COMPOSABLE COMPILERS: EVOLUTION TOWARD A PRACTICAL REALITY

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Chinawat Isradisaikul

August 2017

COMPOSABLE COMPILERS: EVOLUTION TOWARD A PRACTICAL REALITY

Chinawat Isradisaikul, Ph.D.

Cornell University 2017

The ability to add new features to programming languages is essential for language design experimentation and domain-specific developments, but implementing and maintaining small language extensions in traditional compilers remain a challenge. General-purpose programming languages do not have desired mechanisms to integrate small, independently developed extensions into a working programming language. At the same time, domain-specific languages that support such integration struggle to gain popularity in the programming language community. More language mechanisms and tools are needed as a middle ground so that a broader range of programmers can implement, maintain, and combine compilers for individual language features more easily.

At the heart of compiler construction, new design patterns are proposed to allow compilers to be extended in a modular way and to be merged with little effort. These design patterns, implementable in a mainstream programming language, encode dynamic relationships between node types in abstract syntax trees (ASTs) so that inheritance in object-oriented programming still works over the course of language evolution. A new AST representation lets a single AST be viewed as different programs for different languages. Compiler passes are language-neutral, making translations reusable and composable.

At the front end, engineering language syntax can be a painstaking process, especially when individual language syntaxes start to interact. Automatic parser generators, albeit a powerful tool to parse complex grammars, are unhelpful when grammars are faulty, as reports of parsing conflicts do not explain these faults. To improve debugging experience, a semi-decision procedure is added to an LALR parser generator to give compact counterexamples illustrating why the grammar in question is ambiguous. For

unambiguous grammars that cause parsing conflicts, a different kind of counterexample is constructed to aid removal of conflicts.

At the back end, translation passes in compilers require extracting components of AST nodes. Pattern matching, an important feature in functional languages, is a prime candidate for this task. However, data abstraction and extensibility, two concepts central to object-oriented languages, are in conflict with pattern matching. A new language design based on modal abstraction reconciles static, modular reasoning about exhaustiveness in pattern matching with data abstraction.

BIOGRAPHICAL SKETCH

Chinawat (Chin) Isradisaikul [tɕʰīn.nā.wát ɪ̀t.sā.rāː.dì.sǎj.kūn] was born on the Day of Vesak 1985 (Visakha Bucha) in Phitsanulok [pʰít.sā.nú.lôːk], Thailand. His first encounter with a computer was in the fifth grade, when he learned CU Writer, a Thai document processing program on DOS, in his lower school's new computer lab. Although Chin did not own a computer until the tenth grade, his interactions with computers were continuous, thanks to opportunities in his middle school. Installing Windows 95 on dozens of desktop, connecting them to LAN, breaking a few of them, and developing the school's website were part of his ninth-grade routine.

BASIC was Chin's first programming language which he learned upon graduating from middle school and moving to Bangkok to attend Triam Udom Suksa School [trīaːm.ù.dōm.sùk.sǎː]. During his first semester break, he attended a programming camp to learn C, which became his primary programming language throughout high school. Chin wrote an ASCII variant of *Who Wants to Be a Millionaire?* as his first C application for his friends and family to enjoy.

The programming camp would be first of several camps that prepared Chin for representing Thailand in the International Olympiad in Informatics. Even if he missed the cut in the end, that was enough to qualify him for a Royal Thai Government scholarship to study abroad. He attended Westtown School in West Chester, Pennsylvania in 2004 as part of the college preparation program. There, he learned Java in his AP Computer Science class. Upon graduating from Westtown, Chin attended the University of Pennsylvania in Philadelphia, double majoring in computer science and mathematics. At Penn, his first CS course in discrete math introduced him to inference rules, which sparked his interest in logic and formal proofs. Chin's interests in compilers grew out of the Introduction to Compilers course he took in his senior year, in which he learned his first functional programming language, OCaml, and implemented his first compiler without getting

register allocation to work. He then moved to Ithaca, New York, to pursue a Ph.D. in computer science at Cornell University.

Apart from research, Chin appreciates teaching and interacting with students, which in turn helped him become a better researcher. His dedication in teaching earned him the Yahoo! Outstanding Graduate Teaching Award in 2010. For most of his time in the United States, Chin has contributed academically to the Thai Scholar Program from helping new students adjust to American culture in the summer to proofreading their college applications every Christmas. In 2009, he implemented the Thai Scholar College Information System to streamline the college application process for Thai Scholars. The system has been in use since then, minimizing mundane tasks for both students and program staff.

Outside academics, Chin loves nature, enjoys traveling, and practices photography. In 2012, he received training from the National Weather Service in Binghamton to become a SKYWARN spotter volunteer for Tompkins County. He served as the Cornell CS Photo Czar from 2013 to 2017, taking photos of incoming Ph.D. students. Chin is an avid skier, despite getting started after his A exam at Cornell.

To my family, near and far.

ACKNOWLEDGMENTS

This dissertation would not be possible without help and advice from Andrew Myers, my advisor, who knows when to pull me ahead and when to push me from behind. Andrew, you showed me there is always a research problem to solve, whether easy or hard, whether decidable or undecidable. Whenever I seemed to run out of ideas, your suggestions usually gave me a way forward. You taught me how to give a good conference talk, restored my faith that constructive criticisms do exist, and warned me that it is possible to spend countless hours hacking TEX. For that, I am grateful.

My Ph.D. career at Cornell would be less enjoyable without insightful and fun graduate courses taught by Dexter Kozen and Bob Constable, my special committee members. Your courses kept my thoughts organized, kept my reasoning sound, and instilled in me the value of teaching and its connections with research. Your encouragements in our conversations propelled me through the finish line. For that, I am grateful.

Past and present members of the Applied Programming Languages group—Steve Zdancewic, Nate Nystrom, Steve Chong, Jed Liu, Mike George, Krishnaprasad Vikram, Danfeng Zhang, Owen Arden, Tom Magrino, Yizhou Zhang, Isaac Sheff, Matthew Milano, and Ethan Cecchetti—contributed to my success as an undergraduate (Steve in this case) and a graduate student, whether academically, mentally, or morally. Countless interactions, countless hours, and countless comments you gave me made my scattered research ideas solid, made my convoluted paper drafts distinguished, and made my overambitious practice talks appreciated. For that, I am grateful.

Ross Tate, Adrian Sampson, Aslan Askarov, Robert Soulé, Abhishek Anand, Mark Reitblatt, Andrew Hirsch, Fabian Muehlboeck, and Harry Terkelsen are among other members of the programming languages field at Cornell CS who gave me invaluable suggestions and comments on my research papers. For that, I am thankful.

Conveying research ideas would be less effective if communication skills were not kept practiced. I appreciate the opportunities in teaching from the Department of Computer Science, both at the University of Pennsylvania and at Cornell University, that allowed me to reinforce these skills. Val Tannen, Sampath Kannan, Daisy Fan, Andrew Myers, and Adrian Sampson were my role models who set a high expectation in structuring, managing, and running CS courses, small or large, undergrad or grad. They are excellent instructors who bring high-quality students to their courses for which I was on staff. In turn, these students propelled me to maintain a high standard as a teaching assistant. For that, I am thankful.

Becky Stewart, Assistant Director of the Ph.D. Program, deals with everything from processing forms to answering hundreds of questions I had about keeping the department running and keeping my Ph.D. career going, or simply by being there just to chat. Becky, you helped me with bureaucrazy from the beginning to the end. For that, I am thankful.

The Thai students and staff at Cornell not only form a tight-knit community that lets me wind down outside my research time, but also give me support when I need the most. Special thanks go to Apikanya (M) McCarty, Pakawat (Kun) Phalitnonkiat, Chayanee (Namtip) Chawanote, Raksit (A) Pattanapitoon, Kullachate (Oath) Muangnapoh, Tanapong (Nont) Jiarathanakul, Pimbucha (Pim) Rusmevichientong, Bunyarit (Pao) Meksiriporn, Sunsiree (Wahn) Kosindesha, Siraphat (Fay) Taesuwan, Chalermpat (Nong) Pariya-Ekkasut, Visarute (Earth) Pinrod, Ithipong (Billy) Assaranurak, Thapakorn (Hize) Jaroentomeechai, Kittkun (Ob) Songsomboon, Fikri (Fik) Pitsuwan, Pidchapon (Fai) Niruthisard, and Ravi (Note) Laohasurayodhin, who helped me with practice talks, tortured me with checkmate moves in the *Settlers of Catan* and *Dominion*, planned road trips, cooked good food, and went to snowy mountains with me. Guys, you kept me alive, kept me lively, and kept me in check during all these years. We are the force that will make Thailand an even better place. For that, I am hopeful.

My family members—mom, dad, sisters, and uncle—have been of tremendous support even if they could not be with me in person all the time. Mama, thanks for all the phone calls that let me vent and calmed me down before big moments. You all gave me more than I can describe or can pay back.

Studying far away from home kept me a distance from my biological family, but I was lucky to be part of physically closer families over my time here in the States, either by choice or by chance. Lou, Linda, and Michael Riccio, my host family at Westtown, got me on my feet during my first year. Then, I had a family of Thai Scholars whom I got to know every year at the Christmas Program in Stony Point, New York. Last but not least, the staff who ran the Christmas Program itself became a family that made my every Christmas feel like home. A few of these staff members deserve special mentions. Bill Collins and Paula Sandusky were good hosts who operated the Center flawlessly. Ekaphan (Bier) Kraichak was a superb reader and writer who made the program educational yet enjoyable. Christine Brown dealt with logistics, but was a person I felt I could talk to any time about anything. Mananya (Nammon) Tantiwiwat, my first mentor, could not make the program livelier. P'Nammon, you set an example that it is possible to be a great person despite hardship in life. Your dork is now a doc. Thanks for looking after me from above. Finally, John Paul Rorke, the program director, a historian who tried to understand string theory and to comprehend how compilers work, devoted his life to making the journey of Thai Scholars, including mine, a smooth ride when studying abroad. John, sorry I took a bit too long to complete my journey and show you my compilers. Thanks for helping me understand myself and find my confidence. Thanks for your synergy speech you gave to a thousand Thai Scholars. I'll be sure to put that to work. Thanks for all the fun history lessons, the fireside chats, the trips to the Chinese place or the diner. Above all, thanks for being my Santa all these years.

For that, I am eternally grateful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

General-purpose programming languages are often unsatisfactory for building real-world applications [117]. At the front end, general-purpose language constructs can be too unwieldy for concisely expressing desired properties and idioms pertaining to the task at hand. At the back end, machine code generated by general-purpose language compilers might be suboptimal for the specific problems being solved. The introduction of domain-specific languages, therefore, has become a common approach for addressing problems in computer security [74, 76], distributed systems [63], syntactic analysis [48, 51], and formal methods [25, 68], among many other domains of interest.

Programming languages, whether general-purpose or domain-specific, are subject to revisions and evolution. New language features and program analyses may be added to existing languages to reflect programming language research frontiers. Existing features and analyses may be revised to benefit from advances in the underlying hardware technology. Ideally, these language features and program analyses, which shall be denoted collectively as *extensions*, should be implemented in a modular way, forming a *language feature toolbox* from which programmers and language designers can pick just the features best suited to the task at hand. Modular implementations of extensions enable feature sharing across domain-specific languages, and can even be combined into a general-purpose language containing features that can tackle a wide range of domain-specific problems.

A grand, unified programming language that can solve every computable problem efficiently and requires only minimal coding and specifications is to be yearned for. Ideally, this language should be implementable by simply building on individually developed constructs in the language feature toolbox, but in reality, combining small, individual extensions remains a challenge. The syntax in different extensions may conflict,

introducing ambiguities that need to be resolved when constructing the parser for the combined language at the front end. The available target languages in one extension may not overlap with those in another extension, calling for additional translations to a common target language at the back end. To connect the two, individual compilers for these individual extensions need to work in unison; the resulting compiler for the combined language needs to have addressed any semantic tension that may have arisen from the interaction between individual language features.

As code duplication is undesirable for software maintenance and evolution [4, 10], existing compiler code for component extensions should be reused for the combined language. In this way, corrections and changes to the component extensions can be propagated to the combined language automatically. Code reuse would be difficult or even impossible, however, if implementation languages of the component extensions are incompatible with each other. A programming language that can interact with existing implementations is therefore a minimal prerequisite for permitting code reuse. Alternatively, extensions could be implemented with the same programming language so that they can be stitched with short pieces of glue code. Regardless of how extensions are written, we need an implementation language that lets developers combine the features in an efficient way. The end result, a compiler for the combined language, is yet another real-world application, which brings us to where we began: are general-purpose programming languages suitable and satisfactory for this task of *composing* compilers?

Mainstream programming languages have a number of advantages for building compilers. These languages have better support for integrated development environments (IDEs), and are better understood by a wider spectrum of programmers. Better IDE support lets compilers be composed with less trouble, as IDEs can catch trivial programming errors that may arise during the composition. Mainstream languages are understood by a broader community of programmers and are used heavily in industry, so more manpower

is available for new applications, including new compilers, to be developed. Even after release, any lingering errors in compilers implemented in mainstream languages are more likely to be discovered, understood, and resolved. For these reasons, mainstream, general-purpose programming languages appear to be a suitable candidate for composing compilers.

On the other hand, mainstream languages remain unsatisfactory for composing compilers for a number of reasons. They might not be expressive enough to guarantee that the resulting compilers will be type-safe. They might require an unreasonable amount of code to be written for individual features to interact in a meaningful way. They might be inimical to the evolution of existing language features and analyses and make compilers difficult to maintain. Consequently, the problem of composing compilers has been addressed in several ways:

- To support composition, new domain-specific languages have been proposed [14, 35, 80, 83, 118].
- The composability problem itself is relaxed to the *extensibility* problem, in which new language features and analyses can be added in a modular way with a modest amount of code [32, 44, 78], but composing compilers remains challenging.
- The composability problem is limited to the front end, where syntax can be composed, but the possible target languages remain fixed. This approach is usually achieved using macro-rewriting systems [37, 106, 112, 113].
- New programming paradigms have been proposed to organize code in a more modular and understandable way [53, 115].

New programming languages and programming paradigms create obstacles for making compilers composable in practice. Programmers need to spend time learning new features. IDE support may not be readily available. The implementations of these new languages themselves might not be entirely correct, adding another dimension to diagnos-

ing programming errors. Meanwhile, existing general-purpose, mainstream languages continue to evolve. Can we influence this evolution to head in the right direction so that mainstream languages are more satisfactory for composing compilers?

This dissertation aims to address missing concepts and tools that will encourage use of mainstream, general-purpose programming languages for composing compilers and will encourage the evolution of mainstream programming languages to better support this goal.

