C++ does not have direct support for algebraic data types, we saw in §2.4.1 that they can be encoded with classes in a number of ways. *Mach7* supports to different extents all three encodings. The library handles them differently to let the user choose between openness and efficiency. The type switch for tagged encoding (§5.3.1) is simpler and more efficient for many typical use cases, however, making it open eradicates its performance advantages (§5.6.2). The difference in performance is the price we pay for keeping the solution open. We describe pros and cons of each approach in §5.6.2.

5.3.1 Attractive Non-Solution

While Wirth' linked list encoding was considered slow for subtype testing, it can be adopted for quite efficient type switching on a class hierarchy with no repeated inheritance. The idea is to combine fast switching on closed algebraic datatypes with a loop that tries the tags of base classes when switching on derived tags fails.

For simplicity of presentation we assume a pointer to an array of tags be available directly through the subject's taglist data member. The array is of variable size: its first element is always the tag of the subject's dynamic type, while its end is marked with a dedicated end_of_list marker, distinct from all the tags. The tags in between are topologically sorted according to the subtyping relation with incomparable siblings listed in *local precedence order* – the order of the direct base classes used in the class definition. The list resembles the *class precedence list* of object-oriented descendants of Lisp (e.g. Dylan, Flavors, LOOPS, and CLOS) used there for *linearization* of class hierarchies. We also assume the tag-constant associated with a class D_i is accessible through a static member D_i::class_tag. These simplifications are not essential and the library does not rely on any of them.

A type switch below, built on top of a hierarchy of tagged classes, proceeds as a regular switch on the subject's tag. If the jump succeeds, we found an exact match; otherwise, we get into a default clause that obtains the next tag in the list and jumps back to the beginning of the switch statement for a rematch:

```
    size_t attempt = 0;
    size_t tag = subject→ taglist[attempt];
ReMatch:
    switch (tag) {
    default:
        tag = subject→ taglist[++attempt];
        goto ReMatch;
    case end_of_list:
        break;
    case $D_1$::class_tag:
        $D_1$& match = static_cast⟨$D_1$&⟩(∗subject); $s_1$;
        break;
        . . .
    case $D_n$::class_tag:
        $D_n$& match = static_cast⟨$D_n$&⟩(∗subject); $s_n$;
        break;
    }
```

The above structure, which we call a *tag switch*, implements a variation of *best-fit* semantics based on local precedence order. It lets us dispatch to the case clause of the most-specialized class with an overhead of initializing two local variables, compared to an efficient switch used on algebraic data types. Dispatching to a case clause of a base class will take time roughly proportional to the distance between the matched base class and the derived class in the inheritance graph, thus the technique is not constant. When none of the base class tags was matched, we will necessarily reach the end_of_list marker and exit the loop. The default clause, again, can be implemented with a case clause on the subject type's tag: **case** S::class_tag:

The efficiency of the above code crucially depends on the set of tags being small and sequential to justify the use of a jump table instead of a decision tree to implement the switch. This is usually not a problem in closed hierarchies based on tag encoding since the designer of the hierarchy handpicks the tags herself. The use of a static cast however, essentially limits the use of this mechanism to non-repeated inheritance only. This only refers to the way target classes inherit from the subject type – they can freely inherit from other classes. Due to these restrictions, the technique is not open because it may violate independent extensibility. We discuss in §5.6.2 that making the technique more open will also eradicate its performance advantages.

5.3.2 Open but Inefficient Solution

Instead of starting with an efficient solution and trying to make it open, we start with an open solution and try to make it efficient. The following *cascading-if statement* implements the *first-fit* semantics for our type switch in a truly open fashion:

```
if ($T_1$* match = dynamic_cast⟨$T_1$*⟩(subject)) { $s_1$;} else
if ($T_2$* match = dynamic_cast⟨$T_2$*⟩(subject)) { $s_2$;} else
...
if ($T_n$* match = dynamic_cast⟨$T_n$*⟩(subject)) { $s_n$;}
```

Despite the obvious simplicity, its main drawback is degraded performance: a typical implementation of **dynamic_cast** takes time proportional to the distance between base and derived classes in the inheritance tree. What is worse is that due to the sequential order of tests, the time to uncover the type in the i^{th} case clause will be proportional to i, while failure to match will take the longest. This linear increase can be seen in the Figure 5.1, where the above cascading-if was applied to a flat hierarchy encoding an algebraic data type with 100 variants. The same type-switching functionality implemented with the visitor design pattern took only 28 cycles regardless of the case.[2] This is more than 3 times faster than the 93 cycles it took to uncover even the first case with **dynamic_cast**, while it took 22760 cycles to uncover the last.

Relying on **dynamic_cast** also makes an implicit semantic choice where we are no longer looking for the first/best-fitting type that is in subtyping relation, but for the first/best-fitting type to which a cast is possible from the source subobject (§2.3.1).

5.3.3 Memoization Device

Let us look at a slightly more general problem than type switching. Consider a generalization of the switch statement that takes predicates on a subject as its clauses and executes the first statement s_i whose predicate is enabled:

```
switch (x)
{
   case $P_1$(x): $s_1$;
   ...
   case Pn(x): $s_n$;
```

[2] Each case i was timed multiple times, thus turning the experiment into a repetitive benchmark described in §6.4. In a more realistic setting, represented by random and sequential benchmarks, the cost of double dispatch was varying between 52 and 55 cycles.

}

Assuming that predicates are *functional* (i.e. do not involve any side effects), the next time we execute the switch with the same value x, the same predicate will be enabled first. We thus would like to avoid evaluating preceding predicates and jump to the statement it guards. In a way, we would like the switch to memoize the case enabled for a given x.

The idea is to generate a simple cascading-if statement interleaved with jump targets and instructions that associate the original value with enabled target. The code before the statement looks up whether the association for a given value has already been established, and, if so, jumps directly to the target; otherwise, the sequential execution of the cascading-if is started. To ensure that the actual code associated with the predicates remains unaware of this optimization, the code preceding it after the target must re-establish any invariant guaranteed by sequential execution (§5.3.5).

Described code can be easily produced in a compiler setting, but generating it in a library is a challenge. Inspired by Duff's Device [252], we devised a construct that we call *Memoization Device* doing just that in standard C++:

```
typedef decltype(x) T; // T is the type of subject x
static std::unordered_map⟨T,size_t⟩ jump_targets;

switch (size_t& jump_to = jump_targets[x]) {
default: // entered when we have not seen x yet
    if (P1(x)) { jump_to = 1; case 1: s1;} else
    if (P2(x)) { jump_to = 2; case 2: s2;} else
      ...
    if (Pn(x)) { jump_to = n; case n: sn;} else
                 jump_to = n + 1;
case n + 1: // none of the predicates is true on x
}
```

The static jump_targets hash table will be allocated upon first entry to the function. The map is initially empty and according to its logic, request for a key x not yet in the map will allocate a new entry with its associated data default initialized (to 0 for size_t). Since there is no case label 0 in the switch, the default case will be taken, which, in turn, will initiate sequential execution of the interleaved cascading-if statement. Assignments to jump_to effectively establish association between value x and corresponding predicate, since jump_to is a reference to jump_targets[x]. The last assignment records absence of enabled predicates for x.

The sequential execution of the cascading-if statement will keep checking predicates $P_j(x)$ until the first predicate $P_i(x)$ that returns true. By assigning i to target we will effectively associate i with x since target is just a reference to jump_target_map[x]. This association will make sure that the next time we are called with the value x we will jump directly to the label i. When none of the predicates returns true, we will record it by associating x with $N + 1$, so that the next time we can jump directly to the end of the switch on x.

The above construct effectively gives the entire statement *first-fit* semantics. In order to evaluate all the statements whose predicates are true, and thus give the construct *all-fit* semantics, we might want to be able to preserve the fall-through behavior of the switch. In this case, we can still skip the initial predicates returning false and start from the first successful one. This can be easily achieved by removing all else statements and making **if** statements independent as well as wrapping all assignments to target with a condition, to make sure only the first successful predicate executes it:

```
if (P$_i$i(x)) { if (jump_to ==0) jump_to = $i$; case $i$: s$_i$; }
```

Note that the protocol that has to be maintained by this structure does not depend on the actual values of case labels. We only require them to be different and include a predefined default value. The default clause can be replaced with a case clause for the predefined value, but keeping the default clause generates faster code. A more important consideration is to keep the values close to each other. Not following this rule might result in a compiler choosing a decision tree over a jump table implementation of the switch, which in our experience significantly degrades the performance.

The *first-fit* semantics is not an inherent property of the memoization device. Assuming that the conditions are either mutually exclusive or imply one another, we can build a decision-tree-based memoization device that will effectively have *most-specific* semantics – an analog of *best-fit* semantics in predicate dispatching [88].

Imagine that the predicates with the numbers $2i$ and $2i+1$ are mutually exclusive and each imply the value of the predicate with number i, i.e. $\forall i \forall x \in \bigcap_j \mathsf{Domain}(P_j).P_{2i+1}(x) \rightarrow P_i(x) \wedge P_{2i}(x) \rightarrow P_i(x) \wedge \neg(P_{2i+1}(x) \wedge P_{2i}(x))$ holds. Examples of such predicates are class membership tests where the truth of testing membership in a derived class implies the truth of testing membership in its base class.

The following decision-tree-based memoization device will execute the statement s_i associated with the *most-specific* predicate P_i (i.e. the predicate that implies all other predicates true on x) that evaluates to true or will skip the entire statement if none of the predicates is true on x.

```
switch (size_t& jump_to = jump_targets[x]) {
default:
    if ($P_1$(x)) {
        if ($P_2$(x)) {
            if ($P_4$(x)) { jump_to = 4; case 4: $s_4$;} else
            if ($P_5$(x)) { jump_to = 5; case 5: $s_5$;}
            jump_to = 2; case 2: $s_2$;
        } else
        if ($P_3$(x)) {
            if ($P_6$(x)) { jump_to = 6; case 6: $s_6$;} else
            if ($P_7$(x)) { jump_to = 7; case 7: $s_7$;}
            jump_to = 3; case 3: $s_3$;
        }
        jump_to = 1; case 1: $s_1$;
    } else { jump_to = 0; case 0: ; }
}
```

An example of predicates that satisfy this condition are class membership tests where the truth of a predicate that tests membership in a derived class implies the truth of a predicate that tests membership in its base class. Our library solution prefers the simpler cascading-if approach only because the necessary code structure can be laid out with macros. A compiler solution will use the decision-tree approach whenever possible to lower the cost of the first match from linear in case's number to logarithmic.

The main advantage of the memoization device is that it can be built around almost any code, providing that we can re-establish the invariants guaranteed by sequential execution. Its main disadvantage is the size of the hash table that grows proportionally to the number of different values seen. Fortunately, the values can often be grouped into equivalence classes that do not change the outcome of the predicates. The map can then associate the equivalence class of a value with a target instead of associating the value with it.

In application to type switching, the idea is to use the memoization device to learn the outcomes of type-inclusion tests (with **dynamic_cast** used as a predicate). The objects can be grouped into equivalence classes based on their dynamic type: the outcome of each type-inclusion test will be the same on all the objects of the same dynamic type. We can use the address of a class' **type_info** object obtained in constant time with the **typeid**() operator as a unique identifier of each dynamic type. Presence of multiple **type_info** objects for the same class, as is often the case when dynamic linking is involved, is not a problem, as it would effectively split a single equivalence class into multiple ones.

This could have been a solution if we were only interested in class membership. More often than not, however, we will be interested in obtaining a reference to the target type of the subject, and we saw in §2.3.1 that the cast between the source and target subobjects depends on the position of the source subobject in the dynamic type's subobject graph. We thus would like to have different equivalence classes for different subobjects.

5.3.4 Uniqueness of Vtbl-Pointers Under Common Vendor ABI

Given a reference a to polymorphic type A that points to a subobject σ of the dynamic type C (i.e. $C \prec \sigma \succ A$ is true), we will use the traditional field access notation a.vtbl to refer to the virtual table of that subobject. The exact structure of the virtual table as mandated by the common vendor C++ ABI is immaterial for this discussion, but we mention a few fields that are important for the reasoning [55, §2.5.2]:

- rtti(a.vtbl): the *typeinfo pointer* points to the typeinfo object used for RTTI. It is always present and is shown as the first field to the left of any vtbl-pointer in Figure 2.3(1).
- off2top(a.vtbl): the *offset to top* holds the displacement to the top of the object from the location within the object of the vtbl-pointer that addresses this virtual table. It is always present and is shown as the second field to the left of any vtbl-pointer in Figure 2.3(1). The numeric value shown indicates the actual offset based on the object layout from Figure 2.2(1).
- vbase(a.vtbl): *virtual base offsets* are used to access the virtual bases of an object. Such an entry is required for each virtual base class. None is shown in our example in Figure 2.3(1) since it discusses repeated inheritance, but they will occupy further entries to the left of the vtbl-pointer when present.

We also use the notation *offset*(σ) to refer to the offset of a given subobject σ within C, known by the compiler.

Theorem 2. *In an object layout that adheres to the common vendor C++ ABI with RTTI enabled, equality of vtbl-pointers of two objects of the same static type implies that they both belong to subobjects with the same inheritance path in the same dynamic class.*
$\forall a_1, a_2 : A \mid a_1 \in C_1 \prec \sigma_1 \succ A \wedge a_2 \in C_2 \prec \sigma_2 \succ A$
$a_1.vtbl = a_2.vtbl \Rightarrow C_1 = C_2 \wedge \sigma_1 = \sigma_2$

Proof. Let us assume first $a_1.vtbl = a_2.vtbl$ but $C_1 \neq C_2$. In this case we have rtti($a_1.vtbl$) =rtti($a_2.vtbl$). By definition rtti($a_1.vtbl$) = C_1 while rtti($a_2.vtbl$) = C_2, which contradicts that $C_1 \neq C_2$. Thus $C_1 = C_2 = C$.

Let us assume now that $a_1.vtbl = a_2.vtbl$ but $\sigma_1 \neq \sigma_2$. Let $\sigma_1 = (h_1, l_1), \sigma_2 = (h_2, l_2)$

If $h_1 \neq h_2$ then one of them refers to a virtual base while the other to a repeated one. Assuming h_1 refers to a virtual base, vbase($a_1.vtbl$) has to be defined inside the vtable according to the ABI, while vbase($a_2.vtbl$) – should not. This would contradict again that both $vtbl$ refer to the same virtual table.

We thus have $h_1 = h_2 = h$. If $h = \mathsf{Shared}$ then there is only one path to such A in C, which would contradict $\sigma_1 \neq \sigma_2$. If $h = \mathsf{Repeated}$ then we must have that $l_1 \neq l_2$. In this case let k be the first position in which they differ: $\forall j < k.l_1^j = l_2^j \wedge l_1^k \neq l_2^k$. Since our class A is a base class for classes l_1^k and l_2^k, both of which are in turn base classes of C, the object identity requirement of C++ requires that the relevant subobjects of type A have different offsets within class C: $\mathit{offset}(\sigma_1) \neq \mathit{offset}(\sigma_2)$ However $\mathit{offset}(\sigma_1) = \mathsf{off2top}(a_1.\mathit{vtbl}) = \mathsf{off2top}(a_2.\mathit{vtbl}) = \mathit{offset}(\sigma_2)$ since $a_1.\mathit{vtbl} = a_2.\mathit{vtbl}$, which contradicts that the offsets are different. □

Conjecture in the other direction is not true in general as there may be duplicate vtables for the same type present at run-time. This happens in many C++ implementations in the presence of *Dynamically Linked Libraries* (or DLLs for short) as the same class compiled into executable and DLL it loads may have identical vtables inside the executable's and DLL's binaries.

Note also that we require both static types to be the same. Dropping this requirement and saying that equality of vtbl-pointers also implies equality of static types is not true in general because a derived class can share a vtbl-pointer with its primary base class. The theorem can be reformulated, however, stating that one subobject will necessarily have to contain the other, but that would require bringing in the formalism for subobject containment [268]. The current formulation is sufficient for our purposes.

During construction and deconstruction of an object, the value of a given vtbl-pointer may change. In particular, that value will reflect the fact that the dynamic type of the object is the type of its fully constructed part only. This does not affect our reasoning, as during such transition we also treat the object to have the type of its fully constructed base only. Such interpretation is in line with the C++ semantics for virtual function calls and the use of RTTI during construction and destruction of an object. Once the complete object is fully constructed, the value of the vtbl-pointer will remain the same for the lifetime of the object.

5.3.5 Vtable Pointer Memoization

The C++ standard implies that information about types is available at run time for three distinct purposes [55, §2.9.1]:

- to support the **typeid** operator,
- to match an exception handler with a thrown object, and
- to implement the **dynamic_cast** operator.

and if any of these facilities are used in a program that was compiled with RTTI disabled, the compiler shall emit a warning. Some compilers (e.g. Visual C++) additionally let a library check presence of RTTI through a predefined macro, thus letting it report an error if its dependence on RTTI cannot be satisfied. Since our solution depends on **dynamic_cast**, according to the third requirement we implicitly rely on the presence of RTTI and thus fall into the setting that guarantees the preconditions of Theorem 2. Besides, all the objects that will be coming through a particular type switch will have the same static type, and thus the theorem guarantees that different vtbl-pointers will correspond to different subobjects. The idea is thus to group them according to the value of their vtbl-pointer and associate both jump target and the required offset through the memoization device:

```
typedef pair⟨ptrdiff_t,size_t⟩ target_info; //(offset,target)
static unordered_map⟨intptr_t, target_info⟩ jump_targets;
      auto* sptr = &x; // name to access subject
const void* tptr;
```

```
target_info& info = jump_targets[vtbl(sptr)];
switch (info.second) {{ default:
```

We use the virtual table pointer extracted from a polymorphic object pointed to by p as a key for association. The value stored along the key in association now keeps both: the target for the switch as well as a memoized offset for dynamic cast.

The code for the i^{th} case now evaluates the required offset on the first entry and associates it and the target with the vtbl-pointer of the subject. The call to adjust_ptr⟨T$_i$⟩ re-establishes the invariant that match is a reference to type T$_i$ of the subject x.

```
  if (tptr = dynamic_cast⟨const Ti *⟩(sptr)) {
        if (info.second ==0) { // supports fall−through
            info.first = intptr_t(tptr)−intptr_t(sptr); // offset
            info.second = i; // jump target
        }
  case i: // i is a constant − clause's position in switch
    auto match = adjust_ptr⟨Ti⟩(sptr,info.first);
        si;
    }
```

Class std::unordered_map provides amortized constant time access on average and linear in the number of elements in the worst case. We show in the next section that most of the time we will be bypassing traditional access to its elements. We need this extra optimization because, as-is, the type switch is still about 50% slower than the visitor design pattern. [3]

Looking back at the example from §6.1 and allowing for a few unimportant omissions, the first code snippet corresponds to what the macro *Match*(x) is expanded to when given a subject expression x. In order to see what *Case*(T$_i$) is expanded to, the second snippet has to be split on the line containing s$_i$; (excluding s$_i$; itself, which comes from source) and the second part (i.e. } here) moved in front of the first one. The macro thus closes the scope of the previous case clause before starting the new one. *Case*'s expansion only relies on names introduced by *Match*(x), its argument T$_i$, and a constant i, which can be generated from the *__LINE__* macro, or, better yet, the *__COUNTER__* macro when supported by the compiler. The *EndMatch* macro simply closes the scopes (i.e. }} here). We refer the reader to the library source code for further details.

5.3.6 Structure of Virtual Table Pointers

Virtual table pointers are not entirely random addresses in memory and have certain structure when we look at groups of those that are associated with classes related by inheritance. Let us first look at some vtbl-pointers that were present in some of our tests. The 32-bit pointers are shown in binary form (lower bits on the right) and are sorted in ascending order:

```
0000000100111111000000011001001000
0000000100111111000000011001011100
0000000100111111000000011001110000
 ...
0000000100111111000000011111011000
0000000100111111000000011111101100
```

Virtual table pointers are not constant values and are not even guaranteed to be the same between different runs of the same application. Techniques like *address space layout randomization* or simple

[3] We are using speedups throughout the thesis when comparing performance, so "X is 42% faster than Y" and "Y is 42% slower than X" both mean that Y's execution time is 1.42 times X's execution time.

rebasing of the entire module are likely to change these values. The relative distance between them is likely to remain the same, though, as long as they come from the same module.

Comparing all the vtbl-pointers that are coming through a given *Match* statement, we can trace at run time the set of bits in which they do and do not differ. For the above example, it may look as 00000001001111100000X11XXXXXXX00 where positions marked with X represent bits that are different in some vtbl pointers.

When a DLL is loaded, it may have its own copy of vtables for classes also used in other modules as well as vtables for classes it introduces. Similarly comparing all vtbl-pointers coming only from this DLL, we can get a different pattern 01110011100000010111XXXXXXXXX000 and when compared over all the loaded modules the pattern will likely become something like 0XXX00X1X0XXXXXXX0XXXXXXXXXXXX00.

The common bits on the right come from the virtual table size and alignment requirements, and, depending on compiler, configuration, and class hierarchy could easily vary from 2 to 6 bits. Because the vtbl-pointer under the C++ ABI points into an array of function pointers, the alignment requirement of 4 bytes for those pointers on a 32-bit architecture is what causes at least the last 2 bits to be 0. For our purpose, the exact number of bits on the right is not important, as we evaluate this number at run time based on the vtbl-pointers seen so far. Here we only point out that there would be some number of common bits on the right.

Another observation we made during our experiments with the vtbl-pointers of various existing applications was that the values of the pointers changed more frequently in the lower bits than in the higher ones. We believe this happened because programmers tend to group multiple derived classes in the same translation unit so the compiler was emitting virtual tables for them close to each other as well.

Note that derived classes that do not introduce their own virtual functions (even if they override some existing ones) are likely to have virtual tables of the same size as their base class. Even when they do add new virtual functions, the size of their virtual tables can only increase relative to their base classes. This is why the difference between the addresses of many consecutive vtables that came through a given *Match* statement was usually constant or very slightly different.

The changes in higher bits were typically due to separate compilation and especially due to dynamically loaded modules. When a DLL is loaded, it may have its own copies of vtables for classes that are also used in other modules, in addition to vtables for classes it introduces. Comparing all vtbl-pointers coming only from that DLL, we can get a different pattern 01110011100000010111XXXXXXXXX000; when compared over all the loaded modules, the pattern will likely become something like 0XXX00X1X0XXXXXXX0XXXXXXXXXXXX00. Overall, they were not changing the general tendency we saw: smaller bits were changing more frequently than larger ones, with the exception of the lowest common bits, of course.

These observations made virtual table pointers of classes related by inheritance ideally suitable for indexing – the values obtained by throwing away the common bits on the right were compactly distributed in small disjoint ranges. We use those values to address a cache built on top of the hash table in order to eliminate a hash table lookup in most of the cases. The important guarantee about the validity of the cached hash table references comes from the C++11 standard, which states that "insert and emplace members shall not affect the validity of references to container elements" [131, §unord.req(13)].

Depending on the number of actual collisions that happen in the cache, our vtable pointer memoization technique can approach the performance of – and even outperform – the visitor design pattern.

The numbers are, of course, averaged over many runs, as the first run on every vtbl-pointer will take an amount of time as shown in Figure 5.1. We did however test our technique on real code and can confirm that it does perform well in real-world use cases.

The information about jump targets and necessary offsets is just an example of the information we might want to be able to associate with – and access via vtbl-pointers. Our implementation of memoized_cast from §5.5, for example, effectively reuses this general data structure with a different type of element value. The separation of the data structure into hash table and reconfigureable cache is not important and was only used here to ease understanding. The important part is that the data structure must guarantee the validity of references of its elements as well as provide an amorthized-constant-time access to them. Unlike generic unordered maps, the data structure assumes that its keys are vtbl-pointers that can only be inserted or used to access associated value. It also implements the logic presented below to automatically reconfigure the parameters in order to minimize the number of conflicts. Our current implementation of this molithic self-tuning unordered map is provided by a generic class vtblmap⟨T⟩.

5.3.7 Minimization of Conflicts

The small number of cycles that the visitor design pattern uses to uncover a type does not let us put overly-sophisticated cache indexing mechanisms into the critical path of execution if we want to be able to compete with it. This is why we limit our indexing function to shifts and masking operations as well as choose the size of the cache to be a power of 2.

Let Ξ be the domain of integral representations of pointers. Given a cache with 2^k entries, we use a family of hash functions $H_{kl} : \Xi \to [0..2^k - 1]$ defined as $H_{kl}(v) = \lfloor \frac{v}{2^l} \rfloor \mod 2^k$ to index the cache, where $l \in [0..32]$ (assuming 32-bit addresses) is a parameter modeling the number of common bits on the right. Integer division and modulo are implemented with bit operations since the denominator in each case is a power of 2, which in turn explains the choice of the cache size.

Given a hash function H_{kl}, pointers v' and v'' are said to be *in conflict* when $H_{kl}(v') = H_{kl}(v'')$. For a given set of pointers $V \in 2^{\Xi}$, we can always find such k and l that H_{kl} will render no conflicts between its elements, but the required cache size 2^k can be too large to justify the use of memory. The value K such that $2^{K-1} < |V| \leq 2^K$ is the smallest value of k under which absence of conflicts is still possible. We thus allow k to vary only in the range $[K, K+1]$ to ensure that the cache size is never more than 4 times bigger than the minimum required cache size.

Given a set $V = \{v_1, ..., v_n\}$, we would like to find a pair of parameters (k, l) such that H_{kl} will render the least number of conflicts on the elements of V. Since for a fixed set V, the parameters k and l vary over a finite range, we can always find the optimal (k, l) by trying all the combinations. Let $H^V_{kl} : V \to [0..2^k - 1]$ be the hash function corresponding to such optimal (k, l) for the set V.

In our setting, the set V represents the set of vtbl-pointers coming through a particular type switch. While the exact values of these pointers are not known until run time, their offsets from the module's base address are known at link time. This is generally sufficient to estimate the optimal k and l in a compiler setting. In the library setting, we recompute them after a given number of actual collisions in vtblmap's cache. By *collision*, we call a run-time condition in which the cache entry of an incoming vtbl-pointer is occupied by another vtbl-pointer. Presence of conflict does not necessarily imply presence of collisions, but collisions can only happen when there is a conflict.

When H^V_{kl} is injective (renders 0 conflicts on V), the frequency of any given vtbl-pointer v_i coming through the type switch does not affect the overall performance of the switch. However when H^V_{kl} is not injective, we would prefer the conflict to happen on less frequent vtbl-pointers.

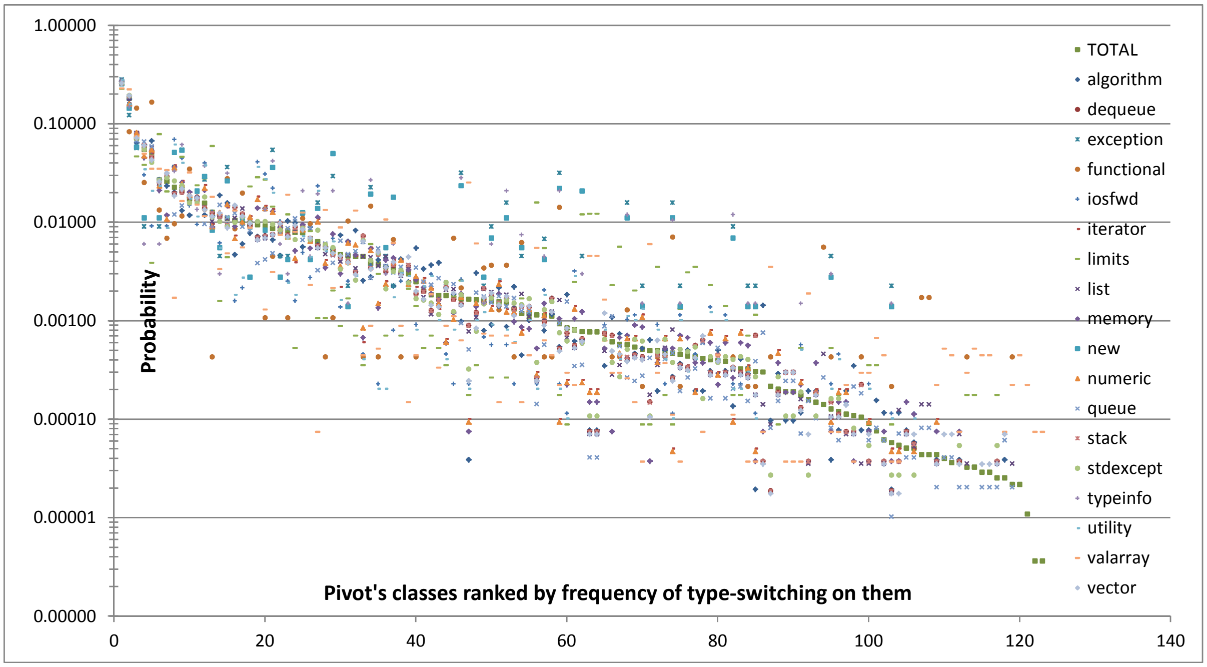


Figure 5.3: Probability distribution of various nodes in the Pivot framework

We studied the frequency of vtbl-pointers that come through various *Match* statements of a C++ pretty-printer that we implemented on top of the Pivot framework [81] using our type switch (see §6.4.3). We ran the pretty-printer on a set of C++ standard library headers and then ranked all the IPR classes from the most-frequent to the least-frequent ones in the header's AST. The resulting probability distribution is shown with a thick line in Figure 5.3.

Note that the Y-Axis in Figure 5.3 uses a logarithmic scale, suggesting that the resulting probability has a power-law distribution. This is likely to be specific to our application; nevertheless, Figure 5.3 demonstrates that the frequency of certain classes can be greater than the combined frequency of all the other classes. In our case, the two most-frequent classes represented the use of a variable in a program, and their combined frequency was greater than the frequency of all the other nodes combined. Naturally, we would like to avoid conflicts on such classes in the vtblmap's cache when possible.

Given a probability $p(v_i)$ of each vtbl-pointer $v_i \in V$ we can compute the probability of conflict rendered by a given H_{kl}:

$$p_{kl}(V) = \sum_{j=0}^{2^k-1} \left(\sum_{v_i \in V_{kl}^j} p(v_i) \right) \left(1 - \frac{\sum_{v_i \in V_{kl}^j} p(v_i)^2}{\left(\sum_{v_i \in V_{kl}^j} p(v_i) \right)^2} \right)$$

where $V_{kl}^j = \{v \in V \mid H_{kl}(v) = j\}$. In this case, the optimal hash function H_{kl}^V can similarly be defined as the H_{kl} that minimizes the above probability of conflict on V.

To calculate the probability of conflict for given l and k parameters, let us consider the j^{th} cache cell and a subset $V_{lk}^j = \{v \in V \mid f_{lk}(v) = j\}$. When the size of this subset $m = |V_{lk}^j|$ is greater than 1, we have a potential conflict, as a subsequent request for a vtbl-pointer v'' might be different from the vtbl-pointer v' currently stored in the cell j. Within the cell, the probability of not having a conflict is the probability of both values v'' and v' being the same:

$$P\left(v''=v'\right)=\sum_{v_i\in V_{lk}^j}P\left(v''=v_i\right)P\left(v'=v_i\right)=\sum_{v_i\in V_{lk}^j}P^2\left(v_i|V_{lk}^j\right)=$$

$$=\sum_{v_i\in V_{lk}^j}\frac{P^2\left(v_i\right)}{P^2\left(V_{lk}^j\right)}=\sum_{v_i\in V_{lk}^j}\frac{p_i^2}{\left(\sum\limits_{v_{i'}\in V_{lk}^j}p_{i'}\right)^2}=\frac{\sum\limits_{v_i\in V_{lk}^j}p_i^2}{\left(\sum\limits_{v_i\in V_{lk}^j}p_i\right)^2}$$

The probability of having a conflict among the vtbl-pointers of a given cell is thus one minus the above value:

$$P\left(v''\neq v'\right)=1-\frac{\sum\limits_{v_i\in V_{lk}^j}p_i^2}{\left(\sum\limits_{v_i\in V_{lk}^j}p_i\right)^2}$$

To obtain the probability of conflict given any vtbl-pointer and not just the one from a given cell, we need to sum up the above probabilities of conflict within a cell multiplied by the probability of a vtbl-pointer falling into that cell:

$$P_{lk}^{conflict}=\sum_{j=0}^{2^k-1}P\left(V_{lk}^j\right)\left(1-\frac{\sum\limits_{v_i\in V_{lk}^j}p_i^2}{\left(\sum\limits_{v_i\in V_{lk}^j}p_i\right)^2}\right)=\sum_{j=0}^{2^k-1}\left(\sum_{v_i\in V_{lk}^j}p_i\right)\left(1-\frac{\sum\limits_{v_i\in V_{lk}^j}p_i^2}{\left(\sum\limits_{v_i\in V_{lk}^j}p_i\right)^2}\right)$$

Our reconfiguration algorithm then iterates over all possible values of l and k and chooses those that minimize the overall probability of conflict $P_{lk}^{conflict}$. The only data still missing are the actual probabilities p_i used by the above formula. They can be approximated in many different ways.

Besides showing the probability distribution on all the tests, Figure 5.3 also shows the probabilities of a given node on each of the tests. The X-axis in this case represents the ordering of all the nodes according to their overall rank in all the tests combined. As can be seen from the picture, the shape of each specific test's distribution still mimics the overall probability distribution. With this in mind, we can simply let the user assign probabilities to each of the classes in the hierarchy and use these values during reconfiguration. The practical problem we found with this solution was that we wanted these probabilities to be inheritable as Pivot separates interface and implementation classes and we preferred letting the user define them on interface rather than on implementation classes. The easiest way to do so was to write a dedicated function that would return the probabilities using a *Match* statement. Unfortunately, such a function would introduce a lot of overhead as it will only be used a few times (since we try to minimize the number of reconfigurations) and thus will not be able to benefit from memoized runs, having to resort to slow sequential type checks.

A simpler and likely more precise way of estimating p_i would be to count the frequencies of each vtbl-pointer directly inside the vtblmap. This introduces an overhead of an increment into the critical path

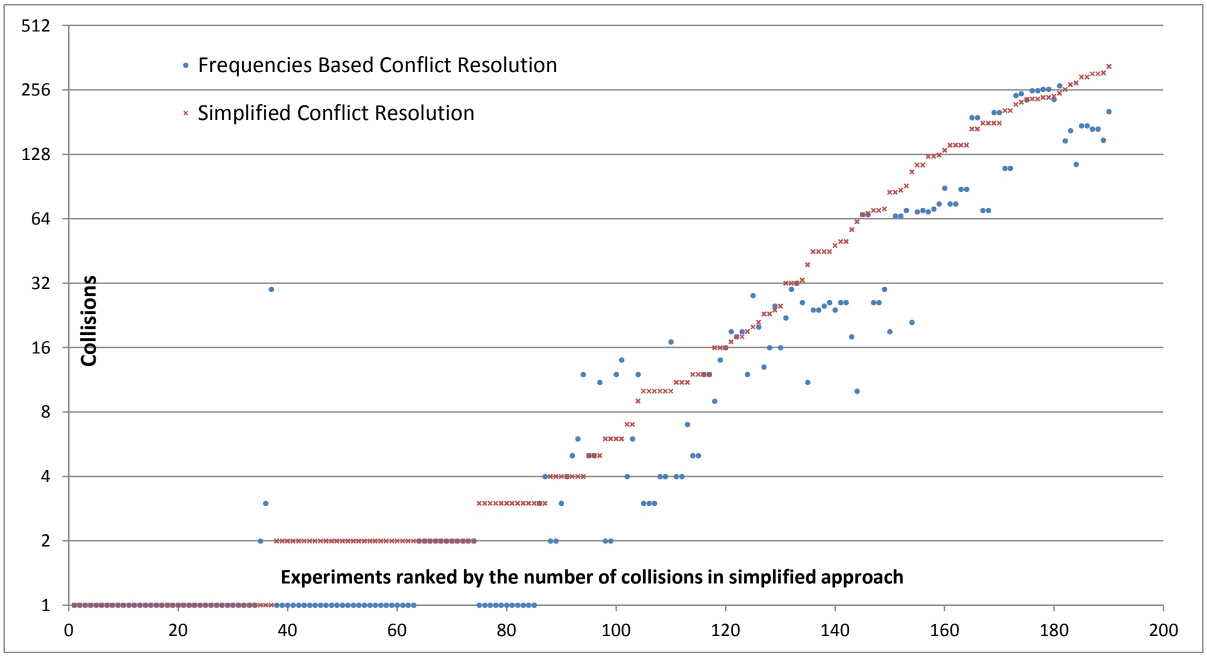


Figure 5.4: Decrease in number of collisions when probabilities of nodes are taken into account

of execution, but according to our tests this only degrades the overall performance by 1-2%. Instead, it was compensating with a smaller number of conflicts and thus a potential gain of performance. We leave the choice of whether the library should count frequencies of each vtbl-pointer to the user of the library as the concrete choice may be to advantage on some class hierarchies and to disadvantage on others.