1.1 Design patterns for composing compilers

Existing mainstream, general-purpose programming languages do not support composing compilers efficiently. To identify their weaknesses, we first implement a collection of compilers for a number of individual language constructs, using Java, a well-known mainstream programming language. Then, we attempt to compose these compilers using as little extra code as possible. Whenever language features that would simplify composition are unavailable in Java, we use design patterns [40], a collection of code templates implementable in Java, to model them. During this process, we also discover that some Java language mechanisms impede compilers from evolving in a scalable way. Again, design patterns are used to simulate desired language mechanisms to make evolution possible under composition. Even if existing language mechanisms by themselves are not adequate for our goal, we can still exploit them as much as possible, using design patterns to achieve it. These design patterns used for implementing composable compilers will help suggest possible directions of mainstream language evolution, including new features to be added to the language, and how these new features can be implemented and translated. The core design patterns constitute the skeleton of a framework for implementing composable compilers.

The key observations for making compilers composable and evolvable are as follows:

- To avoid code duplication, inheritance provided as part of the object-oriented language mechanism should be used so that common code can be written once in a hierarchy of incrementally more specific kinds of objects. In other words, a hierarchy of classes should be used to keep implementations modular. One application of inheritance is with the representations of abstract syntax trees (ASTs), where each AST node type inherits state fields and operations, such as type-checking, from parent node types as much as possible.
- To support the additions of future operations on the AST while maintaining the ability to inherit superclass implementations, the supertype relationships for AST node types are not necessarily fixed across extensions. More specifically, an extension might add a new operation to a subset of existing AST node types, but this operation can be implemented in the same way for all the node types in this subset. To maintain modularity, then, a new AST node type equipped with the implementation of the new operation should be introduced, and the existing node types in this subset should add the new node type as their additional supertype.
- To support the additions of new properties (e.g., fields) in AST nodes without duplicating part of the existing class hierarchy when implementing extensions, instance variables and methods can no longer be used as the primary way to store and access AST node properties. That is, an AST node needs to be represented with a more elaborate data structure, not just a single object.

Our design patterns incorporate these observations.

To illustrate that our design patterns can be used to implement real-world compilers and not only smaller ones, we port the implementation of the Polyglot extensible compiler framework for Java [78] to use our framework, show that the amount of code is comparable, and demonstrate that the performance of the compiler implemented in our framework

does not significantly degrade when compiling large programs. This experiment suggests that our approach is practical for making real-world compilers evolvable and composable.

1.2 Better diagnosis for parsing conflicts

The design patterns for implementing composable compilers primarily deal with the representation of AST nodes and the relationships between AST node types. If all programs were written as ASTs, the design patterns would address all the key parts pertaining to the composability problem. This might be true for visual-based programming languages [9, 46, 64] where source programs are directly represented as ASTs, but a majority of programming languages remains lexical-based. In these languages, programs are still written as text files that require lexical analysis and parsing. Language engineers need to not only resolve any conflicting specifications resulting from the interaction between existing language semantics, but also design a grammar that avoids ambiguities that may result from the interaction between existing language syntax.

Two lines of previous work deal with grammar composition:

- Grammars are composable by way of composing parser tables, but the resulting grammar may be ambiguous [21, 116], although ambiguities can be prevented by working with restricted forms of grammars, e.g., LR(0), which do not use lookahead symbols.
- Only grammars that do not violate a set of restrictions are composable, so that the resulting grammar will not contain ambiguity or conflicts in the parser [104, 105]. These restrictions primarily concern the internal mechanism of parser state machines to ensure that the composed parser is conflict-free.

Either way, some restrictions need to be enforced to ensure that the resulting grammars are acceptable. In general, however, programming language grammars might not always

satisfy such restrictions, especially when programming languages evolve over time. Additions of new language constructs can increase the likelihood of these restrictions being violated. Moreover, modifications or removals of existing language constructs are orthogonal to grammar composition. Techniques for composing grammars alone are hence inadequate for the evolution and composition of compilers.

The ambiguity problem in grammar engineering not only happens when existing grammars are composed, but also occurs firsthand when new language syntax is initially designed. Techniques for detecting ambiguities in grammars will help troubleshoot syntactic flaws. Unfortunately, determining whether a context-free grammar is ambiguous is undecidable [47]. Still, attempting to provide a diagnosis for faulty grammars when possible would be useful.

Although a conservative approximation algorithm that decides when a grammar may be ambiguous is one possible way to detect faults, such simple verdicts are not helpful enough for language engineers to pinpoint the root cause of the ambiguities. A better way to provide diagnosis is to give a *counterexample* that illustrates why a grammar is ambiguous. That is, a counterexample consists of a single sequence of tokens decorated with two distinct ways to parse that sequence. Having seen a counterexample, language designers are notified of the real cause of ambiguity immediately so that they can start fixing relevant parts of the grammar right away.

To make the problem of counterexample generation practical for compiler construction, we focus on diagnosing LR(1) grammars, a specific class of grammars that can describe the syntax of most programming languages. Strings generated by LR(1) grammars are accepted by a left-to-right, rightmost derivation parser that uses a single lookahead terminal symbol. LR(1) parser generators, such as Yacc [51], can transform a valid LR(1) grammar into a parser. On the other hand, LR(1) parser generators attempting

to transform a non-LR(1) grammar will produce a *conflict* indicating that the grammar is faulty.

While the details of a conflict given by LR(1) parser generators can be used as a proof of faulty grammars, parsing conflicts themselves do not immediately suggest why the grammar needs a correction. This is because the conflicts describe what would go wrong in the internal mechanism of the parser were it be generated, rather than the faulty location in the grammar. Instead of requiring parser generator users to understand the clockwork of LR parsers before attempting to fix a bad grammar, we give counterexamples, a different kind of error messages that are easier to interpret by grammar writers.

The key observations for generating useful counterexamples are as follows:

- Counterexamples should not be more specific than necessary. For instance, in the infamous *dangling-else* ambiguity, the counterexample `if 2!=5 then if 4!=7 then 2112 else 2110` is too specific, because the concrete, complete derivation of the expression `2!=5` is unnecessary. That is, a nonterminal in the grammar that does not play a role in causing conflicts in the parser should be left unexpanded.
- Counterexamples should be localized to the ambiguity when possible. In other words, counterexamples need not be a derivation of the start symbol, but rather the innermost nonterminal that can derive an ambiguous string.
- Faulty LR(1) grammars might not be ambiguous, so ambiguous counterexamples cannot always be generated. In addition, since the ambiguity detection problem is undecidable, an ambiguous counterexample might not always be found even if the grammar is ambiguous. As a fallback strategy, a different kind of counterexample should be generated in order to give grammar writers as much debugging information as possible.

Our counterexample-finding procedure takes these observations into account and has been integrated into an existing LALR parser generator [48], so that a meaningful counterexample is displayed along with each conflict, if any, in an input grammar.

To demonstrate that our methods for finding counterexamples is usable in practice, we run our tool against faulty grammars that various programmers having difficulties debugging have asked on an Internet forum and show that counterexamples are generated almost immediately after a conflict is detected. We also run our tool against grammars used in evaluating a previous approach on generating counterexamples [7] and show that, on average, our approach is an improvement. These experiments suggest that our counterexample generator can be user-friendly for debugging grammars.

1.3 Safe pattern matching for objects

So far, we have pointed out issues that may arise at the heart of compiler construction, i.e., the modularity of the implementation of AST nodes, and at the front end, i.e., the difficulty of troubleshooting faulty grammars. An equally important part of the compiler is the back end, where newly defined language constructs are translated into existing, less expressive constructs. Many such translations are syntax-directed: the translations are defined by the types and structures of AST nodes, and components of an AST node are translated recursively. Pattern matching is a powerful technique that can extract these AST components in an expressive way when implementing a compiler in a functional language, but pattern matching has yet to be as powerful in object-oriented programming languages because of a tension between data abstraction and pattern safety.

Unlike in object-oriented languages, declarative programming in functional programming languages makes pattern matching safe. Specifically, the declaration of an algebraic data type in functional languages lists all the possible, distinct cases to which members of the data type may belong. This declaration enables an algorithm [65] that checks whether

(1) a group of patterns handles all members of the data type, i.e., patterns are *exhaustive*, and (2) a pattern in question will handle some members of the data type that have not already been handled by a prior group of patterns, i.e., the pattern is not *redundant*. Upon encountering a nonexhaustive group of patterns or a redundant pattern, the algorithm emits a compilation warning so that programmers can inspect these patterns for potential programming errors.

Even though pattern matching is a dominant feature in functional languages, the safety of pattern matching depends on the availability of information on the concrete implementation of data types. That is, the specified cases of an algebraic data type restricts the number of possible implementations the data type may have, and for each case, exactly one implementation is permitted. In object-oriented languages, where data abstraction prevails, this restriction unnecessarily limits the power of abstract data types that allow multiple concrete implementations. To provide pattern-matching support under the presence of data abstraction, another source of information than the concrete implementations of members of a data type should be used when patterns are checked for exhaustiveness and nonredundancy. Attempts to make pattern matching coexist with data abstraction have partly addressed the safety concerns:

- Pattern matching can be used on abstract types, e.g., interface specifications, but there is no guarantee whether patterns will handle every possible value of the object being matched. This line of work includes Wadler's views [121], the Pizza language [81], extensible algebraic data types with defaults [126], and extractors [33].
- Pattern matching can be used on abstract types, but matching only works safely when the number of concrete implementations of the abstract type is limited. That is, knowledge on concrete data types remains essential for verifying pattern exhaustiveness. This line of work includes active patterns in F# [111], sealed classes in Scala [84], an extension of extractors [31], and the RINV language [123].

- Invertible computation is introduced as a new language feature to provide a stronger guarantee that pattern matching a member of a data type yields a consistent result with the construction of that member. This line of work includes Wadler's views [121], modal abstraction [61], and the RINV language [123].

Still, full-fledged pattern matching on abstract data types, where concrete, specific implementations of objects play no role in checking pattern safety, is hard to come by.

To further address these concerns, we propose a novel language design that reconciles safe pattern matching with objects. The key observations for better pattern matching are as follows:

- An abstract specification independent of concrete implementations is needed to describe possible cases of an algebraic data type. This specification is similar to a data type declaration in functional languages, but different cases do not pin down specific implementations.
- Since pattern verification can no longer depend on a concrete implementation, each implementation now needs a *matching specification* that describes when pattern matching on that implementation will succeed.
- As a single member of an abstract data type may be represented in multiple ways using different implementations of the data type, a new language mechanism is necessary to automatically convert one implementation of an object to another in order for pattern matching to work properly.

We implement these ideas as an extension to modal abstraction [61] and show that verification of patterns under data abstraction is possible using an SMT solver [28]. Further, we demonstrate that our implementation allows multiple implementations of the same abstract data type to interoperate seamlessly when used in pattern matching.

1.4 Dissertation structure

The remainder of this dissertation is organized as follows. Chapter 2 identifies language features needed to make compilers evolvable and composable, and suggests possible ways to implement them by way of design patterns. Chapter 3 describes an incomplete but effective procedure to construct counterexamples that illustrate conflicts in lookahead LR parsers. Chapter 4 explores a new language design for supporting safe pattern matching in the presence of data abstraction, which would make compiler implementations even more concise. Finally, Chapter 5 offers concluding remarks and discusses possible directions for future research.

CHAPTER 2

PRACTICAL DESIGN PATTERNS FOR COMPOSING COMPILERS

A language feature toolbox containing individual constructs that can be cherry-picked and composed into a working language would be useful for domain-specific applications and language-design experimentation. But developments of this toolbox is held back because programming languages that support such composition are widely unknown or lack convenient development environments. In other words, existing mainstream programming languages complicate implementations of programming language composition.

One way to encourage the development of such toolboxes is to identify what is missing in mainstream languages that would make composition easier to implement and whether this absence can be remedied within existing, well-known languages. In this chapter, we present a design-pattern approach to make implementations of compilers composable and evolvable in a scalable way. This is joint work with Andrew Myers.

2.1 Introduction

Few programmers seem completely happy with the programming language they are using. Domain-specific language extensions are frequently introduced to ease the difficulty of developing specific applications. Even mainstream languages continue to evolve through the addition of new language features and program analyses. Ideally, small programming language extensions should be easy to implement modularly, so programmers and language designers are provided a *language feature toolbox* from which they can pick just the features best suited to the task at hand.

In languages that support macros and metaprogramming, language extensions that are sufficiently simple can often be implemented within the language, as libraries [106, 112, 113]. However, there are shortcomings to this approach. It is difficult to produce

satisfactory compile-time error messages, and composing independently developed yet conflicting extensions is problematic.

It is therefore desirable to be able to easily modify compilers in a modular way, so that all parts of the compilation process can be customized, including the language's syntax, type system, static analyses, and translations to target language(s). In the limit, a compiler becomes a framework of various modular transformations that can be freely combined and extended to arrive at the desired language design.

Unfortunately, such compilers do not exist yet. Traditional compilers are relatively monolithic, and changes to the language often involve modifying a large, intricate code base in many locations. This is not a recipe for ease of development or maintenance. Part of the problem seems to be that existing programming languages do not support the development of a highly extensible compilers particularly well.

Software is in general difficult to extend in a modular way, but compilers are a particularly difficult case, because a typical compiler extension involves modifying both the data structures representing program code within the compiler and the code that traverses and rewrites this code. What is desired is *scalable extensibility*, meaning that code changes should be modular and the amount of code written should be proportional to the size of the change being made—and not to the size of the compiler being modified. Various design patterns have aimed to provide solutions to this extensibility challenge [78, 86]. However, these design patterns can also result in a lot of boilerplate code that discourages programmers from implementing full-fledged compilers.

Beyond scalable extensibility, what is missing is an effective, scalable way to compose language features. Composition poses two main challenges that implementation languages have to address: data representations that have been independently extended must be composable, and extended transformations must be integrable while resolving conflicts that may arise from incompatible parent languages. Previous extensibility design patterns

have not solved these problems. To address these problems, some new language designs add features to allow composition of code modules [14, 35, 80, 83]. This language-based approach has the advantage that it can be concise while offering static enforcement that increases code safety. However, boutique implementation languages also have trouble achieving widespread adoption.

Our approach in this paper is to develop new extensibility design patterns that allow compilers to be implemented as composable modules. Because we develop design patterns rather than a new language, we have the advantage that the compiler code remains in a mainstream, well-supported programming language. This language happens to be Java, but the techniques developed here would apply to any other modern, mainstream object-oriented language, e.g., C#, that supports dynamic dispatch, multiple interface inheritance, and traits [100]. Specifically, we make the following contributions:

- We identify challenges for the problem of composing compilers (Section 2.2).
- We propose a desirable relationship between programming languages that enables composability (Section 2.3).
- We present an encoding that permits changes to the inheritance relationships among AST node types during compilation so that compilers can evolve in a natural and scalable way (Section 2.4).
- We present a new representation of AST nodes that allows the state associated with an AST node to be extended and composed, and allows a single AST to represent programs in multiple language variants (Section 2.5).
- We present a new dispatch mechanism that addresses the inflexbility of the standard object-oriented dispatch (Section 2.6).
- We present a new translation mechanism that allows for one-to-many transformations, and makes translation code reusable and composable (Section 2.7).

- We show that our design patterns are effective and scalable by implementing a composable compiler framework for various language variants (Section 2.8). While we have chosen to implement this framework using a mainstream language, there is a lesson for language design here too. The design patterns also suggest useful new language features that could have made our implementation even better.

Section 2.9 discusses related work, and Section 2.10 concludes.

2.2 Challenges in composability

A common goal in research on compiler technology is to make compilers easier to implement. This section overviews challenges in implementing composable compilers, discusses various attempts to address these challenges, and outlines our design-pattern solutions. Additional related work can be found in Section 2.9.

2.2.1 Ease of extension

New programming languages are usually derived from an existing language in several ways. Additional language constructs may be added, available constructs may be removed, or the semantics of certain constructs may be redefined. Even if language constructs do not change, additions of new program analyses can also create a new language, as the same program may yield different compilation results. More precisely, we will use the terms *new language* and *extension* to refer to the result of any modification, however small, to the *base* language.

Extensible compiler frameworks are designed to make extensions easier to implement. JastAdd [32, 44] and Polyglot [78] are frameworks that rely on object orientation to provide scalable extensibility. Design patterns such as factory and visitor patterns [40] are heavily used to maintain modularity. Inspired by experiences with Polyglot, our design

patterns do accommodate scalable extensibility, and, unlike Polyglot, also enable *scalable composability*, in which independently developed extensions can be combined with minimal effort. In Section 2.8, we demonstrate that our design patterns can implement Polyglot extensions with a similar amount of code.

2.2.2 Composability vs multiple inheritance

Research in programming languages often involves experimentation with a small number of language features or program analyses that improve on existing languages. Domain-specific languages often work the same way: only a few new constructs are needed to solve the problems of interest. The ability to freely compose many language extensions to obtain a customized language that meets users' needs—a sort of multiple inheritance at the language level—is an ideal strategy.

An issue with multiple inheritance is the *diamond problem*, in which ambiguity arises when a class extends two classes that happen to implement the same operation. In languages like C++, where multiple class inheritance is permitted, the diamond problem also causes issues with consistency of object representations for shared superclasses. But most mainstream object-oriented programming languages only support single inheritance of classes. As a result, JastAdd resorts to aspect-oriented programming [53] to enable composability. The language J& [80] solves the diamond problem by requiring that the subclass overrides the operation to resolve the ambiguity. J& has an elaborate object structure to ensure that a state is represented by only one location. C++ duplicates objects for the shared superclass unless all subclasses are declared virtual. Scala requires that states only appear in classes, which remain in the single inheritance portion of the language. Our design patterns maintain object consistency in the same way that J& does, but do so within a mainstream language. With a handcrafted type system, J& can detect conflicts from the diamond problem at compile time; our approach trades this static

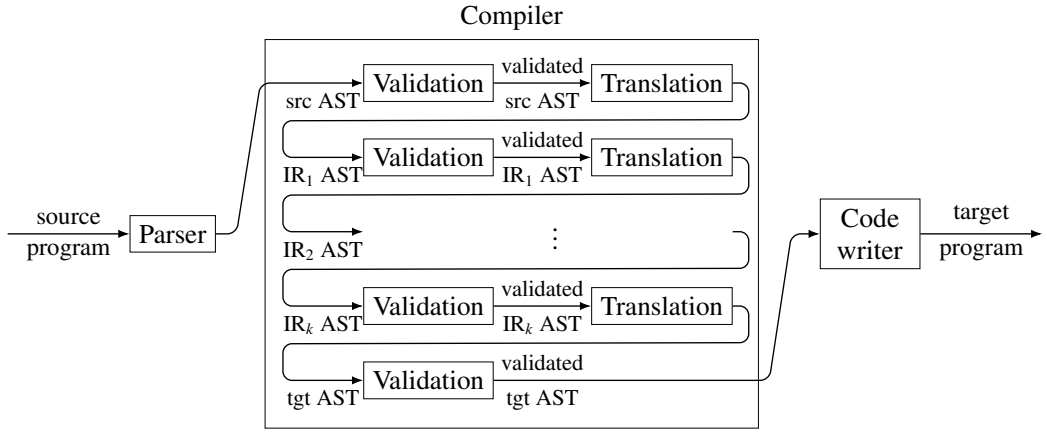


Figure 2.1: Compilation stages, primarily a sequence of AST validations and translations between languages

safety property for a mainstream-language solution to composability, but still flags such conflicts at run time.

2.2.3 Monolithic vs nanopass compilers

Traditional compiler implementations tend to bundle operations on ASTs into one monolithic pass. Figure 2.1 illustrates a typical pipeline of compilation stages. Program text is parsed into an AST; the AST is validated and then translated into a sequence of intermediate languages, prompting additional validations and translations; finally, the target language is reached, and the code is output as an executable or for additional processing by external back-end tools. With all these operations done in a small number of passes, incremental changes to traditional compilers are hard to make modular. A more fine-grained approach is needed to enable extensibility while maintaining modularity. Work on nanopass compilers [52, 99] shows that it is possible to structure compilers into many very small compiler without do not incur significant overhead. Our approach is influenced by version 2 of Polyglot, which introduces a system of *goals* and *passes*: each pass indicates a number of prerequisite passes that need to complete before the pass itself

can be run, forming a dependency graph of passes within each extension. When a new extension is created by composing existing extensions, the new dependency graph can be formed by adding and modifying the lists of prerequisites as appropriate.

2.2.4 Intermediate languages

Part of the canon of compiler design is to have an intermediate representation (IR) independent of any particular source or target language. When a compiler is structured into many small passes, there are effectively many intermediate languages, transitioning across each compiler pass from an intermediate language closer to the source to one closer to the final target. Naively implemented, each compiler pass would create an entirely new AST; instead, our goal is to mostly share the AST data structure between these program representations, while explicitly keeping track of the language in which to interpret it.

2.2.5 Flexibility of target languages

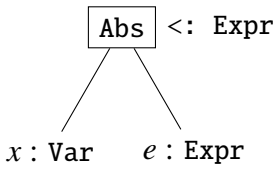
Rewriting is a critical step for translating higher-level programming languages into lower-level languages. Since extensions to programming languages often introduce additional syntactic forms, macros are a prime candidate for implementing compilers. Programming languages such as C++, OpenJava [112], and Racket [37, 113] have integrated macro systems to make rewriting more convenient. However, macros can only translate constructs into a limited number of target languages that are built on top of a predetermined, canonical target language. An ideal implementation of a compiler, on the other hand, should be able to support translations from one source language into any target language of choice, because the target language may vary depending on the hardware architecture and operating system the resulting program will be executed in.

Even though translations can be considered as another compiler pass, our approach decouples the validation phase, e.g., type checking, from the translation phase. This is because validation is independent of the target language and should be performed even if no translations are left, as in validating the target AST in Figure 2.1. In this way, our design patterns can support one-to-many translations.

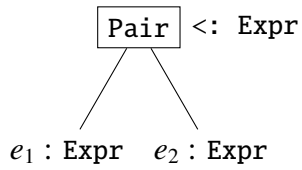
2.3 Programming language hierarchy

Reusability plays an important role in measuring composability. For compilers to be composable, abstract syntax trees (ASTs) that represent programs in an existing programming language should also be representable in extensions of that language. The ability to share ASTs across languages avoids the inconsistency from using different representations for the same program in different languages, even if that program's semantics does not change. Moreover, an AST representation tied to a fixed language risks incompatibility when that language is to be composed with another. In other words, the design of ASTs that is independent of specific programming languages will minimize incompatibility and maximize code reuse across compiler implementations, making composition scalable. To achieve this independence, we first need to consider how programming languages relate to one another so that their compilers can be composed without much difficulty. Developments in this section are driven by examples to give a concrete picture of how to compose languages properly.

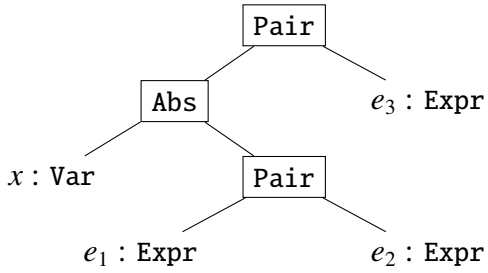
We write the names of programming languages in SMALL CAPS for the remainder of the chapter. Consider two small programming languages: (1) the untyped λ-calculus (denoted LC), containing variables, abstractions, and applications; and (2) expressions on pairs (denoted PAIR), containing pair constructors and projections. Figure 2.2(a) shows a natural AST representation for a λ-expression $\lambda x.e$, and Figure 2.2(b) for a pair (e_1, e_2). Notice that the λ-abstraction and the pair constructor, both subtypes of `Expr`,



(a) AST for $\lambda x.e$ in LC



(b) AST for (e_1, e_2) in PAIR



(c) AST for $(\lambda x.(e_1, e_2), e_3)$ in LCPAIR $\triangleq$ LC $\oplus$ PAIR

Figure 2.2: Composing ASTs across language extensions

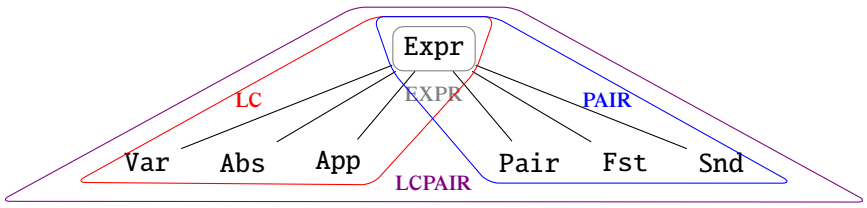
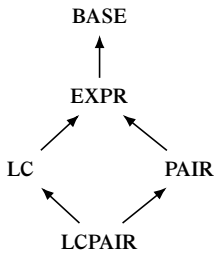


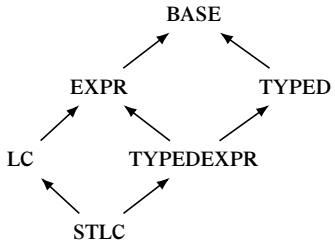
Figure 2.3: Relationships among AST node types across extensions

contain a subexpression of type `Expr`. If LC and PAIR are to be composed, but `Expr` were implemented separately in each extension, then `Abs` could not be a component of a pair, and `Pair` could not be the body of a lambda. This would prevent the two languages from being composed seamlessly, as we would have to reconcile the definition of `Expr` before an AST containing both of these constructs, such as one in Figure 2.2(c), could be constructed.

This observation suggests that abstract language features like `Expr` should be shared across languages. That is, abstract constructs can be collected and defined in an *abstract language*. Actual programming languages should then extend this abstract language to implement concrete constructs. Figure 2.3 shows how this factoring works for our



(a) Language hierarchy for LCPAIR



(b) Language hierarchy for STLC

Figure 2.4: Examples of programming language hierarchies

example: generic expressions of type `Expr` are defined in abstract language EXPR, which is extended by LC and PAIR. Then, the composed language LCPAIR can simply inherit the constructs from both of the parent languages. More generally, all abstract constructs can be defined in a collection of abstract language extensions, leaving us with the empty language (denoted BASE), containing only the top type of all language constructs, at the top of the language hierarchy. The BASE language is then the ancestor of every language variant, extended or composed, abstract or concrete. Figure 2.4(a) illustrates the language hierarchy relevant to LCPAIR.

The BASE language can also be a basis for *mixin languages*, where additional state is added uniformly to every existing construct, without introducing new concrete constructs otherwise. For instance, suppose we would like to add a type system to the LC extension to implement the simply-typed λ-calculus (denoted STLC). In this case, the compiler will have to compute the type of each expression, whether from type annotations in the source program or by type inference. Each expression needs to be decorated with an *attribute* representing the computed type. In general, type attributes could be attached to any AST nodes, so we declare an abstract language TYPED, an extension of BASE that adds type attributes to AST nodes, as shown in Figure 2.4(b). We can then compose EXPR and TYPED to obtain the abstract language for typed expressions (denoted TYPEDEXPR). Finally, the

simply-typed λ-calculus is the composition of LC and TYPEDEXPR, where type attributes are mixed into abstractions and applications.

In what follows, we write "A is an *immediate superlanguage* of B" if B directly extends A, e.g., LC is an immediate superlanguage of LCPAIR; and write "A is a *superlanguage* of B" if B extends from A, possibly indirectly, e.g., BASE is a superlanguage of LC. Further, we use "B is an *immediate sublanguage* of A" if A is an immediate superlanguage of B, and "B is a *sublanguage* of A" if A is a superlanguage of B.

2.3.1 Implementing programming language hierarchy

The implementation of composable compilers begins with the declaration of *language definitions* that capture the relationships among language extensions. Figure 2.5 shows an implementation of language definitions for three of the extensions related to LCPAIR, whose relationships are shown in Figure 2.4(a). First, we define the language definition interface for the BASE language, called `BASELang`, that serves as the root of the programming language hierarchy. This interface extends `Lang`, which contains utility methods for keeping track of superlanguages, and for determining whether one language is a sublanguage of another. This `Lang` interface is shared across all language definitions and only needs to be implemented once. Appendix A.1 lists the core implementation of `Lang`.

Inside the `BASELang` interface, an abstract class `Class` is declared so that an instance representing the BASE language can be constructed. Once again, most of the work is already done in `Lang.Class`, defined in the `Lang` interface; the only job that remains for the BASE language is to make the `Class` class a subtype of `BASELang`. This is achieved by using Java's `implements` clause.