Figure 5.4 compares the number of collisions when frequency information is used versus when it is not used. The data was gathered from 312 tests on multiple *Match* statements present in Pivot's C++ pretty-printer when it was run over standard library headers. In 122 of these tests, both schemes had 0 conflicts; these tests are thus not shown on the graph. The remaining tests were ranked by the number of conflicts in the scheme that does not utilize frequency information.

As can be seen from the graph, both schemes render very few collisions given that there were about 57000 calls in the rightmost test, which had the largest number of conflicts. Taking into account that the Y-axis has logarithmic scale, the use of frequency information in many cases decreased the number of conflicts by a factor of 2. The handful of cases where the use of frequency information increased the number of conflicts can be explained by the fact that the optimal values are not recomputed after each conflict, but after several conflicts and only if the number of vtbl-pointers in the vtblmap increased. These extra conditions sacrifice the optimality of parameters at any given time for the number of times they are recomputed. By varying the number of conflicts we are willing to tolerate before reconfiguration, we can decrease the number of conflicts by increasing the number of recomputations and vice-versa. In our experience, however, we saw that the drop in the number of conflicts does not translate into a proportional drop in execution time, while the number of reconfigurations is proportional to an increase in execution time. This is why we chose to tolerate a relatively large number of conflicts before triggering recomputation, just to keep the number of recomputations low.

Assuming uniform distribution of v_i in V and substituting the probability $p(v_i) = \frac{1}{n}$, where $n = |V|$, into the above formula we get:

$$p_{kl}(V) = \sum_{j=0}^{2^k-1} [|V_{kl}^j| \neq 0] \frac{|V_{kl}^j| - 1}{n}$$

We use the Iverson bracket $[\pi]$ here to refer to the outcome of a predicate π as 0 or 1. The value $|V_{kl}^j|$ represents the number of vtbl-pointers $v_i \in V$ that are mapped to the same location j in cache with H_{kl}^V. Only one such vtbl-pointer will actually be present in that cache location at any given time, which is why the value $|V_{kl}^j| - 1$ represents the number of "extra" pointers mapped into the entry j on which a collision will happen. The overall probability of conflict thus only depends on the total number of these "extra" or conflicting vtbl-pointers. The H_{kl}^V obtained by minimization of the probability of conflict under uniform distribution of v_i in V is thus the same as the original H_{kl}^V that minimizes the number of conflicts. An important observation here is that since the exact location of these "extra" vtbl-pointers is not important and only the total number m is, the probability of conflict under uniform distribution of v_i in V is always going to be of the discrete form $\frac{m}{n}$, where $0 \leq m < n$.

5.3.8 Multi-Argument Type Switching

We considered two different approaches to extending our solution to efficient type switching to N arguments. The first approach was based on maintaining an N-dimensional table indexed by independent $H_{k_i l_i}^{V_i}$ maintained for each of the arguments i. The second approach was to aggregate the information from multiple vtbl-pointers into a single hash in the hope that the hashing would still maintain its favorable properties. The first approach requires an amount of memory proportional to $O(|V|^N)$ regardless of how many different combinations of vtbl-pointers came through the statement. The second approach requires an amount of memory linear in the number of vtbl-pointer combinations seen, which in the worst case becomes the same $O(|V|^N)$. The first approach requires lookup in N caches, with each lookup being subject to potential collisions; the second approach requires non-trivial computations to aggregate N vtbl-pointers into a single hash value and may result in more collisions in comparison to the first approach. Our experience of dealing with multiple dispatch in C++ suggests that we rarely see all combinations of types coming through a given multi-method in real-world applications. With this in mind, we did not expect all combination of types to come through a given *Match* statement and thus preferred the second solution, which grows linearly in memory with the number of combinations seen.

In number theory, *Morton order* (aka *Z-order*) is a function that maps multidimensional data to one dimension while preserving locality of the data points [187]. A Morton number of an N-dimensional coordinate point is obtained by interleaving the binary representations of all coordinates. The original one-dimensional hash function H_{kl}^V applied to arguments $v \in V$ produces hash values in a tight range $[0..2^k[$ where $k \in [K, K+1]$ for $2^{K-1} < |V| \leq 2^K$. The produced values are close to each other, which helps improve the performance of vtblmap's cache due to locality of reference. The idea is to use Morton order on these hash values and not on the original vtbl-pointers in order to maintain locality of reference. To do this, we still maintain a single parameter k reflecting the size of cache, but we also keep N parameters l_i: an optimal offset for each argument i.

Consider a set $V^N = \{\langle v_1^1, ..., v_1^N \rangle, ..., \langle v_n^1, ..., v_n^N \rangle\}$ of N-dimensional tuples representing the set of vtbl-pointer combinations coming through a given *Match* statement. As with the one-dimensional case, we restrict the size 2^k of the cache to be not larger than twice the closest power of two greater than or equal to $n = |V^N|$: i.e. $k \in [K, K+1]$, where $2^{K-1} < |V^N| \leq 2^K$. For a given k and offsets $l_1, ..., l_N$

a hash value of a given combination $\langle v^1, ..., v^N \rangle$ is defined as $H_{kl_1...l_N}(\langle v^1, ..., v^N \rangle) = \mu\left(\lfloor \frac{v^1}{2^{l_1}} \rfloor, ..., \lfloor \frac{v^N}{2^{l_N}} \rfloor\right)$ mod 2^k, where the function μ returns a Morton number (bit interleaving) of N numbers.

Similarly to the one-dimensional case, we vary parameters $k, l_1, ..., l_N$ in their finite and small domains to obtain an optimal hash function $H^{V^N}_{kl_1...l_N}$ by minimizing the probability of conflict on values from V^N. Unlike the one-dimensional case, we do not try to find the optimal parameters every time we reconfigure the cache. Instead, we only try to improve the parameters to render fewer conflicts in comparison to the number of conflicts rendered by the current configuration. This does not prevent us from converging to the same optimal parameters, which we do over time, but is important for maintaining the amortized-constant-time complexity of access. Observe that the domain of each parameter of the optimal hash function $H^{V^N}_{kl_1...l_N}$ only grows since V^N only grows, while any cache configuration is also a valid cache configuration in a larger cache, rendering the same number of conflicts. We demonstrate in §5.6.5 that similarly to the one-dimensional case, such a hash function produces few collisions on real-world class hierarchies, while it is simple enough to compute to make it competitive with the performance of alternatives dealing with multiple dispatch.

5.4 (Ab)using Exceptions for Type Switching

Several authors have noted the relationship between exception handling and type switching [9, 109]. Unsurprisingly, the exception handling mechanism of C++ can be abused to implement *first-fit* semantics of a type switch statement. The idea is to harness the fact that catch handlers in C++ essentially use first-fit semantics to decide which one is going to handle a given exception. The only problem is to raise an exception with a static type equal to the dynamic type of the subject.

To do this, we employ the *polymorphic exception* idiom [247] that introduces a virtual function **virtual void** raise() **const** = 0; into the base class, overridden by each derived class in syntactically the same way: **throw** ∗**this**;. The *Match* statement then simply calls raise on its subject, while case clauses are turned into catch handlers. The exact name of the function is not important, and is communicated to the library as a *raise selector* with an RS specifier in the same way as *kind selectors* and *class members* (§6.3.3). The raise member function can be seen as an analog of the accept member function in the visitor design pattern, whose main purpose is to discover subject's most-specific type. The analog of a call to visit to communicate that type is replaced, in this scheme, with the exception unwinding mechanism.

Just because we can, it does not mean we should abuse the exception handling mechanism to give us the desired control flow. In the table-driven approach commonly used in high-performance implementations of exception handling, the speed of handling an exception is sacrificed to provide a zero execution-time overhead for when exceptions are not thrown [218]. Using exception handling to implement type switching will reverse the common and exceptional cases, significantly degrading performance. As can be seen in Figure 5.1, matching the type of the first case clause using the polymorphic exception approach takes more than 7000 cycles and then grows linearly (with the position of the case clause in the *Match* statement), making it the slowest approach. The numbers illustrate why exception handling should only be used to deal with exceptional cases and not common ones.

Despite its total impracticality, the approach gave us the very practical idea of harnessing a C++ compiler to do *redundancy checking* at compilation time.

5.4.1 Redundancy Checking

As discussed in §2.5, *redundancy* checking is only applicable to first-fit semantics of the *Match* statement, and warns the user of any case clause that will never be entered because of a preceding one being more general.

We provide a library configuration flag, which, when defined, effectively turns the entire *Match* statement into a try-catch block with handlers accepting the target types of the case clauses. This forces the compiler to give a warning when a more general catch handler precedes a more specific one, effectively performing redundancy checking for us, e.g.:

```
filename.cpp(55): warning C4286: 'ipr::Decl*' : is caught by base class ('ipr::Stmt*') on line 42
```

Note that the message contains both the line number of the redundant case clause (55) and the line number of the case clause that makes it redundant (42).

Unfortunately, the flag cannot always be enabled, as the case labels of the underlying switch statement have to be eliminated in order to render a syntactically correct program. Nevertheless, we found the redundancy checking facility of the library extremely useful when rewriting visitor-based code: even though the order of overrides in a visitor's implementation does not matter, for some reason more general ones were inclined to happen before specific ones in the code we looked at. Perhaps programmers are inclined to follow the class declaration order when defining and implementing visitors.

The related *exhaustiveness* checking – a test of whether a given match statement covers all possible cases – needs to be reconsidered for extensible data types like classes, since one can always add a new variant to it. Completeness checking in this case may simply become equivalent to ensuring that there is either a default clause in the type switch or a clause with the static type of a subject as a target type. In fact, our library has an analog of a default clause called an *Otherwise* clause, which is implemented under the hood as a regular *Case* clause with the subject's static type as the target type.

5.5 Memoized Dynamic Cast

The results of §5.3.4 indicate that the outcome of **dynamic_cast** can be mapped to a different instance from within the same subobject. This leads to the simple idea of memoizing the results of **dynamic_cast** and then using them on subsequent casts. In what follows, we will only be dealing with the pointer version of the operator since the version on references that has a slight semantic difference can easily be implemented in terms of the pointer one.

The **dynamic_cast** operator in C++ involves two arguments: a value argument representing an object of a known static type as well as a type argument denoting the runtime type we are querying. Its behavior is twofold: on one hand, it should be able to determine when the object's dynamic type is not a subtype of the queried type (or when the cast is ambiguous), while on the other, it should be able to produce an offset by which to adjust the value argument when it is.

We mimic the syntax of **dynamic_cast** by defining:

```
template <typename T, typename S> inline T memoized_cast(S*);
```

which lets the user replace all the uses of **dynamic_cast** in the program with memoized_cast with a simple:

```
#define dynamic_cast memoized_cast
```

It is important to stress that the offset is not a function of only the source and target types of the **dynamic_cast** operator, which is why we cannot simply memoize the outcome inside the individual instantiations of memoized_cast. The use of repeated multiple inheritance will result in classes having several different offsets associated with the same pair of source and target types depending on which subobject the cast is performed from. We saw in §5.3.4, however, that the offset is a function of target type and the value of the vtbl-pointer stored in the object, because the vtbl-pointer uniquely determines the subobject within the dynamic type. Our memoization of the results of **dynamic_cast** should thus be specific to a vtbl-pointer and the target type.

The easiest way to achieve this would be to use a dedicated global vtblmap⟨std::ptrdiff_t⟩ (§5.3.6) per instantiation of the memoized_cast. This, however, will create an unnecessarily large number of vtblmap structures, many of which will be duplicating information and repeating the work already done. This will happen because instantiations of memoized_cast with the same target type but different source types can share their vtblmap structures since vtbl-pointers of different source types are necessarily different according to Theorem 2.

Even though the above solution can easily be improved to allocate a single vtblmap per target type, a typical application might have many different target types. This is especially true for applications that will use our *Match* statement since we use **dynamic_cast** under the hood in each case clause. Indeed, our C++ pretty-printer was creating 162 vtblmaps of relatively small size each, which increased the executable size quite significantly because of numerous instantiations, as well as noticeably slower compilation time.

To overcome the problem, we turn each target type into a runtime instantiation index of the type and allocate a single vtblmap⟨std::vector⟨std::ptrdiff_t⟩⟩ that associates vtbl-pointers with a vector of offsets indexed by target type. The slight performance overhead incurred by this improvement is specific to our library solution and would not be present in a compiler implementation. As a result, however, we get a much smaller memory footprint, which can be made even smaller once we recognize the fact that global type indexing may effectively enumerate target classes that will never appear in the same *Match* statement. This will result in entries that are never used in the vector of offsets.

Our actual solution uses separate indexing of target types for each source type they are used with and allocates a different vtblmap⟨std::vector⟨std::ptrdiff_t⟩⟩ for each source type. This lets us minimize unused entries within offset vectors by making sure only the plausible target types for a given source type are indexed. This solution should be suitable for most applications, since we expect to have a small number of source types for the **dynamic_cast** operator and a much larger number of target types. For the unlikely case of a small number of target types and large number of source types, we allow the user to revert to the default behavior with a library configuration switch that allocates a single vtblmap per target type, as we have already discussed above.

The use of memoized_cast to implement the *Match* statement reuses the results of **dynamic_cast** computations across multiple independent *Match* statements. This allows leveraging the cost of the expensive first call with a given vtbl-pointer across all the *Match* statements inside the program. The above **#define**, with which a user can easily turn all dynamic casts into memoized casts, can be used to speed up existing code that uses dynamic casting without any refactoring overhead.

5.6 Evaluation

We performed several independent studies of our approach to demonstrate its effectiveness. The first study compares our approach to the visitor design pattern and shows that the type switch is comparable or faster (§5.6.1). While we do not advocate for the closed solution of §5.3.1, we included the comparison of type switching solutions made under open and closed world assumptions (§5.6.2). The second study does a similar comparison with built-in facilities of Haskell and OCaml and shows that the open type switch for extensible and hierarchical data types can be almost as efficient as its equivalent for closed algebraic data types (§5.6.3). In the third study, we looked at how well our caching mechanisms deal with some large real-world class hierarchies in order to demonstrate that our performance numbers were not established in overly idealistic conditions (§5.6.4). In the fourth study, we looked at how well our extension of the *Match* statement to N-arguments using the Morton order deals with the same class hierarchy benchmark (§5.6.5). In the fifth study, we compare the performance of matching N polymorphic arguments against double, triple and quadruple dispatch via visitor design pattern as well as open multi-methods extension to C++ (§5.6.6). In the last study, we rewrote an existing visitor-based application using our approach in order to compare the ease of use, readability and maintainability of each approach, as well as to show the memory usage and the startup costs associated with our approach in a real application (§6.4.3).

5.6.1 Comparison with Visitor Design Pattern

Our comparison methodology involves several benchmarks representing various uses of objects inspected with either visitors or type switching.

The *repetitive* benchmark (REP) performs calls on different objects of the same dynamic type. This scenario happens in object-oriented setting when a group of polymorphic objects is created and passed around (e.g. numerous particles of a given kind in a particle simulation system). We include it because double dispatch becomes twice faster (20 vs. 53 cycles) in this scenario compared to others due to hardware cache and call target prediction mechanisms.

The *sequential* benchmark (SEQ) effectively uses an object of each derived type only once and then moves on to an object of a different type. The cache is typically reused the least in this scenario, which is typical of lookup tables, where each entry is implemented with a different derived class.

The *random* benchmark (RND) is the most representative as it randomly makes calls on different objects – probably be the most common usage scenario in the real world.

Presence of *forwarding* in any of these benchmarks refers to the common technique used by visitors where, for class hierarchies with multiple levels of inheritance, the visit method of a derived class will provide a default implementation of forwarding to its immediate base class, which, in turn, may forward it to its base class, etc. The use of forwarding in visitors is a way to achieve substitutability, which in type switch corresponds to the use of base classes in the case clauses.

The class hierarchy for non-forwarding test was a flat hierarchy of 100 derived classes, encoding an algebraic data type. The class hierarchy for forwarding tests had two levels of inheritance with 5 intermediate base classes and 95 derived ones.

While we do not advocate here for the closed solution of §5.3.1, we included it in our tests to show the performance gains a closed solution might have over the open one. Our library supports both solutions with the same surface syntax, which is why many users will try them both before settling on one.

The benchmarks were executed in the following configurations referred to as *Linux Desktop* and *Windows Laptop* respectively:

- *Lnx*: Dell Dimension® desktop with Intel® Pentium® D (Dual Core) CPU at 2.80 GHz; 1GB of RAM; Fedora Core 13
  - G++ 4.4.5 executed with -O2; x86 binaries
- *Win*: Sony VAIO® laptop with Intel® Core™i5 460M CPU at 2.53 GHz; 6GB of RAM; Windows 7 Professional
  - G++ 4.6.1 / MinGW executed with -O2; x86 binaries
  - MS Visual C++ 2010 Professional x86/x64 binaries with and without Profile-Guided Optimizations

To improve accuracy, timing in all the configurations was performed with the help of RDTSC instruction available on x86 processors. For every number reported here we ran 101 experiments timing 1,000,000 dispatches each (all through either visitors or type switch). The first experiment was serving as a warm-up, during which the optimal caching parameters were inferred, and typically resulted in an outlier with the largest time. Averaged over 1,000,000 dispatches, the number of cycles per dispatch in each of the 101 experiments was sorted and the median was chosen. We preferred median to average to diminish the influence of other applications and OS interrupts as well as to improve reproducibility of timings between the runs of application. In particular, in the diagnostic boot of Windows, where the minimum of drivers and applications are loaded, we were getting the same number of cycles per iteration 70-80 out of 101 times. Timings in non-diagnostic boots had somewhat larger absolute values, however the relative performance of type switch against visitors remained unchanged and equally well reproducible.

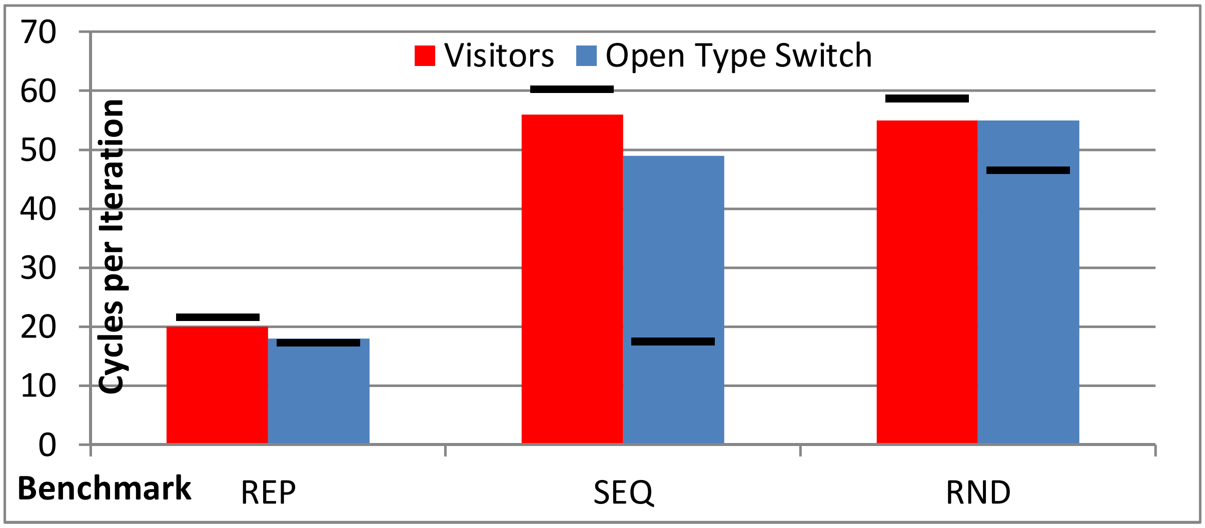


Figure 5.5: Absolute timings for different benchmarks

To understand better the relative numbers of Figure 5.1, we present in Figure 5.5 few absolute timings taken by visitors and open type switch to execute an iteration of a given benchmark. These absolute timings correspond to the relative numbers from column Open/G++/Win of Figure 5.1. The actual bars show the timings without forwarding, while the black lines indicate where the corresponding bar would be in the presence of forwarding. It is easy to see that visitors generally become slower in the presence of forwarding due to extra call, while type switch becomes faster due to smaller jump table. As discussed, both timings are much smaller for repetitive benchmark due to hardware cache.

Figure 5.1 provides a broader overview of how both techniques compare under different compiler/-platform configurations. The values are given as percentages of performance increase against the slower technique.

We can see that type switching wins by a good margin when implemented with tag switch (§5.3.1) as well as in the presence of at least one level of forwarding. Note that the numbers are relative, and thus the ratio depends on both the performance of virtual function calls and the performance of switch

		Open						Closed					
		G++		MS Visual C++				G++		MS Visual C++			
		Lnx	Win	PGO		w/o PGO		Lnx	Win	PGO		w/o PGO	
		x86-32	x86-32	x86-32	x86-64	x86-32	x86-64	x86-32	x86-32	x86-32	x86-64	x86-32	x86-64
	REP	16%	14%	1%	**18%**	2%	**37%**	124%	122%	100%	41%	76%	37%
	SEQ	56%	12%	48%	22%	**2%**	**46%**	640%	467%	29%	15%	30%	10%
	RND	56%	0%	**9%**	**19%**	**5%**	**46%**	603%	470%	35%	20%	32%	6%
Forwarding	REP	33%	22%	8%	**17%**	24%	**36%**	53%	49%	24%	**11%**	20%	**36%**
	SEQ	55%	233%	135%	135%	193%	**32%**	86%	290%	48%	139%	12%	**24%**
	RND	78%	25%	3%	**4%**	13%	**23%**	88%	33%	8%	**1%**	18%	**16%**

Table 5.1: Relative performance of type switching vs. visitors. Numbers in regular font (e.g. 14%), indicate when type switching was faster than visitors were, while underlined numbers in bold (e.g. **18%**), indicate when visitors were faster by corresponding percentage.

statements. Visual C++ was generating faster virtual function calls, while GCC was generating faster switch statements, which is why their relative performance seem to be much more favorable for us in the case of GCC. Similarly, the code for x86-64 is only slower relatively: the actual time spent for both visitors and type switching was smaller than that for x86-32, but it was much smaller for visitors than type switching, which resulted in worse relative performance.

Lastly, the code on the critical path of our type switch implementation benefits significantly from branch hinting as some branches are much more likely than others. We use the branch hinting directives in GCC to guide the compiler, but, unfortunately, Visual C++ does not provide any similar facilities. Instead, Microsoft suggests using *Profile-Guided Optimizations* (PGO) to achieve the same, which is why we list the results for Visual C++ both with and without profile-guided optimizations.

5.6.2 Open vs. Closed Type Switch

With only a few exceptions, we saw in the Table 5.1 that the performance of the closed tag switch dominates the performance of the open type switch. We believe that the difference, often significant, is the price one pays for the true openness of the vtable pointer memoization solution.

As we mentioned in §5.3.1, the use of tags, even when allocated by a compiler, may require integration efforts to ensure that different DLLs have not reused the same tags. Randomization of tags, similar to a proposal of Garrigue [104], will not eliminate the problem and will surely replace jump tables in switches with decision trees. This will likely significantly degrade the numbers for the part of Table 5.1 representing closed tag switch, since the tags in our experiments were all sequential and small.

The reliance of a tag switch on static cast has severe limitations in the presence of multiple inheritance, and thus is not as versatile as open type switch. Overcoming this problem will either require the use of **dynamic_cast** or techniques similar to vtable pointer memoization, which will likely degrade tag switch's performance numbers even further.

Note also that the approach used to implement open type switch can be used to implement both *first-fit* and *best-fit* semantics, while the tag switch is only suitable for best-fit semantics. Their complexity guarantees also differ: open type switch is constant on average, but slow on the first call with given subobject. Tag switch is logarithmic in the size of the class hierarchy (assuming a balanced hierarchy), including the first call. This last point can very well be seen in Table 5.1, where the performance of a closed solution degrades significantly in the presence of forwarding, while the performance of an open

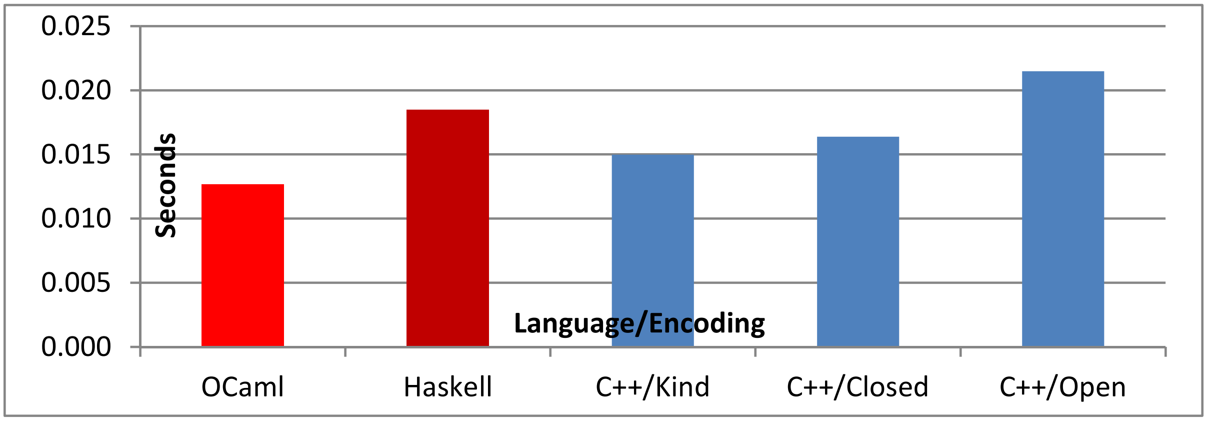


Figure 5.6: Performance comparison with OCaml and Haskell

solution improves.

5.6.3 Comparison with OCaml and Haskell

We now compare our solution to the built-in pattern-matching facility of OCaml [152] and Haskell [141]. In this test, we timed small OCaml and Haskell applications performing our sequential benchmark on an algebraic data type of 100 variants. Corresponding C++ applications were working with a flat class hierarchy of 100 derived classes. The difference between the C++ applications lies in the encoding used. Kind encoding is the same as Tag encoding, but it does not require substitutability, and thus can be implemented with a direct switch on tags without a ReMatch loop. It is only supported through specialized syntax in our library as it differs from the Tag encoding only semantically.

We used the optimizing OCaml compiler `ocamlopt.opt` version 3.11.0 working under the Visual C++ toolset as well as the Glasgow Haskell Compiler version 7.0.3 (with -O switch) working under the MinGW toolset. The C++ applications were compiled with Visual C++ as well and all the tests were performed on the Windows 7 laptop. Similar to comparison with visitors, the timing results presented in Figure 5.6 are averaged over 101 measurements and show the number of seconds it took to perform a 1,000,000 decompositions within our sequential benchmark. We compare here time and not cycles, as that was the only common measurement in all three environments.

5.6.4 Dealing with Real-World Class Hierarchies

For this experiment, we used a class hierarchy benchmark previously used in the literature to study efficiency of type-inclusion testing and dispatching techniques [83, 148, 262, 282]. We use the names of each benchmark from Vitek et al [262, Table 2], since the set of benchmarks we were working with was closest (though not exact) to that work.

While not all class hierarchies originated from C++, for this experiment it was more important for us that the hierarchies were man-made. While converting the hierarchies into C++, we had to prune inaccessible base classes (direct base class that is already an indirect base class) when used with repeated inheritance in order to satisfy semantic requirements of C++. We maintained the same number of virtual functions present in each class as well as the number of data members; the benchmarks, however, did not preserve the exact types of those. The data in Table 5.2 shows various parameters of the class hierarchies in each benchmark, after their adoption to C++.

Library	Language	Classes	Paths	Height	Roots	Leafs	Both	Parents		Children	
								avg	max	avg	max
DG2	Smalltalk	534	534	11	2	381	1	1	1	3.48	59
DG3	Smalltalk	1356	1356	13	2	923	1	1	1	3.13	142
ET+	C++	370	370	8	87	289	79	1	1	3.49	51
GEO	Eiffel	1318	13798	14	1	732	0	1.89	16	4.75	323
JAV	Java	604	792	10	1	445	0	1.08	3	4.64	210
LOV	Eiffel	436	1846	10	1	218	0	1.72	10	3.55	78
NXT	Objective-C	310	310	7	2	246	1	1	1	4.81	142
SLF	Self	1801	36420	17	51	1134	0	1.05	9	2.76	232
UNI	C++	613	633	9	147	481	117	1.02	2	3.61	39
VA2$_a$	Smalltalk	3241	3241	14	1	2582	0	1	1	4.92	249
VA2$_k$	Smalltalk	2320	2320	13	1	1868	0	1	1	5.13	240
VW1	Smalltalk	387	387	9	1	246	0	1	1	2.74	87
VW2	Smalltalk	1956	1956	15	1	1332	0	1	1	3.13	181
	Overalls	15246	63963	17	298	10877	199	1.11	16	3.89	323

Table 5.2: Benchmark class hierarchies

The number of paths represents the number of distinct inheritance paths from the classes in the hierarchy to the roots of the hierarchy. This number reflects the number of possible subobjects in the hierarchy. The roots listed in the table are classes with no base classes. We will subsequently use the term *non-leaf* to refer to the possible root of a subhierarchy. Leafs are classes with no children, while *both* refers to utility classes that are both roots and leafs and thus neither have base nor derived classes. The average for the number of parents and the number of children were computed only among the classes having at least one parent or at least one child correspondingly.

With few useful exceptions, it generally makes sense to apply type switch only to non-leaf nodes of the class hierarchy. 71% of the classes in the entire benchmarks suite were leaf classes. Out of the 4369 non-leaf classes, 36% were spawning a subhierarchy of only 2 classes (including the root), 15% – a subhierarchy of 3 classes, 10% of 4, 7% of 5 and so forth. Turning this into a cumulative distribution, a% of subhierarchies had more than b classes in them:

a	1%	3%	5%	10%	20%	25%	50%	64%	100%
b	700	110	50	20	10	7	3	2	1

These numbers reflect the percentage of use cases one may expect in the real word that have a given number of case clauses in them.

For each non-leaf class A we created a function performing a type switch on every possible derived class D_i of it as well as itself. The function was then executed with every possible subobject $D_i \prec \sigma_j \succ A$ it can possibly be applied to, given the static type A of the subject. It was executed multiple but the same number of times on each subobject to ensure uniformity on one side (since we do not have the data about the actual probabilities of each subobject in the benchmark hierarchies) as well as let the type switch infer the optimal parameters k and l of its cache indexing function H^V_{kl}. We then plotted a point in chart of Figure 5.7 relating 2 characteristics of each of the 4396 type switches tested: the optimal computed probability of conflict p achieved by the type switch and the number of subobjects n that came through that type switch. The actual frequencies of collisions were within one tenth of a percentage point of the computed probabilities, which is why we did not use them in the chart. To

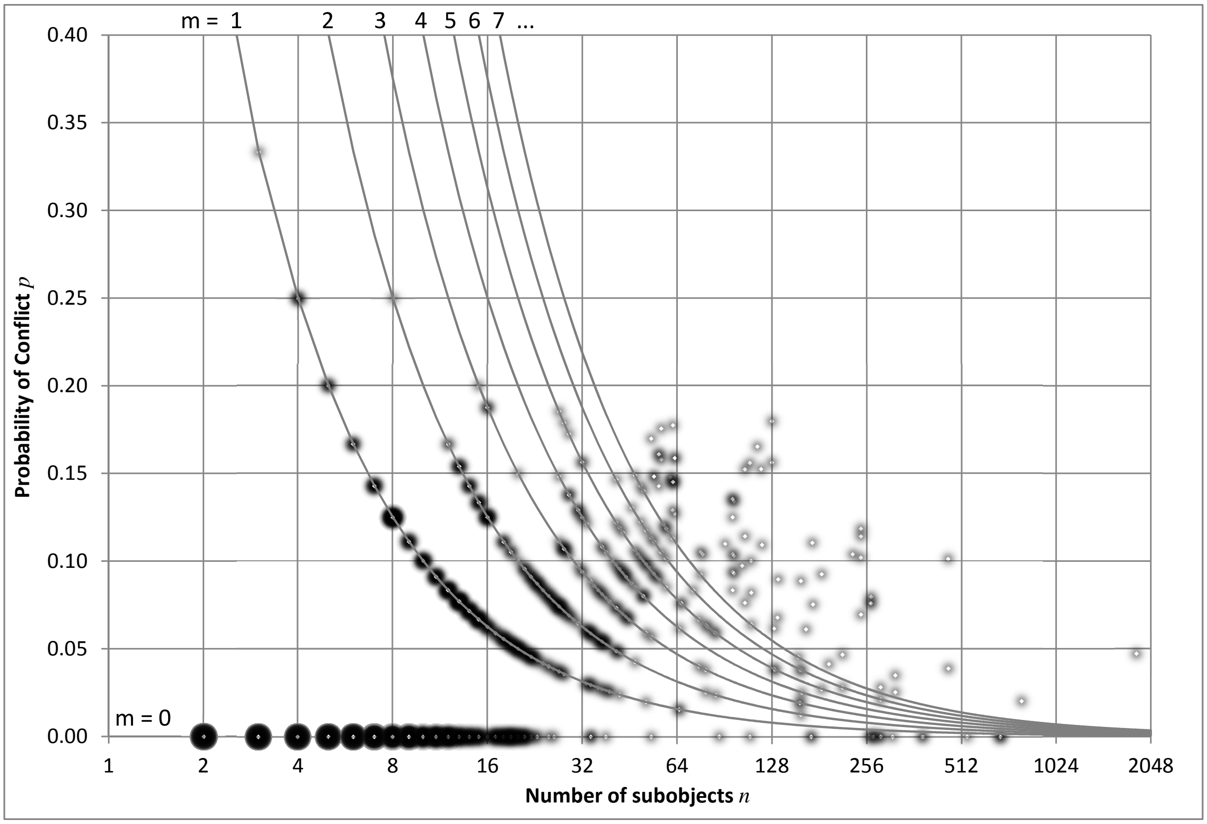


Figure 5.7: Probability of conflict in real hierarchies

account for the fact that multiple experiments could have resulted in the same pair (n,p), we use a shadow of each point to reflect somewhat the number of experiments yielding it.

The curves on which the results of experiments line up correspond to the fact that under uniform distribution of n subobjects, only a finite number of different values representing the probability of conflict p is possible. In particular, all such values $p = \frac{m}{n}$, where $0 \leq m < n$. The number m reflects the number of subobjects an optimal cache indexing function H_{kl}^{V} could not allocate their own entry for and we showed in §5.3.7 that the probability of conflict under uniform distribution of n subobjects depends only on m. The curves thus correspond to graphs of functions $y = \frac{m}{x}$ for different values of m. The points on the same curve (which becomes a line on a log-log plot) all share the same number m of "extra" vtbl-pointers that optimal cache indexing function could not allocate individual entries for.

While it is hard to see from the chart, 87.5% of all the points on the chart lay on the X-axis, which means that the optimal hash function for the corresponding type switches had no conflicts at all ($m = 0$). In other words, only in 12.5% of cases the optimal H_{kl}^{V} for the set of vtbl-pointers V coming through a given type switch had non-zero probability of conflict. Experiments laying on the first curve amount to 5.58% of subhierarchies and represent the cases in which optimal H_{kl}^{V} had only one "extra" vtbl-pointer ($m = 1$). 2.63% of experiments had H_{kl}^{V} with 2 conflicts, 0.87% with 3 and so forth as shown in Table 5.3($K + 1$).

In cases when the user is willing to trade performance for better space efficiency she may restrict k to $[K, K]$ instead of $[K, K + 1]$ as discussed in §5.3.7. We redid all the 4396 experiments under this

m	0	1	2	3	4	5	6	>6
$K+1$	87.50%	5.58%	2.63%	0.87%	0.69%	0.69%	0.30%	1.76%
K	72.55%	12.27%	4.87%	2.61%	1.42%	0.94%	0.80%	4.55%

Table 5.3: Percentage of type switches with given number of conflicts (m) under different size constraints

restriction and obtained a similar histogram shown in Table 5.3(K). The average probability of conflict over the entire set increased from 0.011 to 0.049, while the maximum probability of conflict increased from 0.333 to 0.375. The average load factor of the cache expectedly increased from 75.45% to 82.47%.

It is important to understand that the high ratio of cases in which the hash function could deliver perfect indexing does not indicate that the hash function we used is better than other hash functions. It does indicate instead that the values representing vtbl-pointers in a given application are not random at all and are particularly suitable for such a hash function.

5.6.5 Multi-Argument Hashing

To check the efficiency of hashing in the multi-argument *Match* statement (§6.3.6) we used the same class hierarchy benchmark we used to test the efficiency of hashing in type switch [232, §4.4]. The benchmark consists of 13 libraries describing 15246 classes totally. Not all the class hierarchies originated from C++, but all were written by humans and represent actual relationships in their problem domains.

While a *Match* statement works with both polymorphic and non-polymorphic arguments, only the polymorphic arguments are taken into consideration for efficient type switching and thus efficient hashing. It also generally makes sense to apply type switching to non-leaf nodes of the class hierarchy only. 71% of the classes in the entire benchmarks suite were leaf classes. For each of the remaining 4369 non-leaf classes we created 4 functions, performing case analysis on derived class on a combination of 1, 2, 3 and 4 arguments respectively. Each of the functions was executed with different combinations of possible derived types, including, in case of repeated multiple inheritance, different sub-objects within the same type. There was 63963 different subobjects when the class hierarchies used repeated multiple inheritance and 38856 different subobjects when the virtual multiple inheritance was used.

As with type switching, for each of the 4369 functions (per same number of arguments) we were measuring the number of conflicts m in cache – the number of entries mapped to the same location in cache by the optimal hash function. We then computed the percentage of functions that achieved a given number of conflicts, which we show in Table 5.4.

N/m		[0]	[1]	··· 10]	··· 100]	··· 1000]	··· 10000]	>10000
Repeated	1	88.37%	10.78%	0.85%	0.00%	0.00%	0.00%	0.00%
	2	76.42%	5.51%	10.60%	4.89%	2.22%	0.37%	0.00%
	3	65.18%	0.00%	15.04%	8.92%	5.83%	5.03%	0.00%
	4	64.95%	0.00%	0.14%	14.81%	7.57%	12.54%	0.00%
Virtual	1	89.72%	9.04%	1.24%	0.00%	0.00%	0.00%	0.00%
	2	80.55%	4.20%	8.46%	4.59%	1.67%	0.53%	0.00%
	3	71.26%	0.37%	12.03%	7.32%	4.87%	4.16%	0.00%
	4	71.55%	0.00%	0.23%	11.83%	6.49%	9.90%	0.00%

Table 5.4: Percentage of N-argument *Match* statements with given number of conflicts (m) in cache

We grouped the results in ranges of exponentially increasing size because we noticed that the number of conflicts per *Match* statement for multiple arguments was not as tightly distributed around 0 as it was for a single argument. The main observation however still holds: in most of the cases, we could achieve hashing without conflicts, as can be seen in the first column (marked [0]). The numbers are slightly better when virtual inheritance is used because the overall number of possible subobjects is smaller.

5.6.6 Comparison to Multiple Dispatch Alternatives

Type switching on multiple arguments can be seen as a form of multiple dispatch. In this study, we compare the efficiency of type switching on multiple arguments in comparison to alternatives based on double, triple and quadruple dispatch [128], as well as our own implementation of open multi-methods for C++ [208].

The need for multiple dispatch rarely happens in practice, diminishing with the number of arguments involved in dispatch. Muschevici et al [188] studied a large corpus of applications in 6 languages and estimate that single dispatch amounts to about 30% of all the functions, while multiple dispatch is only used in 3% of functions. In application to type switching, this indicates that we can expect case analysis on dynamic type of a single argument much more often than that on dynamic types of two or more arguments. Note, however, that this does not mean that pattern matching in general reflects the same trend as additional arguments are often introduced into the *Match* statement to check some relational properties. These additional arguments are typically non-polymorphic and thus do not participate in type switching, which is why in this experiment we only deal with polymorphic arguments.

Figure 5.8 contains 4 bar groups that corresponding to the number of arguments used for multiple dispatch. Each group contains 3 wide bars representing the number of cycles per iteration it took N-Dispatch, Open Type Switch and Open Multi-Methods solution to perform the same task. Each of the 3 wide bars is subsequently split into 5 narrow sub-bars representing performance achieved by G++ 4.5.2, 4.6.1, 4.7.2 and Visual C++ 10 and 11 in that order from left to right.

Open multi-methods provide the fastest performance because the dispatch is implemented with an N-dimensional array lookup, requiring only $4N+1$ memory references before an indirect call. N-Dispatch provides the slowest solution, requiring $2N$ virtual function calls (accept/visit per each dimension). Open type switch falls in between the two, thanks to its efficient hashing combined with a jump table.

In terms of memory, given a class hierarchy of n classes (or n subobjects in the subobject graph to be more precise) and a multiple dispatch on N arguments from it, all 3 solutions will require the memory proportional to $O\left(n^N\right)$. More specifically, if δ is the number of bytes used by a pointer, then each of the approaches will use:

- Open Multi-Methods: $\delta\left(n^N + Nn + N\right)$
- N-Dispatch: $\delta\left(n^N + n^{N-1} + \cdots + n^2 + n\right)$
- Open Type Switch: $\delta\left((2N+3)\, n^N + N + 7\right)$

bytes of memory. In all 3 cases, the memory counted represents the non-reusable memory specific to the implementation of a single function dispatched through N polymorphic arguments. Note that n is a variable here since new classes may be loaded at run-time through dynamic linking in all 3 solutions, while N is a constant, representing the number of arguments to dispatch on.

The memory used by each approach is allocated at different stages. The memory used by the virtual tables involved in the N-dispatch solution as well as dispatch tables used by open multi-methods will be allocated at compile/link time and will be reflected in the size of the final executable. Open multi-

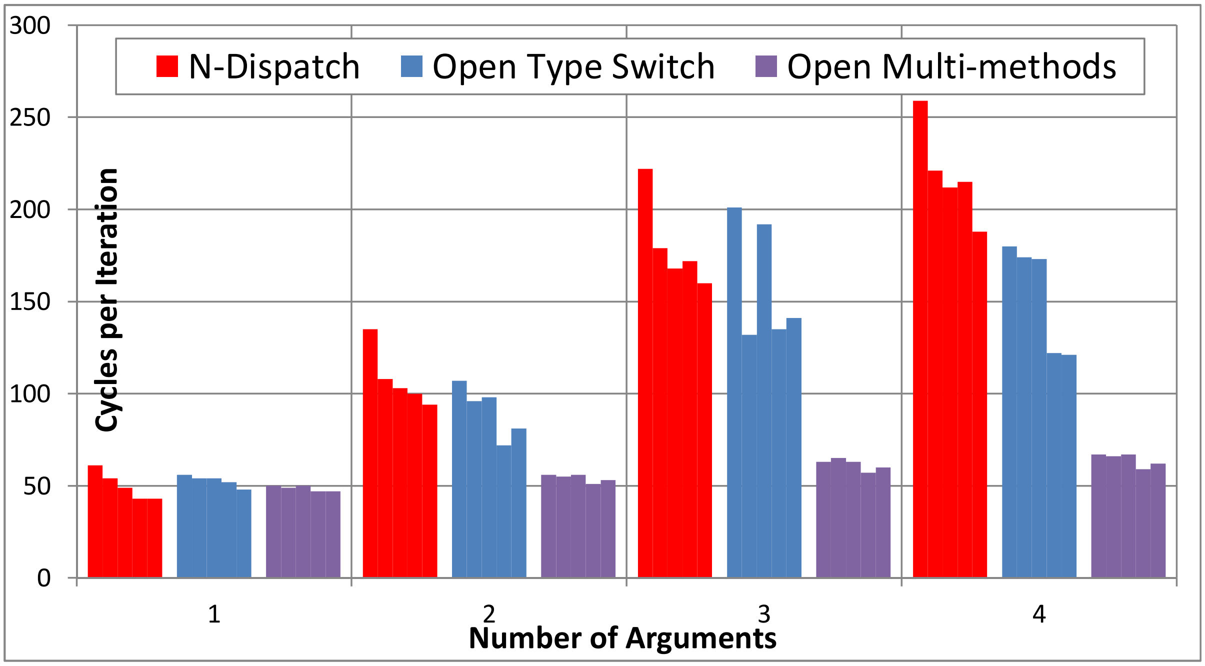


Figure 5.8: N-argument *Match* statement vs. visitor design pattern and open multi-methods

methods might require additional allocations and/or recomputation at load time to account for dynamic linking. In both cases, the memory allocated covers all possible combinations of n classes in N argument positions. In case of open type switch, the memory is only allocated at run-time and grows proportionally to the number of actual argument combinations seen by the type switch (§5.3.8). Only in the worst case when all possible combinations have been seen by the type switch does it reach the size described by the above formula. This is an important distinction as in many applications all-possible combinations will never be seen: for example, in a compiler the entities representing expressions and types might all be derived from a common base class, however they will rarely appear in the same type switch together.

There is also a significant difference in the ease of use of these solutions. N-Dispatch is the most restrictive solution as it is intrusive (and thus cannot be applied retroactively), hinders extensibility (by limiting the set of distinguishable cases) and is surprisingly hard to teach students. While analyzing Java idioms used to emulate multiple dispatch in practice, Muschevici et al [188, Figure 13] noted that there are significantly more uses of cascading instanceof in the real code than the uses of double dispatch, which they also attribute to the obscurity of the second idiom. Both N-Dispatch and open multi-methods also introduce control inversion in which the case analysis is effectively structured in the form of callbacks. Open multi-methods are also subject to ambiguities, which have to be resolved at compile time and in some cases might require the addition of numerous overriders. Neither is the case with open type switch where the case analysis is performed directly, while ambiguities are avoided by the use of first-fit semantics.

5.6.7 Refactoring an Existing Visitor-Based Application

For this experiment, we reimplemented a visitor-based C++ pretty-printer for Pivot [81] using *Mach7*. The Pivot's class hierarchy consists of 162 node kinds representing various entities in the C++ program.

The original code had 8 visitor classes each handling 5, 7, 8, 10, 15, 17, 30 and 63 cases, which we turned into 8 *Match* statements with corresponding numbers of case clauses. Most of the rewrite was performed by sed-like replaces that converted visit methods into respective case clauses. In several cases, we had to manually reorder case clauses to avoid redundancy as visit methods for base classes were often coming before those for derived, while for type switching we needed them to come after due to first-fit semantics. Redundancy checking support provided by *Mach7* (§5.4.1) was invaluable in finding all such cases.

Header	LOC	Faster Time	Performance	Header	LOC	Faster Time	Performance
climits	44	765 mks	**15.26%**	cmath	338	5909 mks	**3.40%**
cassert	47	781 mks	**16.09%**	cwchar	430	6956 mks	3.86%
cerrno	50	894 mks	**11.79%**	functional	588	6352 mks	**2.39%**
cstdarg	51	301 mks	**21.64%**	iosfwd	981	14699 mks	2.17%
csignal	53	1061 mks	**8.17%**	utility	1095	15848 mks	2.30%
cstddef	55	943 mks	**12.29%**	limits	1521	20512 mks	1.53%
cfloat	61	1188 mks	**10.44%**	valarray	1551	30552 mks	1.10%
csetjmp	65	1172 mks	**8.33%**	iterator	2356	29158 mks	2.65%
cctype	98	1667 mks	**3.01%**	numeric	2481	30545 mks	3.70%
cwctype	98	2137 mks	**10.88%**	memory	3171	38761 mks	4.33%
ctime	104	1714 mks	**8.74%**	stdexcept	3907	50049 mks	3.49%
exception	107	1803 mks	**10.02%**	algorithm	5467	63553 mks	3.37%
cstring	144	2560 mks	2.03%	deque	5477	68619 mks	2.75%
typeinfo	145	2267 mks	**8.11%**	stack	5563	68893 mks	3.22%
new	156	2255 mks	**7.94%**	list	5981	72341 mks	3.63%
cstdio	195	3482 mks	**1.58%**	vector	5985	73268 mks	3.94%
cstdlib	195	3211 mks	**0.86%**	queue	9851	115871 mks	3.60%

Table 5.5: Relative performance of type switching vs. visitors achieved by Pivot's C++ pretty-printer on various standard headers. Files are ordered by the number of lines in them. Underlined numbers in bold (e.g. **15.26%**), indicate that pretty-printer based on visitors was faster than the one based on type switch by corresponding percentage, while the numbers in regular font (e.g. 3.60%), indicate that pretty-printer based on type switching was faster than the one based on visitors by corresponding percentage.

We ran both pretty-printers on a set of header files from the C++ standard library to print them back out and made sure the produced outputs of both programs were byte-to-byte the same. We timed execution of the pretty-printing phase (not including loading and termination of the application or parsing of the source file) and presented the results in Table 5.5. Both pretty-printers were compiled with Visual C++ 10. The set of header files presented in the table is only a subset of all the standard header files, because at the time of the experiment Pivot's EDG-based front end could not parse all the headers. The rows in the table are sorted by the number of *Lines of Code* (*LOC*) in the corresponding header file because this number correlates with the number of AST-nodes in the header's IPR.

As can be seen from the table, on small files (e.g. those from C run-time library and few small C++ files) visitors-based implementation is faster because the total number of nodes in AST and thus calls did not justify our set-up calls. In particular, visitor-based implementation of the pretty-printer was faster on files of 44–588 lines of code, with average 136 lines per those inputs, where visitors win. On these input files, visitors are faster by 1.17%–21.42% with an average speed-up of 8.75%. Open type switch

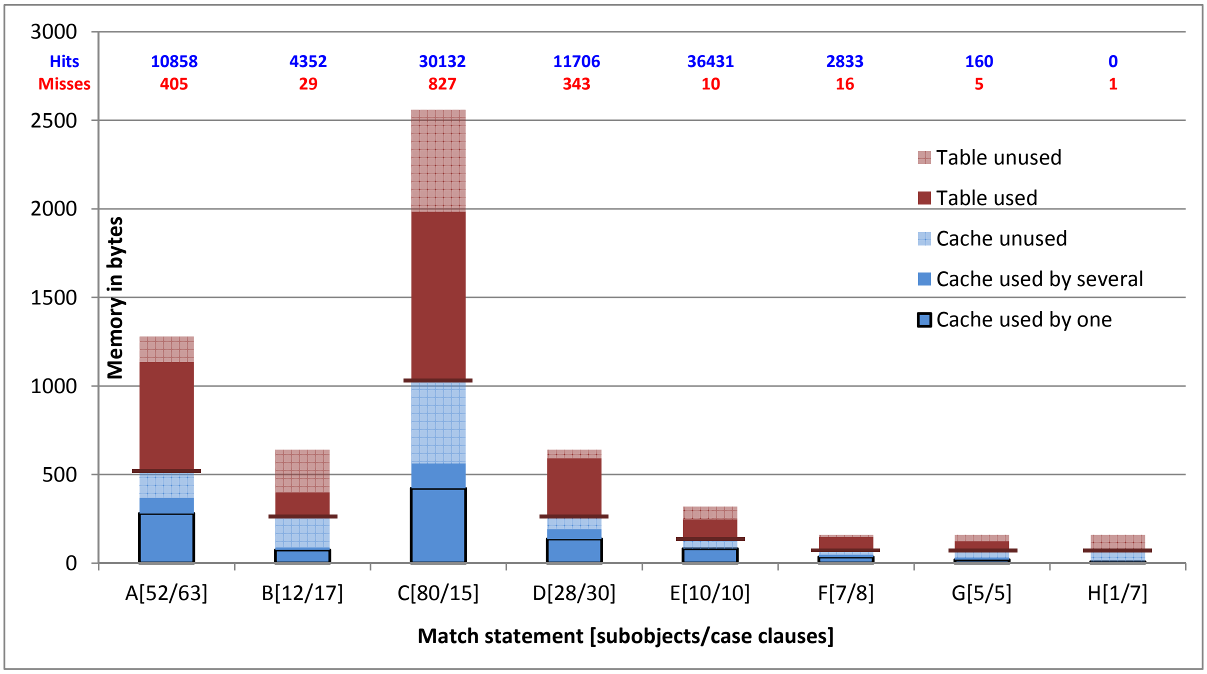


Figure 5.9: Memory usage in real application

based implementation of the pretty-printer was faster on files of 144–9851 lines of code, with average 3497 lines per those input files, where open type switch wins. On these inputs, open type switch is faster by 0.18% – 32.99% with an average speed-up of 5.53%.

Figure 5.9 shows memory usage as well as cache hits and misses for the run of our pretty-printer on ⟨queue⟩ standard library header, which had the largest LOC count after preprocessing in our test set.

The bars represent the total size of memory in bytes each of the 8 match statements (marked A-H) used. Information $[n/c]$ next to the letter indicates the actual number of different subobjects (i.e. vtbl-pointers) n that came through that *Match* statement, and the number of case clauses c the *Match* statement had (the library uses c as an estimate of n). n is also the number of cases the corresponding *Match* statement had to be executed sequentially (instead of a direct jump).

The lower part of each bar (with respect to dividing line) corresponds to the memory used by cache, while the upper part – to the memory used by the hash table. The ratio of the darker section of each part to the entire part indicates the load factors of cache and hash-table respectively. The black box additionally indicates the proportion of cache entries that are allocated for only one vtbl-pointer and thus never result in a cache miss.

The actual number of hits and misses for each of the *Match* statements is indicated on top of the corresponding column. The sum of them is the total number of calls made. The number of misses is always larger than or equal to n since we need to execute the switch sequentially on each of them once in order to memoize the outcome.

The library always preallocates memory for at least 8 subobjects to avoid unnecessary recomputations of optimal parameters k and l – this is the case with the last 3 *Match* statements. In all other cases it allocates the memory proportional to 2^{K+1} where $2^{K-1} < \max(n,c) \le 2^K$. We make c a parameter, because in a library setting n is not known up front and estimating it with c allows us to avoid unnecessary recomputations of l and k even further.

The table does not have to be hash table and can be implemented with any other container i.e. sorted vector, map etc. that let us find quickly by a given vtbl-pointer the data associated with it. In fact we provide a slightly less efficient caching container that avoids the table altogether, thus significantly reducing the memory requirements instead.

5.7 Discussion

In this chapter, we presented type switching as an open alternative to the visitor design pattern, which overcomes the restrictions, inconveniences, and difficulties in teaching and using visitors. Our implementation significantly outperforms the visitor design pattern in most cases and roughly equals it otherwise. This is the case even though we use a library implementation and highly optimized production-quality compilers. An important benefit of our solution is that it does not require any changes to the C++ object-model or require any computations at load time.

To provide a complete solution, we use the same syntax for closed sets of types, where our performance roughly equals the equivalent built-in features in functional languages, such as Haskell and OCaml.

We prove the uniqueness of vtbl-pointers in the presence of RTTI. This is potentially useful in other compiler optimizations that depend on the identity of subobjects. Our memoization device can also become valuable in optimizations that require mapping run-time values to execution paths, and is especially useful in library setting.

5.7.1 Limitations

Currently the definition of each class used in a case clause must be visible to the compiler because **dynamic_cast** operator used in the type switch does not allow incomplete types as a target type. For particularly large type switches (e.g. >1000 case clauses) this may easily reach some compiler limitations. Both GCC and Visual C++, for example, could not generate object files for such translation units simply because the sheer size of v-tables and other compiler data in it were exceeding the limits. The problem is not specific to our technique though and allowing **dynamic_cast** on classes that were declared but not defined yet would solve the problem.

While it might be reasonable to expect from linkers to layout v-tables close to each other – the property that makes our hashing function efficient – they are not required to do so. We believe, nevertheless, that should our approach become popular through the library implementation, its compiler implementation will encourage compiler vendors to enforce the property in order to keep the type switching fast.

Using a library implementation was essential for experimentation and for being able to test our ideas on multiple production-quality compiler systems. However, now we hope to re-implement our ideas in a compiler. This would allow us to improve further surface syntax, diagnostics, and performance.

6. OPEN PATTERN MATCHING[1]

> Within C++, there is a much smaller and cleaner language struggling to get out.
>
> Bjarne Stroustrup

While open type switch and open multi-methods are indispensable in uncovering a combination of dynamic types of several arguments, neither of them is useful in decomposing structure of those arguments. In functional languages, this is traditionally achieved with *pattern matching*, more specifically – with constructor patterns and nesting of patterns. The expressiveness of pattern matching has been so crucial to program analysis applications that often times it was the sole reason in the choice of a language for the job. Instead of introducing pattern matching to C++ as an orthogonal language feature, we show that C++ abstractions are already powerful enough to introduce pattern matching in a form of library. Not only this library solution introduces patterns as first-class citizens, it does so with zero or very small overhead, which was a major advantage over previous approaches to introduction of patterns through library. The term *open* in this respect, indicate that the set of patterns is not fixed and can be extended by the user. In fact, all the patterns traditionally seen in functional languages are implemented as user-defined patterns in our approach.

The actual content of this chapter is based on the paper that is currently under review for publication [233]. An extended abstract covering this work has been accepted for publication at SPLASH'13 [234]. The work has been performed in collaboration with Dr. Dos Reis and Dr. Stroustrup.

6.1 Introduction

Pattern matching is an abstraction mechanism popularized by the functional programming community, most notably ML [111], Hope [39], Miranda [254], OCaml [152] and Haskell [127], and recently adopted by several multi-paradigm and object-oriented programming languages such as Scala [195], F# [249], Newspeak [105], Fortress [217], Thorn [27], Grace [124], and dialects of C++ [154, 192], Eiffel [186] and Java [122, 153, 157, 196, 214]. It provides syntax very close to mathematical notation and allows the user to describe tersely a (possibly infinite) set of values accepted by the pattern. A *pattern* is an immediate predicate on an implicit argument that can be used to check properties and bind parameters inherent to them. The pattern is usually more specialized and thus more concise, expressive and readable than equivalent imperative code, even when the latter is expressed with unnamed λ-expressions. Patterns are similar to λ-expressions, however, in that they are rarely repeated in several places in code, which makes them unattractive for code reuse through a dedicated predicate. The difference between patterns and λ-expressions lies in the brevity of code, their composition and interaction.

Expressivity of pattern matching has been considered so crucial in certain application domains, that it became the number one reason for choosing a functional language for the job in those domains. In their experience report on implementing Frama-C static analyzer, Cuoq et al [71] list expressiveness as the number one reason for using OCaml instead of C++. Similar sentiments are expressed in two other experience reports detailing implementation of a high-performance trading platform in OCaml [183] and

[1] Part of this chapter is reprinted with permission from “Open Pattern Matching for C++ (Extended Abstract)” by Yuriy Solodkyy, Gabriel Dos Reis, Bjarne Stroustrup, 2013. Proceedings of the 4th annual conference on Systems, Programming, and Applications: Software for Humanity, Copyright 2013 by ACM.

implementation of a hardware-design toolset in Haskell [190].

We thus consider how functional-style pattern matching can be added to C++. Introduction of pattern matching into a new language is nontrivial; introducing it into an existing language in widespread use is an even greater challenge. The obvious utility of the feature is easily compromised by the need to fit into the language's syntax, semantics, and toolchains. A prototype implementation requires more effort than for an experimental language. It is also harder to get into use because mainstream users are unwilling to try non-portable, non-standard, and unoptimized features, so we need to provide an optimized implementation. To balance utility and effort we took the Semantically Enhanced Library Language (SELL) approach [245, 246] that advocates for extending a general-purpose programming language with a library, aided by a tool. With pattern matching, we (somewhat surprisingly) managed to avoid external tool support by relying on some pretty nasty macro hacking and template metaprogramming to provide a conventional and convenient interface to an efficient library implementation.

6.1.1 Summary

This chapter presents functional-style pattern matching for C++ built as an ISO C++11 library. Our solution:

- Is open to introduction of new patterns into the library, while not making any assumptions about existing ones.
- Is type safe: inappropriate applications of patterns to subjects are manifested as compile-time errors.
- Makes patterns first-class citizens in the language (§6.3.2).
- Is non-intrusive and can be retroactively applied to existing types (§6.3.3).
- Provides an alternative interpretation of the controversial n+k patterns (in line with that of constructor patterns), leaving the choice of exact semantics to the user (§6.3.4).
- Supports a limited form of views (§6.3.5).
- Generalizes open type switch to accept patterns in case clauses (§6.3.6).
- Demonstrates that compile-time composition of patterns through concepts is superior to run-time composition of patterns through polymorphic interfaces in terms of performance, expressiveness and static type checking (§6.4.1).

Our library sets a minimum for performance, extensibility, brevity, clarity and usefulness of any language solution for pattern matching in C++. It provides full functionality, so we can experiment with the use of pattern matching in C++ and compare it to existing language alternatives. Our solution can be used today given current support of C++11 (e.g., GCC and Microsoft C++) without any additional tool support.

6.2 Pattern Matching in C++

Since the 1990s, C++ has had a pure functional sublanguage in it with striking similarity to ML and Haskell that has a very extensive compile-time pattern-matching facility in it. The sublanguage in question is the template facilities of C++, which allows one to encode computations at compile time, and has been shown to be Turing complete [259]. The key observation in the analogy to pattern matching is that partial and explicit template specializations of C++ class templates are similar to defining equations for Haskell functions [173]. It is also used widely in C++ for similar purposes: case analysis over families of types, structural decomposition of types, overload resolution etc. We can use this compile-time pattern matching facility to build a very expressive, efficient and extensible run-time pattern matching solution. When not stated otherwise, in the remainder of this chapter we will use the term *pattern matching* to mean run-time pattern matching – a facility that can be used to analyze and decompose run-time values.

The object analyzed through pattern matching is called *subject*, while its static type – *subject type*. The subject is generally implicit in patterns, which is the first major difference from λ-expressions, where the subject is explicit. Typically, the subject will be listed at the beginning of the statement performing the case analysis; however, in other cases it may be functionally related to the original subject or completely unrelated to it. We will show examples of each case throughout the section. Consider for example the following definition of factorial in *Mach7*:

```
int factorial(int n)
{
  unsigned short m;
  Match(n) {
    Case(0)  return 1;
    Case(m) return m*factorial(m-1);
    Case(_)  throw std::invalid_argument("factorial");
  } EndMatch
}
```

The subject n is passed as an argument to the *Match* statement and is then analyzed through *Case* clauses that list various patterns. In the *first-fit* strategy typically adopted by functional languages, the matching proceeds in sequential order while the patterns guarding their respective clauses are *rejected*. Eventually, the statement guarded by the first *accepted* pattern is executed or the control reaches the end of the *Match* statement.

The value 0 in the first case clause represents *value pattern*, variable m in the second case clause is used as a *variable pattern*, while the name _ in the last case clause refers to the common instance of the *wildcard pattern*, which are all instances of *primitive patterns* (§2.5). We extend the list of primitive patterns with a *predicate pattern* whereby we allow the use of any unary predicate or nullary member predicate as a pattern: e.g. *Case*(even) ... (assuming **bool** even(**int**);) or *Case*([](**int** m) { **return** m^m−1; }) ... etc.

In functional languages based on type inference, variable pattern is *irrefutable* since its type can be inferred from the pattern-matching context, however, in C++, it becomes a *refutable* pattern since all the variables have to be pre-declared and the type of the variable may be different from the type expected by the pattern. In the above example, of type **unsigned short**, hence last case clause is not *redundant*.

We are neutral in the debate of utility of n+k patterns (§2.5), but would like to point out that in C++, where variables have to be explicitly declared and given a type, one can use a variable's type to avoid at least some semantic issues. For example, matching $n + 1$ against 0 can be rejected when type of n is **unsigned int**, but accepted (binding n to -1) when its type is **int**. For the sake of experiment and as a demonstration that n+k patterns can be brought on first class principles, *Mach7* provides an implementation that treats them as application patterns. We do not restrict ourselves with the equational view, however, and instead offer an alternative point of view on n+k patterns that let the user choose a suitable semantics (§6.3.4). The following example in *Mach7* defines a function for fast computation of Fibonacci numbers with n+k patterns:

```
int fib(int n)
{
  var⟨int⟩ m;
  Match(n) {
    Case(1)        return 1;
    Case(2)        return 1;
    Case(2*m)      return sqr(fib(m+1)) − sqr(fib(m−1));
    Case(2*m+1) return sqr(fib(m+1)) + sqr(fib(m));
  } EndMatch
}
```

Note that the variable m is now declared with library's type var⟨**int**⟩ instead of a simple **int**. This is necessary because in the context of pattern matching induced by the *Case* clause we would like to capture the structure of the expression 2*m+1 instead of just computing its value. Type var⟨T⟩ facilitates exactly that in the pattern-matching context. In non-pattern-matching contexts, like the return statement above, it provides an implicit conversion to type T and thus behaves as a value of type T. In the rest of this chapter, we use cursive font on all the variables of library's type in order to help the reader visually distinguish them from the variables of built-in or any non-*Mach7* types.

Since the underlying type of variable m is still **int**, the third case clause will only match even numbers n, binding the value of m to $\frac{n}{2}$, while the fourth clause will only match odd numbers n, binding the value of m to $\frac{n-1}{2}$. Note that in these clauses the subject to a variable pattern m is not the original subject n anymore, but $\frac{n}{2}$ and $\frac{n-1}{2}$ respectively. Matching against even more complex expressions is discussed in §6.3.4, where we do not restrict ourselves with the equational interpretation, and instead offer an alternative point of view on n+k patterns that let the user choose a suitable semantics.

We saw the utility of *guard*s in §2.5 where they were used to attach arbitrary predicates to case clauses. In *Mach7* we allow combining patterns with guards, which gives raise to *guard patterns* – expressions of the form $P|{=}E$, where P is a pattern and E is an expression guarding it. As with n+k patterns, the variables involved in the condition must be of the library's type var⟨T⟩ to allow for lazy evaluation of condition at match time. For example, a case clause of the form *Case*(m|= m%2==1) ... in the example above will match any odd value n, binding m to n. Similarly, a case clause of the form *Case*(3*m|= m%2==0) ... will only match values n that are divisible by 6, while binding m to $\frac{n}{3}$.

Often times, guard patterns take the form that checks equality or inequality of two variables bound via pattern matching: e.g. *Case*(x, y|= y==x) ... is a two-argument case clause taking a variable and guard patterns to check that both subjects are the same. When there are many of them, the code quickly becomes obscure and hard to understand. Naturally, we would prefer to write something like *Case*(x,x) ... with the semantics of *equivalence patterns*, which match only when all the independent uses of variable x get bound to the same value. Neither OCaml nor Haskell support equivalence patterns, while Miranda and Tom's extension [186] of C, Java and Eiffel does. Grace and Thorn approximate the simplicity of equivalence patterns by explicitly marking the non-binding uses of a variable. We follow the same approach in *Mach7* – a unary + placed in front of a variable (or an n+k pattern) turns it into an expression evaluated at match time that behaves as a value pattern. As an example, consider the following *Mach7* implementation of the slower Euclidian algorithm with subtractions:

```
unsigned int gcd(const unsigned int a, const unsigned int b)
{
  var⟨unsigned int⟩ x, y;
  Match(a,b) {
    Case(x,+x)        return x;
    Case(x,(+x)+y) return gcd(x,y);
    Case(x,+x−y)   return gcd(x−y,y);
  } EndMatch
}
```

The subjects in *Mach7* are matched left to right, so the value of x is bound before its value gets used in the equivalence pattern $+x$ in the first clause. Patterns $+x + y$ and $+x - y$ in the second and third clauses are then simply n+k patterns with respect to variable y and a constant value bound in x. We use parentheses in the second clause only for clarity as many novices complained it was hard to decipher the pattern. The meaning is the same as in the other two case clauses since unary + has higher priority than the binary one.

Unary + in a pattern expression $+x$ is an example of *pattern combinator* – an operation that allows composing patterns to produce a new pattern. Besides *equivalence combinator*, other typical pattern combinators supported by many languages are *conjunction*, *disjunction* and *negation* combinators. *Conjunction combinator* in *Mach7* is predictably represented by P_1&& P_2 and succeeds only when both patterns P_1 and P_2 accept the subject. The set of variables bound by the resulting pattern is the union of variables bound by each of the patterns. Similarly, *disjunction combinator* is represented by $P_1 || P_2$ and succeeds when at least one pattern accepts the subject. Languages supporting disjunction combinator traditionally either require the set of variables bound by each pattern to be the same or only make the intersection of bound variables available after the match. In a library solution, we cannot enforce any of these requirements so it becomes user's responsibility to ensure she only uses variables from the intersection of bound variables. *Negation combinator* $!P_1$ binds nothing and succeeds only when pattern P_1 is rejected for a given subject. Negation combinator can be combined with equivalence combinator to check two subjects for inequivalence: x vs. $!+x$. Note that simply writing $!x$ would not have the desired semantics of "anything but x", because it will only match values outside the domain of type of x, and if the subject is of the same type as x, the pattern $!x$ will never be accepted.

We add two non-standard combinators that reflect the specifics of C++ – presence of pointers and references in the language. An *address combinator* &P_1 can be used to match against the subjects of type `T*` when the pattern P_1 expects subjects of type `T&`. It fails when the subject is **nullptr** and otherwise forwards the match to pattern P_1 applied to dereferenced subject. The utility of this combinator becomes obvious once we realize that most of the time we need to match against values pointed to by a pointer instead of matching against the pointer itself. The alternative would have been to require patterns match against both pointer and reference types, however for some cases, it produces non-intuitive ambiguity errors and we decided to require the user be more explicit instead. Dual to address combinator is *dereferencing combinator* written as $*P_1$ that can only match against l-values and forwards the matching to pattern P_1 applied to the address of the l-value. It is less frequently used than the address combinator is, nevertheless, it is indispensable when we need to obtain an address of pattern's subject, as that subject may not be easily accessible otherwise.

In functional languages, a syntactically the same expression may have different semantics within pattern-matching context and outside of it. In particular, given an algebraic data type

$$D = C_1(T_{11},...,T_{1m_1}) \mid \cdots \mid C_k(T_{k1},...,T_{km_k})$$

an expression of the form $C_i(x_1,...,x_{m_i})$ in a non-pattern-matching context is called a *value constructor* and refers to a value of type D created via constructor C_i and arguments $x_1,...,x_{m_i}$. The same expression in the pattern-matching context is called *constructor pattern* and is used to check whether the subject is of type D and was created with constructor C_i. If so, it binds the actual values it was constructed with to variables $x_1, ..., x_{m_i}$ (or matches against nested patterns if x_j are other kinds of patterns).