An instance of the language definition interface for a given programming language can be thought of as the identity of that language. Consequently, there should be at most one such instance per language: language definition classes should be singleton. To

```
/** A {@code BASELang} represents the root programming language. */
public interface BASELang extends Lang {
   abstract class Class extends Lang.Class implements BASELang {
      protected Class(Lang... superlangs) {
         super(superlangs);
      }
   }
   BASELang.Class instance = new Class() {
      // Define various factories for the language (later).
      ...
   };
   ...
}

/**
 * An {@code EXPRLang} represents a programming language
 * containing expressions.
 */
public interface EXPRLang extends BASELang {
   abstract class Class extends Lang.Class implements EXPRLang {
      protected Class(Lang... superlangs) {
         super(superlangs);
      }
   }
   EXPRLang.Class instance = new Class(BASELang.instance) {
      ...
   };
   ...
}

/** An {@code LCPAIRLang} represents a lambda calculus with pairs. */
public interface LCPAIRLang extends LCLang, PAIRLang {
   abstract class Class extends Lang.Class implements LCPAIRLang {
      protected Class(Lang... superlangs) {
         super(superlangs);
      }
   }
   LCPAIRLang.Class instance =
     new Class(LCLang.instance, PAIRLang.instance) {
      ...
   };
   ...
}
```

Figure 2.5: Example implementations of language definition interfaces

enforce this requirement, each language definition interface declares an interface constant named `instance` that instantiates the unique language definition object. For example, `BASELang.instance` represents the identity of the BASE language.

Language definition interfaces for further extensions are implemented in the same way. The main difference is the presence of superlanguage identities as additional arguments to the constructor of the language definition class. For instance, the EXPR language is an immediate sublanguage of BASE, so we need to pass `BASELang.instance` as the argument to the constructor to establish this relationship. For composed languages, the implementation is similar, as shown in the initialization of the language identity instance for the LCPAIR language.

The readers might notice a redundancy in indicating superlanguages. The identity instances of superlanguages are passed as arguments to the constructor, but the declaration the language definition interface itself also extends superlanguage definition interfaces. The constructor arguments account for the lack of features in the implementation language (Java) so that composition can work properly. On the other hand, interface extensions still rely on existing language mechanism, to enjoy as many static safety properties offered by the implementation language as possible. Specifically, the language identity instances are used when the compiler is run, while the declaration of superinterfaces helps ensure the consistency of inherited method implementations in language definition interfaces. Although the redundancy appears undesirable, our experience indicates that static errors are helpful for catching bugs early on, even if run-time errors remain possible. The technique of introducing compromises for missing language features and relying on static checking whenever possible will be a common theme in the design patterns described here.

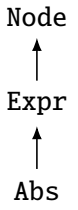
2.4 Evolvable class hierarchy

Programming languages change over time. Changes to an existing language create a new version of the language, requiring a new version of the compiler. To maintain backward compatibility, the source code of existing compilers cannot simply be altered to handle the changes. Rather, the new version of the compiler should reuse as much code as possible from the existing compiler, and new code can be added to handle changes where necessary. Therefore, to minimize the cost of compiler construction and maintenance, we need a good way to make compilers for a single language evolvable before we even attempt to compose compilers from multiple languages.

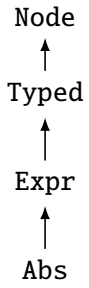
In the traditional object-oriented programming paradigm, the hierarchy of classes is statically determined. That is, a class declaration specifies a list of superclasses or superinterfaces the class extends or implements. However, compiler evolution can change the relationship between classes. For instance, mixin composition [20] adds new IS-A relationships to existing classes. To keep AST nodes independent of a particular language, the class hierarchy for AST node types effectively must be able to change across languages. We need design patterns that support schema evolution, like in language-based approaches such as open classes [24] and nested interitance [79].

For example, suppose the λ-calculus is to evolve from untyped (LC) to simply-typed (STLC)[1]. Suppose further that, in LC, the class hierarchy relevant to the AST node type for λ-abstractions (denoted `Abs`) is as shown in Figure 2.6(a), where `Abs` is an immediate subclass of `Expr` that represents all expressions, and `Expr` is an immediate subclass of `Node` that represents all AST nodes. The addition of a type attribute to every expression in STLC means that `Expr` should now be a subclass of `Typed` that represents all AST nodes containing type attributes. As such, the class hierarchy relevant to `Abs` now has

[1]This in fact does limit the expressiveness of λ-calculus, but adds safety properties to the evaluation of λ-terms [23].



(a) AST class hierarchy in LC



(b) AST class hierarchy in STLC

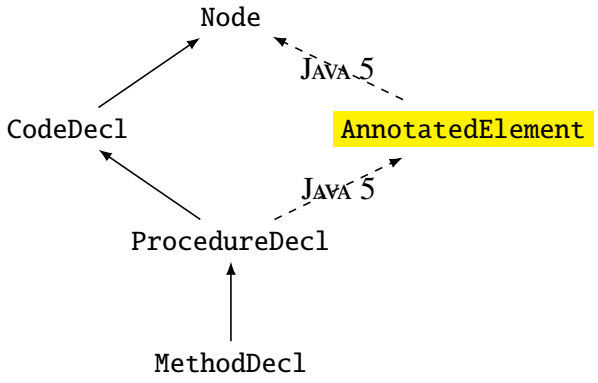
Figure 2.6: AST class hierarchies for `Abs` under the evolution of λ-calculus

to change to one shown in Figure 2.6(b). Changing the AST class hierarchy across languages enables code reuse, where state and operations defined in new superclasses can be inherited in subclasses. In our example, the type attribute state declared in `Typed` can be inherited by `Expr` and `Abs`.

As another example, consider the changes that were made to the implementation of Java 1.4 in Polyglot [78] in order to handle the evolution of method declarations from Java 1.4 to Java 5. A simplified description of these changes is depicted in Figure 2.7(b). Class `MethodDecl` (method declaration) is a direct subclass of class `ProcedureDecl`, which also represents constructor declarations. Class `ProcedureDecl` is a direct subclass of class `CodeDecl`, which represents nodes that may contain a block of code, including initializers. Finally, class `CodeDecl` is a direct subclass of the base class `Node`, which in Polyglot represents all possible Java language constructs. The Java 5 extension adds annotation support to method declarations in the manner shown in Figure 2.7(a). A new class `AnnotatedElement` is introduced to represent any Java language constructs that may be annotated, including class declarations, method declarations, and instance variable declarations. Since `ProcedureDecl` may now be annotated, a new IS-A relationship between `ProcedureDecl` and `AnnotatedElement` arises. In addition to `CodeDecl`, `ProcedureDecl` now has `AnnotatedElement` as another direct superclass, as shown in the highlighted portion of Figure 2.7(b).

```
@Deprecated
private void foo(...)
  throws AnException { ... }
```

(a) Syntax changes



(b) Class hierarchy changes in the Java 5 extension

Figure 2.7: Evolution of method declaration code from Java 1.4 to Java 5. Java 5 additions are highlighted.

Again, changing the class hierarchy permits code reuse. In the Polyglot evolution example, annotations in Java 5 are type-checked to ensure each annotation occurs at most once. This type-checking code is implemented in `AnnotatedElement` class. Unless the AST class hierarchy is changed, this implementation must be duplicated in the Java 5's implementation of `ProcedureDecl` and of any other kind of nodes that exist in Java 1.4 but can be annotated in Java 5.

2.4.1 Implementing evolvable class hierarchy

To support evolution of the IS-A relationship, we add a *hierarchy factory* interface, as shown in Figure 2.8, to each extension. A hierarchy factory interface has two parts, one to represent node classes themselves, and the other to represent inheritance relationships (arrows). This decomposition respects an observation that the only changes that evolution causes are the movements of arrows. In other words, the existence of AST node classes does not change once they are declared[2], but their superclasses might.

[2]We do not directly support removal of classes, but removals can be simulated by simply not using such classes.

```
public interface JLNodeClassFactory extends NodeClassFactory {
    abstract class JLNodeClass extends NodeClass.Class {
        /** Return direct superclasses of this class. */
        @Override
        public final Set<NodeClass> superclasses(NodeClassFactory cf) {
            return superclasses((JLNodeClassFactory) cf);
        }
        protected abstract Set<NodeClass>
          superclasses(JLNodeClassFactory cf);
        ...
    }
    NodeClass ProcedureDecl = new JLNodeClass("ProcedureDecl") {
        @Override
        protected Set<NodeClass> superclasses(JLNodeClassFactory cf) {
            // Delegate to language-specific definition.
            return cf.ProcedureDecl_super();
        }
    };
    /** direct superclasses of ProcedureDecl in Java 1.4 */
    default Set<NodeClass> ProcedureDecl_super() {
        return Collections.singleton(CodeDecl);
    }
    ...
}
```

```
public interface JL5NodeClassFactory extends JLNodeClassFactory {
    NodeClass AnnotatedElement = new JL5NodeClass("AnnotatedElement") {
        @Override
        protected Set<NodeClass> superclasses(JL5NodeClassFactory cf) {
            return cf.AnnotatedElement_super();
        }
    };
    default Set<NodeClass> AnnotatedElement_super() {
        return Collections.singleton(Node);
    }
    /** direct superclasses of ProcedureDecl in Java 5 */
    @Override
    default Set<NodeClass> ProcedureDecl_super() {
        Set<NodeClass> sup =
          new HashSet<>(JLNodeClassFactory.super.ProcedureDecl_super());
        sup.add(AnnotatedElement);
        return sup;
    }
    ...
}
```

Figure 2.8: Example implementations of hierarchy factory interfaces for node types

Representing declarations of node classes Each node class declaration is a static final member of the hierarchy factory interface[3] and is represented by an instance of the `NodeClass` interface. Like the `Lang` interface, `NodeClass` is shared among AST node types and only needs to be implemented once. Most of the work is already done in `NodeClass.Class` defined in the `NodeClass` interface. Appendix A.2 lists the core implementation of `NodeClass`.

The only difference among `NodeClass` instances is the `superclasses()` method implementation, which simply delegates to an appropriate factory method in the hierarchy factory interface. In this fashion, the hierarchy factory interface can implement such factory methods freely to reflect changes to direct-superclass arrows. For example, the hierarchy factory interface `JLNodeClassFactory` for the Java 1.4 compiler in Figure 2.8 contains a declaration of `ProcedureDecl`, whose implementation of `superclasses` method simply calls the `ProcedureDecl_super` factory method of the given hierarchy factory interface.

The hierarchy factory interface for an extension simply extends one or more existing hierarchy factory interfaces. For instance, the hierarchy factory interface for the Java 5 compiler (`JL5NodeClassFactory`), also in Figure 2.8, extends `JLNodeClassFactory` and adds a declaration of the `AnnotatedElement` node class, which is introduced in Java 5.

Representing direct-superclass relationships As direct-superclass relationships may differ among various languages, they are implemented as instance methods in the hierarchy factory interface so that they may be overridden. These methods are implemented as Java 8's `default` interface methods [42]. Java interfaces containing default methods are similar to traits in Scala [83], but do not use the order of declared superinterfaces to resolve conflicts that may occur when inherited traits provide different method implementations. Instead, like in the original traits proposal [100], the composite interface

[3] A final factory method that returns the `NodeClass` instance would also suffice.

must override the implementation or declare the method abstract. Requiring developers to explicitly specify the desired resolution prevents accidentally preferring one implementation over another, and makes the code more robust against lexical code changes. This property becomes more important when compiler passes are implemented.

Using default methods, an extension that combines multiple existing extensions can inherit all factory method implementations by way of Java interface inheritance but still have an opportunity to override them as necessary. For example, the implementation of method `ProcedureDecl_super` in `JLNodeClassFactory` returns a singleton set indicating that the direct superclass of `ProcedureDecl` in Java 1.4 is `CodeDecl`. Meanwhile, `JL5NodeClassFactory` overrides this method to indicate that the direct superclasses of `ProcedureDecl` in Java 5 now include `AnnotatedElement`, whose direct superclass is `Node`, as implemented in method `AnnotatedElement_super`.

Similar to language definition instances, an instance of a hierarchy factory interface is considered the identity of the hierarchy of AST node types in a given language, so hierarchy factories should be singleton. To enforce this property, we declare a factory method in the `Lang` interface that returns the singleton instance:

```
NodeClassFactory nodeClassFactory();
```

A unique instance of the hierarchy factory interface is then added to the language definition instance for each language. For example, we add the following code to `LCPAIRLang.instance`:

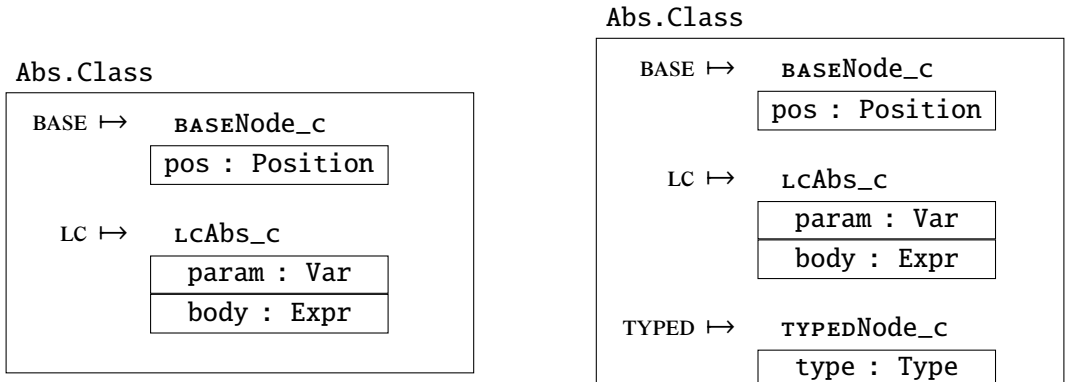
```
protected LCPAIRNodeClassFactory af =
       new LCPAIRNodeClassFactory.Class();
@Override
public NodeClassFactory nodeClassFactory() {
   return af;
}
```

2.5 Composable abstract syntax trees

We now have enough machinery to describe the data structure for AST nodes that can be composed across languages. The new AST design also allows a single AST to represent programs in multiple language variants.

In traditional object-oriented programming languages, as in Java, declaration of states (fields) can only occur in classes. As subclassing in Java is single inheritance, meaning that a class can only be a direct subclass of at most one other class, any implementation of an AST node as a single object is unfriendly for making compilers composable. The problem is that in composable compilers, an AST nodes might have to inherit states from multiple extensions. For example, a typed λ-abstraction node in STLC needs to inherit the fields representing the parameter binding and the body of the abstraction from LC, and also inherit the type-attribute field from TYPED. We need a way to structure AST nodes so that extensions can inherit states from multiple parent languages.

To avoid resorting to multiple implementation inheritance, the actual locations of states are partitioned into a collection of *extension objects* by the language extension that defines them. An AST node is then represented by a *master object* that contains a map from language extensions to corresponding extension objects. Figure 2.9(a) depicts the components of an AST node for LC's untyped λ-abstraction, whose states are partitioned into two extension objects. Extension class `BASENode_C`, associated with the BASE extension, declares a field representing the source position of an AST node, while extension class `LCAbs_C`, associated with the LC extension, declares fields for the parameter and body of a lambda expression. Figure 2.9(b) depicts the components of an AST node for STLC's typed λ-abstraction, extending the AST node for LC with one additional extension object that contains a type-attribute field.