C++ allows a class to have several constructors and does not allow overloading the meaning of construction for the use in pattern matching. This is why in *Mach7* we have to be slightly more explicit about constructor patterns, which take form $\mathsf{C}\langle\mathsf{T}_i\rangle(P_1, ..., P_{m_i})$, where T_i is the name of the user-defined type we are decomposing and $P_1, ..., P_{m_i}$ are patterns that will be matched against members of T_i (§6.3.3). "C" was chosen to abbreviate "Constructor pattern" or "Case class" as its use resembles the use of case classes in Scala [195]. It was also the shortest notations we could come up with, which becomes important for nesting of constructor patterns, as we will see shortly. Consider for example the following *polymorphic encoding* of terms in λ-calculus in C++ together with a recursive implementation of an equality of two lambda terms in it using *Mach7*:

```
struct Term       { virtual ~Term() {} };
struct Var : Term { std::string name; };
struct Abs : Term { Var* var;  Term* body; };
struct App : Term { Term* func; Term* arg; };

bool operator==(const Term& left, const Term& right)
{
  var⟨const std::string&⟩ s; var⟨const Term&⟩ x,y;

  Match(left               , right               ) {
    Case(C⟨Var⟩(s)         , C⟨Var⟩(+s)          ) return true;
    Case(C⟨Abs⟩(&x,&y), C⟨Abs⟩(&+x,&+y)) return true;
    Case(C⟨App⟩(&x,&y), C⟨App⟩(&+x,&+y)) return true;
    Otherwise()                                    return false;
  } EndMatch
}
```

The above equality is an example of a *binary method* – an operation that requires both arguments to have the same type [36]. The first case clause can be interpreted as following: if both subjects are instances of class Var, bind the name of the first one to variable s and then check, that the name of the other variable is the same through equivalence combinator: $+s$. To understand the second and third clauses, note that the corresponding members of classes Abs and App are pointers to terms, while to ensure equality we are interested in checking deep equality of the terms themselves. Thus instead of binding pointers to variables, checking they are not **nullptr** and forwarding the call to the values they point to etc., we declare our pattern-matching variables of type var⟨**const** Term&⟩ and apply an address combinator to them to be able to match against pointer type. Passing variables without address combinator will not type check because the value bound in each position is of pointer type, while the variable pattern expects the reference type. Note also that unlike the variable patterns of type var⟨T⟩ that contain the bound value of type T, variable patterns of type var⟨T&⟩ do not contain the value and only bind the reference to a memory location holding an actual value during matching. This is important for polymorphic classes like Term to avoid slicing. The right argument is matched in a similar manner: the constructor pattern ensures the dynamic type is Abs or App respectively, after which the address combinator ensures the actual pointers matched are not **nullptr**, forwarding match to equivalence pattern applied to a variable of underlying type **const** Term&, which in turn recursively calls this equality operator on sub-terms.

To demonstrate nesting of patterns we reimplement in *Mach7* a well-known functional solution to balancing red-black trees with pattern matching due to Chris Okasaki [198, §3.3].

```
class T {
  enum color {black,red} col;
  T∗ left; K key; T∗ right;
};
T∗ balance(T::color clr, T∗ left, const K& key, T∗ right)
{
  const T::color B = T::black, R = T::red;
  var⟨T∗⟩ a, b, c, d; var⟨K&⟩ x, y, z; T::color col;
  Match(clr, left, key, right) {
    Case(B, C⟨T⟩(R, C⟨T⟩(R, a, x, b), y, c), z, d) ...
    Case(B, C⟨T⟩(R, a, x, C⟨T⟩(R, b, y, c)), z, d) ...
    Case(B, a, x, C⟨T⟩(R, C⟨T⟩(R, b, y, c), z, d)) ...
    Case(B, a, x, C⟨T⟩(R, b, y, C⟨T⟩(R, c, z, d))) ...
    Case(col, a, x, b) return new T{col, a, x, b};
  } EndMatch
}
```

The ... in the first four case clauses above stands for

return new T{R, **new** T{B,a,x,b}, y, **new** T{B,c,z,d}};. Class T implements a tree-node type. For the details of how exactly the balancing by pattern matching works, we refer the reader to Okasaki's original manuscript [198, §3.3].

To demonstrate the openness of the library, we also implemented numerous specialized patterns that often appear in practice and are even built into some languages. For example, the following combination of regular-expression and one-of patterns can be used to match against a string and see whether the string represents a toll-free phone number.

```
rex("([0–9]+)–([0–9]+)–([0–9]+)", any({800,888,877,866,855}), n, m)
```

The regular-expression pattern takes a C++11 regular expression and an arbitrary number of sub-patterns. It then uses matching groups to match against the sub-patterns. A *one-of* pattern any simply takes an initializer-list with a set of values and checks that the subject matches one of them. Note that variables n and m are integers, not strings; their value will be bound to properly parsed integers in the local part of the number. The parsing is generic and will work with any data type that can be read from an input stream – a common idiom in C++. Should we also need the exact area code, we can mix in a variable pattern with conjunction combinator: a && any(...).

6.3 Implementation

We begin by explaining the traditional object-oriented approach to implementing first class patterns, which is based on run-time compositions through interfaces, and point to its problems and shortcomings. We then explain our approach based on compile-time composition through concepts. Finally, we detail how the patterns fit into and extend our open type switch statement [232].

6.3.1 Patterns as Objects

The idea of implementing patterns through objects is not new and has been explored before in different languages [105, 124, 204, 261]. Variations of it differ in where bindings are stored and what is returned as a result of the match, but in its most basic form it consists of the following interface:

```
struct object { // root of the object class hierarchy
  virtual bool operator==(const object&) const = 0;
};
struct pattern { // pattern interface
  virtual bool match(const object&) = 0;
};
```

A particular kind of pattern then implements pattern interface according to its logic. A value pattern, for example, will look as following:

```
struct value : pattern {
  value(const object& obj) : m_obj(obj) {}
  virtual bool match(const object& obj)
    { return m_obj == obj; }
  const object& m_obj;
};
```

The dual approach of passing a pattern to an object has also been explored, notably by Newspeak [105], however its default implementation (which is rarely overridden in practice) dispatches back to the first solution. Regardless of whether the emphasis is made on the pattern or on the object, however, the solution suffers from the same bottleneck – the cost of a virtual function call(s) and the cost of identifying the dynamic type of either the subject or the pattern. We demonstrate in §6.4.1 that the overhead incurred by this solution is incomparably larger than the overhead of our *patterns as expression templates* approach.

While the approach is open to new patterns and pattern combinators (the patterns are composed at run-time by holding references to other pattern objects), it has some design problems. For example, mismatch in the type of the subject and the type accepted by the pattern can only be detected at run-time, while in languages with built-in support of pattern matching it is typically detected at type-checking phase. The approach may also unnecessarily clutter the code by requiring lots of similar boilerplate code be written. For example, modeling n+k patterns requires additional interface for evaluating the expression. With it, we have a dilemma of whether expression should be derived from pattern, pattern from expression, or neither of those. Independently of the choice, implementation of pattern combinators will require that the class of the combinator conditionally derives from pattern, expression or both depending on which of these interfaces its arguments implement. On one hand, this will require a separate implementation of the combinator for each of the cases, while on the other it makes the combinators dependent on something that was only needed to implement n+k patterns.

6.3.2 Patterns as Expression Templates

Patterns in *Mach7* are also represented as objects, however their composition happens at compile time and is based on C++ concepts. *Concept* is the C++ community's long-established term for a set of requirements for template parameters. Concepts were not included in C++11, but techniques for emulating them with enable_if [135] have been in use for a while. Here we use the notation for *template constraints* – a lighter version of the concepts proposal [248], because the actual code is less readable due to numerous idioms used to emulate concepts. Template constraints are briefly explained in § 2.8.3.

There are two main constraints on which the entire library is built: Pattern and LazyExpression.

```
template ⟨typename P⟩ constexpr bool Pattern() {
  return Copyable⟨P⟩
      && is_pattern⟨P⟩::value
      && requires (typename S, P p, S s) {
            bool = { p(s) };
            AcceptedType⟨P,S⟩;
         };
}
```

The Pattern constraint is the analog of the pattern interface from the *patterns as objects* solution. Objects of any class P satisfying this constraint are patterns and can be composed with any other patterns in the library as well as be used in the *Match* statement. We now break down the meaning of the above definition.

Patterns can be passed as arguments of a function, which is why the constraint subsumes a Copyable constraint. Implementation of pattern combinators requires the library to overload certain operators on all the types satisfying the Pattern constraint. To avoid overloading these operators for types that satisfy the requirements accidentally, Pattern constraint is a semantic constraint and classes that claim to satisfy it have to state this explicitly by specializing is_pattern⟨P⟩ trait. The constraint introduces also some syntactic requirements, described by the **requires** clause. In particular, patterns require presence of an application operator that serves as an analog of pattern::match(**const** object&) interface method in the *patterns as objects* approach; there is no analog of the object interface, however. The Pattern constraint itself does not impose further restrictions on the type of the subject S. Patterns like wildcard pattern will leave the S type completely unrestricted, while other patterns may require it to satisfy certain constraints, model a given concept, inherit from a certain type, etc. Application operator will typically return a value of type **bool** indicating whether the pattern is *accepted* on a given subject (**true**) or *rejected* (**false**). For convenience reasons, application operator is allowed to return any type that is convertible to **bool** instead, e.g. a pointer to a casted subject, which is useful in emulating the support of *as-patterns*.

Most of the patterns are applicable only to subjects of a given *expected type* or types convertible to it. This is the case, for example, with value and variable patterns, where the expected type is the type of the underlying value, as well as with constructor pattern, where the expected type is the type of the user-defined type it decomposes. Some patterns, however, do not have a single expected type and may work with subjects of many unrelated types. A wildcard pattern, for example, can accept values of any type without involving a conversion. To account for this, the Pattern constraint requires presence of a type alias AcceptedType, which given a pattern of type P and a subject of type S returns an expected type AcceptedType⟨P,S⟩ that will accept subjects of type S with no or minimum conversions. By default, the alias is defined in terms of a nested type function accepted_type_for as following:

```
template⟨typename P, typename S⟩
  using AcceptedType = P::accepted_type_for⟨S⟩::type;
```

The wildcard pattern defines accepted_type_for to be an identity function, while variable and value patterns define it to be their underlying type. Here is an example of how variable pattern satisfies the Pattern constraint:

```
template ⟨RegularT⟩ struct var {
  template ⟨typename⟩
    struct accepted_type_for { typedef T type; };
  bool operator()(const T& t) const // exact match
    { m_value = t; return true; }
  template ⟨RegularS⟩
  bool operator()(const S& s) const // with conversion
    { m_value = s; return m_value ==s; }
  mutable T m_value; // value bound during matching
};
template ⟨RegularT⟩ struct is_pattern⟨var⟨T⟩⟩
  { static const bool value = true; };
```

Note that for semantic or efficiency reasons a pattern may have several overloads of application operator as in the example above. The first alternative is used when no conversion is required and thus the variable pattern is guaranteed to be accepted, while the second may involve a possibly narrowing conversion, which is why we check that the values compare equal after assignment. Similarly, for type checking reasons, accepted_type_for may and typically will provide several partial or full specializations to limit the set of acceptable subjects. For example, *address combinator* can only be applied to subjects of pointer types. Its implementation manifests this by deriving unrestricted case of the type function accepted_type_for from invalid_subject_type⟨S⟩. This will trigger a static assertion when its associated type type gets instantiated, resulting in a compile-time error that states that a given subject type S cannot be used as an argument of the address pattern. The second case of the type function indicates through partial specialization of class templates that for any subject of a pointer type S∗, the accepted type is going to be a pointer to the type accepted by the argument pattern P_1 of the address combinator.

```
template ⟨PatternP1⟩
struct address
{ // ...
  template ⟨typename S⟩
    struct accepted_type_for : invalid_subject_type⟨S⟩ {};
  template ⟨typename S⟩ struct accepted_type_for⟨S∗⟩ {
    typedef typename P1::template
      accepted_type_for⟨S⟩::type∗ type;
  };
  template ⟨typename S⟩
    bool operator()(const S∗ s) const
      { return s && m_p1(∗s); }
  P1m_p1;
};
```

Checking whether a given subject type can be accepted is inherently late and happens at instantiation time of the nested accepted_type_for type function and possibly parameterized application operator. For this reason, pattern's implementation may have to provide a set of overloads of the application operator that will be able to accept all possible outcomes of accepted_type_for⟨S⟩::type on any valid subject type S.

Guard and n+k patterns, the equivalence combinator, and potentially some new user-defined patterns, depend on capturing the structure (term) of lazily evaluated expressions. All such expressions are objects of some type E that must satisfy the LazyExpression constraint:

```
template ⟨typename E⟩ constexpr bool LazyExpression() {
  return Copyable⟨E⟩
      && is_expression⟨E⟩::value
      && requires (E e) {
           ResultType⟨E⟩;
           ResultType⟨E⟩ =={ eval(e) };
           ResultType⟨E⟩ { e };
         };
}
template⟨typename E⟩ using ResultType = E::result_type;
```

The constraint is again semantic and the classes claiming to satisfy it must assert it through is_expression⟨E⟩ trait. Template alias ResultType⟨E⟩ is defined to return expression's associated type result_type, which defines the type of the result of a lazily evaluated expression. Any class satisfying LazyExpression constraint must also provide an implementation of function eval that evaluates the result of the expression. Conversion to the result_type should call eval on the object in order to allow the use of lazily evaluated expressions in the contexts where their eagerly computed value is expected: e.g. non-pattern matching context of the right hand side of the *Case* clause. Class var⟨T⟩, for example, models concept LazyExpression as following:

```
template ⟨RegularT⟩ struct var {
  // ...definitions from before
  typedef T result_type; // type when used in expression
  friend const result_type& eval(const var& v) // eager evaluation
    { return v.m_value; }
  operator result_type() const { return eval(*this); }
};
```

To capture the structure of an expression, the library employs a commonly used technique called "expression templates" [256,257]. It captures the structure of expression through the type, which for binary addition may look as following:

```
template ⟨LazyExpression E1, LazyExpression E2⟩
struct plus {
  E1 m_e1; E2 m_e2; // subexpressions
  plus(const E1& e1, const E2& e2) : m_e1(e1), m_e2(e2) {}
  typedef decltype(std:: declval ⟨E1::result_type ⟩()
                 + std:: declval ⟨E2::result_type ⟩()
                  ) result_type ; // type of result
  friend result_type  eval(const plus& e)
    { return eval(e.m_e1) + eval(e.m_e2); }
  friend plus operator+(const E1& e1, const E2& e2) noexcept
    { return plus(e1, e2); }
};
```

The user of the library never sees this definition, instead she implicitly creates its objects with the help of overloaded **operator**+ on any LazyExpression arguments. The type itself models LazyExpression concept as well so that the lazy expressions can be composed. Notice that all the requirements of the concept are implemented in terms of the requirements on the types of the arguments. The key point to efficiency of expression templates is that all the types in the final expression are known at compile time, while all the function calls are trivial and fully inlined. Use of new C++11 features like move constructors and perfect forwarding allows us to ensure further that no temporary objects will ever be created at run-time and that the evaluation of the expression template will be as efficient as a hand coded function.

In general, an *expression template* is an algebra $\langle T_C, \{f_1, f_2, ...\}\rangle$ defined over the set $T_C = \{\tau \mid \tau \vDash C\}$ of all the types τ modeling a given concept C. Operations f_i allow one to compose new types modeling concept C out of existing ones. In this sense, the types of all lazy expressions in *Mach7* stem from a set of few possibly parameterized basic types like var⟨T⟩ and value⟨T⟩ (which model LazyExpression) by applying type functors plus, minus ... etc. to them. Every type in the resulting family then has a function eval defined on it that returns a value of the associated type result_type. Similarly, the types of all the patterns stem from a set of few possibly parameterized patterns like wildcard, var⟨T⟩, value⟨T⟩, C⟨T⟩ etc. by applying to them pattern combinators like conjunction, disjunction, equivalence, address etc. The user is allowed to extend both algebras with either basic expressions and patterns or functors and combinators.

Sets $T_{LazyExpression}$ and $T_{Pattern}$ have non-empty intersection, which slightly complicates the matter. Basic types var⟨T⟩ and value⟨T⟩ belong to both families and so are some of the combinators: e.g. conjunction. Since we can only have one overloaded **operator**&& for a given combination of argument types, we have to state conditionally whether requirements of Pattern, LazyExpression or both are satisfied in a given instantiation of conjunction⟨T_1,T_2⟩ depending on what combination of these concepts the argument types T_1 and T_2 model. Concepts, unlike interfaces, allow modeling such behavior without multiplying implementations or introducing dependencies.

6.3.3 Structural Decomposition

Mach7's support of constructor patterns C⟨T⟩(P_1,. . .,Pn) requires the library to know which member of class T should be used as the subject to P_1, which should be matched against P_2 etc. In functional languages supporting algebraic data types, such decomposition is unambiguous as each variant has only one constructor, which is thus also used as *deconstructor* – a term used by several languages [27, 122] to define explicitly decomposition of a type through pattern matching. In C++, a class may have several constructors, which may not even reflect the data members it contains, some of which may be inaccessible or unrepresentative of the class' logic etc. We thus require the user to be more explicit as to class' decomposition, which she does by specializing the library template class bindings. We decided not to use the name "deconstructor" as it is already commonly confused with "destructor" in C++. Here are the definitions required to decompose the lambda terms we introduced in §6.2:

```
template ⟨⟩struct bindings⟨Var⟩ { Members(Var::name); };
template ⟨⟩struct bindings⟨Abs⟩ { Members(Abs::var, Abs::body); };
template ⟨⟩struct bindings⟨App⟩ { Members(App::func, App::arg); };
```

Variadic macro Members simply expands each of its argument into the following definition, demonstrated here on App::func:

```
static inline decltype(&App::func) member1() noexcept { return &App::func; }
```

Each of such functions returns a pointer-to-member that should be bound in position i. The library applies corresponding members to the subject in order to obtain subjects for sub-patterns $P_1, ..., P_n$. The functions get inlined so the code to access a member in a given position becomes the same as the code to access that member directly. Note that binding definitions made this way are *non-intrusive* since the original class definition is not touched. They also respect *encapsulation* since only the public members of the target type will be accessible from within bindings specialization. Members do not have to be data members only, which can be inaccessible, but any of the following three categories:

- Data member of the target type T
- Nullary member function of the target type T
- Unary external function taking the target type T by pointer, reference or value.

Binding definitions have to be written only once for a given class hierarchy and can be used everywhere. This is also true for parameterized classes, as can be seen in an example in §6.3.5. Unfortunately, at this point C++ does not provide sufficient compile-time introspection capabilities to let the library generate these definitions implicitly.

6.3.4 Algebraic Decomposition

Traditional approaches to generalizing n+k patterns treat matching a pattern $f(x, y)$ against a value v as solving an equation $f(x, y) = v$ [199]. This interpretation is well defined when there are zero or one solutions, but alternative interpretations are possible when there are multiple solutions. Instead of discussing which interpretation is the most general or appropriate, we propose to look at n+k patterns as a *notational decomposition* of mathematical objects. The elements of the notation are associated with sub-components of the matched mathematical entity, which effectively lets us decompose it into parts. The structure of the expression tree used in the notion is an analog of a constructor symbol in structural decomposition, while its leaves are placeholders for parameters to be matched against or inferred from the mathematical object in question. In essence, *algebraic decomposition* is to mathematical objects what structural decomposition is to algebraic data types. While the analogy is somewhat ad-hoc, it resembles the situation with operator overloading: you do not strictly need it, but it is so syntactically convenient it is virtually impossible not to have it. We demonstrate this alternative interpretation of the n+k patterns with examples.

- An expression n/m is often used to decompose a rational number into numerator and denominator.
- Euler notation $a + bi$ with i being an imaginary unit is used to decompose a complex number into real and imaginary parts. Similarly, expressions $r(cos\phi + i\sin\phi)$ and $re^{i\phi}$ are used to decompose it into polar form.
- An object representing 2D line can be decomposed with slope-intercept form $mX+c$, linear equation form $aX + bY = c$ or two-points form $(Y - y_0)(x_1 - x_0) = (y_1 - y_0)(X - x_0)$.
- An object representing polynomial can be decomposed for a specific degree: a_0, $a_1X^1 + a_0$, $a_2X^2 + a_1X^1 + a_0$ etc.
- An element of a vector space can be decomposed along some sub-spaces of interest. For example a 2D vector can be matched against $(0,0)$, aX, bY, $aX + bY$ to separate the general case from those when one or both components of vector are 0.

Expressions i, X and Y in these examples are not variables, but named constants of some dedicated type that lets the expression be generically decomposed into orthogonal parts. Note also that the linear equation and the two-point form for decomposing lines already include an equality sign, which makes it hard to give them semantics in an equational approach. It turns out that the equational approach can be generically expressed in our framework for many interesting cases of interest.

Applying equational approach to floating-point arithmetic creates even more problems. Even when the solution is unique, it may not be representable by a given floating-point type and thus not satisfy the equation. Once we settle for an approximation, we open ourselves to even more decompositions that become possible with our approach.

- Matching n/m with integer variables n and m against a floating-point value can be given semantics of finding the closest fraction to the value.
- Matching an object representing sampling of some random variable against expressions like $Gaussian(\mu,\sigma^2)$, $Poisson(\lambda)$ or $Binomial(n,p)$ can be seen as distribution fitting.
- Any curve fitting in this sense becomes an application of pattern matching. Precision in this case can be a global constant or explicitly passed parameter of the matching expression.

The user of our library defines the semantics of decomposing a value of a given type S against an expression of shape E by overloading a function:

```
template ⟨LazyExpressionE, typename S⟩ bool solve(const E&, const S&);
```

The first argument of the function takes an expression template representing a term we are matching against, while the second argument represents the expected result. Note that even though the first argument is passed with const-qualifier, it may still modify state in E. For example, when E is var⟨T⟩, the application operator for const-object that will eventually be called will update a mutable member m_value. The following example defines a generic solver for multiplication by a constant:

```
template ⟨LazyExpressionE, typename T⟩
    requires Field⟨E::result_type⟩()
bool solve(const mult⟨E,value⟨T⟩⟩& e, const E::result_type& r)
    { return solve(e.m_e1,r/eval(e.m_e2)); }
template ⟨LazyExpressionE, typename T⟩
    requires Integral⟨E::result_type⟩()
bool solve(const mult⟨E,value⟨T⟩⟩& e, const E::result_type& r) {
    T t = eval(e.m_e2);
    return r%t ==0 && solve(e.m_e1,r/t);
}
```

The first overload is only applicable when the type of the result of the sub-expression models the Field concept. In this case, we can rely on the presence of a unique inverse and simply call division without any additional checks. The second overload uses integer division, which does not guarantee the unique inverse, and thus we have to verify that the result is divisible by the constant first. This last overload combined with a similar solver for addition of integral types is everything the library needs to know to handle properly the definition of the fib function from §6.2. It also demonstrates how an equational approach can be generically implemented for a number of expressions.

A generic solver capable of decomposing a complex value using the Euler notation is very easy to define by fixing the structure of expression:

```
template ⟨LazyExpression E1, LazyExpression E2⟩
   requires SameType⟨E1::result_type,E2::result_type⟩()
bool solve(
      const plus⟨mult⟨E1,value⟨complex⟨E1::result_type⟩⟩⟩,E2⟩& e,
      const complex⟨E1::result_type⟩& r);
```

As we mentioned in §6.2, the template facilities of C++ resemble pattern-matching facilities of other languages. Here, we essentially use these compile-time patterns to describe the structure of the expression this solver is applicable to: $e_1 * c + e_2$ with types of e_1 and e_2 being the same as type on which a complex value c is defined. The actual value of the complex constant c will not be known until run-time, but assuming its imaginary part is not 0, we will be able to generically obtain the values for sub-expressions.

Our approach is largely possible due to the fact that the library only serves as an interface between expressions and functions defining their semantics and algebraic decomposition. The fact that the user explicitly defines the variables she would like to use in patterns is also a key as it lets us specialize not only on the structure of the expression, but also on the types involved. Inference of such types in functional languages would be hard or impossible as the expression may have entirely different semantics depending on the types of arguments involved. Concept-based overloading simplifies significantly the case analysis on the properties of types, making the solvers generic and composable. The approach is also viable as expressions are decomposed at compile-time and not at run-time, letting the compiler inline the entire composition of solvers.

An obvious disadvantage of this approach is that the more complex expression becomes, the more overloads the user will have to provide to cover all expressions of interest. The set of overloads will also have to be made unambiguous for any given expression, which may be challenging for novices. An important restriction of this approach is its inability to detect multiple uses of the same variable in an expression at compile time. This happens because expression templates remember the form of an expression in a type, so use of two variables of the same type is indistinguishable from the use of the same variable twice. This can be worked around by giving different variables (slightly) different types or making additional checks as to the structure of expression at run-time, but that will make the library even more verbose or incur a significant run-time overhead.

6.3.5 Views

Any type T may have an arbitrary number of *binding*s associated with it, which are specified by varying the second parameter of the bindings template – *layout*. The layout is a non-type template parameter of an integral type that has a default value and is thus omitted most of the time. Support of multiple bindings through layouts in our library effectively enables a facility similar to Wadler's *views* [264]. Consider:

```
enum { cartesian = default_layout, polar }; // Layouts
template ⟨typename T⟩ struct bindings⟨std::complex⟨T⟩⟩ { Members(std::real⟨T⟩,std::imag⟨T⟩); };
template ⟨typename T⟩ struct bindings⟨std::complex⟨T⟩, polar⟩ { Members(std::abs⟨T⟩, std::arg⟨T⟩); };
template ⟨typename T⟩ using Cartesian = view⟨std::complex⟨T⟩⟩;
template ⟨typename T⟩ using Polar = view⟨std::complex⟨T⟩, polar⟩;
  std::complex⟨double⟩ c; double a,b,r,f;
  Match(c)
    Case(Cartesian⟨double⟩)(a,b)) ...// default layout
    Case( Polar⟨double⟩)(r,f)) ...// view for polar layout
  EndMatch
```

The C++ standard effectively enforces the standard library to use Cartesian representation [131, §26.4-4], which is why we choose the Cartesian layout to be the default. We then define bindings for each layout and introduce template aliases for each of the views. *Mach7* class view⟨T,l⟩ binds together a target type with one of its layouts, which can be used everywhere where an original target type was expected.

The important difference from Wadler's solution is that our views can only be used in a match expression and not as a constructor or arguments of a function etc.

6.3.6 Match Statement

The *Match* statement presented in this chapter extends the efficient type switch from Chapter 5. The core of this extension amounts to ability of detecting polymorphic arguments and using only them for efficient type switching, as well as the ability to accept patterns in case clauses.

Given a statement *Match*(e$_1$,...,e$_N$) applied to arbitrary expressions e_i, the library introduces several names into the scope of the statement: e.g. number of arguments N, subject types subject_type$_i$ (defined as **decltype**(e$_i$) modulo type qualifiers), number of polymorphic arguments M etc. When $M > 0$ it also introduces the necessary data structures to implement efficient type switching [232]. Only the M arguments whose subject_type$_i$ are polymorphic will be used for fast type switching.

For each case clause *Case*(p$_1$,...,p$_N$) the library ensures that the number of arguments to the case clause N matches the number of arguments to the *Match* statement, and that the type P$_i$ of every expression p$_i$ passed as its argument models the Pattern concept. Initially we allowed case clauses to accept less than N patterns, assuming the missing patterns to be the wildcard, however, brittleness of the macro system made us reconsider this. The problem is that macro system is blind to C++ syntax and template instantiation like A⟨B,C⟩ used in a pattern will be treated by the preprocessor as 2 macro arguments. This resulted in errors that were hard for the users to comprehend. For each subject_type$_i$ it then introduces target_type$_i$ into the scope of the case clause, defined as the result of evaluating type function P$_i$::accepted_type_for⟨subject_type$_i$⟩. This is the type the pattern expects as an argument on the subject of type subject_type$_i$ (§6.3.2), which is used by the type switching mechanism to properly cast the subject if necessary. The library then introduces names match$_i$ of type target_type$_i$& bound to properly casted subjects and available to the user in the right-hand side of the case clause in case of a successful match. The qualifiers applied to the type of match$_i$ reflect the qualifiers applied to the type of subject e$_i$. Finally, the library generates the code that sequentially applies each pattern to properly casted subjects, making the clause's body conditional:

```
if (p1(match1) && ...&& pN(matchN)) { /* body */ }
```

When type switching is not involved, the generated code implements the naive backtracking strategy, which is known to be inefficient as it can produce redundant computations [43, §5]. More efficient algorithms for compiling pattern matching have been developed since [11, 152, 164, 165, 210]. Unfortunately, while these algorithms cover most of the typical kinds of patterns, they are not pattern agnostic as they make assumptions about semantics of concrete patterns. A library-based approach to pattern matching is agnostic of the semantics of any given user-defined pattern. The interesting research question in this context would be: what language support is required to be able to optimize open patterns. While we do not address this question in its generality, our solution makes a small step in that direction.

The main advantage from using pattern matching in *Mach7* comes from the fast type switching weaved into the *Match* statement. It effectively skips case clauses that will definitely be rejected because their target types are not subtypes of subjects' dynamic types. This, of course, is only applicable to polymorphic arguments, for non-polymorphic arguments the matching is done naively with cascade of conditional statements.

6.4 Evaluation

We performed three independent studies of our pattern matching solution to test its efficiency and impact on compilation process. In the first study we compare various functions written with pattern matching and hand-optimized manually in order to estimate the overhead added by the composition of patterns (§6.4.1). We demonstrate this overhead for both our solution based on compile-time composition of patterns and the suggested alternatives based on the run-time composition of patterns. In the second study we compare a relevant impact on the compilation times brought by both approaches (§6.4.2). In the last study, we rewrote a code optimizer of an experimental language from Haskell into C++ with *Mach7*. We compare the ease of use, readability and maintainability of the original Haskell code and its *Mach7* equivalent (§6.4.3).

The studies involving performance comparisons have been performed in *Windows Laptop* configuration, described in §5.6.1, using the same timing methodology.

6.4.1 Pattern Matching Overhead

The overhead associated with pattern matching may originate from several different sources:

- Naive (sequential and often duplicated) order of tests due to a pure library solution.
- Compiler's inability to inline the test expressed by the pattern in the left-hand side of the case clause (e.g. due to lack of [type] information or complexity of the expression).
- Compiler's inability to elide construction of pattern trees when used in the right-hand side of the case clause.

To estimate the overhead introduced by commonly used *patterns as objects* approach (§6.3.1) and our *patterns as expression templates* approach (§6.3.2), we implemented several simple functions with and without pattern matching. The handcrafted code we compared against was hand-optimized by us to render the same results, without changes to the underlying algorithm. Some functions were implemented in several ways with different patterns in order to show the impact of different patterns and pattern combinations on performance. The overhead of both approaches on a range of recent C++ compilers is shown in Table 6.1.

		Open Patterns					Patterns as Objects				
		G++			Visual C++		G++			Visual C++	
Test	Patterns	4.5.2	4.6.1	4.7.2	10.0	11.0	4.5.2	4.6.1	4.7.2	10.0	11.0
factorial$_0^*$	1,v,_	**15%**	**13%**	**17%**	85%	35%	347%	408%	419%	2121%	1788%
factorial$_1$	1,v	0%	6%	0%	83%	21%	410%	519%	504%	2380%	1812%
factorial$_2$	1,n+k	7%	9%	6%	78%	18%	797%	911%	803%	3554%	3057%
fibonacci*	1,n+k	17%	**2%**	2%	62%	15%	340%	431%	395%	2730%	2597%
gcd$_1^*$	v,n+k,+	21%	25%	25%	309%	179%	1503%	1333%	1208%	8876%	7810%
gcd$_2$	1,n+k,_	5%	13%	19%	373%	303%	962%	1080%	779%	5332%	4674%
gcd$_3$	1,v	**1%**	0%	**1%**	38%	15%	119%	102%	108%	1575%	1319%
lambdas*	&,v,C,+	58%	54%	56%	**29%**	**34%**	837%	780%	875%	259%	289%
power	1,n+k	10%	8%	13%	50%	6%	291%	337%	338%	1950%	1648%

Table 6.1: Pattern Matching Overhead

The experiments marked with $*$ correspond to the code of their respective functions shown in §6.2. The rest of the functions, including all the implementations with "patterns as objects" approach, are available on the project's web page. The patterns involved in each experiment are abbreviated as following:

1 – value pattern	**n+k** – n+k pattern
v – variable pattern	**+** – equivalence combinator
_ – wildcard pattern	**&** – address combinator
	C – constructor pattern

It is easy to see that the overhead incurred by compile-time composition of patterns in *patterns as expression templates* approach is significantly smaller than the overhead of run-time composition of patterns in *patterns as objects* approach. In fact, in several cases, shown in the table in bold font[1], the compiler was able to eliminate the overhead entirely and even generate a faster code. In case of "lambdas" experiment, the advantage was mainly due to the underlying type switch, while in the rest of the cases the generated code was feeding better the instruction pipeline and the branch predictor.

In each experiment, the handcrafted baseline implementation was the same in both cases (compile-time and run-time composition) and reflected our idea of the fastest code without pattern matching describing the same algorithm. For example, gcd$_3$ was implementing the fast Euclidian algorithm with remainders, while gcd$_1$ and gcd$_2$ were implementing its slower version with subtractions. The baseline code was correspondingly implementing fast Euclidian algorithm for gcd$_3$ and slow for gcd$_1$ and gcd$_2$.

The comparison of the overhead incurred by both approaches would be incomplete without detailing the implementation of patterns as objects solution. In particular, dealing with objects in object-oriented

[1] Note that this is the opposite from the convention used in Table 5.1, where bold font was used to indicate slower code

languages often involves heap allocation, subtype tests, garbage collection etc., which all can significantly affect the performance. To make this comparison applicable to a wider range of object-oriented languages, we took the following precautions in the "objects as patterns" implementations:

- All the objects involved were stack-allocated or statically allocated. This measure was taken to avoid allocating objects on the heap, which is known to be much slower and is an optimization compilers of many object-oriented languages undertake.
- Objects representing constant values as well as patterns whose state does not change during pattern matching (e.g. wildcard and value patterns) were all statically allocated.
- Patterns that modify their own state were constructed only when they were actually used, since a successful match by a previous pattern may return early from the function.
- Only the arguments on which pattern matching was effectively performed were boxed into the object class hierarchy, e.g. in case of power function only the second argument was boxed.
- Boxed arguments were statically typed with their most derived type to avoid unnecessary type checks and conversions: e.g. object_of⟨**int**⟩&, which is a class derived from object to represent a boxed integer, instead of just object&.
- No objects were returned as a result of a function as in truly object-oriented approach that would typically require heap allocation.
- n+k patterns that effectively require evaluating a result of an expression where implemented with an additional virtual function instead that simply checks whether a result is a given value. This does not allow expressing all n+k patterns of *Mach7*, but was sufficient to express all those involved in the experiments and allowed us to avoid heap allocating the results.
- When run-time type checks were unavoidable (e.g. inside pattern::match implementation) we were comparing type-ids first, and only when the comparison failed were invoking the dynamic cast. This typical optimization avoids the much slower dynamic cast in the common case.

With these precautions in place, we believe that the main overhead of the patterns-as-objects solution that we were measuring was in the cost of a virtual function call (pattern::match) and the cost of run-time type identification and conversion on its argument – the subject. Both are specific to the approach and not to our implementation of it, which is why we believe a similar overhead will be present in other object-oriented languages following this strategy to pattern matching.

6.4.2 Compilation Time Overhead

Several people expressed their concerns about possible significant increase in the compilation time due to openness of our pattern matching solution. While this might be the case for some future patterns that require a lot of compile time computations, it is certainly not the case with any of the patterns we have developed and included in the library thus far. All the existing patterns are simple top-down instantiations that rarely go beyond standard overload resolution or occasional enable_if condition. Nevertheless, to disperse these worries we compared relative increase or decrease of compilation time for each of the examples discussed in §6.4.1. We slightly modified each code to compile either the handcrafted version or the pattern-matching one. We removed the definition of the unused function in each case, however we kept the set of included headers unchanged to avoid affecting of overall timing by the time spent in the parser. We then measured the time spent in the compilation phase of the compiler and report the percentage increase of the slower one against the faster one. Each compilation time has been measure 3 times and the median was used in the actual comparison.