(a) Untyped λ-abstraction in LC

(b) Typed λ-abstraction in STLC

Figure 2.9: Representation of states for λ-abstractions in different languages

The partitioning of states into extension objects maintains subtyping between AST nodes in the absence of subclassing. In our example, a typed `Abs` node can be used anywhere an untyped `Abs` node is expected. This is because the extension objects for a typed `Abs` are a superset of the extension objects for an untyped `Abs`. The additional level of indirection for accessing states makes ASTs composable, simply by adding different extension objects. The indirection also automatically implements certain translations, such as type erasure, simply by changing how the same AST node is viewed. This reduces the compiler's memory footprint.

Despite the convenience of composability and reusability, the new representation of AST nodes introduces one wrinkle to be addressed. The exact type of an AST node is determined by the collection of extension objects that are present in the node. Duck typing appears inevitable. Fortunately, the node class instance and language definition instance, defined previously, come to the rescue. These instances uniquely characterize all the supertypes of a given AST node type, which in turn determine the extension objects to be expected in an AST node of that type. Therefore, in addition to extension objects, each AST node includes a field indicating the type to be expected of an AST

node. A language definition instance is then used to select the appropriate view of the node.

2.5.1 Implementing AST nodes

The implementation of AST nodes starts with the declaration of the `Node` interface that represents master objects, as shown in Figure 2.10. Each `Node` object provides a method `rep()` that returns the *representation type*, the node class instance representing the actual type of the node, a method `get()` that retrieves the extension object associated with a given language definition instance, and a method `put()` that updates these bindings with new extension objects. The abstract class `Class`, which implements the `Node` interface, can be written in a straightforward fashion.

The `check()` method is a utility function that enforces the subtyping relationship on node types when AST nodes are used at run time. The parameter `expected` represents the node type expected within a compilation context, and `lang` is the language definition instance representing the extension in which the AST node `n` is to be viewed. The implementation of `check()` queries the node class hierarchy defined by the given extension and checks that the underlying type of the given node is a subtype of the expected type. When this check fails, the methods fails with an error comparable to the `ClassCastException` in Java, which indicates that the object in question cannot be cast to the desired type.

As an example of using the `check()` method, suppose that in the compiler for STLC, an AST node is to be used in a context that expects a node containing a type attribute. The following invocation enforces the validity of the AST node:

```
Node.check(n, stlcLang.instance, typedNodeClassFactory.Typed);
```

On the other hand, in the compiler for LC, if the context expects a node that represents a λ-abstracton, the following code can be used:

```
Node.check(n, lcLang.instance, lcNodeClassFactory.Abs);
```

```
interface Node {
  /** Return the node type represented by this master object. */
  NodeClass rep();
  /**
   * Return the extension object associated
   * with the given language extension.
   */
  <E extends Ext> E get(Lang lang);
  /**
   * Register a new extension object for the given language extension.
   * Return this master object.
   */
  <N extends Node> N put(Lang lang, Ext extNew);
  /**
   * Check that the node type is a subtype of the expected node type
   * in the given language.
   */
  static void check(Node n, Lang lang, NodeClass expected) {
    if (!n.rep().isSubclass(lang.nodeClassFactory(), expected)) {
      // Can't use this node in the given context.
      // Throw an error comparable to ClassCastException in Java.
      throw new InternalCompilerError("Unexpected node type");
    }
  }
  abstract class Class implements Node {
    /** node type represented by this master object */
    protected NodeClass rep;
    /** map from language definitions to extension objects */
    protected Map<Lang, Ext> map;
    protected Class(NodeClass rep, Map<Lang, Ext> map) {
      this.rep = rep;
      this.map = map;
    }
    @Override
    public final NodeClass rep() { return rep; }
    @Override
    public final <E extends Ext> E get(Lang lang) { ... }
    @Override
    public final <N extends Node> N put(Lang lang, Ext extNew) { ... }
    ...
  }
  ...
}
```

Figure 2.10: Declaration of AST master node interface

Observe that a typed λ-abstraction node as in Figure 2.9(b) will pass both checks above. In particular, the first check passes because in STLC, `Abs` is a subtype of `Typed` in the node class hierarchy.

Using static types as hints The implementation of the master node thus far suggests that the object type of the node does not play a role in determining the actual type of the node; what matters is the representation type. Therefore, in principle, the `Node` interface need not be specialized for different kinds of nodes. For instance, declaring the `Abs` interface that extends `Node` to specifically represent λ-abstractions is unnecessary. In practice, however, programmers rely on static types to discern one kind of object from another. Uses of types in the implementation language—Java in our case—will help reduce potential coding errors.

Once again, we use a middle-ground approach to give programmers as much static typing information as possible without compromising composability. The goal is to define enough types in Java for constructing master objects that provide clues as to what the representation type of a node may be. To achieve this goal, we designate certain types in the node class hierarchy as *principal types* that should be present in the Java class hierarchy. Intuitively, a principal type represents the core components of an AST node that distinguish one node type from another. For example, in STLC, the core components of a typed λ-abstraction are the variable binding and the body of the abstraction, while the type attribute is not part of the core. The most general type that represents these two core components is the untyped λ-abstractions `Abs`, so `Abs` should be considered a principal type.

Although there is no set rule for classifying core components, they are normally newly introduced states when a new AST node type is created, and not states added as part of mixin compositions. Since mixins are a main factor that causes the evolution of the class hierarchy, excluding mixin classes from principal types makes the superclass

(a) Hierarchy for LC

(b) Hierarchy for STLC

Figure 2.11: Node class hierarchies for λ-abstractions in various extensions. Principal types are highlighted.

relationships static. When the supertypes of principal types are fixed, the class hierarchy generated by principal types can be represented in Java. For instance, Figure 2.11 shows the node class hierarchies for LC and STLC, along with their principal types. In STLC, the addition of types as a mixin composition does not affect the hierarchy of principal types. The intermediate class `Typed` is bypassed so that `Node` remains the supertype of `Expr`.

Having determined principal types, we can now implement them as traditional Java interfaces. Figure 2.12 lists the implementation of `Expr` and `Abs` interfaces. For concrete node types, such as `Abs`, a concrete class extending `Node.Class` and implementing `Abs` is written so that a master object of type `Abs` can be constructed. On the other hand, abstract node types, such as `Expr`, do not need a concrete class.

Interfaces and classes for extension objects are implemented in a straightforward way, as in Figure 2.12. An instance of interface `LcAbs` is an extension object that will store the variable binding and the body of a λ-abstraction, to be associated with `LcLang.instance` in a master object of type `Abs`. Accessor methods in extension classes take a master object as an argument, as in line 22, to account for a state that might not be represented simply by a field, but by a combination of other states. The master object argument allows the accessor methods to access states in other extension objects of the same AST

```
1  interface Expr extends Node { ... }
2
3  interface Abs extends Expr {
4    static Expr body(Abs n, Lang lang) {
5      // First, check that the node is of the desired type,
6      // and that this state is accessible in the given language.
7      Node.check(n, lang, LcNodeClassFactory.Abs);
8      // Forward request to extension object.
9      // Pass this master object in case the state
10     // is derived from other states.
11     return n.<LcAbs> get(LcLang.instance).body(n);
12   }
13   class Class extends Node.Class implements Abs {
14     public Class(NodeClass rep, Map<Lang, Ext> map) {
15         super(rep, map);
16     }
17   }
18   ...
19 }
20
21 interface LcAbs extends Ext {
22   Var param(Abs n);
23   Expr body(Abs n);
24   class LcAbs_c implements LcAbs {
25     Var param;
26     Expr body;
27     public LcAbs_c(Var param, Expr body) {
28       this.param = param;
29       this.body = body;
30     }
31     @Override
32     public Var param(Abs n) { return param; }
33     @Override
34     public Expr body(Abs n) { return body; }
35   }
36 }
```

Figure 2.12: Example implementations of AST master interfaces, master classes, extension interfaces, and extension classes

node to compute this derived state. Alternatively, the derivation of such states could be implemented directly within the master class, but we choose to maintain modularity by keeping the traditional compiler implementations within extension classes, separate from the forwarding mechanism in master classes.

2.5.2 Accessing state in AST nodes

Accessing a field for an AST node requires looking up the extension object associated with the appropriate language extension, and then accessing the actual field within that extension object. To encapsulate this mechanism from client code, which should not know the exact representation of AST nodes, we need to provide accessor methods (getters). Traditionally, these accessor methods are implemented as instance methods that belong to each object, but doing so would make composition less scalable. To see why, consider accessing the type attribute for AST nodes in STLC. If the accessor method for type attributes were implemented as an instance method of the `Typed` node type, every node type in STLC would need to extend `Typed` to inherit the implementation. Since node types in STLC have already been declared in LC, extending `Typed` would entail duplicating the entire node type hierarchy, only to add one more accessor method to each node type. To make composition more scalable, an alternative to instance accessor methods is needed.

Instead of instance methods, we use static methods to access fields. Each accessor method takes an AST node, along with a language definition instance to select an appropriate view of the node. In the `Abs` interface in Figure 2.12, the `body()` method shows how to retrieve the body of a λ-abstraction.

The implementation of `body()` works as follows. First, it checks that the given `Abs` node does in fact permit access to the body field, as some extension might disallow this view. We use the `check()` method in the `Node` interface to enforce accessibility. If the test

succeeds, we then retrieve the extension object that contains the body field—in our case, the one associated with the LC language—and invoke the accessor method as appropriate.

The Java type system is not powerful enough to let us avoid type casts in accessor methods. In particular, the value type of the map from language definition instances to extension objects is fixed, requiring a cast to the desired extension type: the desired type is passed as a type argument to `get()` (line 11). Nevertheless, the target types of these casts are clear from the context, so such casts are unlikely to lead to programmer mistakes.

2.5.3 Constructing AST nodes

So far, we have a representation of ASTs that is modular, reusable, and composable, but we still need to be able to construct these ASTs in a modular and extensible way. To enable extensibility, object creation is encapsulated in a factory method. Factory methods in AST node factories are either a master factory method, to create a master object, or an extension factory method, to create extension objects. A master factory method passes the created master object, along with states that need to be stored, to an extension factory method. An extension factory method populates the given master object.

Figure 2.13 shows a partial implementation of AST node factories used by STLC. The primordial `BASENodeFactory` does not contain a master factory method, because the BASE language does not have a concrete node type. It does contain an extension factory method `Node` for storing the position in the source file. Before creating a `Node` extension object, the `Node` extension factory method checks that the master object's map does not already contain an extension object for the BASE language that might otherwise have been created by other calls to this factory method. Extension factory methods in sublanguages eventually invoke the `Node` extension factory method. The `EXPRNodeFactory` extends `BASENodeFactory` and adds another extension factory method to handle all subtypes

```
public interface baseNodeFactory {
  default <N extends Node> N Node(N n, Position pos) {
    // Nothing to do if extension object is already there.
    if (n.containsExtObj(baseLang.instance)) return n;
    // Add extension object for base to master object.
    n.put(baseLang.instance, new baseNode_c(pos));
    return n;
  }
}
```

```
public interface exprNodeFactory extends baseNodeFactory {
  default <N extends Expr> N Expr(N n, Position pos) {
    // Let parent node type handle the rest of extension objects.
    return Node(n, pos);
  }
  ...
}
```

```
public interface lcNodeFactory extends exprNodeFactory {
  default Abs Abs(Position pos, Var param, Expr body) {
    // Create master object.
    Abs n = new Abs_c(lcNodeClassFactory.Abs);
    // Populate the master object with extension objects.
    return Abs(n, pos, param, body);
  }
  default <N extends Abs> N Abs(
      N n, Position pos, Var param, Expr body) {
    // Nothing to do if extension object is already there.
    if (n.containsExtObj(lcLang.instance)) return n;
    // Add extension object for lc to master object.
    n.put(lcLang.instance, new lcAbs_c(param, body));
    // Let parent node type handle the rest of extension objects.
    return Expr(n, pos);
  }
  ...
}
```

Figure 2.13: Example implementations of AST node factories

```
1  public interface STLcNodeFactory
2      extends LcNodeFactory, TYPEDEXPRNodeFactory {
3    @Override
4    default Abs Abs(Position pos, Var param, Expr body) {
5      // Factory method no longer usable in this extension.
6      throw new UnsupportedOperationException();
7    }
8    default Abs TypedAbs(
9        Position pos, Type type, Var param, Expr body) {
10     // Create master object.
11     Abs n = new Abs_c(LcNodeClassFactory.Abs);
12     // Populate the master object with extension objects.
13     return TypedAbs(n, pos, type, param, body);
14   }
15   default <N extends Abs> TypedAbs(
16       N n, Position pos, Type type, Var param, Expr body) {
17     // Let parent node types handle the rest of extension objects.
18     n = TypedExpr(n, pos, type);
19     return Abs(n, pos, param, body);
20   }
21   ...
22 }
```

Figure 2.13: Example implementations of AST node factories (continued)

of Expr, but since Expr itself does not have any state, this method simply let the Node extension factory method deal with extension objects. That is, the EXPR language does not have any extension object associated with it.

In the LC extension, we start seeing concrete node types. The first method in LcNodeFactory is a master factory method for Abs, creating a master object and passing it to the second method, which is an extension factory method. The extension factory method creates an LcAbs_C object to store states pertaining to a λ-abstraction, and the Expr extension factory method handles the rest of the extension objects.

Finally, STLcNodeFactory is a composition of two node factories: LcNodeFactory and TYPEDEXPRNodeFactory (not shown in Figure 2.13). The inherited Abs master factory method is first disabled to prevent untyped Abs nodes from being created. To construct a typed λ-abstraction, the new TypedAbs master factory method creates a master object of the appropriate principal type, Abs in this case. Since a typed λ-abstraction does not have any state associated with STLC itself, the TypedAbs extension factory method can simply let the extension factory methods for parent node types create extension objects. Specifically, TypedExpr extension factory method will populate BASE and TYPED entries, and Abs extension factory method will populate BASE and LC entries. Notice that the Node extension factory method is invoked twice, but only one BASENode_C extension object will be created since we avoid duplicate extension object creation in line 4 of BASENodeFactory. The existence of an extension object for a language means that the extension objects for all its superlanguages have already been created also. This check effectively prunes further invocations of extension factory methods in parent languages.

2.6 Composable dispatch mechanism

The next step after having defined the data structure for abstract syntax trees is to implement operations on ASTs. These implementations should encourage code reuse,

account for the potential evolution of the node class hierarchy, and enable composability across language extensions. One important objective is to resolve the diamond problem that occurs when an AST node inherits two conflicting implementations of an operation.

Let us return to the evolution of the Java compiler from Java 1.4 to Java 5 as implemented in the Polyglot compiler framework (recall Figure 2.7), and consider how type checking for method declarations changes between the two Java versions. The type checker for the Java 1.4 compiler works as follows:

- In `MethodDecl`, the method flags are checked for conformance, e.g., that the `abstract` flag only appears in abstract method declarations. Then, the `ProcedureDecl` implementation is called to do the rest of type checking.
- In `ProcedureDecl`, the types in `throws` clause are checked to be a subtype of Throwable.
- In `Node`, nothing is checked by default; the node is simply returned as is.

The introduction of annotations in Java 5 adds new type-checking obligations:

- In `AnnotatedElement`, annotations are checked to ensure no duplicates.

To integrate this addition into type-checking method declarations, we need to invoke the implementations in `MethodDecl`, `ProcedureDecl`, and `AnnotatedElement` at some point. Depending on the first implementation invoked, certain changes need to be made either to the Java 1.4 implementation or to the code in `AnnotatedElement`:

- If the code in `MethodDecl` is to be invoked first: In `ProcedureDecl`, a call to the type-checking implementation in `AnnotatedElement` must be added so that type checking for annotations occurs. This approach corresponds to asymmetric multiple dispatching in which the node type has higher priority than the language. This dispatch ordering is also used in the J& language [80].

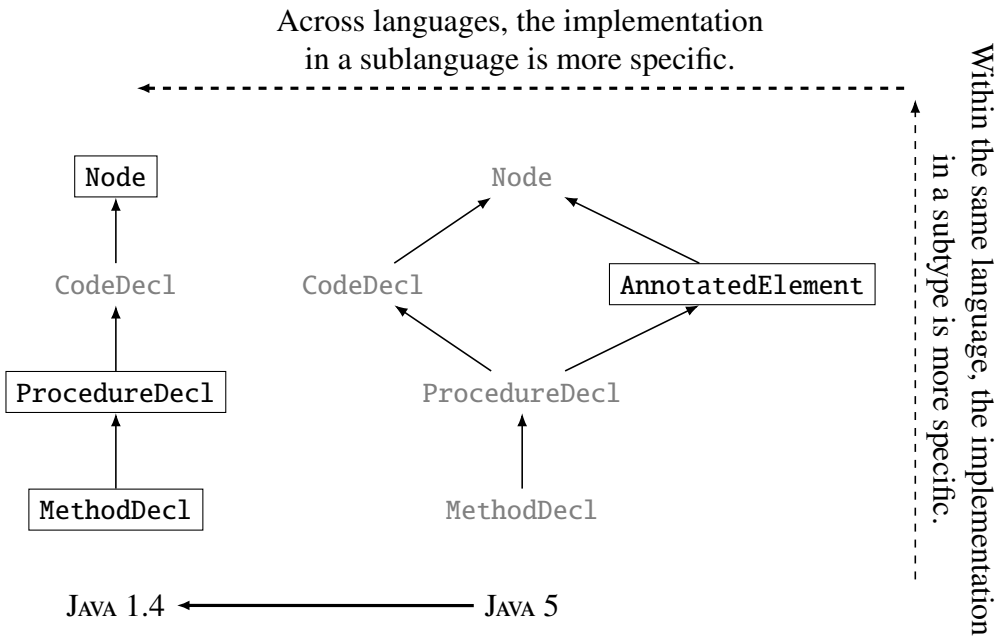


Figure 2.14: Determination of the most specific type-checking implementation for Java 5's method declarations. A boxed node type contains a type-checking implementation, while a grayed-out node type has no implementation and inherits a type-checking implementation from its ancestors.

- If the code in `AnnotatedElement` is to be invoked first: In `AnnotatedElement`, a call to the implementation in `MethodDecl` must be added so that type checking for Java 1.4 still occurs. This approach corresponds to asymmetric multiple dispatching in which the language has higher priority than the node type. That is, even though `AnnotatedElement` is a superclass of `MethodDecl`, the implementation in `AnnotatedElement` is more specific than that in `MethodDecl` because the code for `AnnotatedElement` is implemented in Java 5 and the code for `MethodDecl` is implemented in Java 1.4. Figure 2.14 depicts this dispatch ordering.

Either way, we need to implement multiple dispatching while taking the evolution of class hierarchy into account. Although choosing between the two dispatch orderings could be a matter of taste, we argue that the latter approach—where languages have higher priority—better supports extensibility. In both approaches, new code is added for

`AnnotatedElement`, but the first dispatch ordering also requires modifying existing code. This modification is only possible if the source code for the existing implementation is available. Plus, the resulting code is now specialized to the Java 5 compiler and would be incompatible with other, orthogonal extensions. In contrast, the second dispatch ordering only requires new code to be added in one place, without specializing existing code.

2.6.1 Implementing dispatch mechanism

Method dispatch occurs in two scenarios. Dispatch happens when client code requests that an operation be performed, e.g., type checking of method declarations. In Java, this is simply a method invocation. Dispatch also happens within a method implementation. In Java, this is the `super` method call to the superclass implementation, which is uniquely defined thanks to single inheritance. Unlike in Java, however, our desire to compose compilers means multiple inheritance may yield multiple superclass implementations, resulting in an ambiguity. Multiple inheritance also occurs at the language level when extensions are composed. Our multiple dispatch mechanism handles multiple inheritance in both dispatch scenarios; we describe each of them in turn.

Determining the most specific implementation Suppose an operation on a given node type T_0 in a given language L_0, denoted (L_0, T_0), is requested. If (L_0, T_0) itself implements (or overrides) this operation, then we have found the most specific implementation. In the general case, this operation might not be implemented in (L_0, T_0), but instead inherited from a supertype of T_0, from a superlanguage of L_0, or both, so we need to explore the hierarchy of node types and possibly the hierarchy of languages to find the desired implementation.

We introduce some notation before defining the most specific implementation. For any languages L_1 and L_2, let $L_1 <: L_2$ denote that L_1 is a sublanguage of L_2. For any node types T_1 and T_2, let $T_1 <: T_2$ denote that T_1 is a subtype of T_2, e.g., `MethodDecl` $<:$ `Node`.

Next, we define a partial ordering $<:$ on (L, T)-pairs corresponding to our dispatch ordering as follows:

- If $L_1 <: L_2$, then $(L_1, T_1) <: (L_2, T_2)$ for any T_1 and T_2.
- For a fixed L, if $T_1 <: T_2$ in L, then $(L, T_1) <: (L, T_2)$.

Then, given a set P of (L, T)-pairs, call $(L', T') \in P$ *maximally specific* if $(L', T') <: (L, T)$ for any $(L, T) \in P$. Notice that, due to multiple inheritance, P could contain multiple maximally specific pairs, all of which are pairwise incomparable. Finally, let $M(P) = \{(L, T) \in P \mid (L, T) \text{ is maximally specific in } P\}$.

Now we are ready to define the most specific implementation for (L_0, T_0). Let predicate $I(L, T)$ denote that (L, T) implements or overrides the desired operation. Given a set $P_I(L_0, T_0) = \{(L, T) \mid I(L, T) \wedge (L_0, T_0) <: (L, T) \wedge T_0 <: T\}$ of all applicable implementations for (L_0, T_0), consider $M(P_I(L_0, T_0))$:

- If $M(P_I(L_0, T_0)) = \emptyset$, then there is no applicable implementation for (L_0, T_0): a *message not understood* error.
- If $M(P_I(L_0, T_0))$ is a singleton, then its unique element has the most specific implementation for (L_0, T_0).
- If $|M(P_I(L_0, T_0))| > 1$, there are multiple applicable implementations for (L_0, T_0): a *message ambiguous* error.

To determine $I(L, T)$, we define an *operator factory* containing *operator factory methods*, each of which returns an implementation for each node type within a language. By default, each node type inherits the operation, so the operator factory methods return `null`, as shown in the `JLOperatorFactory` interface in Figure 2.15. The default operator factory for an extension simply extends one or more existing operator factory interfaces in the same way as for the hierarchy factory interfaces, as shown in `JL5OperatorFactory`. For a language composed of multiple immediate superlanguages, multiple inheritance

```
public interface JLOperatorFactory<O extends Operator> {
    default O Node() { return null; }
    default O CodeDecl() { return null; }
    default O ProcedureDecl() { return null; }
    default O MethodDecl() { return null; }
    ...
}
```

```
public interface JL5OperatorFactory<O extends Operator>
        extends JLOperatorFactory<O> {
    default O AnnotatedElement() { return null; }
    ...
}
```

Figure 2.15: Example declarations of operator factory classes

for interfaces takes care of collecting the operator factory methods for all the node types in the composed language.

To define an operator, we extend the `Operator` interface and add the desired method declaration, as in `TypeCheckOperator` in Figure 2.16(a). To implement an operator, we implement the operator factory interface and override an operator factory method for each node type of interest, as shown in `JLTypeCheckFactory` in Figure 2.16(b). An overridden operator factory method returns a Java's lambda expression [42], which implements the declared operator.

The dispatcher method, called by client code to invoke the operator on (L_0, T_0), computes $M(P_I(L_0, T_0))$. To avoid exploring all operator factories in all extensions, the dispatcher first determines $I(L_0, T_0)$ by querying the operator factory method for T_0 in L_0's operator factory. If this is `null`, the dispatcher queries operator factory methods for superclasses of T_0, as defined by L_0's node class hierarchy factory, in the same operator factory. If there is still no applicable implementation, the dispatcher then explores operator factories for $L :> L_0$. Appendix A.3 describes the implementation of exploration code and data structure for quickly computing $M(P_I(L_0, T_0))$. This exploration code is shared by all operators and only needs to be implemented once. Furthermore, the dispatch

```
public interface TypeCheckOperator extends Operator {
    Node typeCheck(Node n, TypeChecker tc);
}
```

(a) Operator declaration

```
public class JLTypeCheckFactory implements
  JLOperatorFactory<TypeCheckOperator> {
    public static final instance = new JLTypeCheckFactory();
    @Override
    public TypeCheckOperator Node() {
        return (n, tc) -> n; // Do nothing.
    }
    @Override
    public TypeCheckOperator MethodDecl() {
        return (n_, tc) -> {
            MethodDecl n = (MethodDecl) n_;
            ...
        };
    }
    ...
}
```

```
public class JL5TypeCheckFactory implements
  JL5OperatorFactory<TypeCheckOperator> {
    public static final instance = new JL5TypeCheckFactory();
    @Override
    public TypeCheckOperator AnnotatedElement() {
        return (n, tc) -> {
            ...
            // Do Java 1.4 type checking.
            return Typed.typeCheckSuper(n, ...);
        };
    }
    ...
}
```

(b) Factory method implementations

Figure 2.16: Example implementations of type-checking operator

ordering can be changed by simply altering the definition of <: and the exploration code. In other words, our design pattern makes both symmetric and asymmetric multiple dispatch implementable with similar effort.

Invoking the next most specific implementation To invoke the next most specific implementation according to <:, we need to determine the superclass S associated with the call site. In Java, S is the superclass of the enclosing class of the method whose body contains the `super` call. By contrast, operators in our design pattern are no longer implemented in the traditional object-oriented style, so we have to specify the enclosing class explicitly. Moreover, since the superclasses of a given enclosing class may differ depending on a particular node class hierarchy, we also have to specify the current language explicitly. For example, line 9 of `JL5TypeCheckFactory` in Figure 2.16(b) has a super call to handle the rest of type-checking an `AnnotatedElement` node in Java 5. In full, this call is as follows:

```
Typed.typeCheckSuper(n, tc, JL5Lang.instance,
                     JL5NodeClassFactory.AnnotatedElement);
```

Appendix A.3 contains the listing of `typeCheckSuper`, which uses the same exploration code mentioned above to find the most specific superclass implementation.

Multiple implementation inheritance could result in ambiguous superclass implementations. To resolve conflicts, the call site can specify the pair (L, T) for the desired superclass implementation. For instance, the following code invokes the most specific type-checking implementation applicable to (JL5, `CodeDecl`):

```
Typed.typeCheck(n, tc, JL5Lang.instance, JLNodeClassFactory.CodeDecl);
```

The implementation of this method is also listed in Appendix A.3.

Operator factory factories The last step in implementing operations on ASTs is to inform the compiler for a specific language about these new implementations. One way to do so is to add a method in the language definition interface that returns the operator

factory for each operation, or `null` if that operation is not overridden at all for any AST node type in the language. This approach, however, can be difficult to maintain, as the number of methods to be added to the language definition interface is proportional to the number of operators that have changed. To make language definition interfaces more manageable, we introduce *operator factory factories* as a level of indirection. An operator factory factory interface is a Java interface that declares factory methods, one per operation. Each factory method returns the operator factory for the operation that has been overridden in a given language.

Figure 2.17 lists some examples of operator factory factory interfaces and classes. Suppose that the BASE language declares an operator that pretty-prints the AST. The operator factory factory interface for the BASE language (`baseOperatorFactoryFactory`) defines a corresponding method that returns `null` by default to indicate the absence of overriding pretty-printing. However, the BASE language itself does override this operation for `Node`, to implement printing nothing. This override requires a concrete implementation of the operator factory factory interface (`baseOperatorFactoryFactory_c`), which overrides the `prettyPrint()` factory method to return the appropriate operator factory.

A language extension that introduces a new operation extends an existing operator factory factory interface with an appropriate factory method. For instance, TYPED declares an operation for type-checking AST nodes, so `typedOperatorFactoryFactory` adds the `typeCheck()` factory method. For a language composed of multiple parent languages, multiple inheritance for interfaces once again takes care of combining all available operations into the composed operator factory factory interface, as in `stlcOperatorFactoryFactory`. In this way, a concrete implementation of the composed interface can override factory methods for operators that are overridden in the composed language. For example, STLC adds new type-checking implementations for the λ-calculus

```
public interface baseOperatorFactoryFactory
       extends OperatorFactoryFactory {
    default OperatorFactory<PrettyPrintOperator> prettyPrint() {
       return null;
    }
    ...
}
public class baseOperatorFactoryFactory_c
       implements baseOperatorFactoryFactory {
    @Override
    public basePrettyPrintFactory prettyPrint() {
       return basePrettyPrintFactory.instance;
    }
    ...
}
```

```
public interface typedOperatorFactoryFactory
       extends baseOperatorFactoryFactory {
    default OperatorFactory<TypeCheckOperator> typeCheck() {
       return null;
    }
}
public class typedOperatorFactoryFactory_c
       implements typedOperatorFactoryFactory {
    @Override
    public typedTypeCheckFactory typeCheck() {
       return typedTypeCheckFactory.instance;
    }
}
```

```
public interface stlcOperatorFactoryFactory
       extends lcOperatorFactoryFactory,
              typedexprOperatorFactoryFactory { }
public class stlcOperatorFactoryFactory_c
       implements stlcOperatorFactoryFactory {
    @Override
    public stlcTypeCheckFactory typeCheck() {
       return stlcTypeCheckFactory.instance;
    }
}
```

Figure 2.17: Example implementations of operator factory factories

constructs. The stLcOperatorFactoryFactory_c class overrides the typeCheck() factory method to return the operator factory containing the additional type-checking code.