		Open Patterns			Patterns as Objects		
		G++	Visual C++		G++	Visual C++	
Test	Patterns	4.7.2	10.0	11.0	4.7.2	10.0	11.0
factorial$_0^*$	1,v,_	1.65%	1.65%	2.95%	7.10%	**10.00%**	10.68%
factorial$_1$	1,v	2.46%	1.60%	10.92%	7.14%	0.00%	1.37%
factorial$_2$	1,n+k	2.87%	3.15%	3.01%	8.93%	4.05%	**3.83%**
fibonacci*	1,n+k	3.66%	1.60%	2.95%	11.31%	**4.03%**	1.37%
gcd$_1^*$	v,n+k,+	4.07%	4.68%	**0.91%**	9.94%	2.05%	8.05%
gcd$_2$	1,n+k,_	1.21%	1.53%	**0.92%**	8.19%	**2.05%**	**2.58%**
gcd$_3$	1,v	2.03%	3.15%	7.86%	5.29%	2.05%	0.08%
lambdas*	&,v,C,+	18.91%	7.25%	**4.27%**	4.57%	**3.82%**	0.00%
power	1,n+k	2.00%	6.40%	3.92%	8.14%	0.13%	4.02%

Table 6.2: Compilation Time Overhead

As can be seen in Table 6.2, most of the time the handcrafted code was compiling faster (entries indicated with regular font), however in few cases the pattern-matching code was compiling faster (entries indicated in bold font). In any case the difference in compilation times was not significant – on average 3.99% for open patterns and 4.84% for patterns as objects. Bear in mind that we were compiling relatively small files with only difference in a single function and that the difference on real-world projects with significant percentage of non-pattern matching code will be even less noticeable.

6.4.3 Rewriting Haskell Code in C++

For this experiment we took an existing code in Haskell and asked its author to rewrite it in C++ with *Mach7*. The code in question is a simple peephole optimizer for an experimental GPU language called *Versity*. We assisted the author along the way to see which patterns he uses and what kind of mistakes he makes.

Somewhat surprising to us we found out that pattern-matching clauses generally became shorter, but their right-hand side became longer. The shortening of case clauses was perhaps specific to this application and mainly stemmed from the fact that Haskell does not support equivalence patterns or equivalence combinator and had to use guards to relate different arguments. This was particularly cumbersome when optimizer was looking at several arguments of several instructions in the stream, e.g.:

```
peep2 (x1:x2:xs) =
  case x1 of
    InstMove a b →
      case x2 of
        InstMove c d | (a ==d) && (b ==c) → peep2 $ x1:xs
  ...
```

compared to *Mach7* version:

```
  Match(∗x1,∗x2) {
    Case(C⟨InstMove⟩(a,b), C⟨InstMove⟩(+b,+a)) ...
  ...
```

Haskell also requires to use wildcard pattern in every unused position of a constructor pattern (e.g. InstBin _ _ _ _), while *Mach7* allows one to omit all the trailing wildcards in constructor patterns (e.g. C⟨InstBin⟩()). And while this pays off only for constructor patterns with at least 3 arguments, the use of named patterns avoided many repeated pattern expressions and actually improved both performance and readability:

```
auto either  = val(src) || val(dst);
Match(inst) {
  Case(C⟨InstMove⟩(_,       either )) ...
  Case(C⟨InstUn⟩  (_, _,    either )) ...
  Case(C⟨InstBin⟩ (_, _, _, either )) ...
} EndMatch
```

The disadvantage for *Mach7* was coming after pattern matching, as we had to explicitly manage memory when inserting, removing or replacing instructions in the stream as well as explicitly manage the stream itself. Eventually we could hide some of this boilerplate behind smart pointers and other standard library classes.

During the initial rewrite into C++ the developer simply mapped Haskell patterns into their *Mach7* equivalents, at which point we intervened and showed how some of the patterns can be expressed simpler. We reiterated the process until we could not improve the patterns ourselves without getting into the details of the actual simplifier. At one point we noticed that the developer started passing data via pattern-matching variables var⟨T⟩ in and out of functions, which was not the intended use. This was hard to prevent statically, because in general, as first-class citizens, patterns can be passed in and out of functions, however in this case they were not used as patterns, but rather as values.

6.5 Discussion

Presented pattern-matching library provides fairly standard functional style pattern-matching facilities for C++. The solution is open to new patterns, with the traditional patterns implemented as an example. It is also non-intrusive and can be applied retroactively.

We generalize n+k patterns to arbitrary expressions by letting the user define the exact semantics of such patterns. Our approach is more general than traditional approaches as it does not require an equational view of such patterns. It also avoids hardcoding the exact semantics of n+k patterns into the language.

In the future we would like to implement an actual language extension capable of working with open patterns and look into how code for such patterns can be optimized without hardcoding the knowledge of pattern's semantics into the compiler. We would like to experiment with other kinds of patterns, including those defined by the user; look at the interaction of patterns with other facilities in the language and the standard library; make views less ad hoc etc. For example, standard containers in C++ do not have the implicit recursive structure present in data types of functional languages and viewing them as such with views would incur significant overheads. We will experiment with very general patterns as first-class citizens.

6.5.1 Limitations

While our patterns can be saved in variables and passed over to functions, they are not true first-class citizens. In fact, we believe that the notion of being a first-class citizen in a language should be updated to take parametric polymorphism of C++ into account. The reason is that as is, our solution does not allow for composing patterns based on user's input (e.g. by letting user type in a pattern to match against). This can potentially be solved by mixing "patterns as objects" approach in, however the performance overhead we saw in §6.4.1 is too costly to be adopted.

7. REFINING TYPE SYSTEM[1]

> There will always be things we wish to say
> in our programs that in all known languages
> can only be said poorly.
>
> Alan J. Perlis

Often time a property of interest can be enforced within an extended type system, not directly supported by general-purpose programming language. In this chapter, we demonstrate that C++ type system is powerful enough to model many interesting domain-specific typing relations in a library setting. For example, it is often useful to track sign of a number or the associated unit of quantity, safety of its value (taintedness analysis) etc. The facility is general and is based on *customizable* subtyping relation that the user can define on his types. The library then extends it to traditional typing relations on functions, arrays, variants and other standard and built-in types. As a demonstration of this approach, we have successfully implemented two domain-specific type systems that have previously required preprocessor: a type system for type qualifiers, as well as a type system for regular expression types.

This work has been done in collaboration with Dr. Jaakko Järvi, who came up with the idea of harnessing C++ type system to model the semantic subtyping relation of regular-expression types. While working on the implementation of his idea, I came across Jeff Foster's PhD Thesis [96] and realized that we can extend the framework to model the subtyping of type qualifiers as well. From then on we were trying to separate the core of the XTL framework from any given type system we model, allowing the user to extend it to new domain-specific type systems. Esam Mlaih contributed an implementation of XTL's XML I/O based on Expat XML Parser [91]. The content of this chapter is based on the published article [235, 236].

7.1 Introduction

It is in general not possible to decide statically the exact set of all safe programs (programs whose behavior for all inputs is specified by the language semantics). Type systems of practical programming languages can only approximate this set, rejecting some safe programs, and accepting some unsafe ones. For example, the **if** statement below is rejected by a C++ compiler, even though the type-incorrect execution path would never be taken, and the initialization of j is accepted, even though i will always lead to a "division by zero" error:

```
int i = 1;
if (i ==1) i = 0; else i = "error";
int j = 1/i;
```

Replace i ==1 in the condition with an arbitrarily complex computation, and it is evident why practical type systems have this behavior: it is too inefficient, or impossible, to statically keep track of computations with certain abstractions to guarantee safety. There are, however, many abstractions for which ensuring their safe use with a type system would be neither inefficient nor impossible. Consider the following piece of code, accepted by a C++ compiler:

[1] Reprinted with permission from "Extending Type Systems in a Library: Type-Safe XML Processing in C++" by Yuriy Solodkyy, Jaakko Järvi, 2011. Science of Computer Programming, Volume 76, Issue 4, Pages 290-306, Copyright 2010 by Elsevier B.V.

```
double w; // width in cm
double h; // height in m
  ...
double p = 2 *(w + h); // perimeter; oops, incompatible units!
```

The variables obviously correspond to physical quantities, but the *units* of those are outside of the type system, and the easy error goes undetected.

There are numerous domain-specific abstractions for which type systems could in principle guarantee important run-time invariants—but the abstractions are not modeled as part of the type system of the programming language used. Of course, many type systems for domain-specific abstractions have been developed. For example, type systems for rejecting incorrect computations with physical quantities, such as the one in our example above, can be found [143]. As other examples, there are type systems for tracking memory usage errors with a *non-null* annotation [89, 97, 98], automatically detecting format-string security vulnerabilities [223], keeping track of positive and negative values [50], ensuring that user pointers are never dereferenced in kernel space [139], preventing data races and deadlocks [32], and so forth. All of the above type systems can be based on annotating types with different kinds of *type qualifiers*, and tracking their use in expressions.

We note that none of the above type systems has found their way to mainstream languages. Whether programmers can benefit from such type systems becomes a question of whether the abstractions involved are common enough and safety properties important enough to warrant complicating the specification of a general-purpose programming language and the implementation of its compilers and interpreters. It is clear that programming languages cannot be extended to support typing disciplines for every possible domain. Ideally, it would be possible to *extend* type systems to guarantee run-time invariants of new domain-specific abstractions.

Work towards extensible type systems exists. Chin, Markstrum and Millstein [50] share our view that language designers cannot anticipate all of the practical ways in which types may be refined in a particular type system in order to enforce a particular invariant. The proposed solution is a framework for user-defined *type refinements*, allowing programmers to augment a language's type system with new type annotations to ensure invariants of interest. The framework allows the generation of a type checker based on declarative rules. Other work with similar goals includes that of optional, "pluggable" type systems [34]. While clearly beneficial, the above kind of frameworks has not yet found widespread use.

In this chapter, instead of a special purpose framework, we advocate a more lightweight mechanism for refining type systems with domain-specific abstractions: as software libraries. We show that most of type refinements presented, for example, in [50, 98, 163], and available through dedicated frameworks can also be implemented as a library in a general-purpose programming language, namely C++. Our approach is therefore to refine the C++ type system with domain-specific abstractions via libraries. The underlying C++ type system cannot obviously be altered—by *refining* the type system we primarily mean defining the *convertibility* relations between data types of particular domains, and how these user-defined data types behave with respect to the built-in types of C++. The approach is constrained by what can be expressed in a library, and thus some capabilities of special purpose frameworks are not offered. For example, some frameworks [50] ensure the soundness of the generated type-systems, which our library solution does not automatically guarantee.

Libraries taking the role of the type checker have been proposed before. E.g., C++ libraries for tracking physical units are presented in [15, 35]. The introduction of several recent programming techniques and foundational C++ libraries, however, enable a more disciplined approach to defining such type system refinements—such that separately defined refinements compose. In this chapter, we collect these techniques together, and show how to apply them for refining the C++ type system.

7.1.1 Summary

The contributions of this chapter can be summarized as following:

- We identify the necessary library tools for extending the C++ type system for domain-specific types and abstractions.
- We identify the necessary language features of C++ that enable the definition of an arbitrary "subtyping" relation (in quotes, since a conversion may be needed; we omit the quotes from now on).
- We provide a library of primitives for easy extension of the C++ type system with user-defined (sub)typing rules.
- We demonstrate with two extensive examples: a type system for building type-qualifiers and a type system for XML documents. The latter can, for example, guarantee statically that a program only produces XML documents that are valid according to a given XML Schema.

The library and all examples are available for download [237].

To whet an appetite, consider the following example using our XML type-checking library:

```
typedef alt⟨seq⟨Name, Email⟩, seq⟨Name, Tel⟩⟩ old_contact;
typedef seq⟨Name, alt⟨Email, Tel⟩⟩ new_contact;
  ...
new_contact n = old_contact();
```

The alt and seq types represent, respectively, alternation and sequencing of XML elements while Name, Email, and Tel types represent particular XML elements. Thus, old_contact and new_contact are types of objects that represent fragments of XML data. We discuss these types in detail in Section 7.4.2. Our library statically assures that the two XML types are compatible and that the initialization of n with an object of type old_contact is safe, and generates the necessary code to conduct such a transformation. With our library, arbitrarily complex XML schemas can be represented as C++ types. These types provide static guarantees about dynamic content of their values, can aid in parsing conforming XML documents, and provide safe conversion operations between fragments of XML data.

7.2 Necessary Building Blocks for Type System Refinements

To be able to refine a type system in a library, several capabilities are required of the host language: first, the host language must support some form of metaprogramming, that is, the definition and evaluation of compile-time computations; second, a representation of types needs to be accessible to metaprograms; and third, the host languages should support non-intrusively grouping types into different classes defined by metaprograms, and then defining operations and functions that work for types belonging to one or more of such classes. The toolbox of a C++ programmer has grown significantly during the recent years, and can support these key capabilities. Below, we identify several techniques and libraries that are invaluable for defining type refinements.

7.2.1 Language for Metaprogramming

The ability to express interesting typing rules necessitates that one can encode computations in a library. C++ templates are a Turing-complete language [259] allowing arbitrary computations on types and constants to be performed at compile time. Such *template metaprograms* [258] have found uses in various C++ libraries (see e.g. Boost.type_traits [162], and numerous other libraries in the Boost library collection [30]). Template metaprogramming, however, remained a relatively ad-hoc activity until the introduction of the Boost Metaprogramming Library (MPL) [1, 119]. MPL provides a solid foundation for metaprogramming in C++, defining essentially a little programming language and a supporting library for defining *metafunctions*; in MPL *metafunctions* are functions from types to types. MPL allows one to define higher-order metafunctions, lambda metafunctions etc., and provides a host of data structures and algorithms for storing and manipulating types.

For complex typing rules, a framework like MPL is essential; we use MPL extensively to define relations between types, in particular in the user-defined subtyping relation. For example, the following application of the is_subtype metafunction determines whether the types old_contact and new_contact shown above are in a subtyping relation:

```
is_subtype⟨old_contact, new_contact⟩::type;
```

Following the conventions of MPL, the result of the is_subtype metafunction is not a Boolean constant but rather a type, either mpl::true_ or mpl::false_.

7.2.2 Constraining and Specializing Functions Based on Arbitrary Properties of Types

Type systems typically define in which context the use of objects of certain types is allowed, what operators between objects of different types are allowed, and so forth. With metafunctions, it is possible to define arbitrary sets of types and relations between types. The ability to *enable* or *disable* functions based on conditions defined by arbitrary metafunctions then allows one to define the contexts where the specified sets of types are valid. This ability is offered with the enable_if templates [135, 136]. We use these templates to enable certain operations, such as assignment, only when its parameters are in a subtyping relation. For example, the following assignment operation is defined only if the right-hand side of the assignment is an *arithmetic* type:

```
class A {
  . . .
  template ⟨typename T⟩
    typename enable_if⟨is_arithmetic⟨T⟩, A&⟩::type operator=(const T&);
};
```

The first argument to enable_if is a condition, a metafunction that has to evaluate to true for the assignment operator to be considered as a candidate for overload resolution; the is_arithmetic metafunction is defined in [162] and now also in the C++ Standard Library [131]. The second argument is the type of the entire enable_if⟨...⟩::type type expression in the case where the condition is true. Thus, in the definition above, the return type of the assignment operation is A&.

In addition to function overloads, the enable_if templates can be applied to enable and disable class template specializations based on arbitrary conditions.

7.2.3 Access to Structure of Types

To be able to define typing rules and conversion operators based on structural properties of types, a representation of the structure of types must be accessible to template metaprograms. The **class** construct of C++ is not useful in this regard—apart from modest (and inadequate for our purposes) compile-time reflection obtained by clever uses of templates, C++ offers no language support for inspecting the structure of classes at compile time. Instead of classes, we thus use the *tuple* types from the Boost Fusion Library [73]. When types are represented as nested instantiations of tuples, their structure becomes accessible to metaprograms; we can inspect and manipulate tuples with Boost Fusion's algorithms at compile time.

Fusion draws its design from that of MPL, but where, say, an MPL vector only contains types, a Fusion tuple contains types and values. Similar to MPL, we can define metafunctions in Fusion, but Fusion's metafunctions can also have a run-time component: Fusion's metafunctions map types to types and also values to values.

As a simple demonstration of the functionality offered by the Fusion library, below we first create a tuple type, populate its elements with values, define a function object that prints out its argument, filter out all non-arithmetic types from the tuple, and then print out the values that remain:

```
typedef fusion::tuple⟨std::string, int, char⟩ grading_record;
grading_record rec = fusion::make_tuple("Humpty Dumpty", 89, 'B');

struct print {
  template ⟨typename T⟩ void operator()(const T& x) const { std::cout ≪ x; }
}
fusion::for_each(filter_if⟨is_arithmetic⟨_⟩⟩(rec), print());
```

Both MPL metafunctions (such as is_arithmetic) and "traditional" function objects (such as print()) can be given as inputs to Fusion algorithms. Some Fusion algorithms, for example transform, requires a *hybrid* of a metafunction class and a function object. This algorithm transforms a tuple to another tuple, potentially transforming both the types and the values of the elements. We use Fusion tuples to represent XML types and Fusion algorithms in defining implicit conversions between XML types.

7.2.4 Variants and Other Powerful Abstractions

During the past few years, several new C++ Libraries that implement new "language constructs" have been introduced. The Boost Variant [99] and the Boost Optional [41] libraries, both of which provide notable new functionality to C++, are very useful in expressing complex types. For example, in the code below, Contact is a discriminated union type that can hold an object of any of the three types Email, Tel, or ICQ; MiddleName is a type that possibly contains a string:

```
typedef boost::variant⟨Email, Tel, ICQ⟩ Contact;
typedef boost::optional⟨std::string⟩ MiddleName;
```

Such *alternation* and *optionality* are central in XML typing.

We point out that Boost MPL, Fusion, Variant, and Optional, even enable_if, have all been developed using the *generic programming* methodology (see e.g. [103,189]), building their interfaces to a large extent against common *concepts* (in the technical sense of Stepanov, as established with the Standard Template Library [240]). As a result, the above libraries are highly interoperable. For example, the list of the element types of a variant type can be viewed as an MPL sequence, and thus manipulated either with the MPL or Fusion algorithms; enable_if expects MPL metafunctions as its condition argument, and so forth. Though mere libraries, the above set extends the C++ language in a significant way.

Though we have not attempted to implement XTL in any other language besides C++, we note that, e.g., Haskell supports the key capabilities that we identified as necessary for implementing type refinements. Template Haskell [224] offers powerful metaprogramming capabilities, as well as access to the representations of data types. The type class system of Haskell supports non-intrusive classification of types, and also a form of metaprogramming; for example, the approach of *strongly typed heterogeneous collections* [146] is very similar to that of Boost Fusion.

7.3 XTL: an eXtensible Typing Library

In this section, we demonstrate how the library techniques from the previous section enable extending the C++ type system. We present the rationale and design of the library components that support this task. We refer to these components and the accompanying conventions collectively as the *eXtensible Typing Library* (*XTL*). The core of XTL is very small, consisting of only a handful of template definitions, providing hooks for extension. A particular domain-specific type system refinement is achieved by extending the core according to XTL's conventions, which provide a uniform interface for each refinement, and interoperability between them.

We start with a simple example, presented in [50], that extends the integer type with qualifiers pos and neg to track statically when a value is positive or negative. A straightforward wrapping of a type with a template provides a simple but incomplete solution, shown in Figure 7.1. The constructors and the assignment and conversion operators are supposed to capture the relationship of the new type pos⟨T⟩ and the original type T. The technique of capturing such a relationship between types can be traced back to the early days of C++ [241, §6.3.2]. Objects of the underlying numeric type (T) can be used to initialize objects of pos⟨T⟩ and neg⟨T⟩ types. This is an unsafe operation and thus equipped with a run-time assertion. Conversions back to the underlying numeric type are safe, and provided with the user-defined conversion operators to T. How the pos and neg qualifiers behave with various operators is encoded by overloading those operators; here we show the overloads of **operator**+.

```
template ⟨typename T⟩ class pos {
  T m_t;
public:
  explicit  pos(const T& t) : m_t(t)   { assert (t  >0);  }
  operator T() const { return m_t; }
};
template ⟨typename T⟩ class neg;
template ⟨typename T⟩ pos⟨T⟩ operator+(const pos⟨T⟩& a, const pos⟨T⟩& b);
template ⟨typename T⟩    T  operator+(const pos⟨T⟩& a, const neg⟨T⟩& b);
template ⟨typename T⟩    T  operator+(const neg⟨T⟩& a, const pos⟨T⟩& b);
template ⟨typename T⟩ neg⟨T⟩ operator+(const neg⟨T⟩& a, const neg⟨T⟩& b);
```

Figure 7.1: A straightforward implementation of type qualifiers pos and neg. We only show the definition of the class pos and the definition of **operator**+; class neg, and other operators are defined analogously.

This straightforward solution is fairly limited. When using solely the types pos⟨T⟩ and neg⟨T⟩, the behavior is well-defined, but the interaction of these types with other types, either built-in or user-defined, or with other possible qualifiers, is not. We can identify several questions, the answers to which are not clear in this simple approach. What is the relationship between the element type T and type pos⟨T⟩? The provided constructor and conversion operator make them convertible to one another, but does this conversion lose any semantic information? Can values of one type always be implicitly converted to and used in place of the other? Are these types in a subtyping relation? How about the relationship of instantiations of pos and neg with different element types? What should, e.g., be the relationship between pos⟨**int**⟩ and pos⟨**double**⟩? The straightforward approach is lacking in many respects.

7.3.1 XTL Subtyping

The central notion in XTL is a *user-definable subtyping* relation (not based on inheritance). XTL sets the policies of how the subtyping relation is extended for new user-defined types, and provides the general building blocks to make the task effortless. In particular, when a user defines a type to be a subtype of another type, the rest of the framework assures that objects of the first type can be used in contexts where objects of the second type are expected. Note that even though we use the term *subtyping*, a conversion may in some cases be involved, e.g., in the case of XML types, described in Section 7.4.2. As part of defining the XTL subtyping relation for data types of a domain, the programmer defines the necessary conversion operators as well. In practice, when an object of a subtype is used in the context where supertype is expected, a user-defined conversion is often implicitly performed.

XTL's user-extensible subtyping relation is defined by the is_subtype metafunction: if the metafunction invocation is_subtype⟨S, T⟩::type evaluates to mpl::true_, XTL considers the type S to be a subtype of type T. The metafunction's implementation consists of a primary template and a set of template specializations. The primary template is defined as follows:

```
template ⟨class T, class U, class Cond=void⟩ struct is_subtype : is_same⟨T, U⟩ {};
```

The is_same metafunction, defined in the standard library [131, §20.5.5], compares types for equality; XTL's subtyping relation is thus reflexive. The third template parameter Cond is a hook that allows (with the help of the enable_if template, see §7.2.2) one to attach an arbitrary type predicate to a template specialization, to determine when the specialization is enabled.

New pairs of types are added to the subtyping relation by partially or explicitly specializing the is_subtype template. To refine a type system for a particular domain, it generally suffices to specialize is_subtype for the types specific to that domain. Once these basic relations have been established, XTL provides an elaborate set of ready to use subtyping algorithms for compound types, including support for subtyping of function types (see §7.3.2), standard container types, type sequences and discriminated unions (see §7.4.1), and types refined with type qualifiers (see §7.3.4) similar to those described in [97].

The XTL subtyping relation is fully under the control of the programmer. It is also non-intrusive as the classes and types involved do not have to be altered when extending the is_subtype metafunction. For example, if deemed useful, the C++ type **char** can be defined to be a subtype of **int**, **int** a subtype of **double**, **double** a subtype of complex⟨**double**⟩, and so forth. In fact, such safe conversions are commonly useful, so they are available through inclusion of a dedicated XTL header file. The following assignments demonstrate the effect of these rules:

```
neg⟨int⟩       ni (−20);
neg⟨double⟩ nd = ni;        // OK
ni  = nd;                   // error
```

7.3.2 Subtype Casting

Since physical representations of values in different types may vary, our definition of subtyping relation implies existence of a unified conversion mechanism capable of transforming physical representation of a subtype into a physical representation of a supertype. Such a conversion in XTL is accomplished with the subtype_cast function template, invoked as subtype_cast⟨T⟩(val), where T represents a supertype of val's type. This function deduces the type of val and, if it is a subtype of T, converts val to an object of type T. Otherwise the subtype_cast function template is disabled with the enable_if mechanism, and a call to it results in a compile-time error.

We demonstrate the use of the XTL subtyping relation and subtype_cast with subtyping of function types, instances of the std::function template from the standard library [131, §20.7.16.2]. Consider the following types A and B:

```
typedef tainted⟨int⟩ A;
typedef     pos⟨int⟩ B;
```

that are in a subtyping relationship B <: A; the tainted type qualifier is explained in § 7.3.4:

We further define two function types B_to_A and A_to_B. The function types are instances of the std::function template, which is the standard library's generic wrapper for different kinds of function types, giving a uniform interface to function pointers, pointers to member functions, and function objects.

```
typedef std::function⟨A(B)⟩ B_to_A; // function type B → A
typedef std::function⟨B(A)⟩ A_to_B; // function type A → B
```

According to the usual subtyping rules between function types (covariant on return types, contravariant on parameter types), a function type A → B is a subtype of B → A, and we thus would like to be able to use functions of type A_to_B everywhere where functions of type B_to_A are expected, but not vice versa. For example:

```
A f(B);
B g(A);
B_to_A b2a = &f; // Wrap f into std::function
A_to_B a2b = &g; // Wrap g into std::function
b2a = a2b;       // OK, but rejected!
a2b = b2a;       // Error
```

The first assignment above, even though safe, is rejected. If implicit conversions apply between corresponding argument types and return types of two functions, std::function defines an implicit conversion between those two function types. This is of no help, since there is no implicit conversion from B to A. XTL, however, recognizes the subtyping relation between A_to_B and B_to_A, and with a subtype_cast the safe assignment is accepted:

```
b2a = subtype_cast⟨B_to_A⟩(a2b); // OK
a2b = subtype_cast⟨A_to_B⟩(b2a); // Error
```

XTL derives the subtyping fact A_to_B <: B_to_A from a general subtyping rule for function types, and the rules for type qualifiers that establish the fact pos⟨**int**⟩<: tainted⟨**int**⟩. All these facts are expressed by extending the is_subtype metafunction, and all conversions are performed using subtype_cast. As a consequence, types recognized by the XTL compose. For example, instead of types A and B above, the parameter and return types of the above functions could be other function types, or any other types recognized by the XTL, and the framework would recursively check whether the appropriate subtyping relations between those types hold. The framework thus allows extension with many domain-specific types independently, resulting in the expected behavior when the types are used together.

Before we proceed to revising the example in Figure 7.1, we note that explicit casting is not very elegant. In practice, we can often avoid it by providing implicit conversions, either as constructors in the supertype or conversion operators in the subtype that perform the call to subtype_cast. Such implicit conversions are not, however, possible when both types involved in the conversion are types that the developer of a type system cannot alter, such as built-in or standard types. In generic code, it is advisable not to rely on implicit conversions between types encoded as part of the XTL framework, but rather use subtype_cast explicitly if conversions are necessary. This will ensure that the code works with all applicable types. We follow this rule consistently in XTL.

7.3.3 Revisiting pos and neg

To demonstrate the use of the XTL's subtyping relation, we rewrite our naïve implementation of the pos and neg class templates, extend the is_subtype metafunction appropriately, and define the subtype casts. The pos class template is shown in Figure 7.2. The definition of neg is similar.

```
template ⟨typename T⟩ class pos {
  T m_t;
public:
  explicit pos(const T& t) : m_t(t) { assert(t >0); }
  template ⟨typename U⟩
    pos(const U& u, typename enable_if⟨is_subtype⟨U, pos⟨T⟩⟩, void⟩::type* = 0)
      : m_t(subtype_cast⟨T⟩(u)) {}
  template ⟨typename U⟩
    typename enable_if⟨is_subtype⟨U, pos⟨T⟩⟩, pos&⟩::type operator=(const U& u) {
      m_t = subtype_cast⟨T⟩(u);
      return *this;
    }
  operator T() const { return subtype_cast⟨T⟩(*this); }
};
```

Figure 7.2: Revisited definition of the pos class template.

We can observe that there is a new constructor in the pos class. Though taking two arguments, the second one has a default value, and thus the constructor implements an implicit conversion. The first argument seemingly matches any type, but in reality, only types that are subtypes of pos⟨T⟩ will be considered. This is made possible by the second parameter that acts as a guard: the constructor is only enabled if U is defined to be a subtype of pos⟨T⟩ by the is_subtype metafunction. The rather complex type expression boils down to the type **void*** when the function is enabled, thus the parameter can accept the default value 0. This is an idiomatic use of the enable_if template when placing a constraint to a constructor. The body of the constructor performs a conversion between the representations using the subtype_cast function.

The assignment operation has the same guard as the converting constructor described above. The condition is now, however, expressed as part of the return type of the operator. Again, this is idiomatic use of enable_if. The effect is that an object of type U can be assigned to a variable of type pos⟨T⟩ exactly when U is a subtype of pos⟨T⟩.

The subtyping rules for pos are defined outside the pos class, by specializing the is_subtype metafunction:

```
template ⟨typename S⟩
struct is_subtype⟨pos⟨S⟩, S⟩ : mpl::true_ {};
template ⟨typename S, typename T⟩
struct is_subtype⟨pos⟨S⟩, pos⟨T⟩⟩ : is_subtype⟨S, T⟩ {};
```

Here, the first specialization states that a pos type is a subtype of its element type, and the second that two pos types are in a subtyping relation when their element types are.

Calls to the subtype_cast function express the target type as an explicitly specified template parameter. This function is disabled when the source type is not a subtype of the target type, but otherwise it performs the cast by delegating the task to the subtype_cast_impl function. The target type is carried in the type of the first argument to subtype_cast_impl. This arrangement makes extending XTL with new types easier. First, the disabling condition in subtype_cast remains the same for all types, and does not need to be repeated for each extension. Second, some extensions require partially specializing the target type. We can rely on the function overloading mechanism for this with the subtype_cast_impl functions, where the target type is deducible. In contrast, the target type template parameter in subtype_cast is explicitly specified; partially specializing such parameters is not supported in C++.

The subtype_cast_impl function is thus the function overloaded when extending XTL with new types. According to the above subtyping rules, we overload subtype_cast_impl to specify how to convert between a subtype and a supertype in the case of pos-qualified types:

```
template ⟨typename T⟩
T subtype_cast_impl(target⟨T⟩, const pos⟨T⟩& p) { return p.m_t; }
template ⟨typename T, typename S⟩
pos⟨T⟩ subtype_cast_impl(target⟨pos⟨T⟩⟩, const pos⟨S⟩& p) {
  return pos⟨T⟩(subtype_cast⟨T⟩(p.m_t));
}
```

Definitions of operators now also change slightly to take subtyping into account:

```
template ⟨typename T, typename U⟩
pos⟨decltype(std::declval⟨T⟩()+std::declval⟨U⟩())⟩ operator+(const pos⟨T⟩& a, const pos⟨U⟩& b) {
  typedef decltype(std::declval⟨T⟩()+std::declval⟨U⟩()) result_type;
  return pos⟨result_type⟩(a.m_t + b.m_t);
}
```

7.3.4 Type Qualifiers

The pos and neg qualifiers presented above are a simple example of an important direction for enriching type systems: refining a type with qualifiers. Type qualifiers modify existing types to capture additional semantic properties of the values flowing through the program. A well-known example of a type qualifier is the **const** qualifier of C++, used for tracking immutability of values at different program points. Other examples include type qualifiers for distinguishing between user and kernel level pointers [139], safe handling of format strings [223], and tracking of values with certain mathematical properties [50].

Instead of implementing different type qualifiers to type systems in an ad-hoc manner, several systems, based on a general theory of type qualifiers, have been described [96–98]. These systems allow an economical definition of behavior of domain-specific qualifiers.

In this section, we review common properties of type qualifiers, and show how to implement type qualifiers as a C++ template library using the XTL framework. To give a brief example, we use *taintedness* analysis [223] that uses the qualifiers untainted and tainted to tag data coming from trustworthy and potentially untrustworthy sources, respectively. The requirement is that tainted data may never flow where untainted data is expected. We may want to ensure that, say, data originating from measurements, considered as trustworthy (untainted) data, is not mixed with tainted data from untrustworthy sources (e.g. assumptions, values obtained from modeling etc.) to produce untrustworthy results. Note that type qualifiers can be composed—besides trustworthiness, values may have other properties we want to track: positiveness, constness, measurement units etc. The following pseudocode involves multiple type qualifiers applied to the same type:

```
extern untainted kg double get_weight();
const kg double a = get_weight();   // OK, untainted dropped
kg untainted double b = a;          // Error, no untainted in the right −hand side
b = get_weight();                   // OK, qualifiers are preserved
```

We discuss later in this section how to verify type safety of an assignment that involves multiple type qualifiers; here we just note that the order of application of type qualifiers to a type should not matter—it does not in our framework—and types that differ only in the order of qualifiers should be semantically equivalent. In what follows, by *qualified type* we mean a type that is obtained through applying one or more type qualifiers to an *unqualified type*.

As with pos and neg, we represent a type qualifier as a template class with a single parameter that represents the type being qualified. By taking advantage of the common properties of all type qualifiers, we can reduce the work that is necessary for defining a new qualifier. The developer of a type qualifier explicitly marks his template class as a type qualifier through specialization of a traits-like class is_qualifier. It is not necessary to provide new specializations for the is_subtype metafunction or overloads for the subtype_cast_impl functions. The behavior follows according to whether the qualifier is *positive* or *negative* [97], which the programmer states in the definition of the qualifier class. An example definition is shown in Figure 7.3.

Definition 6.
A type qualifier q *is* positive *if* T <: q T *for all types* T *for which* q T *is defined.*
A type qualifier q *is* negative *if* q T <: T *for all types* T *for which* q T *is defined.*

The C++ qualifier **const**, type qualifier tainted [223], and optional [41] are examples of positive type qualifiers because T <: **const** T, T <: tainted⟨T⟩, and T <: optional⟨T⟩, respectively. Qualifiers pos, nonzero, and untainted are examples of negative type qualifiers because pos⟨T⟩<: T, nonzero⟨T⟩<: T, and untainted⟨T⟩<: T.