Each language that overrides any operator at all has a corresponding operator factory factory instance. Like language definition instance and node type hierarchy instance, operator factory factory instance is the part of the identity of the language, hence singleton. Like with the nodeClassFactory() method, we declare another factory method in the Lang interface to enforce the singleton property:

```
default OperatorFactoryFactory opFactoryFactory() { return null; }
```

The method returns null by default to indicate no overriding. A language that overrides some operations overrides this method to return a unique instance of the operator factory factory interface. For example, the following code is added to stLcLang.instance:

```
protected stLcOperatorFactoryFactory off =
       new stLcOperatorFactoryFactory_c();
@Override
public OperatorFactoryFactory opFactoryFactory() {
   return off;
}
```

2.7 Composable translations

In Section 2.6.1, we described an implementation of multiple dispatch that enables composition of language extensions. There, the dispatcher follows the <: relation, walking up the node-type and language hierarchies to find the most specific implementation. An extension can simply reuse operator implementations from its base language. Translations, on the other hand, are a bit more complicated to implement because they are defined in terms of source–target language pairs. We can no longer rely on the language hierarchy for dispatching. A slightly different design pattern is needed to ensure that

translations are composable, and that one language can define multiple translations to a number of target languages. Despite this difference at the language level, we can still treat a translation as another set of operations at the AST level. We declare a new operator `Transformer`, extending the previously mentioned `Operator` interface:

```
public interface Transformer extends Operator {
    /**
     * Translate the given node n using translation driver t.
     * Return either a translated node in the target language,
     * or null if n's children should be translated
     * but there is no change to n itself after translating children.
     */
    Node translate(Node n, Translator t);
}
```

A translation driver provides the AST node factory for the target language, and implements a method to facilitate the translation of children within the AST.

For any translation, some constructs may be translated, while others can remain if the target language also recognizes them. For example, when translating LCPAIR into LC, pair constructors and projections must be rewritten, while λ-abstractions and applications can be preserved, provided their children are translated as appropriate. First, we implement the most trivial translation, the identity translation, which serves as a basis for any other translation. In the BASE language, a transformer factory provides the identity translation for all node types:

```
public interface BASETransformerFactory
        extends BASEOperatorFactory<Transformer> {
    @Override
    default Transformer Node() {
        // No translation defined; translate children as needed.
```

```
1  public interface LCPAIRTransformerFactory_LC
2        extends LCPAIROperatorFactory<Transformer>,
3              BASETransformerFactory {
4    @Override default Transformer Pair() {
5       return (n_, t) -> {
6          Pair n = (Pair) n_;
7          LCNodeFactory nf = t.tgtNodeFactory();
8          Position pos = n.position();
9          // [[(e1, e2)]] ≜ λf.f [[e1]] [[e2]]
10         Expr fst = t.translate(n.fst());
11         Expr snd = t.translate(n.snd());
12         return nf.Abs(pos, nf.Var(...), nf.App(...));
13      };
14   }
15   @Override default Transformer Fst() { ... }
16   @Override default Transformer Snd() { ... }
17 }
```

Figure 2.18: Example implementation of a transformer factory

```
      return (n, t) -> null;
    }
  }
```

Other, more concrete translations extend BASETransformerFactory to include translator method implementations for appropriate node types. An example of a transformer factory that translates LCPAIR to LC is shown in Figure 2.18.

The main difference between normal operator factories and transformer factories is that operator factories do not inherit other operator factories, while transformer factories inherit other transformer factories that contain desired translator methods. In this way, we need not rely on the language hierarchy for dispatching, since we have all the translation code in one factory. To demonstrate composability of transformer factories, suppose further that we have implemented a translation from LCSUM, the untyped λ-calculus with sums, to LC by translating away injections and case expressions using lambda expressions; have implemented LCPAIRSUM, a composition of LCPAIR and LCSUM; and would like to add three translations: (1) from LCPAIRSUM to LCSUM, rewriting pairs,

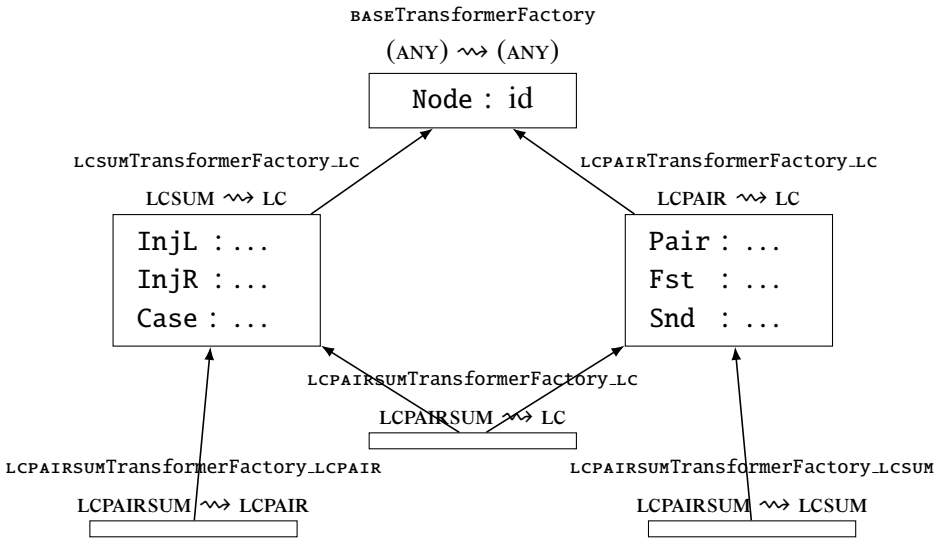


Figure 2.19: Compositions of transformer factory interfaces

(2) from LCPAIRSUM to LCPAIR, rewriting sums, and (3) from LCPAIRSUM to LC, rewriting both pairs and sums. Figure 2.19 shows how existing translations can be reused in these three translations. Since we already have a translation for pairs from LCPAIR to LC (denoted LCPAIR ⇝ LC), we can reuse this translation for LCPAIRSUM ⇝ LCSUM by simply extending LCPAIRTransformerFactory_LC, without adding any more implementations, and similarly for LCPAIRSUM ⇝ LCPAIR. Finally, multiple inheritance on interfaces allows us to compose both translations on pairs and sums, yielding a translation for LCPAIRSUM ⇝ LC.

2.8 Experience and evaluation

Our implementation and compositions of language extensions aim to answer the following questions:

- Are our design patterns effective for implementing extensible compilers and translations between languages?
- Are our design patterns effective for composing compilers and translations?

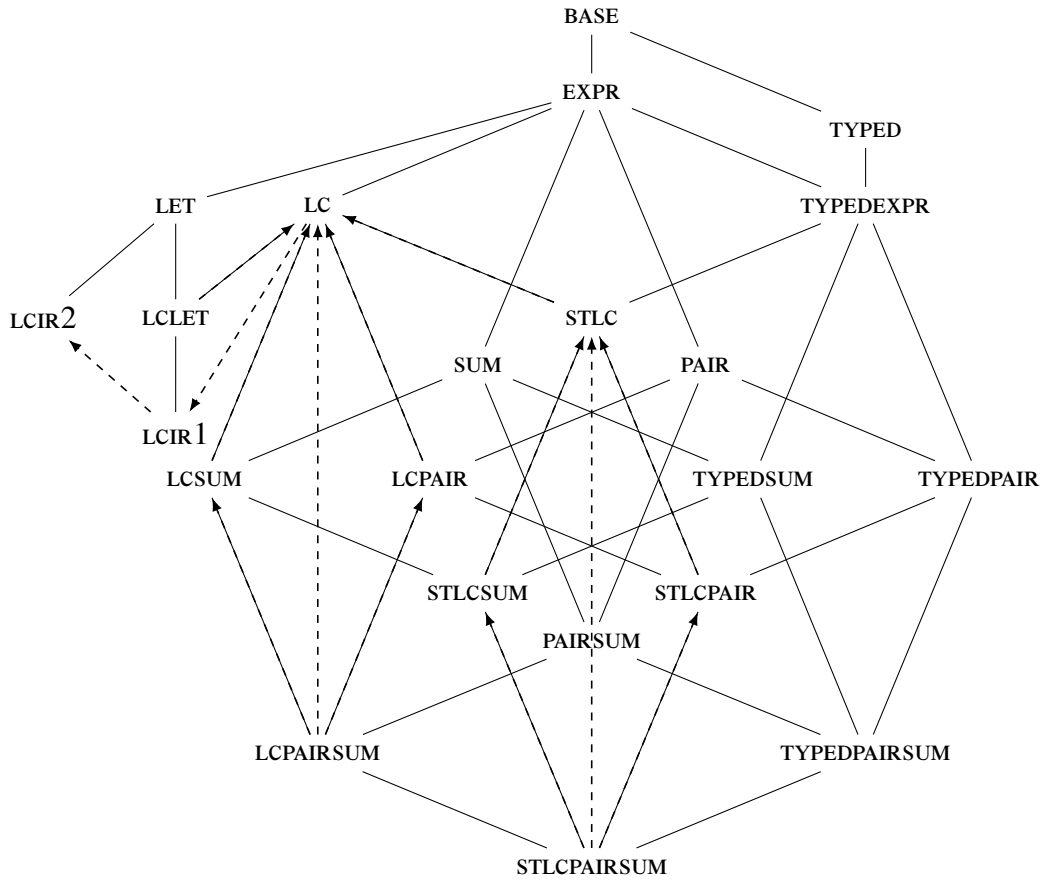


Figure 2.20: Portion of the language hierarchy and translation ordering for implemented extensions. A solid edge means the lower language extends the upper language. A directed edge indicates an available translation from one language to another. These edges may be superimposed to indicate that a language both extends from and translates to another.

- Do our design patterns scale to deeply layered language extensions?
- Are our design patterns applicable to compilers for industrial-scale languages?
- What are the primary sources of boilerplate code that could be reduced by additional language constructs?

2.8.1 Language extensions and their compositions

To evaluate the extensibility and composability of compilers based on our design patterns, we implemented compilers for 35 language extensions. Figure 2.20 illustrates the relationships on a subset of extensions, showing that among the languages implemented are every possible combination of four independent extensions: LC, PAIR, SUM, and TYPEDEXPR. The descriptions of the 35 extensions follow:

- BASE is the root of the language hierarchy that includes `Node` class to represent all possible language constructs, and `Id` type to represent string identifiers.
- EXPR is the abstract language that contains expressions. Untyped variables, and arithmetic and logical expressions are also implemented here.
- TYPED is the mixin that adds typed constructs. An AST node of type `Typed` has a type attribute. Type-checking operators are declared here.
- TYPEDEXPR is the composition of EXPR and TYPED, adding type-checking to arithmetic expressions. Additionally, type nodes are introduced so that variables can be annotated with types such as `int` and `bool`. For instance, the operands of a binary addition are checked to have an integer type.
- LC is the untyped λ-calculus, extending EXPR with abstractions and applications.
- STLC is the simply-typed λ-calculus, composing LC and TYPEDEXPR. Type-checking for abstractions and applications is implemented here.
- PAIR extends EXPR with pair constructors and projections.
- TYPEDPAIR is the composition of PAIR and TYPEDEXPR, adding type-checking to pair constructors and projections.
- LCPAIR is the composition of PAIR and LC.

- STLCPAIR is the composition of LCPAIR, STLC, and TYPEDPAIR. No additional type-checking code is required since every construct needing type-checking has already been handled in either STLC or TYPEDPAIR.
- SUM extends EXPR with injections and case expressions.
- TYPEDSUM is the composition of SUM and TYPEDEXPR, adding type-checking to injections and case expressions.
- LCSUM is the composition of SUM and LC.
- STLCSUM is the composition of LCSUM, STLC, and TYPEDSUM. No additional type-checking code is required.
- PAIRSUM is the composition of PAIR and SUM.
- TYPEDPAIRSUM is the composition of PAIRSUM, TYPEDPAIR, and TYPEDSUM.
- LCPAIRSUM is the composition of PAIRSUM, LCPAIR, and LCSUM.
- STLCPAIRSUM is the composition of LCPAIRSUM STLCPAIR, STLCSUM, and TYPEDPAIRSUM.
- LET adds the let expressions.
- TYPEDLET is the composition of LET and TYPEDEXPR, adding type-checking to let expressions.
- LCLET is the composition of LET and LC.
- STLCLET is the composition of LCLET, STLC, and TYPEDLET. No additional type-checking code is required.
- STMT is the abstract language that contains statements, extending BASE. Statement blocks and empty statements are also implemented here.
- TYPEDSTMT is the composition of STMT and TYPED, adding type-checking code to statement blocks. (Empty statements need not be type-checked.)

- IMP is the imperative language, extending both EXPR and STMT, and contains statements such as variable declarations, assignments, and loops.
- TYPEDIMP is the composition of IMP, TYPEDSTMT, and TYPEDEXPR, adding type-checking code to new constructs declared in IMP.
- LCIR1 and LCIR2 are intermediate languages for translating LC into an assembly-like target language. The LCIR1 language contains *administrative lambdas* [38] that result from CPS-converting λ-expressions. Expressions in LCIR1 are then closure-converted to ones in LCIR2 that closely resembles an assembly language.
- POLYTYPE adds type variables for use in type inference, extending TYPEDEXPR. Infrastructure for performing type inference is implemented here.
- POLYSTLC is the composition of STLC and POLYTYPE, adding type inference to simply-typed λ-calculus so that type annotations can be omitted.
- POLYSTLCLET is the composition of STLCLET and POLYSTLC, adding type inference to let expressions so that variable bindings do not require annotated types.
- LETPOLY extends POLYSTLCLET to implement let-polymorphism. Universal types are introduced here for use in inferring and checking types on let-polymorphism expressions.
- COND adds the conditional expressions to EXPR.
- TYPEDCOND is the composition of COND and TYPEDEXPR, adding type-checking code to conditional expressions.
- LETREC implements recursive let expressions, composing POLYSTLC and TYPEDCOND. The let-rec construct is introduced here along with code for type checking the construct and code for inferencing the type of recursive functions.

The most sophisticated languages implemented are LETREC and LETPOLY, which provide recursive function definitions and ML-style let-polymorphism, respectively. Thus, the

languages implemented in this highly decomposed fashion come close to reaching the richness of core ML [73].