As mentioned above, the order of applying type qualifiers to a type should not affect the resulting type's behavior. Thus, definitions of operations on type qualifiers must be made ignorant of the particular order of qualifiers. Directly overloading an operator for a particular qualifier, as in the addition operator in §7.3.3, is insufficient. We note that type qualifiers do not change the underlying operation, only the type of the result. For example, when we apply the pos qualifier to type **int** we still use the addition operation defined on **int**s, but ask pos to be applied to the result type whenever both argument types are qualified with pos. Consequently, XTL defines generic operations that match all qualified types and ignore the order of qualifiers. Using metafunctions, these operators can then be customized with the typing rules for particular qualifiers. For example, the rules for how the untainted and tainted type qualifiers are propagated in the addition operation are as follows:

```
untainted  + untainted → untainted
untainted  + tainted   → tainted
tainted    + untainted → tainted
tainted    + tainted   → tainted
```

The encoding of typing rules for the addition operator is done by specializing XTL's add_op template:

```
template ⟨template⟨typename⟩ typename A, template⟨typename⟩ typename B⟩
struct add_op { typedef mpl::identity⟨mpl::_1⟩ type; };
```

For the tainted and untainted rules above, the specializations are as follows:

```
template ⟨typename T⟩
struct untainted : negative_qualifier⟨typename unqualified_type⟨T⟩::type⟩ {
  typedef negative_qualifier⟨typename unqualified_type⟨T⟩::type⟩ base;

  untainted() : base() {}
  explicit untainted(const typename base::unqualified_type& t) : base(t) {}

  template ⟨typename U⟩
  explicit untainted(const U& u, typename enable_if⟨is_subtype⟨U, untainted⟨T⟩⟩, void⟩::type* = 0)
    : base(subtype_cast⟨T⟩(u)) {}

  template ⟨typename U⟩
  typename enable_if⟨is_subtype⟨U, untainted⟨T⟩⟩, untainted&⟩::type operator=(const U& u) {
    base::operator=(subtype_cast⟨T⟩(u));
    return *this;
  }

  operator T() const { return subtype_cast⟨T⟩(m_t); }
};
```

Figure 7.3: The definition of the untainted type qualifier class using the XTL framework. The negative_qualifier class is defined in XTL, as well as the subtyping rules common to all negative qualifiers. To access the underlying unqualified type, XTL provides the unqualified_type metafunction.

```
template ⟨⟩ struct add_op⟨untainted, untainted⟩ { typedef add_qualifier ⟨untainted⟩ type; };
template ⟨⟩ struct add_op⟨untainted,   tainted⟩ { typedef add_qualifier ⟨  tainted⟩ type; };
template ⟨⟩ struct add_op⟨  tainted, untainted⟩ { typedef add_qualifier ⟨  tainted⟩ type; };
template ⟨⟩ struct add_op⟨  tainted,   tainted⟩ { typedef add_qualifier ⟨  tainted⟩ type; };
```

The class template add_op takes two qualifier templates as template template parameters, and defines a metafunction that will be applied to compute what qualifiers should be present in the result type of **operator**+. The primary template sets the default to mpl::identity: qualifiers are neither added nor removed from the result type. The metafunction add_qualifier, defined by XTL, applies a given qualifier template to the result type. XTL's generic implementation of a particular operation will loop through all possible combinations of qualifiers in the arguments' types and apply the corresponding metafunctions to compute the qualifiers that should be applied to the result type.

To arrange that a particular operator is not dependent of the order of qualifiers in its argument types (for example, untainted⟨nonzero⟨optional⟨**int**⟩⟩⟩ should have the same overloading behavior as optional⟨untainted⟨nonzero⟨**int**⟩⟩⟩), the XTL uses enable_if in the overloaded operators for qualified types. Focusing a particular operator, say, **operator**+, no separate overloads for optional⟨T⟩, untainted⟨T⟩, etc., are provided. Instead, a single overload matches any composition of qualifiers. This is arranged by guarding the overload with enable_if and the metafunction is_qualifier_type⟨T⟩. Similarly, the is_subtype metafunction, that determines when an object of one qualified type can be assigned to that of another type, is agnostic of the exact order of the qualifiers. This metafunction inspects the set of type qualifiers, and bases the subtyping relation on the negativeness or positiveness of the qualifiers. Omitting some details, to preserve subtyping, a positive type qualifier can only be added to the right-hand side, and a negative type qualifier can only be removed from the left-hand side. For example: nonzero⟨optional⟨untainted⟨T⟩⟩⟩ is a subtype of tainted⟨optional⟨nonzero⟨U⟩⟩⟩ whenever T <: U. Here the negative type qualifier untainted was dropped from the left while positive type qualifier tainted was added to the right. Other qualifiers were preserved. Dropping the optional qualifier in the right-hand side would have failed the subtyping.

To give a feel of working with type qualifiers built with XTL, Figure 7.4 shows code using some of the type qualifiers mentioned above.

```
struct SomeType {};
void foo(pos⟨int⟩)        { ... }
void foo(pos⟨SomeType⟩) { ... }
int main() {
    nonzero⟨neg⟨int⟩⟩              a(−44);
    untainted⟨pos⟨nonzero⟨int⟩⟩⟩  b(2);
    neg⟨nonzero⟨long⟩⟩          m = a *b; // OK
  //nonzero⟨pos⟨double⟩⟩        e = b − a; // Error: difference of two non−zeros can be zero
    pos⟨double⟩                 d = b − a; // OK: no nonzero
    foo(b);    // OK, picks the right overload
  //foo(d);    // Error: double is not a subtype of int
    nonzero⟨tainted⟨double⟩⟩ bc = subtype_cast⟨nonzero⟨tainted⟨double⟩⟩⟩(b);
    nonzero⟨tainted⟨double⟩⟩ bi = b; // same as above
    bi = b;
    pos⟨nonzero⟨int⟩⟩ c(3);
    std::string                 ci = subtype_cast⟨std::string⟩(c); // user defined that int <: std::
         string
    tainted⟨std::string⟩        cc = subtype_cast⟨std::string⟩(c); // OK to add a positive qualifier to
         the left
}
```

Figure 7.4: Example of working with XTL's qualifiers. Demonstrates how subtyping relation can be established between types non-intrusively. Above, the programmer has defined **int** to be a subtype of the std::string type.

Even with the subtyping and casting functionality provided by the XTL, the definition of an individual type qualifier class is still fairly elaborate, but mostly boilerplate code. For cases where no special run-time checks are needed, the XTL provides two macros for taking care of this boilerplate. For example, the two macro invocations below generate the type qualifier classes for the tainted and untainted qualifiers:

```
DECLARE_POSITIVE_QUALIFIER( tainted);
DECLARE_NEGATIVE_QUALIFIER(untainted);
```

The metafunctions add_op, sub_op etc. that describe how different operations carry the qualifiers must still be defined.

7.4 Typing XML in C++

In this section, we describe how the XTL, with the help of several C++ libraries, allow an elaborate extension to the C++'s type system: static typing of *XML* – an *eXtensible Markup Language* [277].

7.4.1 Background: Regular Expression Types

Type systems that understand XML data have gained considerable interest. The central idea is to harness the type system to guarantee statically that a particular program cannot manipulate or produce XML documents that do not conform to a particular DTD [277] or Schema [118]. The insight is that

```
⟨?xml version =”1.0” encoding=”ISO–8859–1” ?⟩
⟨xsd:schema xmlns:xsd =”http://www.w3.org/2001/XMLSchema”⟩
⟨xsd:element  name=”name” type=”xsd:string” /⟩
⟨xsd:element  name=”tel”      type=”xsd:string” /⟩
⟨xsd:element  name=”email” type=”xsd:string” /⟩
⟨xsd:element  name=”contact” ⟩
  ⟨xsd:complexType⟩
    ⟨xsd:sequence⟩
      ⟨xsd:element  ref =”name” /⟩
      ⟨xsd:element  ref =”tel”    maxOccurs=”unbounded” /⟩
      ⟨xsd:element  ref =”email” maxOccurs=”unbounded” /⟩
    ⟨/xsd:sequence⟩
  ⟨/xsd:complexType⟩
⟨/xsd:element⟩
⟨/xsd:schema⟩
```

Figure 7.5: An example XML schema.

XML data corresponds directly to *regular expression types*, which then can be given a representation in the type systems of various languages. Some of the recent efforts in this direction include the XDuce language [125], specifically designed for XML processing, that has a direct representation for regular expression types; the C$_\omega$ [22] and Xtatic [102] languages that extend C# with regular expression types; and the HaXml [267] toolset, that uses Haskell's algebraic data types to represent XML data.

A *regular expression type*, e.g., as defined in XDuce, is a set of sequences over certain domains. Values from those domains denote singleton and composite sequences. Composite sequences are formed with the *regular expression operators* “,” (*concatenation*), “|” (*alternation*), “$*$” (*repetition*), and “?” (*optionality*), together with *type constructors* of the form “$l[\cdots]$”. If S and T are types, then S, T denotes all the sequences formed by concatenating a sequence from S and a sequence from T. $S|T$ denotes a type that is a union of sequences from S and sequences from T. Type $l[T]$, where T is a type and l ranges over a set of labels, defines a set of labeled sequences where each sequence from T becomes classified with the label l. Type $T*$ denotes a set of sequences obtained by concatenating a finite number of elements from T. The empty sequence is denoted with $()$, and $T?$ denotes any sequence from T or an empty sequence.

Consider for example the following XML snippet describing a contact:

```
⟨contact⟩
  ⟨name⟩Humpty Dumpty⟨/name⟩
  ⟨tel⟩555–4321⟨/tel⟩
  ⟨email⟩humpty.dumpty@tamu.edu⟨/email⟩
⟨/contact⟩
```

This snippet conforms to the XML Schema in Figure 7.5. Labels classify types similar to how XML tags classify their content; For example, the type email[...] corresponds to ⟨email⟩...⟨/email⟩ in XML parlance. Using XDuce syntax, the regular expression type

```
contact[name[string], tel[string]∗, email[string]∗]
```

corresponds to the schema in Figure 7.5; both define the set of XML snippets with “contact” as the root element containing single “name” element, followed by zero or more “tel”, followed by zero or more “email” elements.

An interesting feature of the XDuce language is the *semantic* subtyping relation between regular expression types, defined as the subset relation between languages generated by two tree automata [125]. For example, the following subtyping relationships hold:

```
T*, U* <: (T | U)*
T, (T)* <: T*
```

A *subtyping* between two regular expression types corresponds to safe convertibility between XML fragments. Subtyping between XML fragments is useful, for example, in providing backward compatibility of documents that correspond to an older schema: code written against a newer schema should work for older schemas, as long as the type defined by the newer schema is a supertype of the type defined by the older one.

The decision problem for subtyping between regular expression types is EXPTIME-hard [126, 221] in the worst case, but the cases that lead to this worst-case complexity are reported to be rarely seen in practice [3].

7.4.2 Regular Expression Types in C++

We define an *encoding* of regular expression types in C++. Regular expression types are represented as nested template instantiations, consisting of sequence types, variants, and lists. We represent XML elements in our system as a struct parameterized with two types, the first of which represents the element's tag and second the element's data:

```
template <typename Tag, typename T = detail::empty> struct element { T data; };
```

The Tag type denotes the name of the XML element, or the label in XDuce's regular expression types. Empty XML elements can be represented by an element instantiated with nothing but a tag type. Complex XML elements may have several levels of instantiations of element as their data type. Consider for example the following XML snippet:

```
<contact><name>Humpty Dumpty</name></contact>
```

It can be given the following type in our library:

```
struct contact { ... }; struct name { ... };
typedef element<contact, element<name, string >> Contact;
```

Tag-classes are also used to keep additional information about the tag: the name as a character array, XML node type, additional restrictions etc. For example, the full definition of the contact tag is:

```
struct contact {
  static const char* tag_name() { return "contact"; }
  typedef attribute node_type;
};
```

Sequencing of XML elements is represented with the seq template. Here is an example of using sequencing of elements:

```
typedef element<name,     string >  Name;
typedef element<tel ,     string >  Tel;
typedef element<email,    string >  Email;
typedef element<contact,  seq<Name, Tel, Email>> Contact;
```

The empty sequence () is represented by seq<>.

The seq template is a simple wrapper around Fusion's tuple class [73]. We use the wrapper to change the behavior of certain operations, e.g., to perform preprocessing on the tuple types. For example, we define the I/O operators for seqs to perform a *flattening* of sequences prior to delegating the call to Fusion's tuple I/O. For example, the sequence (A, B, (C, D), E) is flattened to (A, B, C, D, E). Fusion tuples are MPL-compliant sequences [1,119], and in our subtyping algorithm we operate on tuples using MPL algorithms.

Alternation of XML elements is represented with the alt class template that is again just a simple wrapper, now around Boost's variant template [99]. In case of alternation, the wrapping is done to allow the redefinition of the I/O routines. Here is a small example of using alternation:

```
typedef alt⟨Tel, Email⟩ ContactInfo;
typedef element⟨contact, seq⟨Name, ContactInfo, ContactInfo⟩⟩ AlternativeContact;
```

The empty union is represented by alt⟨⟩.

A repetition of XML elements is represented with the rep template, a wrapper over a std::vector of Fusion tuples. Using repetition, the contact definition from Section 7.4.1 can be expressed as follows:

```
typedef element⟨contact, seq⟨Name, rep⟨Tel⟩, rep⟨Email⟩⟩⟩ FlexibleContact;
```

7.4.2.1 Subtyping Relation

We utilize the static metaprogramming capabilities of C++ to establish a *subtyping* relation between two regular expression types. Again, the is_subtype metafunction is harnessed for this purpose. We note one restriction. XDuce allows the definition of recursive data types, and can decide subtyping between right/tail recursive data types (the decision problem of subtyping between general recursive data types is undecidable). An implementation of subtyping between recursively-defined data types, analogous to that in XDuce, has so far eluded us. However, we support *repetition*, which is functionally equivalent, but possibly syntactically more cumbersome, to right/tail recursion (analogously to the equivalence between right-linear grammars and regular expressions). The restriction is that subtyping of regular expression types with repetition, is weaker than in XDuce—there are cases where two regular expression types are in the semantic subtyping relation, but the is_subtype metafunction does not agree.

The implementation of is_subtype metafunction for XML types is lengthy and we do not show it here. It amounts to implementing the subtyping rules of XDuce (really a limited form of them per the restrictions mentioned above) using MPL. Once the is_subtype metafunction has been defined to recognize our XML types, we can exploit it to implement guards similar to those in the constructors of the qualified types—Figure 7.6 demonstrates with the converting constructor of the element class template.

We do not overload subtype_cast_impl to define conversions on element types because we define such conversions on the element class itself. Calling subtype_cast on the element type then calls the most general implementation of subtype_cast_impl, which simply tries to apply either a standard or a user-defined conversion on the type, which is exactly what we need.

7.4.3 Working with the Library

In addition to the core type system, we have implemented some supporting functionality as part of our XML processing library. This includes I/O and automatic generation of the C++ types from an XML Schema. For I/O, we provide direct streaming operations. To automate generation of the C++ types from XML types, we provide an XSL transformation from an XML Schema to C++ source code. To give a general feel for the use of our XML framework, Figure 7.7 presents a complete example of turning an XML Schema into the corresponding C++ encodings of XML types, and working with them. The comments in the code point out the parts of the code that were generated from an `.xsd` file (the

```
template ⟨typename TTag, typename T = detail::empty⟩
class element {
  ...
  template ⟨typename UTag, typename U⟩
  element(const element⟨UTag, U⟩&,
          typename enable_if⟨
                      is_subtype⟨element⟨UTag, U⟩, element⟨TTag, T⟩⟩
                    ⟩::type∗ = 0) { ...}
};
```

Figure 7.6: The converting constructor of the element class template. The enable_if guard allows the constructor to match exactly when the argument's type is a subtype of the class to be constructed, according to the library's type system.

XML Schema description), where our support functionality (input and output) is invoked, and where the subtyping checks are performed.

To demonstrate the practicality of the library, as a larger example, we generated XML types for two established Internet standards for syndication, the RSS [28] and Atom [117], and modeled a subtyping relation between documents of these standards, allowing thus a type safe conversion from one to another. The standards themselves do not provide a ready to use XML schema to validate the documents. Several slightly different schemata exist; our XML types are generated from the schemas in [278].

To establish a subtyping relation between the types representing the two different syndication formats, call these types T_{RSS} and T_{Atom}, we defined a mapping, presented in Table 7.1, between tags from one schema (RSS) to another (Atom), making T_{RSS} a subtype of T_{Atom}. In many cases, a tag in one schema had a natural "semantic" match in the other. For example, an RSS tag *pubDate* indicating when an item was published, can be matched to Atom's tag *published* with the same meaning. Similarly, RSS's notion of *author* can be represented with Atom's broader notion of *contributor*. With "subtagging" relations defined, the subtyping relation between XML types are taken care of by the XTL. Both feeds accept unrecognized XML tags (for future extensibility) via an "extension element" any[any]. Elements in RSS that do not have a match in Atom are thus matched to a wildcard element any[any]. Our implementation of any_element is a supertype of all XML element types. Objects of the any_element type simply store the XML source code of an element cast to it, and thus does not lose information. For our experiment, we ignored some of the differences in representations of data stored as strings within certain XML elements. For example, we did not account for differences in the date formats. We also dropped attributes from XML elements, as our current implementation does not support them. Atom's feed element consists of a single repetition of a large alternation, which we unrolled to a sequence of three such repetitions (which is semantically equivalent), to match the structure of the RSS's channel element.

Table 7.1 shows the regular expression types representing the schema of the RSS and Atom feeds. We further zoom in to the types representing a single news story within a feed: item in RSS and entry in Atom. These are the most interesting types; the elements whose definitions we omit, are much simpler.

7.4.4 Impact on Compilation Times

Heavy use of template metaprogramming is known to increase compile times of C++ programs, often significantly. We conducted experiments to estimate the impact that library-defined type systems written using the XTL have on compile times. We tested both the uses of type qualifiers and the use of the

RSS	Atom
item[(^{1}title[string]? | ^{2}description[string]? | ^{3}link[anyURI]? | ^{4}author[tEmailAddress]? | ^{5}category[tCategory]? | ^{6}guid[tGuid]? | ^{7}pubDate[tRfc822FormatDate]? | ^{8}source[tSource]? | ^{0}comments[anyURI]? | ^{0}enclosure[tEnclosure]? | ^{0}any[any]*)+]	entry[(^{1}title[textType] | ^{2}content[contentType]? | ^{3}link[linkType]* | ^{4}author[personType]* | ^{5}category[categoryType]* | ^{6}id[idType] | ^{7}published[dateTimeType]? | ^{8}source[textType]? | contributor[personType]* | rights[textType]? | summary[textType]? | updated[dateTimeType] | issued[dateTimeType] | modified[dateTimeType] | ^{0}any[any]*)*]
channel[(^{1}title[string] | ^{2}link[anyURI] | ^{3}category[tCategory]? | ^{4}copyright[string]? | ^{5}managingEditor[tEmailAddress]? | ^{5}webMaster[tEmailAddress]? | ^{6}lastBuildDate[tRfc822FormatDate]? | ^{6}pubDate[tRfc822FormatDate]? | ^{7}generator[string]? | ^{8}image[tImage] | ^{0}description[string] | ^{0}language[language]? | ^{0}docs[anyURI]? | ^{0}cloud[tCloud]? | ^{0}ttl[nonNegativeInteger]? | ^{0}textInput[tTextInput]? | ^{0}skipHours[tSkipHoursList]? | ^{0}skipDays[tSkipDaysList]? | ^{0}any[any]*)+, ^{9}item[tRssItem]+, ^{0}any[any]*]	feed[(^{1}title[textType] | ^{2}link[linkType]* | ^{3}category[categoryType]* | ^{4}rights[textType]? | ^{5}contributor[personType]* | ^{6}updated[dateTimeType] | ^{7}generator[generatorType]? | ^{8}icon[iconType]? | ^{9}entry[entryType]* | author[personType]* | id[idType] | logo[logoType]? | subtitle[textType]? | ^{0}any[any]*)+]

Table 7.1: RSS and Atom types for news items and feeds. Tags with the same numbers in both columns represent our semantic matching between tags.

```
using namespace xml;
using namespace std;
// −−− definitions  generated by XSLT transformation −→
struct name { ... }; struct email   { ... };
struct tel   { ... }; struct contact { ... };
typedef element⟨name,    string ⟩                     Name;
typedef element⟨email,   string ⟩                     Email;
typedef element⟨tel ,      alt ⟨string , int ⟩⟩        Tel;
typedef element⟨contact, seq⟨Name, Tel, Email⟩⟩ Contact;
typedef element⟨contact, seq⟨Name, rep⟨Tel⟩, rep⟨Email⟩⟩⟩ ContactEx;
// ←− definitions  generated by XSLT transformation −−−
int main() {
  try {
    ifstream  xml("contact.xml");
    Contact contact;
    xml ≫ contact;                       // Parse file
    ContactEx contact_ex = contact;   // OK, implicit  subtype_cast ⟨ContactEx⟩(contact)
    cout ≪ contact_ex;                 // Output XML
    // contact = contact_ex;             // Error ,  not in  subtyping relation
  }
  catch(invalid_input & x) {
    cerr ≪ "Error parsing " ≪ x. what();
    return −1;
  }
  return 0;
}
```

Figure 7.7: An example of working with our XML library. Type Contact is a subtype of ContactEx, which is why assignment contact_ex = contact is allowed while contact = contact_ex is not.

XML framework. We used the GCC 3.4.4 compiler for all tests, on Intel Pentium M processor running at 2 GHz with 512 MB of RAM.

Our test-suite for type qualifiers consisted of 11 versions of the same program, each working with types qualified with a different number of qualifiers: program with index $n \in \{0, \ldots, 10\}$ used types qualified with n qualifiers. Each program defined 20 pairs of qualified types where first type of the pair was always made to be a subtype of the second type of the pair. Pairs used various combinations of positive and negative qualifiers that preserved the subtyping relation. For all type pairs (A, B), the compiler was forced to verify subtyping as: is_subtype⟨A, B⟩::type::value;. Additionally, each program contained 20 functions, each of which instantiated values of two types from the pair and then performed a multiplication of them to trigger inference of qualifiers from an operation. Each applied qualifier had up to 4 rules defined on it. The full test suit can be obtained from XTL's website [237]. Compilation times (in seconds) for these tests are given in Table 7.2. The compile time should be compared against the value in row 0, which defines the baseline: the equivalent program without any qualifiers. Our current implementation of the subtyping algorithm for qualifiers has (compile-time) complexity $O(n^2)$ where n is the number of qualifiers applied. Though there is noticeable increase in compilation times of the last two cases, for a reasonable number of qualifiers slowdowns are moderate—ten different qualifiers applied to the same type seems unlikely. Note also, that the qualifier use in the test programs is proportionally very high—the programs contain practically no other code than code that triggers the is_subtype test

N	0	1	2	3	4	5	6	7	8	9	10
Time	1.12	1.72	2.61	2.81	3.18	3.94	4.97	5.55	6.28	15.40	19.22

Table 7.2: Compilation times, in seconds, of the type qualifier test programs.

with different inputs. Though not supported in XTL, a "release mode" that sidesteps subtype tests for faster compile times would be possible for the type qualifier library. In particular, with *template aliases*, a qualified type, such as pos⟨T⟩ for any T, could be defined to be a type alias for the unqualified type T. Template aliases are a new feature of the current C++ standard [80] [131, §14.5.7].

The subtyping relation of the XML types is computationally more expensive than that of type qualifiers. As mentioned earlier, in the general case deciding subtyping of regular expression types is EXPTIME-hard. The computationally expensive cases are subtyping relations of the following form:

$$l(A_1, \cdots, A_n) \texttt{ <: } l(B_{11}, \cdots, B_{1n})|\cdots|l(B_{k1}, \cdots, B_{kn})$$

Verification of such a relation in the general case involves a number of steps that is proportional to the number of ways a k-element set can be split into n disjoint sets. Detailed discussion can be found in [126].

We wanted to test the effect of this worst-case scenario on compile times. Our test suite for the XML type system consisted of 81 tests—one for each combination of n and k (ranging from 1 to 9) from the above relation. In each of the tests, we invoked is_subtype metafunction using XML types that trigger the exponential case with (essentially) the following code:

```
is_subtype ⟨
  element⟨a, seq⟨Ak0, ... , Akn⟩⟩,
  alt ⟨element⟨a, seq⟨A00, ... , A0n⟩⟩,
      element⟨a, seq⟨A10, ... , A1n⟩⟩,
      ... ,
      element⟨a, seq⟨Ak0, ... , Akn⟩⟩
  ⟩
⟩:: type:: value
```

n/k	1	2	3	4	5	6	7	8	9
1	1.48	1.66	1.73	1.81	2.03	2.14	2.35	2.54	2.79
2	1.52	1.77	2.29	3.67	8.28	27.38			
3	2.43	2.00	2.66	6.00	31.43				
4	2.62	2.03	3.63	25.57					
5	2.32	2.28	7.15						
6	2.73	4.13	19.30						
7	2.48	2.86	56.78						
8	1.74	3.65							
9	1.76	4.93							

Table 7.3: Compilation times, in seconds, of the test programs for the XML type system. The compilation time of an "empty" program, containing the same include files but no template instantiations, was 1.31 seconds.

Table 7.3 represents compilation times in seconds for different values of n and k. Empty entries correspond to tests that did not finish within 10 minutes or exceeded the compiler's limit on the number of nested template instantiations (500). With small n and k, effect to compile times is small. With larger values, they become infeasible as expected. Other implementations of the subtyping algorithm have been reported to behave satisfactorily on practical examples [126], suggesting that the exponential case with large n and k does not manifest often in practice.

Summarizing the test results, using our approach to refining type systems can have a notable negative impact on compile times. In the case of the type qualifiers, the slowdown is quite reasonable, and typical for libraries relying on template metaprogramming. The XML case behaves similarly, except that the pathological cases that lead to exponential growth in the cost of deciding subtyping also obviously increase the compilation times exponentially.

7.5 Discussion

Type systems are traditionally closely tied to the implementation of a compiler or an interpreter, and typically are not extensible. In this chapter, we presented a library solution for extending the type system of a general-purpose programming language with typing of domain-specific abstractions. This is a very economical and lightweight approach to building type systems. The presented solution does not require any compiler support and can be fully implemented in standard C++ [130]. We demonstrated that it is feasible to implement elaborate typing behavior purely as a library. We used our framework to build two extensions to the C++ type system: type qualifiers and regular expression types. The library of type qualifiers allows effortless definition of new type qualifiers and their order-independent composition. Regular expression types can directly describe the structure of XML documents. A subtyping relation between regular expression types can express safe conversions between different structures of XML data. For example, with our library we can write programs that are statically guaranteed to produce only XML data that conforms to a particular schema. For more convenient use of the library, we additionally provide machinery to map XML Schema definitions to corresponding library abstractions.

Besides the two domain-specific type systems, our approach can be equally well used to model other subtyping relations, like record subtyping, variant parametric types etc. In the future, we would like to explore the limits of the approach, by implementing different kinds of type system extensions in terms of the XTL tools we described. We are currently looking into extending XTL with the possibility to define co- and contravariancy of template arguments on multi-argument class templates. If proven successful, current implementation of the type qualifiers will become a partial case of this more general subtyping algorithm. Within the XML domain, we are currently working on support for attributes and integrate our pattern matching solution from chapter 6. In addition, currently we only support a basic subset of primitive XML data types. We plan to extend this support to other built-in types as well as possibly provide a compile- and run-time support of facets.

7.5.1 Limitations

One of the most important restrictions that refining type systems possesses is that it cannot handle control-flow dependence or properties that are not directly related to types. This is a consequence of the fact that we encode required semantic properties as properties of types and types in C++ can only be given to values once.

8. SAIL ABSTRACT INTERPRETATION LIBRARY

> Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.
>
> Edsger W. Dijkstra

While our approach to refining type systems could not handle control-flow dependent properties, its main advantage was that we were able to encode many interesting properties of interest without having to deal with the abstract syntax tree of a program externally from a preprocessor. Instead, we could deal with the program directly from a library by requiring the user to use its certain constructs. We take this idea into a different setting in order to let users build control-flow aware analyses without having to deal with AST of a program.

We combine all of the solutions presented thus far to the design and implementation of a generic abstract interpretation library, called *SAIL*[1]. The library allows the user to express static analyses of C++ programs without having to deal with abstract syntax trees or tricky corner cases of the language semantics. This work was mainly influenced by the calculational design of a generic abstract interpreter presented by Patrick Cousot at Marktoberdorf Summer School [61]. Francesco Logozzo's PhD thesis [160] on modular static analysis of object-oriented languages also played a major role in favor of abstract interpretation as the theoretic framework for the generic library we were trying to build.

The initial implementation of SAIL has been performed on top of the Pivot framework [81, 246], but is otherwise front end agnostic. It serves as a proof of concept, as currently it can only handle very limited subset of C++ and supports only non-relational abstract domains. The research on the framework is less concerned with the development of new abstract domains and instead concentrates on generic programming and library design issues that allow one to combine existing abstract domains into scalable, precise and useful analyses.

The work presented in this chapter has not been published yet. It represents the ideals, design and the state of the current implementation, which we would like to expand in the future.

8.1 Introduction

Abstract Interpretation is a theory of sound approximation of the semantics of computer programs, formalized and developed by Patrick and Radhia Cousot [60, 65]. General idea of abstract interpretation of imperative programs was introduced by Sintzoff [228]. The theory formalizes the relationship between the concrete semantics of a syntactically correct program and an abstract semantics, which is a safe approximation of the concrete semantics. Quoting Cousot [65]:

> ”A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the results of abstract execution give some information on the actual computations”

[1] SAIL is a recursive acronym that stands for “SAIL Abstract Interpretation Library”

The goal is to find the simplest universe of abstract objects, in which the resulting abstraction of the program semantics is precise enough to imply the desired property and coarse enough to be efficiently computable.

To give some intuition behind the theory, consider for example an arithmetic expression 4+7 and suppose that we are interested only in parity of the result. On one hand we can evaluate the result concretely obtaining the value 11 and see that it is odd, however on the other hand we may abstract each of the values by their parity Odd or Even and consider the problem of figuring out the parity of expression Even+Odd instead. The last one can be abstractly evaluated rendering Odd. The important observation is that we have abstract versions of the concrete values (4 and 7) as well as abstract versions of the concrete operations on them. When expression contains variables: $x*3+7$, its concrete evaluation relies on an environment to provide a concrete value for x, say 8, and evaluates the expression using this value. Abstract evaluation of this expression naturally relies on an *abstract environment* to provide an abstract value for x – Even. Deterministic concrete evaluations always render concrete results, however when these evaluations are mapped into the abstract, determinism of the result may be lost. Consider abstract evaluation of $\mathsf{Even} \mod \mathsf{Odd}$: on one hand it can be an abstraction of 4 mod 3 resulting in 1 (Odd), but on the other hand it can be an abstraction of 6 mod 3 resulting in 0 (Even), which means we need to have a value representing "either Odd or Even". This last value, usually represented as $\top$, is clearly less precise than either Odd or Even and during evaluation we would prefer to get as precise abstract value as possible. The precision of information between two abstract values is usually represented by a partial order $\sqsubseteq$, while the domain of abstract values in general case has to form a lattice.

This does not seem like a useful abstraction, as simple constant folding will result in a more precise answer. Consider, however, a more likely expression in a program $x = 4 + x$, which happens to be in the middle of a loop. Even if the initial value of x is 7, constant folding may not help us learn what the value of x is going to be after the loop, since the number of iterations of the loop may be unknown at compile time. Its evaluation in the abstract, however, will tell us that regardless of the number of iterations, the value of x is going to be Odd, since the initial value of $x = 7$ is Odd and the result of evaluating 4+ Odd in the abstract is Odd. Knowing that x is Odd may not be sufficient to prove some properties, like $x > 0$, however it is sufficient to prove other properties, like $x \neq 0$, often required when dividing by x.

In the next section, we present some general ideas and facts about abstract interpretation from [61, §4], [160, §2] and [62, §2, §3]. We found these to be the easiest to comprehend introductory material on abstract interpretation and suggest the reader to refer to them for a more detailed discussion.

8.2 Background

A *partially ordered set* (*poset*) $\langle L, \sqsubseteq \rangle$ is a set L with a partial order $\sqsubseteq$ that is a reflexive, antisymmetric and transitive binary relation on L. A *complete lattice* $\langle L, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ is a poset $\langle L, \sqsubseteq \rangle$ such that any subset $X \subseteq L$ has a least upper bound $\sqcup X$ (supremum). A complete lattice will have both the infimum $\bot = \sqcap \varnothing$, and the supremum $\top = \sqcup L$.

A *value property* is a set of values having the property in question. Assuming that values belong to some domain $\mathbb{C}$, $\langle \wp(\mathbb{C}), \subseteq, \varnothing, \mathbb{C}, \cup, \cap, \neg \rangle$ is a complete boolean lattice, elements of which are understood as properties of values with subset inclusion $\subseteq$ as logical implication, $\varnothing$ as false, $\mathbb{C}$ as true, $\cup$ as the disjunction, $\cap$ as the conjunction and $\neg$ as the negation.

Domain $\mathbb{C}$ can be infinite or very large so that in practice we cannot represent (or keep) any given subset of it (e.g. a set of all odd integers). Therefore, for program analysis we can only use a machine encoding $\mathbb{A}$ of a subset of all the value properties. An *abstract property* $a \in \mathbb{A}$ is the encoding of some property $\gamma(a) \in \wp(\mathbb{C})$ specified by the *concretization function* $\gamma \in \mathbb{A} \to \wp(\mathbb{C})$. It is assumed that $\langle \mathbb{A}, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ forms a complete lattice with $\sqsubseteq$ (called *approximation ordering*) being an abstract logical implication, $\bot$ encoding false, $\top$ encoding true, join $\sqcup$ representing abstract disjunction and $\sqcap$ being abstract conjunction. Since the approximation ordering $\sqsubseteq$ should encode logical implication on abstract properties, the concretization function γ is assumed to be monotone: $a_1 \sqsubset a_2 \implies \gamma(a_1) \subseteq \gamma(a_2)$.

Since $\mathbb{A}$ only approximates $\wp(\mathbb{C})$, an arbitrary value property $C \in \wp(\mathbb{C})$ may not have abstract equivalent in $\mathbb{A}$. Nevertheless, it can be overapproximated by any abstract property $a \in \mathbb{A}$ such that $C \subseteq \gamma(a)$. Looking at all abstract properties a that overapproximate given concrete property C one may notice that the intersection of meanings of all such properties $\cap\{\gamma(a) \mid C \subseteq \gamma(a)\}$ better approximates the original property C than any of the abstract properties a. The ideal situation corresponds to *Galois connections*, where for all concrete properties, the best approximation has a concrete encoding in the abstract domain. The encoding of the best approximation is provided by the *abstraction function* $\alpha \in \wp(\mathbb{C}) \to \mathbb{A}$ satisfying the following properties:

$C_1 \subseteq C_2 \implies \alpha(C_1) \sqsubseteq \alpha(C_2)$ (i.e. α preserves implication)

$\forall C \in \wp(\mathbb{C}) : C \subseteq \gamma(\alpha(C))$ (i.e. $\alpha(C)$ overapproximates C)

$\forall a \in \mathbb{A} : \alpha(\gamma(a)) \sqsubseteq a$ (i.e. γ introduces no loss of information)

The conjunction of the above properties is equivalent to:

$\forall C \in \wp(\mathbb{C}), a \in \mathbb{A} : \alpha(C) \sqsubseteq a \iff C \subseteq \gamma(a)$

which is a characteristic property of *Galois connections* denoted as: $\langle \wp(\mathbb{C}), \subseteq \rangle \leftrightarrows \langle \mathbb{A}, \sqsubseteq \rangle$.

We used value properties to introduce concrete and abstract domains as well as concretization and abstraction functions linking them. The properties of a program one may be interested in can be broader though: in some cases we might be interested in *relational properties* – value properties of a group of variables or even properties of the entire execution environment, in other cases we might be interested in *temporal properties* of a given program (e.g. which states can and cannot be reached from or follow other states, in how many steps one state can be reached from another etc.). What is common between all these properties however is that they can be represented by some lattice (whether finite or infinite and thus computable or incomputable).

Before we start abstracting certain properties away, we should have a domain capable of representing all the information about program execution. To do this, abstract interpretation starts with a standard semantics of a programming language and associates a *discrete transition system* to each program $\mathbb{P}$ of the language that is a pair $\langle \Sigma, \tau_{\mathbb{P}} \rangle$ where Σ is a (non-empty) set of states, and $\tau_{\mathbb{P}} \subseteq \Sigma \times \Sigma$ is the binary *transition* relation between a state and its possible successors. A *state* $\sigma \in \Sigma$ can informally be understood as an object representing complete state of computer's memory, its registers, state of any external devices (e.g. clock) that might affect program's execution, etc. Any input on which a program $\mathbb{P}$ runs is thus also already encoded as part of the state σ.

Given the notion of state, the theory of abstract interpretation then proceeds defining a notion of *trace* that informally represents an either finite or infinite sequence of states. The domain of traces over a given domain of states Σ is usually denoted by $T(\Sigma)$, while a concrete trace is denoted as $\sigma_0 \to \sigma_1 \to \cdots \to \sigma_n \to \cdots$. When a given set of traces was obtained by successive (possibly infinite) application of $\tau_{\mathbb{P}}$ starting from some initial set of states $S_0 \subseteq \Sigma$ (denoted as $\mathbf{tr}[\![\mathbb{P}]\!](S_0)$), such set of traces represents all possible evolutions of the discrete transition system (see program $\mathbb{P}$) starting from S_0 and thus contains all possible information about program's execution. We have to talk about a set of initial states S_0 instead of just a single initial state σ_0 because any input to program $\mathbb{P}$ only restricts part of the state: e.g., we do not know anything about the state of computer's memory not holding input values and thus have to assume that in can hold any values. This is why there are multiple initial states representing the same input to a program.

$\mathbf{tr}[\![\mathbb{P}]\!](S_0)$ defines the *semantics* of program $\mathbb{P}$ started from a set of initial states S_0. The most important bit for understanding the abstract interpretation is the fact that $\mathbf{tr}[\![\mathbb{P}]\!](S_0)$ can be computed as a fixpoint of some function, given by the following

Theorem 3. Fixpoint Partial Traces Semantics [60, 66] *Let Σ be a set of states, $\tau_{\mathbb{P}} \subseteq \Sigma \times \Sigma$ be the transition relation associated with a program $\mathbb{P}$, $S_0 \subseteq \Sigma$ be a set of initial states and $F \in \wp(\Sigma) \to \wp(T(\Sigma)) \to \wp(T(\Sigma))$ be $F(S_0) : \wp(T(\Sigma)) \to \wp(T(\Sigma)) = \lambda X.S_0 \cup \{\sigma_0 \to \dots \to \sigma_n \to \sigma_{n+1} \mid \sigma_0 \to \dots \to \sigma_n \in X \wedge \langle \sigma_n, \sigma_{n+1} \rangle \in \tau_{\mathbb{P}}\}$ Then the* partial trace semantics *of $\mathbb{P}$, $\boldsymbol{tr}[\![\mathbb{P}]\!] \in \wp(\Sigma) \to \wp(T(\Sigma))$ is $\boldsymbol{tr}[\![\mathbb{P}]\!](S_0) = \mathrm{lfp}^{\subseteq}_{\varnothing} F(S_0)$.*

The notation $\mathrm{lfp}^{\subseteq}_{\varnothing} F(S_0)$ in the above means *least fix point* that is greater than $\varnothing$ with respect to the partial ordering $\subseteq$ of a monotone function $F(S_0)$.

Galois connections compose, which is why typically we will not have to abstract the concrete semantics from the partial trace semantics every time. The theory of abstract interpretation builds Galois connections between partial trace semantics and some less precise abstract semantics. As long as we can establish Galois connection with one of those, our approximation will still be sound and equivalent to building the Galois connection directly to the partial trace semantics. This is why at each step of the abstraction process we will be talking about some *concrete semantics* – a mathematical object that belongs to a *concrete semantic domain* D, described by a partially ordered set $\langle D, \sqsubseteq \rangle$ and an *abstract semantics* – a mathematical object from an *abstract semantic domain* $\bar{D}$, given by a partially ordered set $\langle \bar{D}, \bar{\sqsubseteq} \rangle$. Because of the composition property of Galois connections, the abstract semantic domain $\bar{D}$ at one step of the abstraction process can be regarded as the concrete semantic domain D at the next step of the abstraction process. Both partially ordered sets will typically be complete lattices $\langle D, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ and $\langle \bar{D}, \bar{\sqsubseteq}, \bar{\bot}, \bar{\top}, \bar{\sqcup}, \bar{\sqcap} \rangle$ respectively.

To understand this better, consider the type of F and note that the type of partial application of it $F(S_0)$ is $\wp(T(\Sigma)) \to \wp(T(\Sigma))$ and in particular, we have a function defined on a lattice into itself. In general, the concrete $\mathbf{tr}[\![\mathbb{P}]\!]$ and abstract $\bar{\mathbf{tr}}[\![\mathbb{P}]\!]$ semantics of a program $\mathbb{P}$ are usually given in a fixpoint form as $\mathbf{tr}[\![\mathbb{P}]\!] = \mathrm{lfp}\ \ t[\![\mathbb{P}]\!]$ and $\bar{\mathbf{tr}}[\![\mathbb{P}]\!] = \mathrm{lfp}\ \ \bar{t}[\![\mathbb{P}]\!]$, where the *semantic transformers* $t[\![\mathbb{P}]\!]$ and $\bar{t}[\![\mathbb{P}]\!]$ are monotonic functions respectively on $D \to D$ and $\bar{D} \to \bar{D}$ for concrete and abstract lattices D and $\bar{D}$ respectively. If the concrete and abstract transformers satisfy the *local commutative condition* $t[\![\mathbb{P}]\!] \circ \gamma \sqsubseteq \gamma \circ \bar{t}[\![\mathbb{P}]\!]$ then the next theorem assures the soundness of abstract interpretation

Theorem 4. Fixpoint Transfer [66] *Let $\langle D, \sqsubseteq \rangle$ and $\langle \bar{D}, \bar{\sqsubseteq} \rangle$ be complete lattices. Let $t[\![\mathbb{P}]\!]$ and $\bar{t}[\![\mathbb{P}]\!]$ be monotonic functions defined respectively on $D \to D$ and $\bar{D} \to \bar{D}$. If $\langle D, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} \langle \bar{D}, \bar{\sqsubseteq} \rangle$ and if $t[\![\mathbb{P}]\!] \circ \gamma \sqsubseteq \gamma \circ \bar{t}[\![\mathbb{P}]\!]$ then $\boldsymbol{tr}[\![\mathbb{P}]\!] = \mathrm{lfp}\ \ t[\![\mathbb{P}]\!] \sqsubseteq \gamma(\mathrm{lfp}\ \ \bar{t}[\![\mathbb{P}]\!]) = \gamma(\bar{\boldsymbol{tr}}[\![\mathbb{P}]\!])$.*

What this essentially means is that if we have two lattices linked by Galois connection and any result computed by the semantic transformer $t[\![\mathbb{P}]\!]$ on the first lattice is approximated by the result computed by the semantic transformer $\bar{t}[\![\mathbb{P}]\!]$ on the second lattice (local commutative condition) then the fixpoint of semantic transformer defined on the first lattice will also be approximated by the fixpoint of the semantic transformer defined on the second lattice. In other words as long as we can prove that abstract semantic transformer properly approximates the concrete semantic transformer, the concrete semantics expressed as a fixpoint of the concrete semantic transformer on the concrete domain will be approximated by the abstract semantics expressed as a fixpoint of the abstract transformer on the abstract domain.

A general recipe of using abstract interpretation for analysis of programs can then be summarized as following:

- Design an abstract domain that forms a complete lattice with respect to an ordering that models logical implication of properties you are trying to analyze. This domain will have to be finitely representable to be computable.
- Pick as concrete lattice a domain that was previously shown to be related by Galois connection to the domain that describes semantics of a program (the domain of semantics itself in the worst case – ideally you have to establish Galois connection with the powerset of traces).
- Establish Galois connection between your concrete and abstract domains by writing down concretization and abstraction functions and showing that they satisfy the characteristic property of Galois connection.
- Show how each operation of the language transforms abstract domain remembering that the result should always overapproximate what the concrete operation will evaluate to on concrete domain with the concretization of arguments. This is to maintain local commutative condition true at all times.
- Express the concrete semantics as a fixpoint of your concrete semantic transformer.
- Express the abstract semantics as a fixpoint of your abstract semantic transformer.
- Run the fixpoint computation on the abstract domain starting with the bottom of the abstract domain. The result that you will get will safely approximate the hypothetical (as it is usually incomputable) result of running the same fixpoint computation on the concrete domain to find the concrete semantics of a program.

8.3 Putting Theory to Practice

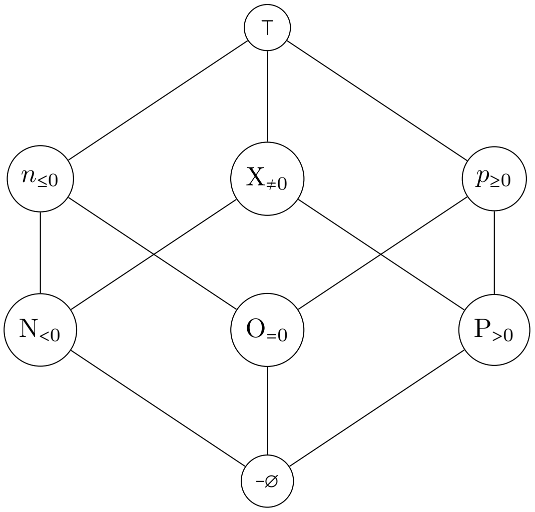
The abstract interpreter that the user will construct with SAIL is parameterized over abstract domains. Unfortunately, SAIL currently only supports *non-relational* abstract domains – domains that approximate the value properties of a given program variable at each program point. The set of states Σ is approximated with a map-lattice $\mathbf{Vars} \to \mathbf{D}$ from the set of program variables **Vars** to the domain **D** (*value domain*) that abstracts values of individual variables. The Galois connection between $\mathbf{Vars} \to \mathbf{D}$ and partial trace semantics is derived in [61, §6.3]. The user is thus restricted at the moment to only parameterization of the abstract interpreter over the value domain **D**.

To define new analyses, the user has to define the following four entities (or use pre-defined ones) in order to be able to run the analysis:

- Define abstract domain.
- Define abstractions of operations over that domain.
- Map concrete operations to their abstractions.
- Map concrete types to abstract domains.

In what follows, we demonstrate each step on the example of sign abstract domain used to perform sign analysis on values. The abstract domain is represented by the lattice shown in Figure 8.1, which is encoded with the following class definition in SAIL:

```
struct sign
```



_ – bottom of the lattice, representing empty set ∅
T – top of the lattice, representing any (all) numbers
O – number 0, or set consisting of it: $\{x \mid x = 0\}$
X – numbers excluding 0, or set consisting of it: $\{x \mid x \neq 0\}$
N – strictly negative numbers: $\{x \mid x < 0\}$
P – strictly positive numbers: $\{x \mid x > 0\}$
n – 0 or negative numbers: $\{x \mid x \leq 0\}$
p – 0 or positive numbers: $\{x \mid x \geq 0\}$

Figure 8.1: Sign lattice and the identifiers used to enumerate corresponding values

```
{
    enum values {_,N,O,n,P,X,p,T};
    sign(values v = _) : m_value(v) {}
    template ⟨typename T⟩ static sign alpha(T t);
    static bool      eq(const sign& a, const sign& b);
    static sign     top();
    static sign     bot();
    static sign    join (const sign& a, const sign& b);
    static sign    meet(const sign& a, const sign& b);
    // Non−required functions
    static bool     leq (const sign& a, const sign& b);
    static sign   widen(const sign& a, const sign& b);
    static sign  narrow(const sign& a, const sign& b);
    static sign  complement(const sign& a);
    values m_value;
};
```

Functions top, bot, join and meet define corresponding lattice operations with leq defining the partial order ⊑ on value domain sign. widen and narrow are overapproximations operators used to force convergence of fixpoint computations. alpha is the abstraction function that takes a concrete value and returns its abstraction. The user then implements abstractions of the forward and backward semantics of various operations, which are defined simply as functions on the class representing the abstract domain:

```
sign operator+(const sign& a);
sign operator+(const sign& a, const sign& b);
void solve_unary_plus (const sign& r, sign& a);
void          solve_plus (const sign& r, sign& a, sign& b);
```

The forward abstractions should overapproximate the result with the smallest abstract value representing the outcomes of concrete operation on all possible combinations of concretizations of its arguments. Backward abstractions (called solvers by the library) refine (tighten) the arguments, providing the result is known to be abstracted by r. For finite abstract domains like sign all the operations mentioned thus far can be implemented with table lookup on arguments.

In the third step, the user registers abstractions of operations that will be used in place of concrete operations in the analyzed program, for values abstracted by this domain. Note that the user only specifies the type of the concrete operation C (C,C) and its name:

```
template ⟨typename C, typename AI⟩
void register_abstractions_ex  (concrete_type⟨C⟩, abstract_type⟨sign⟩, AI& ai)
{
  // Register abstraction function
  ai . register_abstraction_function  (&sign:: alpha⟨C⟩);

  // Register unary operators
  ai . register_operation ⟨C (C)⟩  (”operator+”, &sign::operator+);

  // Register binary operators
  ai . register_operation ⟨C (C,C)⟩(”operator+”, &sign::operator+);

  // Register solvers
  ai . register_solver  ⟨C (C)⟩  (”operator+”, &solve_unary_plus);
  ai . register_solver  ⟨C (C,C)⟩(”operator+”, &solve_plus);
}
```

Registration is parameterized over concrete type C, because the library will use the same registration routine to register additionally abstractions for references, pointers and arrays of given type.

In the last step, the user tells the library which concrete type C should be abstracted with which abstract domain A, e.g:

```
register_abstractions  ⟨bool,        belnap_bool⟩    (interpreter  );
register_abstractions  ⟨char,        parity ⟩        (interpreter  );
register_abstractions  ⟨int ,        sign ⟩          (interpreter  );
register_abstractions  ⟨double,      sign ⟩          (interpreter  );
register_abstractions  ⟨long,        interval ⟨long⟩⟩(interpreter  );
register_abstractions  ⟨class_type , any_class ⟩     (interpreter  );
register_abstractions  ⟨any_type,    unit ⟩          (interpreter  );
```

With this, the user can now run semantic analyses on his programs and obtain invariants at each program point (mentioned in comments below). The following listing shows the outcome of performing abstract interpretation of Euclidian algorithm:

```
int gcd(int a, int b)
{
    // {a→ T,b→ T}
    if ((a <0) || (b <0))
        return 0; // {a→ T,b→ T}

    // {a→ p,b→ p}
    if (a ==0)
        return b; // {a→ O,b→ p}

    // {a→ P,b→ p}
    while (b ≠0)
    {
        // {a→ T,b→ X}
        if (a >b)
            a = a − b; // {a→ T,b→ X}
        else
            b = b − a; // {a→ T,b→ X}
        // {a→ T,b→ X}
    }

    return a; // {a→ T,b→ O}
}
```

The definition of abstractions of operations and solvers for them requires certain degree of scrupulousness and carefulness in order to ensure their monotonicity, while maintaining the precision of abstraction. Their semantics is discussed in details in [61]. Here we will only mention that for finite lattices we provide ways to generate their source code automatically (see Figure 8.2).

The library provides many ready-to-use abstract domains, as well as many adaptors and utility classes that can be used to build other domains:

- Belnap's four-valued logic
- Intervals
- Congruences
- Sign
- Arrays
- Classes
- Pointers and references
- Member pointers

Note that the user did not have to deal with abstract syntax tree of a program at all – everything used to define the analysis was specified with the usual C++ syntax. The library takes care of mapping these high-level abstractions into a concrete code that navigates ASTs, recognizes operations and abstractly evaluates them using the abstract operations provided.

8.4 Implementation

Function alpha represents the abstraction function α in a Galois connection. The difference is that our functions takes a single value t, while α is defined on sets. This design choice was motivated by several reasons:

- The literals in a program this function will be applied to are typically given alone, not in the set of possible values, thus it will be an extra burden on the user to convert that value to a singleton set every time
- Function α must return the smallest abstract value representing the concrete set given by its value. Ensuring this property might be not obvious for many programmers and thus we prefer to ensure it ourselves inside the library.

Here is an example definition of the function alpha for our sign abstract domain:

```
template ⟨typename T⟩ static sign sign::alpha(T t) { return t >0 ? P : t <0 ? N : O; }
```

Notice that for each concrete value the function must return the smallest abstract value representing it, e.g. P instead of p here for positive numbers etc.

Function eq defines the equality of lattice elements (not the underlying concrete values). We use this dedicated function instead of overloading traditional **operator**== because we use operators as uniform names to refer to forward abstractions of corresponding operations. Uniformity of names of different abstract operations is needed to be able to compose domains through generic programming.

Since our sign lattice is isomorphic to lattice formed by all subsets of a 3-element set, we chose the concrete lattice values to be such that the corresponding meet and join operators can be implemented with bitwise operations:

```
static sign sign::join(const sign& a, const sign& b) { return values(a.m_value | b.m_value); }
static sign sign::meet(const sign& a, const sign& b) { return values(a.m_value & b.m_value); }
```

For other finite lattices, these operators can be trivially implemented with a look-up table. Note that the library is not limited to finite lattices, which we only use here for the ease of understanding. Here is an example of implementing forward abstract semantics of addition with look-up tables:

```
sign sign::operator+(const sign& a, const sign& b)
{
    static const values op[8][8] = {
    // b=_,N,O,n,P,X,p,T    // a=
        {_,_,_,_,_,_,_,_}, // _
        {_,N,N,N,T,T,T,T}, // N
        {_,N,O,n,P,X,p,T}, // O
        {_,N,n,n,T,T,T,T}, // n
        {_,T,P,T,P,T,P,T}, // P
        {_,T,X,T,T,T,T,T}, // X
        {_,T,p,T,P,T,p,T}, // p
        {_,T,T,T,T,T,T,T}, // T
    };
    return op[a.m_value][b.m_value];
}
```

The values in the table correspond to the smallest abstract value that represents all possible outcomes of addition, given that arguments are drawn from sets abstracted by abstract values a and b.

Finally, to be able to infer additional information in conditional branches, the user has to provide backward abstract semantics of operations. Backward abstract semantics of an expression relates its result to the smallest set of arguments on which such result is still achievable. We call functions implementing backward abstract semantics of operations *solvers* as they mimic solving the equation in the abstract domain. Figure 8.2 is an example of solver for addition of sign values. For each resulting abstract value, it contains a table that can be looked up by abstract values of arguments. The pair it contains shows how the arguments can be “shrunk” given such arguments and the result.

Definitions of forward and backward abstract semantics are too complex to be understood by non-expert in abstract interpretation as well as too brittle and error prone to be written by hand. Naturally, we do not require them to be written by hand and provide a tool that generates their source code from a small and simple definition:

```
int c[] = {−7,−4,−1,0,+1,+2,+4,+5};
sign a[] = {_,N,O,n,P,X,p,T};

fwd_abs_sem_binary(”sign::operator+”, std::plus<int>(), c,a);
bkw_abs_sem_binary(”sign::solve_plus”, std::plus<int>(), c,a);
```

For a given finite abstract domain sign, this small program will print out the source code for forward and backward abstract semantics of addition shown above. It uses concrete values from array c to generate all possible outcomes of a given operation std::plus⟨int⟩, abstracting the arguments and results according to the abstract semantics.

Naturally, this approach is error prone since the choice of concrete values might result in certain outcomes never produced. In practice, however, it is much more reliable than deriving these tables calculationally by hand, as it helped us found several mistakes we made during initial encoding of some finite domains. We found it also much easier to verify an automatically generated look-up table than produce a correct one by hand.

```
void sign::solve_plus(const sign& r, sign& a, sign& b)
{
 static const values op[8][8][8][2] = {
 // b=  _ ,  N ,  O ,  n ,  P ,  X ,  p ,  T     // a= // r=_
   {
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // -
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // N
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // O
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // n
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // P
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // X
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // p
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // T
   },
 // b=  _ ,  N ,  O ,  n ,  P ,  X ,  p ,  T     // a= // r=N
   {
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // -
     {{_,_},{N,N},{N,O},{N,n},{N,P},{N,X},{N,p},{N,T}}, // N
     {{_,_},{O,N},{_,_},{O,N},{_,_},{O,N},{_,_},{O,N}}, // O
     {{_,_},{n,N},{N,O},{n,n},{N,P},{n,X},{N,p},{n,T}}, // n
     {{_,_},{P,N},{_,_},{P,N},{_,_},{P,N},{_,_},{P,N}}, // P
     {{_,_},{X,N},{N,O},{X,n},{N,P},{X,X},{N,p},{X,T}}, // X
     {{_,_},{p,N},{_,_},{p,N},{_,_},{p,N},{_,_},{p,N}}, // p
     {{_,_},{T,N},{N,O},{T,n},{N,P},{T,X},{N,p},{T,T}}, // T
   },
 // b=  _ ,  N ,  O ,  n ,  P ,  X ,  p ,  T     // a= // r=O
   {
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // -
     {{_,_},{-,-},{_,_},{_,_},{N,P},{N,P},{N,P},{N,P}}, // N
     {{_,_},{-,-},{O,O},{O,O},{_,_},{_,_},{O,O},{O,O}}, // O
     {{_,_},{_,_},{O,O},{O,O},{N,P},{N,P},{n,p},{n,p}}, // n
     {{_,_},{P,N},{_,_},{P,N},{_,_},{P,N},{_,_},{P,N}}, // P
     {{_,_},{P,N},{_,_},{P,N},{N,P},{X,X},{N,P},{X,X}}, // X
     {{_,_},{P,N},{O,O},{p,n},{_,_},{P,N},{O,O},{p,n}}, // p
     {{_,_},{P,N},{O,O},{p,n},{N,P},{X,X},{n,p},{T,T}}, // T
   },
 // b=  _ ,  N ,  O ,  n ,  P ,  X ,  p ,  T     // a= // r=n
   {
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // -
     {{_,_},{N,N},{N,O},{N,n},{N,P},{N,X},{N,p},{N,T}}, // N
     {{_,_},{O,N},{O,O},{O,n},{_,_},{O,N},{O,O},{O,n}}, // O
     {{_,_},{n,N},{n,O},{n,n},{N,P},{n,X},{n,p},{n,T}}, // n
     {{_,_},{P,N},{_,_},{P,N},{_,_},{P,N},{_,_},{P,N}}, // P
     {{_,_},{X,N},{N,O},{X,n},{N,P},{X,X},{N,p},{X,T}}, // X
     {{_,_},{p,N},{O,O},{p,n},{_,_},{p,N},{O,O},{p,n}}, // p
     {{_,_},{T,N},{n,O},{T,n},{N,P},{T,X},{n,p},{T,T}}, // T
   },
 // b=  _ ,  N ,  O ,  n ,  P ,  X ,  p ,  T     // a= // r=P
   {
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // -
     {{_,_},{-,-},{-,-},{_,_},{N,P},{N,P},{N,P},{N,P}}, // N
     {{_,_},{-,-},{-,-},{_,_},{O,P},{O,P},{O,P},{O,P}}, // O
     {{_,_},{-,-},{-,-},{_,_},{n,P},{n,P},{n,P},{n,P}}, // n
     {{_,_},{P,N},{P,O},{P,n},{P,P},{P,X},{P,p},{P,T}}, // P
     {{_,_},{P,N},{P,O},{P,n},{X,P},{X,X},{X,p},{X,T}}, // X
     {{_,_},{P,N},{P,O},{P,n},{p,P},{p,X},{p,p},{p,T}}, // p
     {{_,_},{P,N},{P,O},{P,n},{T,P},{T,X},{T,p},{T,T}}, // T
   },
 // b=  _ ,  N ,  O ,  n ,  P ,  X ,  p ,  T     // a= // r=X
   {
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // -
     {{_,_},{N,N},{N,O},{N,n},{N,P},{N,X},{N,p},{N,T}}, // N
     {{_,_},{O,N},{_,_},{O,N},{O,P},{O,X},{O,P},{O,X}}, // O
     {{_,_},{n,N},{N,O},{n,n},{n,P},{n,X},{n,p},{n,T}}, // n
     {{_,_},{P,N},{P,O},{P,n},{P,P},{P,X},{P,p},{P,T}}, // P
     {{_,_},{X,N},{X,O},{X,n},{X,P},{X,X},{X,p},{X,T}}, // X
     {{_,_},{p,N},{P,O},{p,n},{p,P},{p,X},{p,p},{p,T}}, // p
     {{_,_},{T,N},{X,O},{T,n},{T,P},{T,X},{T,p},{T,T}}, // T
   },
 // b=  _ ,  N ,  O ,  n ,  P ,  X ,  p ,  T     // a= // r=p
   {
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // -
     {{_,_},{-,-},{_,_},{_,_},{N,P},{N,P},{N,P},{N,P}}, // N
     {{_,_},{-,-},{O,O},{O,O},{O,P},{O,P},{O,p},{O,p}}, // O
     {{_,_},{_,_},{O,O},{O,O},{n,P},{n,P},{n,p},{n,p}}, // n
     {{_,_},{P,N},{P,O},{P,n},{P,P},{P,X},{P,p},{P,T}}, // P
     {{_,_},{P,N},{P,O},{P,n},{X,P},{X,X},{X,p},{X,T}}, // X
     {{_,_},{P,N},{p,O},{p,n},{p,P},{p,X},{p,p},{p,T}}, // p
     {{_,_},{P,N},{p,O},{p,n},{T,P},{T,X},{T,p},{T,T}}, // T
   },
 // b=  _ ,  N ,  O ,  n ,  P ,  X ,  p ,  T     // a= // r=T
   {
     {{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-},{-,-}}, // -
     {{_,_},{N,N},{N,O},{N,n},{N,P},{N,X},{N,p},{N,T}}, // N
     {{_,_},{O,N},{O,O},{O,n},{O,P},{O,X},{O,p},{O,T}}, // O
     {{_,_},{n,N},{n,O},{n,n},{n,P},{n,X},{n,p},{n,T}}, // n
     {{_,_},{P,N},{P,O},{P,n},{P,P},{P,X},{P,p},{P,T}}, // P
     {{_,_},{X,N},{X,O},{X,n},{X,P},{X,X},{X,p},{X,T}}, // X
     {{_,_},{p,N},{p,O},{p,n},{p,P},{p,X},{p,p},{p,T}}, // p
     {{_,_},{T,N},{T,O},{T,n},{T,P},{T,X},{T,p},{T,T}}, // T
   },
 };
 a = op[r.m_value][a.m_value][b.m_value][0];
 b = op[r.m_value][a.m_value][b.m_value][1];
}
```

Figure 8.2: Solver encoding Backward Abstract Semantics of Addition

The produced look-up tables effectively state many little lemmas about the properties of concrete values. Our ultimate goal is to produce together with these tables a script for proof assistant that will be able to prove these lemmas and thus remove the dependence on the choice of concrete values.

The workflow of the user of our library trying to produce a finite abstract domain looks as following:

- Define the class encoding the finite lattice of the abstract domain
- Implement its smallest set of operations: alpha, eq, bot, top, join and meet.
- Feed this definition to the tool for creating abstractions of operations.
- Incorporate the produced output into the C++ source file with definitions of abstractions.
- Map concrete operations to their abstractions in the registration routine.
- Use the abstract domain for analysis by associating it with some concrete types.

Abstractions of operations for infinite abstract domains, like interval, cannot be produced with similar ease and require scrupulous calculational design [61]. SAIL provides few of such domains for the use by its users, but their definition requires more elaborate knowledge of abstract interpretation theory and thus remains the privilege of experts. Nevertheless, the theory of abstract interpretation states the following facts that can be used to justify our approach [67]:

- For each program, there exists a finite lattice, which can be used for this program to obtain results equivalent to those obtained using widening/narrowing operators;
- No such a finite lattice will do for all programs;
- For all programs, infinitely many abstract values are necessary;
- For a particular program, it is not possible to infer the set of needed abstract values by a simple inspection of the text of the program.

8.5 Discussion

In this chapter we presented a work in progress towards implementing a generic abstract interpreter for C++. Current prototype implementation supports a fairly representative subset of C++ and establishes feasibility of its extension to the scope of the entire language. As promissed, we demonstrated that the user of our framework does not have to deal with the AST of a program at all. This, in turn, allowed the framework to hide from the user the details of the C++ semantics as well as the specifics of a given dialect of the language used. As such, definitions of abstract domains together with the necessary library mappings can be seen as front-end independent encoding of various static analyses for C++.

To lower the level of expertise required from a user in defining a finite abstract domain, we offered a tool that generates the necessary forward and backward abstractions for any given C++ function. Even thought the precision and correctness of approximations produced by the tool depends on the set of input values that the user provided, the tool turned out to be particularly useful in making quick turnarounds between refinement of abstractions, as well as pointed out to few mistakes we made while computing abstractions of operations by hand.

8.5.1 Limitations

In its current implementation, SAIL supports only non-relational value domains, however, we do not believe there to be any problems once we extend it to support relational domains. The framework is also intra-procedural at the moment, but we foresee adding inter-procedural capabilities in the future.

9. CONCLUSIONS AND FUTURE WORK

> Je n'ai fait celle-ci plus longue que parce que n'ai pas eu le loisir de la faire plus courte.[1]
>
> Blaise Pascal
> *Les provinciales (1656), XVI*

There is an increasing gap between programming languages that advanced research is conducted on and the languages actually used in practice. Consequently, the fruits of such research efforts are not immediately available in where they are needed the most. C++ is one such language: its reach abstractions and uncompromised performance proved invaluable in the real world; however, the complexity of the language has also made it difficult for both researchers and users to deal with programs in the language in automated fashion. We believe the gap can be made significantly smaller by addressing two main issues:

- the complexity of dealing with the AST of a program
- the complexity of the language semantics

Consequently, this thesis offers solutions along both of these directions and includes the design of abstractions that aid in analysis and verification of programs written in C++.

Dealing with an AST of a program in an object-oriented language is a subject of well-known expression problem [265], which we address on several fronts.

First, we have implemented *open multi-methods* [208, 209] for C++. They allow adding new classes and new polymorphic functions freely and provide symmetric dispatch that is based on the same rules as overload resolution.

Second, we developed an efficient type switch construct that allows for a more direct expression of the program's logic than multi-methods and visitors do [232]. It allows for multi-way switching on the most-derived type of an object and avoids any ambiguities by enforcing the first-fit semantics.

Third, since neither open type switch nor open multi-methods are useful in decomposing the structure of AST, we introduce functional-style pattern matching into C++. Instead of introducing pattern matching to C++ as an orthogonal language feature, we show that C++ abstractions are already powerful enough to introduce pattern matching in a form of library, where all the patterns traditionally seen in functional languages are implemented as user-defined patterns. Not only does this library solution introduce patterns as first-class citizens; it does so with zero or very small overhead.

To address the complexity of the language, we offer two approaches that avoid dealing with AST of the program altogether.

The first approach is based on refining the type system of a general-purpose programming language with domain-specific type-checking rules [235, 236]. We successfully applied it to two very different domain-specific type systems, allowing the user ensure semantic properties that usually fall out of scope of the type checker of a general-purpose programming language.

The second approach is based on abstract interpretation and allows for static inference of program invariants based on user-defined abstract domains. The initial implementation is based on Pivot [81, 246],

[1] I made this very long, because I did not have the leisure to make it shorter.

but, in principle, is front end agnostic. A mapping from user-provided definitions to entities in the analyzed program is implemented with the help of other solutions presented in this thesis. This mapping is what allows us to hide the complexity of the C++ semantics from the user, letting a particular C++ compiler implement the mapping according to the compiler's specifics.

In conclusion, the research presented in this thesis aims at simplifying the analysis and transformation of C++ programs sufficiently for non-experts to be able to design and implement new language tools for real-world applications. It extends the C++ tradition of offering the developers everything they need to help themselves into the domain of program analysis and transformation. The programming constructs and examples of their use address interesting use-cases for the development of higher-level facilities and improved support of generic programming in C++. Applying this research to other programming languages should offer insights into efficient language support of open first-class pattern matching, a composable framework for creating static analysis and transformation tools for a variety of languages, and a set of reusable and compile-time composable abstract domains.

In the future, we would like to explore further both directions, with initial emphasis on type switching and pattern matching. First, we would add language support to open type switch and open pattern matching to overcome restrictions imposed by a library solution. This will involve general facilities to allow the compiler to optimize match statements for any user-defined pattern or subject types as well as extending the deducible contexts in which variables can be introduced. This should give us a more elegant syntax, improved type checking, better diagnostics and better object code in implementing the semantic direction. Second, we would like to expand the abstract interpretation framework to handle all of C++; support relational abstract domains; scale to real-world projects; and serve as a front end independent way of defining static analysis of C++ programs. This research is going to be less concerned with the development of new abstract domains and more with generic programming and library design issues that allow one to combine existing abstract domains into scalable, precise and useful analyses.

REFERENCES

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond.* Addison-Wesley Professional, Boston, MA, USA, 2004.

[2] Rakesh Agrawal, Linda G. Demichiel, and Bruce G. Lindsay. Static type checking of multi-methods. In *Proceedings of the 6th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '91, pages 113–128, New York, NY, USA, 1991. ACM.

[3] Alexander Aiken and Brian R. Murphy. Implementing regular tree expressions. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 427–447, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[4] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[5] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., Santa Clara, CA, USA, March 2008. version 1.0, `http://research.sun.com/projects/plrg/Publications/fortress.1.0.pdf`.

[6] Eric Allen, Joseph Hallett, Victor Luchangco, Sukyoung Ryu, and Guy L. Steele Jr. Modular multiple dispatch with multiple inheritance. In *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 1117–1121, New York, NY, USA, 2007. ACM.

[7] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS '96: Proceedings of the 3rd International Symposium on Static Analysis*, SAS '96, pages 52–66, London, UK, UK, 1996. Springer-Verlag.

[8] Eric Amiel, Olivier Gruber, and Eric Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *Proceedings of the 9th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '94, pages 244–258, New York, NY, USA, 1994. ACM Press.

[9] Andrew Appel, Luca Cardelli, Kathleen Fisher, Carl Gunter, Robert Harper, Xavier Leroy, Mark Lillibridge, David B. MacQueen, John Mitchell, Greg Morrisett, John H. Reppy, Jon G. Riecke, Zhong Shao, and Christopher A. Stone. Principles and a preliminary design for ML2000. `http://flint.cs.yale.edu/flint/publications/ml2000.html`, March 1999.

[10] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language.* Addison-Wesley Professional, Boston, MA, USA, 4th edition, August 2005.

[11] Lennart Augustsson. Compiling pattern matching. In *Proceedings of a conference on Functional Programming Languages and Computer Architecture*, pages 368–381, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[12] Matthew H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[13] James Bailey and Alexandra Poulovassilis. Abstract interpretation for termination analysis in functional active databases. *Journal of Intelligent Information Systems*, 12(2-3):243–273, April 1999.

[14] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming Language Design and Implementation*, pages 203–213, New York, NY, USA, 2001. ACM.

[15] John J. Barton and Lee R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1994.

[16] Pete Becker. *The C++ Standard Library Extensions: A Tutorial and Reference.* Addison-Wesley Professional, Boston, MA, USA, 1st edition, 2006.

[17] Peter Becker. Working Draft, Standard for Programming Language C++. Technical Report N2857, JTC1/SC22/WG21 C++ Standards Committee, March 2009.

[18] Nuel D. Belnap. A Useful Four-Valued Logic. In J. Michael Dunn and George Epstein, editors, *Modern Uses of Multiple-Valued Logics*, pages 8–37. Reidel, Dordrecht, 1977.

[19] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, February 2010.

[20] Lorenzo Bettini, Sara Capecchi, and Betti Venneri. Double dispatch in C++. *Software – Practice and Experience*, 36(6):581–613, May 2006.

[21] Dirk Beyer, Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST. *International Journal on Software Tools for Technology Transfer*, 9:505–525, 2007.

[22] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cω. In Andrew P. Black, editor, *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311, Berlin, Heidelberg, July 2005. Springer.

[23] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula BEGIN.* Auerbach Press, Philadelphia, 1973.

[24] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In Torben Mogensen, David A. Schmidt, and Hal I. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 85–108. Springer-Verlag, Berlin, Heidelberg, October 2002.

[25] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[26] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[27] Bard Bloom and Martin J. Hirzel. Robust scripting via patterns. In *Proceedings of the 8th symposium on Dynamic languages*, DLS '12, pages 29–40, New York, NY, USA, 2012. ACM.

[28] RSS Advisory Board. RSS 2.0 specification. `http://www.rssboard.org/rss-specification`, 2009.

[29] Daniel Bonniot, Bryn Keller, and Francis Barber. The Nice user's manual. `http://nice.sourceforge.net/manual.html`, 2008.

[30] Boost.org. *The Boost C++ Libraries*. Boost. `http://www.boost.org/`.

[31] Lubomir Bourdev and Jaakko Järvi. Efficient Run-Time Dispatching in Generic Programming with Minimal Code Bloat. In *Workshop of Library-Centric Software Design at OOPSLA'06, Portland Oregon*, oct 2006.

[32] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM Press.

[33] John Boyland and Giuseppe Castagna. Parasitic methods: an implementation of multi-methods for Java. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 66–76, New York, NY, USA, 1997. ACM Press.

[34] Gilad Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, October 2004.

[35] Walter E. Brown. Applied Template Metaprogramming in SIUNITS: the Library of Unit-Based Computation. In *Second Workshop on C++ Template Programming*, October 2001. in conjunction with OOPSLA'01, `http://www.oonumerics.org/tmpw01/brown.pdf`.

[36] Kim Bruce, Luca Cardelli, Giuseppe Castagna, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[37] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching: using temporal logic and model checking. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '09, pages 114–126, New York, NY, USA, 2009. ACM.

[38] Rod M. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 1969.

[39] Rod M. Burstall, David B. MacQueen, and Donald T. Sannella. Hope: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and Functional Programming*, LFP '80, pages 136–143, New York, NY, USA, 1980. ACM.

[40] Warren Burton, Erik Meijer, Patrick Sansom, Simon Thompson, and Philip Wadler. Views: an extension to Haskell pattern matching. 1996. `http://haskell.org/development/views.html`.

[41] Fernando Cacciola. The Boost Optional Library. `http://www.boost.org/libs/optional/`. The Boost Library documentation.

[42] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In Mihaela Bobaru, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 459–465. Springer, Berlin, Heidelberg, 2011. 10.1007/978-3-642-20398-5_33.

[43] Luca Cardelli. Compiling a functional language. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 208–217, New York, NY, USA, 1984. ACM.

[44] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '89, pages 202–212, New York, NY, USA, 1989. ACM.

[45] Yves Caseau. Efficient handling of multiple inheritance hierarchies. In *Proceedings of the 8th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '93, pages 271–287, New York, NY, USA, 1993. ACM.

[46] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In *ECOOP '92: Proceedings of the 6th European Conference on Object-Oriented Programming*, pages 33–56, London, UK, 1992. Springer-Verlag.

[47] Craig Chambers. The Cecil Language: Specification and Rationale. 3.2. Technical report, Department of Computer Science and Engineering. University of Washington, 2004. `http://students.cs.tamu.edu/pmp9599/mm4cc/cecil-spec.pdf`.

[48] Craig Chambers. The Diesel Language, specification and rationale. `http://www.cs.washington.edu/research/projects/cecil/www/Release/doc-diesel-lang/diesel-spec.pdf`, 2006.

[49] Craig Chambers and Weimin Chen. Efficient multiple and predicated dispatching. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 238–255, New York, NY, USA, 1999. ACM Press.

[50] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 85–95, New York, NY, USA, 2005. ACM Press.

[51] Clang: a C language family frontend for LLVM. `http://clang.llvm.org/`, 2007.

[52] Robert Clariso and Jordi Cortadella. The octahedron abstract domain. In *In Static Analysis Symposium (2004*, pages 312–327. Springer-Verlag, 2004.

[53] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 130–145, New York, NY, USA, 2000. ACM Press.

[54] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3):517–575, 2006.

[55] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. The Itanium C++ ABI. `http://www.codesourcery.com/public/cxx-abi/`, March 2001.

[56] Norman H. Cohen. Type-extension type test can be performed in constant time. *ACM Transactions on Programming Languages and Systems*, 13(4):626–629, October 1991.

[57] Stephen Cook, Damian Dechev, and Peter Pirkelbauer. The IPR Query Language. Technical report, Texas A&M University, October 2004. `http://parasol.tamu.edu/pivot/`.

[58] William R. Cook. Object-oriented programming versus abstract data types. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 151–178, London, UK, 1991. Springer-Verlag.

[59] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[60] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French)*. Thèse d'État ès sciences mathématiques, Université Joseph Fourier, Grenoble, France, 21 March 1978.

[61] Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[62] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.

[63] Patrick Cousot. Abstract interpretation. `http://www.di.ens.fr/~cousot/AI/`, August 2008.

[64] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[65] Patrick Cousot and Radhia 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*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[66] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[67] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[68] Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 95–112, Toulouse, France, 16–19 May 1994. IEEE Computer Society Press, Los Alamitos, California.

[69] Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, Portland, Oregon, January 2002. ACM Press, New York, NY.

[70] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[71] Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto, and Armand Puccetti. Experience report: Ocaml for an industrial-strength static analysis framework. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 281–286, New York, NY, USA, 2009. ACM.

[72] Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *SIMULA 67 common base language.* Norwegian Computing Center, May 1968.

[73] Joel de Guzman, Dan Marsden, and Tobias Schwinger. The Boost Fusion Library. `http://spirit.sourceforge.net/dl_more/fusion_v2/libs/fusion/doc/html/index.html`, Sep 2008. The Boost Library documentation.

[74] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: an optimizing compiler for object-oriented languages. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 83–100, New York, NY, USA, 1996. ACM.

[75] Alain Deutsch. Static verification of dynamic properties. In *Proceedings of the 2003 annual ACM SIGAda international conference on Ada: the engineering of correct and reliable software for real-time & distributed systems using ada and related technologies*, SigAda '03, New York, NY, USA, 2003. ACM.

[76] Peter L. Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[77] Edsger W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD472.PDF, January 1975.

[78] Gabriel Dos Reis and Bjarne Stroustrup. Internal Program Representation for The Pivot. Technical report, Programming Tecniques, Tools and Languages Group at Texas A&M University, College Station, TX, USA, 2004. `http://parasol.tamu.edu/pivot/publications/ref.pdf`.

[79] Gabriel Dos Reis and Bjarne Stroustrup. *The Pivot.* Programming Tecniques, Tools and Languages Group at Texas A&M University, 2005. `http://parasol.tamu.edu/pivot/`.

[80] Gabriel Dos Reis and Bjarne Stroustrup. Templates aliases (revision 3). Technical Report WG21/N2258=07-0118, JTC1/SC22/WG21 C++ Standards Committee, 2007. `http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2007/n2258.pdf`.

[81] Gabriel Dos Reis and Bjarne Stroustrup. A principled, complete, and efficient representation of C++. In *Proceedings Joint Conference of ASCM 2009 and MACIS 2009*, volume 22 of *COE Lecture Notes*, pages 407–421, December 2009.

[82] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in C++. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 306–323, New York, NY, USA, 1996. ACM.

[83] Roland Ducournau. Perfect hashing as an almost perfect subtype test. *ACM Transactions on Programming Languages and Systems*, 30(6):33:1–33:56, October 2008.

[84] Edison Design Group. C++ front end. `http://www.edg.com/`, July 2008.

[85] Simon Marlow (editor). Haskell 2010 language report, July 2010. `http://www.haskell.org/onlinereport/haskell2010/`.

[86] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[87] Burak Emir. *Object-oriented pattern matching*. PhD thesis, EPFL, Lausanne, 2007.

[88] Michael D. Ernst, Craig S. Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 186–211, Brussels, Belgium, July 20-24, 1998.

[89] David Evans. Static detection of dynamic memory errors. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming Language Design and Implementation*, pages 44–53, New York, NY, USA, 1996. ACM.

[90] David Evans and David Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19:42–51, 2002.

[91] The Expat XML Parser. `http://expat.sourceforge.net/`, 2006.

[92] David J. Farber, Ralph E. Griswold, and Ivan P. Polonsky. Snobol, a string manipulation language. *Journal of The ACM*, 11:21–30, January 1964.

[93] Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch. Goanna – A Static Model Checker. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, pages 297–300. Springer, 2006.

[94] Benjamin C. Flynn and David Wonnacott. Reconciling Encapsulation and Dynamic Dispatch via Accessory Functions. Technical Report 387, Rutgers University Department of Computer Science , 1999. `http://citeseer.ist.psu.edu/flynn99reconciling.html`.

[95] Brian Foote, Ralph E. Johnson, and James Noble. Efficient Multimethods in a Single Dispatch Language. *Proceedings of the European Conference on Object-Oriented Programming, Glasgow, Scotland, July*, 2005.

[96] Jeffrey S. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, 2002.

[97] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation*, pages 192–203, New York, NY, USA, 1999. ACM Press.

[98] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 2002. ACM Press.

[99] Eric Friedman and Itay Maman. The Boost Variant Library. `http://www.boost.org/doc/html/variant.html`, 2002. The Boost Library documentation.

[100] Christopher Frost and Todd Millstein. Modularly Typesafe Interface Dispatch in JPred. In *2006 International Workshop on Foundations and Development of Object-Oriented Languages (FOOL/-WOOD'07)*, Charleston, SC, USA, January 2006.

[101] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 406–431, London, UK, UK, 1993. Springer-Verlag.

[102] Vladimir Gapeyev, Michael Y. Levin, Benjamin C. Pierce, and Alan Schmitt. The Xtatic experience. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, January 2005. University of Pennsylvania Technical Report MS-CIS-04-24, Oct 2004.

[103] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 115–134, New York, NY, USA, 2003. ACM Press.

[104] Jacques Garrigue. Programming with polymorphic variants. In *ACM Workshop on ML*, September 1998.

[105] Felix Geller, Robert Hirschfeld, and Gilad Bracha. *Pattern Matching for an object-oriented and dynamically typed programming language.* Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam. Univ.-Verlag, 2010.

[106] Wolfgang Gellerich, Mustafa Hagog, Dorit Naishlos, Mircea Namolaru, Eberhard Pasch, Hartmut Penner, Ulrich Weigand, and Ayal Zaks. Contributions to the GNU compiler collection. *IBM Systems Journal*, 44(2):259–278, 2005.

[107] Michael Gibbs and Bjarne Stroustrup. Fast dynamic casting. *Software – Practice and Experience*, 36:139–156, February 2006.

[108] James F. Gimpel. The theory and implementation of pattern matching in snobol4 and other programming languages. *Bibliography of Numbered SNOBOL4 Documents*, 1971.

[109] Neal Glew. Type dispatch for named hierarchical types. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, pages 172–182, New York, NY, USA, 1999. ACM.

[110] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[111] Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A metalanguage for interactive proof in LCF. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, POPL '78, pages 119–130, New York, NY, USA, 1978. ACM.

[112] Eric Goubault, Matthieu Martel, and Sylvie Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In *ESOP '02: Proceedings of the 11th European Symposium on Programming Languages and Systems*, pages 209–212, London, UK, 2002. Springer-Verlag.

[113] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT '91: Proceedings of the international joint conference on theory and practice of software development on Colloquium on trees in algebra and programming (CAAP '91): vol 1*, pages 169–192, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[114] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '06, pages 291–310, New York, NY, USA, 2006. ACM Press.

[115] Gabor Greif. Chinese Dispatch – Unpublished notes for a talk, July 2002. Dylan Hackers Conference, Berlin, `http://www.opendylan.org/cgi-bin/viewcvs.cgi/trunk/www/papers/ChineseDispatch.lout?rev=8014&view=markup`.

[116] Christian Grothoff. Walkabout revisited: The runabout. In *ECOOP '03: Proceedings of the 17th European Conference on Object-Oriented Programming*, pages 103–125. Springer-Verlag, 2003.

[117] Network Working Group. The Atom syndication format. Technical Report RFC 4287, The Internet Engineering Task Force, December 2005. `http://tools.ietf.org/html/rfc4287`.

[118] The XML Schema Working Group. XML Schema. `http://www.w3.org/XML/Schema`, October 2004.

[119] Aleksey Gurtovoy. The Boost MPL library. `http://www.boost.org/libs/mpl/doc/index.html`, July 2002. The Boost Library documentation.

[120] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '95, pages 130–141, New York, NY, USA, 1995. ACM.

[121] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *Proceedings of the 10th international conference on Model checking software*, SPIN'03, pages 235–239, Berlin, Heidelberg, 2003. Springer-Verlag.

[122] Martin Hirzel, Nathaniel Nystrom, Bard Bloom, and Jan Vitek. Matchete: Paths through the pattern matching jungle. In *10th International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 150–166. Springer, 2008.

[123] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38, Berlin, Germany, 1991. Springer-Verlag.

[124] Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. Patterns as objects in Grace. In *Proceedings of the 8th symposium on Dynamic languages*, DLS '12, pages 17–28, New York, NY, USA, 2012. ACM.

[125] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language (preliminary report). In Dan Suciu and Gottfried Vossen, editors, *International Workshop on the Web and Databases (WebDB)*, May 2000. Reprinted in *The Web and Databases, Selected Papers*, Springer LNCS volume 1997, 2001.

[126] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2005.

[127] Paul Hudak, Philip Wadler, and Simon Peyton Jones. *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language : Version 1.0*, volume 54 of *ML Library*. Haskell Committee, 1990.

[128] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 347–349, New York, NY, USA, 1986. ACM.

[129] International Organization for Standardization. *ISO/IEC 14882: Programming languages – C++*. American National Standards Institute, September 1998.

[130] International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages: C++*. American National Standards Institute, Geneva, Switzerland, 2nd edition, October 2003.

[131] International Organization for Standardization. *ISO/IEC 14882:2011: Programming languages: C++*. American National Standards Institute, Geneva, Switzerland, 3rd edition, 2011.

[132] International Standardization Organization. *ISO/IEC 10918-1:1994: Information technology – - Digital compression and coding of continuous-tone still images: Requirements and guidelines*. pub-ISO, pub-ISO:adr, 1994.

[133] Warwick Irwin and Neville Churcher. A generated parser of C++. In *NZCSRSC-01: Proceedings of The Fourth New Zealand Computer Science Research Students Conference*, 2001.

[134] Jaakko Järvi, Douglas Gregor, Jeremiah Willcock, Andrew Lumsdaine, and Jeremy Siek. Algorithm specialization in generic programming: challenges of constrained generics in C++. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 272–282, New York, NY, USA, 2006. ACM.

[135] Jaakko Järvi, Jeremiah Willcock, Howard Hinnant, and Andrew Lumsdaine. Function overloading based on arbitrary properties of types. *C/C++ Users Journal*, 21(6):25–32, June 2003.

[136] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. *The Boost Enable_if Library*. Boost, 2003. http://www.boost.org/libs/utility/enable_if.html.

[137] Barry Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[138] Barry Jay and Delia Kesner. First-class patterns. *Journal Of Functional Programming*, 19(2):191–225, March 2009.

[139] Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, pages 119–134, 2004.

[140] Stephen Johnson. Lint, a C program checker. Technical Report 65, Bell Laboratories, December 1977. http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.1841.

[141] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[142] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[143] Andrew Kennedy. Dimension types. In *Proceedings of the 5th European Symposium on Programming (ESOP)*, volume 788 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[144] Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 21–40, New York, NY, USA, 2005. ACM.

[145] Ken Kennedy, Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnsson, John Mellor-Crummey, and Linda Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803&ndash;1826, 2001.

[146] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107, New York, NY, USA, 2004. ACM Press.

[147] Steven Cole Kleene. *Introduction to Metamathematics.* North-Holland, Amsterdam, second edition, 1987.

[148] Andreas Krall, Jan Vitek, and R. Nigel Horspool. Near optimal hierarchical encoding of types. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 128–145. Springer-Verlag, 1997.

[149] Shriram Krishnamurthi, Matthias Felleisen, and Daniel Friedman. Synthesizing object-oriented and functional design to promote re-use. In Eric Jul, editor, *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113. Springer, Berlin, Heidelberg, 1998. 10.1007/BFb0054088.

[150] Marek Kucharski and Miroslaw Zielinski. Integrating error-detection techniques to find more bugs in embedded C software. `http://embedded-computing.com/integrating-error-detection-techniques-find-bugs-embedded-software`, November 2011.

[151] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[152] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 26–37, New York, NY, USA, 2001. ACM.

[153] Keunwoo Lee, Anthony LaMarca, and Craig Chambers. Hydroj: object-oriented pattern matching for evolvable distributed systems. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 205–223, New York, NY, USA, 2003. ACM.

[154] Allen Leung. Prop: A C++ based pattern matching language. Technical report, Courant Institute of Mathematical Sciences, New York University, March 1996. `http://www.cs.nyu.edu/leunga/prop.html`.

[155] Barbara Liskov. Keynote address – data abstraction and hierarchy. In *Addendum to the Proceedings of the 2nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '87, pages 17–34, New York, NY, USA, 1987. ACM Press.

[156] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, Craig J. Schaffert, Robert Scheifler, and Alan Snyder. Clu reference manual. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1979. `http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail\&id=oai%3Ancstrlh%3Amitai%3AMIT-LCS%2F%2FMIT%2FLCS%2FTR-225`.

[157] Jed Liu and Andrew C. Myers. JMatch: Iterable Abstract Pattern Matching for Java. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03, pages 110–127, London, UK, UK, 2003. Springer-Verlag.

[158] Liz: A System for Axiomatic Programming. `http://liz.axiomatics.org/`, 2012.

[159] Lockheed Martin Corporation. Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program. (2RDU00001 Rev C), December 2005.

[160] Francesco Logozzo. *Analyse statique modulaire de langages á objets.* PhD thesis, L'Ècole Polytechnique, June 2004.

[161] Andres Löh and Ralf Hinze. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '06, pages 133–144, New York, NY, USA, 2006. ACM.

[162] John Maddock, Steve Cleary, et al. The Boost type_traits library. `www.boost.org/libs/type_traits`, 2002. The Boost Library documentation.

[163] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 213–225, New York, NY, USA, 2003. ACM Press.

[164] Luc Maranget. Compiling lazy pattern matching. In *Proceedings of the 1992 ACM conference on LISP and Functional Programming*, LFP '92, pages 21–31, New York, NY, USA, 1992. ACM.

[165] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, ML '08, pages 35–46, New York, NY, USA, 2008. ACM.

[166] Matthieu Martel. Static analysis of the numerical stability of loops. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*, pages 133–150, London, UK, 2002. Springer-Verlag.

[167] Matthieu Martel. An overview of semantics for the validation of numerical programs. In *VMCAI*, pages 59–77, 2005.

[168] Lockheed Martin. *Joint Strike Fighter, Air Vehicle, C++ Coding Standard.* Lockheed Martin, December 2005.

[169] MathWorks. PolySpace verifier. `http://www.mathworks.com/products/polyspace/`, 2009.

[170] Scott McPeak. Elkhound: A fast, practical GLR parser generator. Technical report, University of California, Berkeley, December 2002. `http://www.cs.berkeley.edu/~smcpeak/elkhound/elkhound.ps`.

[171] Bertrand Meyer. *Eiffel: The Language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[172] Microsoft Research. Phoenix Compiler and Shared Source Common Language Infrastructure. `http://research.microsoft.com/phoenix`, 2009.

[173] Bartosz Milewski. Haskell – the pseudocode language for C++ template metaprogramming. In *BoostCon'11*, 2011.

[174] Todd Millstein and Craig Chambers. Modular Statically Typed Multimethods. *Information and Computation*, 175(1):76–118, May 2002.

[175] Todd Millstein, Mark Reay, and Craig Chambers. Relaxed MultiJava: balancing extensibility and modular typechecking. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 224–240, New York, NY, USA, 2003. ACM Press.

[176] Todd D. Millstein and Craig Chambers. Modular Statically Typed Multimethods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, volume 1628 of *LNCS*, pages 279–303, London, UK, 1999. Springer-Verlag.

[177] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1990.

[178] Antoine Miné. The octagon abstract domain. In *Proceedings of the Workshop on Analysis, Slicing, and Transformation (AST'01)*, IEEE, pages 310–319, Stuttgart, Gernamy, October 2001. IEEE CS Press. `http://www.di.ens.fr/~mine/publi/article-mine-ast01.pdf`.

[179] Antoine Miné. The octagon abstract domain library. `http://www.di.ens.fr/~mine/oct/`, 2001.

[180] Antoine Miné. A few graph-based relational numerical abstract domains. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*, volume 2477 of *Lecture Notes in Computer Science*, pages 117–132, Madrid, Spain, September 2002. Springer. `http://www.di.ens.fr/~mine/publi/article-mine-sas02.pdf`.

[181] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP*, pages 3–17, 2004.

[182] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006. `http://www.di.ens.fr/~mine/publi/article-mine-HOSC06.pdf`.

[183] Yaron Minsky. Caml trading – experiences with functional programming on Wall Street. *Journal of Functional Programming*, 18:553–564, 2008.

[184] MISRA Consortium. Guidelines for the use of the C++ language in critical systems. `http://misra-cpp.org/`, 2008.

[185] Markus Mohnen. A graph-free approach to data-flow analysis. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 46–61, London, UK, 2002. Springer-Verlag.

[186] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 61–76, Berlin, Heidelberg, 2003. Springer-Verlag.

[187] G. M. Morton. A computer-oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd, Ottawa, Canada, 1966.

[188] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '08, pages 563–582, New York, NY, USA, 2008. ACM.

[189] David A. Musser and Alexander A. Stepanov. Generic Programming. In *Proceedings of International Symposium on Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Rome, Italy, 1988.

[190] Ravi Nanavati. Experience report: a pure shirt fits. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 347–352, New York, NY, USA, 2008. ACM.

[191] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag.

[192] George Nelan. An algebraic typing & pattern matching preprocessor for C++. Available on: `http://web.archive.org/web/20021214012815/http://www.primenet.com/~georgen/app.html`, 2000. Originally published at http://www.primenet.com/ georgen/app.html.

[193] Michael Norrish. *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, February 1998. Also published as Technical Report 453, available from http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-453.pdf.

[194] Michael Norrish. A formal semantics for C++. Technical report, NICTA, 2008. `http://nicta.com.au/people/norrishm/attachments/bibliographies_and_papers/C-TR.pdf`.

[195] Martin Odersky, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean Mcdirmid, Stephane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An overview of the Scala programming language (2nd edition). Technical Report LAMP-REPORT-2006-001, Ecole Polytechnique Federale de Lausanne, 2006.

[196] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *In Proceedings 24th ACM Symposium on Principles of Programming Languages*, pages 146–159. ACM Press, 1997.

[197] Chris Okasaki. Views for standard ML. In *Workshop on ML*, 1998.

[198] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[199] Nikolaas N. Oosterhof. Application patterns in functional languages. `http://essay.utwente.nl/57656/`, 2005.

[200] Yoann Padioleau, Julia L. Lawall, Rene Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *EuroSys Conference*, pages 247–260, 2008.

[201] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference*, COMPSAC '98, pages 9–15, Washington, DC, USA, 1998. IEEE Computer Society.

[202] Emanuele Panizzi and Bernardo Pastorelli. Multimethods and separate static typechecking in a language with C++-like object model. *The Computing Research Repository*, cs.PL/0005033, 2000.

[203] Nikolaos S. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, February 1998.

[204] Edmondo Pentangelo. Functional C#. `http://functionalcsharp.codeplex.com/`, 2011.

[205] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.

[206] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 50–61, New York, NY, USA, 2006. ACM.

[207] Peter Pirkelbauer. *Programming Language Evolution and Source Code Rejuvenation.* PhD thesis, Texas A&M University, December 2010.

[208] Peter Pirkelbauer, Yuriy Solodkyy, and Bjarne Stroustrup. Open multi-methods for C++. In *Proceedings of the 6th international conference on Generative Programming and Component Engineering*, GPCE '07, pages 123–134, New York, NY, USA, 2007. ACM.

[209] Peter Pirkelbauer, Yuriy Solodkyy, and Bjarne Stroustrup. Design and evaluation of C++ open multi-methods. *Science of Computer Programming*, 75(7):638–667, July 2010.

[210] Laurence Puel and Ascander Suarez. Compiling pattern matching by term decomposition. *Journal of Symbolic Computation*, 15(1):1–26, January 1993.

[211] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. Formal verification of object layout for C++ multiple inheritance. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '11, pages 67–80, New York, NY, USA, 2011. ACM.

[212] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. A mechanized semantics for C++ object construction and destruction, with applications to resource management. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '12, pages 521–532, New York, NY, USA, 2012. ACM.

[213] Morten Rhiger. Type-safe pattern combinators. *Journal Of Functional Programming*, 19(2):145–156, March 2009.

[214] Adam Richard. OOMatch: pattern matching as dispatch in Java. Master's thesis, University of Waterloo, October 2007.

[215] Jonathan G. Rossie, Jr. and Daniel P. Friedman. An algebraic semantics of subobjects. In *Proceedings of the 10th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '95, pages 187–199, New York, NY, USA, 1995. ACM.

[216] Guido van Rossum. *The Python Language Reference Manual.* Network Theory Ltd., September 2003. Paperback.

[217] Sukyoung Ryu, Changhee Park, and Guy L. Steele Jr. Adding pattern matching to existing object-oriented languages. In *2010 International Workshop on Foundations of Object-Oriented Languages*, October 2010.

[218] Jonathan L. Schilling. Optimizing away C++ exception handling. *SIGPLAN Not.*, 33:40–47, August 1998.

[219] Markus Schordan and Dan Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of LNCS, pages 214–223. Springer-Verlag, August 2003.

[220] Lenhart K. Schubert, Mary Angela Papalaskaris, and Jay Taugher. Determining type, part, color, and time relationships. *Computer*, 16(10):53–60, October 1983.

[221] Helmut Seidl. Deciding equivalence of finite tree automata. *SIAM Journal on Computing*, 19(3):424–437, 1990.

[222] Andrew Shalit. *The Dylan Reference Manual. 2nd edition.* Apple Press, 1996.

[223] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[224] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

[225] Jeremy Siek and Andrew Lumsdaine. Concept Checking: Binding Parametric Polymorphism in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[226] Jeremy Siek and Walid Taha. A semantic analysis of C++ templates. In *ECOOP '06: Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 304–327, Berlin, Heidelberg, 2006. Springer-Verlag.

[227] Axel Simon, Andy King, and Jacob M. Howe. Two Variables per Linear Inequality as an Abstract Domain. In M. Leuschel, editor, *Proceedings of Logic Based Program Development and Transformation*, volume 2664 of *Lecture Notes in Computer Science*, pages 71–89. Springer-Verlag, unknown 2002. see http://www.springer.de./comp/lncs/index.html.

[228] Michel Sintzoff. Calculating properties of programs by valuations on specific models. In *Proceedings of ACM conference on Proving assertions about programs*, pages 203–207, New York, NY, USA, 1972. ACM.

[229] Julian Smith. Draft proposal for adding Multimethods to C++. Technical Report N1463, JTC1/SC22/WG21 C++ Standards Committee, 2003. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1463.html`.

[230] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, New York, NY, USA, 1986. ACM.

[231] Software Engineering Institute, Carnegie Mellon. CERT C++ Secure Coding Standard. `https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637`, 2012.

[232] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. Open and efficient type switch for C++. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '12, pages 963–982, New York, NY, USA, 2012. ACM.

[233] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. Open pattern matching for C++. Under review to GPCE'13, October 2013.

[234] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. Open pattern matching for C++ (extended abstract). In *Proceedings of the 4th annual conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '13, New York, NY, USA, 2013. ACM.

[235] Yuriy Solodkyy and Jaakko Järvi. Extending type systems in a library: Type-safe XML processing in C++. *Science of Computer Programming*, 76(4):290–306, April 2011.

[236] Yuriy Solodkyy, Jaakko Järvi, and Esam Mlaih. Extending type systems in a library — type-safe XML processing in C++. In *Proceedings of the Second International Workshop on Library-Centric*

Software Design (LCSD'06), pages 55–64, October 2006. Technical Report No. 06-18 in Computer Science and Engineering at Chalmers University of Technology and Göteborg University.

[237] Yuriy Solodkyy, Jaakko Järvi, and Esam Mlaih. *eXtensible Typing Library*, 2007.

[238] David A. Spuler. Compiler Code Generation for Multiway Branch Statements as a Static Search Problem. Technical Report Technical Report 94/03, James Cook University, January 1994.

[239] Guy L. Steele Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

[240] A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, April 1994. `http://www.hpl.hp.com/techreports`.

[241] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, USA, 1986.

[242] Bjarne Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring'87 EUUG Conference*, EUUG '87, May 1987. Revised version in AT&T C++ Translator Release Notes, June 1989. Also, USENIX Computing Systems, V2 no 4, Fall 1989, pp 367-396.

[243] Bjarne Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[244] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[245] Bjarne Stroustrup. A rationale for Semantically Enhanced Library Languages. In *LCSD '05*, October 2005.

[246] Bjarne Stroustrup and Gabriel Dos Reis. Supporting SELL for High-Performance Computing. In *18th International Workshop on Languages and Compilers for Parallel Computing*, volume 4339 of LNCS, pages 458–465. Springer-Verlag, October 2005.

[247] Herb Sutter and Jim Hyslop. Polymorphic exceptions. *C/C++ Users Journal*, 23(4), April 2005.

[248] Andrew Sutton, Bjarne Stroustrup, and Gabriel Dos Reis. Concepts lite: Constraining templates with predicates. Technical Report WG21/N3580, JTC1/SC22/WG21 C++ Standards Committee, March 2013.

[249] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 29–40, New York, NY, USA, 2007. ACM.

[250] David Thomas and Andrew Hunt. *Programming Ruby: the pragmatic programmer's guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[251] Sam Tobin-Hochstadt. Extensible pattern matching in an extensible language. In *Proceedings of the 22nd Symposium on Implementation and Application of Functional Languages (IFL 2010)*, September 2010. Utrecht University Technical Report UU-CS-2010-020, August 2010.

[252] Tom Duff. Duff's Device. `http://www.lysator.liu.se/c/duffs-device.html`, Aug 1988.

[253] Mark Tullsen. First class patterns. In *Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages*, PADL '00, pages 1–15, 2000.

[254] David A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proceedings of a conference on Functional programming languages and computer architecture*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[255] David Ungar and David Patterson. Berkeley smalltalk: Who knows where the time goes? In Glenn Krasner, editor, *Smalltalk-80: Bits of History and Words of Advice*, pages 189–206. Addison-Wesley, 1983.

[256] David Vandevoorde and Nicolai M. Josuttis. *C++ templates: the complete guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[257] Todd Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995.

[258] Todd L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in *C++ Gems*, ed. Stanley Lippman.

[259] Todd L. Veldhuizen. C++ templates are Turing complete. `www.osl.iu.edu/~tveldhui/papers/2003/turing.pdf`, 2003.

[260] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[261] Joost Visser. Matching objects without language extension. *Journal of Object Technology*, 5:81–100, November-December 2006.

[262] Jan Vitek, R. Nigel Horspool, and Andreas Krall. Efficient type inclusion tests. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 142–157, New York, NY, USA, 1997. ACM.

[263] Dimitrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An open and shut typecase. In *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '05, pages 13–24, New York, NY, USA, 2005. ACM.

[264] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. ACM.

[265] Philip Wadler. The expression problem. `http://www.daimi.au.dk/~madst/tool/papers/expression.txt`, November 1998. Mail to the java-genericity mailing list.

[266] Charles Wallace. Specification and validation methods. In Egon Börger, editor, *Specification and Validation Methods*, chapter The semantics of the C++ programming language, pages 131–164. Oxford University Press, Inc., New York, NY, USA, September 1995.

[267] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*, volume 34–9 of *ICFP '99*, pages 148–159, N.Y., 27–29 1999. ACM Press.

[268] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '06, pages 345–362, New York, NY, USA, 2006. ACM.

[269] Reinhard Wilhelm and Björn Wachter. Abstract interpretation with applications to timing validation. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 22–36, Berlin, Heidelberg, 2008. Springer-Verlag.

[270] Edward D. Willink. *Meta-Compilation for C++*. PhD thesis, University of Surrey, June 2001. Computer Science Research Group.

[271] Nikolaus Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, April 1988.

[272] David Wonnacott. Using Accessory Functions to Generalize Dynamic Dispatch in Single-Dispatch Object-Oriented Languages. In *COOTS'01: 6th USENIX Conference on Object-Oriented Technologies and Systems, San Antonio, TX*, pages 93–102. USENIX, 2001.

[273] Albrecht Wöß, Markus Löberbauer, and Hanspeter Mössenböck. LL(1) Conflict Resolution in a Recursive Descent Compiler Generator. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of LNCS, pages 192–201. Springer-Verlag, 2003.

[274] Andrew Wright and Bruce Duba. Pattern matching for Scheme. 1995.

[275] www.fourcc.org. Video codec and pixel format definitions. `http://www.fourcc.org`. retrieved on February 20th, 2007.

[276] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems*, 29(3), May 2007.

[277] XML Activity. Extensible markup language (XML^{TM}). `http://www.w3.org/XML`, 2005.

[278] XSD Schema for RSS and Atom. `http://www.cs.cornell.edu/courses/cs431/2008sp/assignments.htm`, 2009. Part of the course CS/INFO 431/631: Web Information Systems.

[279] Victor H. Yngve. A programming language for mechanical translation. *Mechanical Translation*, 5:25–41, July 1958.

[280] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 241–252, New York, NY, USA, 2001. ACM.

[281] Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *Proceedings FOOL 12*, January 2005. `http://homepages.inf.ed.ac.uk/wadler/fool`.

[282] Yoav Zibin and Joseph Yossi Gil. Efficient subtyping tests with PQ-encoding. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 96–107, New York, NY, USA, 2001. ACM.

[283] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Software Engineering Notes*, 29(6):97–106, 2004.

APPENDIX A

SUPPLEMENTAL MATERIAL

solodkyy-dissertation-sources.zip contains the LaTeX source used to produce this dissertation.

type-switch.zip contains the source code of the *Mach7* library that was used to produce the measurement data and the syntax presented in Chapter 5. The file was submitted to accompany our OOPSLA'12 publication [232], and reflects the state of the library as of April 2012, with the exception of the pattern-matching functionality, which was intentionally removed.

pattern-matching.zip has the full source of the *Mach7* library, presented and evaluated in Chapter 6. In June 2013 it was submitted to accompnay our GPCE'13 publication, currently under review [233].

sail.zip contains *SAIL* source code alone. Requires Pivot's source code to compile.

ComparingPerformance.xlsx is the measurement spreadsheet we built to see how reproducible the experiments are. It convinced us to use the median instead of the average to aggregate the data, as discussed in §5.6.1 when describing our comparison methodology.

DCast-vs-Visitors.xlsx contains the detailed measurement data that, after aggregation, was presented in Figure 5.1 and Figure 5.2.

ClassHierarchies.xlsx contains the relevant information for the class hierarchies benchmark shown in Table 5.2 as well as the measurements of the probabilities of conflict shown in Figure 5.7 and Table 5.3.

ClassHierarchiesMultipleArguments-2013-02-20.xlsx is similar to the preceding, but contains the measurements for the multi-argument type switching presented in Table 5.4.

OCamlComparison.xlsx contains results of comparing our approaches to the built-in solutions of OCaml and Haskell, as summarized in Figure 5.6.

Timing.xlsx is a visualization of absolute times taken by each synthetic benchmark of §5.6.1 on a range of compilers, options and parameters. A particular snapshot of this dataset is shown in Figure 5.5 with a larger part of the data aggregated in relative form in Table 5.1.

Timing-N-arg.xlsx has the measurements data behind the multi-argument experiments of §5.6.6 summarized in Figure 5.8.

PatternMatchingOverhead.xlsx measures the overheads of patterns as objects and our open patterns approach in comparison to a handcrafted solution. The data is presented in Table 6.1.

Timing-Compilation.xlsx contains the measured impact of our open pattern matching solution on compilation times, as summarized in Table 6.2.

std-lib-power-law-distributions-msvc.xlsx lists measured frequencies of type switching on IPR nodes in Pivot's pretty-printer (see §5.3.7), as summarized in Figure 5.3.

CollisionsWithAndWithoutFrequencies.xlsx contains the data for the observed decrease in the number of collisions when the frequency of any given class is take into account, as depicted in Figure 5.4.

UseOfFrequency.xlsx complements the above with the timing data on an actual application when frequency tracing is enabled. These results convinced us to not use frequency tracing by default, as the drop in collisions did not translate into any noticeable drop in execution time (see §5.3.7 for details). The file also contains data on the overhead incurred on the first call with each dynamic type, as summarized in Table 5.5.

APPENDIX B

INDEX