2.8.2 Extensibility and composability of compilers

Table 2.1 shows the number of Java source files that need to be created to implement the validation phase of the compiler for each extension, along with the number of lines of code, excluding comments and blank lines, to implement various components. Other components include type objects for extensions of TYPED, and utility functions such as the free-variable finder for lambda expressions.

The "skeleton" is code providing shared infrastructure. The 182 lines of skeleton language definition provide most of the functionality for adding a new language extension to the hierarchy. On average, 47 lines of code are needed to implement a language definition class, which declares parent languages, creates factory objects for a new extensions, and refines return types of various methods. Specifications of supported target languages, included in the reported lines of code, significantly influence the length of a language definition. For instance, STLCPAIRSUM has three translations that require 15 lines of specifications, while LC's only translation requires just five lines.

Additions of syntactic constructs and associated operations require a modest amount of code for AST definitions and operators that is proportional to the number of new constructs and operations. For example, LC's two new constructs require an average of 103 lines of code per construct, while EXPR adds 18 new constructs, requiring an average of 39 lines each. These averages differ depending on the similarity among added constructs. In our example, 13 kinds of binary expressions share code for traversing children and pretty-printing the node.

Compositions involving mixin languages introduce interactions between mixin states and existing constructs. For instance, adding types to EXPR requires that every expression

Extension	# files	Total	Language definition	Hierarchy factory	Node factory	AST definitions	Operators	Other components
(skeleton)	62	3758	182	79	N/A	N/A	67	3430
BASE	13	586	38	47	49	335	117	0
EXPR	30	1284	41	315	225	482	221	0
TYPED	30	598	57	55	11	121	111	243
TYPEDEXPR	30	902	60	61	81	128	356	216
LC	13	446	50	56	56	93	113	78
STLC	18	332	54	7	7	0	116	148
PAIR	12	418	41	82	65	110	120	0
TYPEDPAIR	15	306	55	8	8	0	96	139
LCPAIR	5	71	38	7	7	0	19	0
STLCPAIR	6	102	53	9	9	0	23	8
SUM	12	479	41	82	70	145	141	0
TYPEDSUM	22	696	63	44	84	95	272	138
LCSUM	5	71	38	7	7	0	19	0
STLCSUM	6	101	52	9	9	0	23	8
PAIRSUM	5	62	29	7	7	0	19	0
TYPEDPAIRSUM	6	92	43	9	9	0	23	8
LCPAIRSUM	5	96	55	9	9	0	23	0
STLCPAIRSUM	6	119	65	10	10	0	25	9
LET	9	266	41	43	40	53	89	0
TYPEDLET	8	145	47	7	8	0	77	6
LCLET	6	96	38	7	7	0	19	25
STLCLET	6	101	52	9	9	0	23	8
STMT	11	298	41	68	48	56	85	0
TYPEDSTMT	10	145	44	14	6	5	72	4
IMP	13	644	42	96	102	209	195	0
TYPEDIMP	8	196	52	9	9	0	118	8
LCIR1	13	462	48	69	65	81	155	44
LCIR2	13	606	41	95	102	176	192	0
POLYTYPE	28	718	47	6	11	0	48	606
POLYSTLC	6	90	41	7	8	0	19	15
POLYSTLCLET	6	93	50	8	8	0	19	8
LETPOLY	20	469	52	6	6	0	92	313
COND	9	271	42	43	42	52	92	0
TYPEDCOND	8	138	47	8	8	0	69	6
LETREC	15	512	59	58	61	88	207	39

Table 2.1: Code statistics for implementations of 35 language extensions

Source	Target	Translator
LCIR1	LCIR2	353
LC	LCIR1	182
LCLET	LC	25
LCPAIR	LC	72
LCSUM	LC	69
STLC	LC	21
LCPAIRSUM	LCPAIR	7
LCPAIRSUM	LCSUM	7
LCPAIRSUM	LC	9
STLCLET	STLC	7
STLCPAIR	STLC	7
STLCSUM	STLC	7
STLCPAIRSUM	STLCPAIR	8
STLCPAIRSUM	STLCSUM	8
STLCPAIRSUM	STLC	8

Table 2.2: Lines of code for implementing translations between extensions.

be type-checked, incurring 484 lines of code that implement type checking in extension classes (24 lines per construct on average). Composing independent language extensions is trivial, requiring no additional AST classes whatsoever, even if these extensions share a mixin language. For example, TYPEDPAIRSUM extends TYPEDPAIR and TYPEDSUM, both of which extend TYPEDEXPR. No new AST classes are needed because type checking is already implemented fully in the parent extensions. These results indicate that our design patterns make ASTs highly composable.

2.8.3 Extensibility and composability of translations

Table 2.2 reports the line counts of Java code, excluding comments and blank lines, needed to implement the translation phase of the compiler between certain pairs of extensions. The amount of rewriter code is proportional to the number and complexity of constructs that need rewriting. For example, an average of 39 lines is needed to rewrite

each of the nine constructs in LCIR1 into LCIR2, while 24 lines are needed on average to rewrite one of the three constructs in LCPAIR into LC.

Translations can be reused and composed easily for deeper extensions than the ones they were developed for. As an example, to translate LCPAIRSUM into LCSUM, the translation from LCPAIR to LC is reused completely; the only glue code needed is the additional translator factory. Meanwhile, the translation from LCPAIRSUM to LC is the composition of the translations from LCPAIR to LC and from LCSUM to LC, using multiple interface inheritance on translator factories. Again, only one more factory specific to this translation is needed. The code size shows that highly reusable and composable translations are achievable with our design patterns.

2.8.4 Scalability for deeply layered extensions

Our results indicate that the extension depth, i.e., the distance in the language hierarchy from BASE to the extension of interest, does not affect the amount of code that implements a new extension. As an example, STLCPAIRSUM, a depth-5 extension, requires nearly the same amount of code as LCLET, a depth-3 extension. Also, the amount of code for translations between extensions is not affected by extension depth. These observations suggest that our design patterns enable scalable composition of compiler extensions.

2.8.5 Application to larger-scale compilers

To demonstrate adaptability of our design pattern to real-world implementations of compilers, we ported the Polyglot extensible compiler framework [78] and compared the number of lines of code, excluding comments and blank lines, between the two implementations. Table 2.3 lists these numbers for Java extensions within Polyglot, along with the percentage change in the number of lines between the original and ported

Language	Polyglot # files	Polyglot # lines	Polyglot$_{\oplus}$ # files	Polyglot$_{\oplus}$ # lines	% line change
Java 1.4	499	52060	461	52074	0.03
Java 5	201	19999	196	20655	3.28
Java 7	34	2076	40	2218	6.84

Table 2.3: Comparisons of number of source files and lines of code between the original Polyglot extensible compiler framework and the implementation using our design pattern (denoted Polyglot$_{\oplus}$)

versions. The results show that, in terms of the amount of code, not much change is needed to make compilers composable.

2.8.6 Language constructs for mainstream languages

A primary goal of our design patterns is to encourage adoption. One way to accomplish this goal is to ensure that programmers write only as much code as necessary. Nevertheless, some code duplication and boilerplate are still required to implement our composable compilers. We now identify language features unavailable in Java 8 that would have minimized such unnecessary code.

Protected default methods To enforce encapsulation in AST node factory interfaces, extension factory methods are only used by master factory methods and should be protected to prevent client code from invoking them by accident. Although default methods in Java 8 interfaces can only be public, future support for protected default methods looks promising, as private default methods will be permitted in Java 9 [1].

Multiple dispatch Most type casts in our implementation of extensions are required because the formal types in method declarations must be fixed for overriding to work properly. Covariant formal types can eliminate these casts. One way to support covariant formal types is multimethods [17], where actual argument types, in addition to the receiver type, play a role in determining the correct method implementation to be invoked.

MultiJava [24, 72] extends Java with multimethods by allowing a formal type to be appended with a desired type refinement, e.g., `Node@Pair` to indicate that the associated method implementation is only applicable when the argument is of type `Pair`.

Higher-order polymorphism As an alternative to multiple dispatch, higher-order polymorphism can eliminate node type casts appearing in operator implementations. The `Operator` interface could be parameterized with the operator's input node type, but without higher-order polymorphism, the return types of all operator factory methods need to be the same, e.g., `TypeCheckOperator<Node>`. With higher-order polymorphism, however, the type arguments of these return types could be tailored to specific node types, eliminating casts. A decidable type system for higher-order polymorphic multimethods has been proposed [19].

Pattern matching Some rewriting is applicable only to a specific form of the AST node and its children. Additional checks on the children are needed on top of method dispatch based on the type of the parent AST node. These checks are similar to deep pattern matching on the type of children nodes. Some extensions in our implementation contain basic pattern-matching support to make such rewriting more convenient to write. Native support for pattern matching, as in Scala [84], F# [111], or JMatch [49], would eliminate the need for this boilerplate code.

2.9 Related work

Extensibility of compilers is an extension of the expression problem [122], which addresses the tension between adding new data variants and adding new operations. Composability of compilers can be formulated as the independent extensibility criterion [82], or the expression families problem [85], a larger-scale version of the expression problem that addresses the reusability and composability of families of data variants and associated

operations. One requirement of a solution to the expression problem is static type safety: no casts should be used. We choose to trade this requirement for multiple-view ASTs, as casts in our framework are either checked or appear where the target type is clear from context.

Erdweg et al. [34] define terminology concerning programming language evolution and provide a survey on related technologies. In their terminology, our framework supports language extension (implementation of the base language can be reused unchanged to implement extension) and language unification (implementation of existing languages can be reused unchanged by adding glue code only), but not self-extension (embedding domain-specific language into host language), as Java remains our host implementation language.

Class hierarchy mutation can be traced back to approaches in metamorphic programming [67, 108]. In those approaches, the run-time type of an object can change depending on its state, allowing different implementations of the same method to be used. Instead of relying on conditional statements to select a desired method behavior based on the object's state value, dynamic class hierarchy mutation redirects pointers in the object's virtual function table to a desired method implementation specialized to a particular state value, thereby improving performance. Unlike metamorphic programming, which focuses on changing the run-time types to exhibit different behaviors at the object level, our approach in making the node class hierarchy evolvable focuses on changing the relationship between AST node types to exhibit different behaviors at the programming-language extension level. Once the extension is fixed, the type of an AST node created in that extension and the relationship between node types remains unchanged.

To address the multiple implementation inheritance problem, new host languages such as `gbeta` [35], Scala [83], and J& [80] were designed. Scala provides traits that may contain implementations, but not states; ambiguities are resolved by textual ordering of

trait names in a subclass declaration. Meanwhile, `gbeta` assumes that such ambiguity does not arise for composition to succeed. Our design patterns do not make these assumptions, yet still handle multiple inheritance gracefully.

The Grace object-based language [14, 15] builds on an observation that fully qualified class names used in object creation expressions such as `new` in Java are inimical to extensibility. Like in our approach, Grace uses factory methods to defer determination of the actual class to be instantiated until run time. Grace supports independent extensibility, but only along orthogonal dimensions.

Covariant type refinement [124] may be used to refine the possible types of fields, thereby widening the available operations on AST nodes. This approach is similar to our interface subtyping for AST node types and does support independent extensibility, but requires more boilerplate code than our approach and does not address the object construction of mixin compositions.

The Turnstile metalanguage [22] uses Racket macros to implement composable type systems. To improve performance, Turnstile simultaneously type-checks and rewrites source programs into a target language, as opposed to running these two passes separately as in the nanopass approach. As usual with approaches using macros, the implemented translations tightly couple source languages to a limited number of target languages. In this case, the target languages can only be Racket or extensions thereof, restricting the resulting executables to the platforms targeted by the Racket compiler. Our design patterns do not have this limitation, allowing translations to any target language of choice.

One advantage of traditional, monolithic compilers is better performance, as multiple, small compiler passes require repetitive traversals of the AST, each with only slight changes. For larger programs, larger ASTs may not fit in memory, incurring more cache miss across multiple traversals. Design patterns for the Miniphases approach [93] in the Dotty compiler for Scala reconcile these two camps, allowing compiler passes to be

implemented in a modular way, but providing infrastructure for noninterfering passes to be run in a single AST traversal. Implemented in Scala, Miniphases design patterns use sealed classes that sacrifice code inheritance and extensibility of AST node types for the convenience of pattern matching. Our design patterns would be complementary to these performance improvements, and vice versa.

Although this paper has focused on composability of ASTs, grammar extension and composition remain important for most programming languages. Previous work does addresses grammar extension and composition [78, 116, 120].

2.10 Conclusion

The research frontier in compiler technology is moving towards domain-specific solutions, but the burden for understanding these ideas and the shortage of supporting tools delay the widespread, practical adoption of these solutions. Our design patterns narrow this gap: a mainstream programming language *can* address most of the composability problem, but to fully solve it, more language constructs are needed. With these constructs identified, mainstream languages can evolve in the right direction. We are now one step closer to providing a language feature toolbox to programmers and language designers.

CHAPTER 3

FINDING COUNTEREXAMPLES FROM PARSING CONFLICTS

The design patterns to support compiler evolution and composition presented in the last chapter only deal with the compilation process once abstract syntax trees are available. To make evolution and composition possible at the front end, syntax engineering should receive the same level of attention. The classification of context-free grammars based on the number of lookahead symbols needed to uniquely determine the next action a parser should take has made parsing a largely solved problem [54]. In particular, LR(1) grammars, which require only one lookahead token, can be easily recognized; parser generators can automatically construct parsers for such grammars. Nevertheless, upon encountering a non-LR(1) grammar, these parser generators still emit cryptic error messages that do not pinpoint the source of errors in the grammar. In other words, the problem of troubleshooting erroneous grammars remains largely unaddressed.

One impediment to debugging grammars is that determining whether a grammar is ambiguous is undecidable [47], but discouraging theoretical findings should not be an excuse for not improving error messages given by parser generators. Whenever possible, parser generators should give sample derivations from an erroneous grammar to suggest why the grammar itself should be altered.

In this chapter, we present a practical method for constructing counterexamples from LALR parsing conflicts to help programmers diagnose faulty grammars. This joint work with Andrew Myers appeared in the Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation [50].

3.1 Introduction

An early triumph of programming language research was the development of parser generators, tools that in principle provide a concise, declarative way to solve the ubiqui-

tous problem of parsing. Although LALR parser generators are powerful and have been available since the 1970s [51], they remain difficult to use, largely because of the challenges that arise when debugging grammars to eliminate shift/reduce and reduce/reduce conflicts.

Currently, debugging LALR grammars requires a solid understanding of the internal mechanism of LR parsers, a topic that is often but not always taught in undergraduate-level compiler courses. Even with this understanding, language designers can spend hours trying to understand how a grammar specification leads to the observed conflicts. The predictable result is that software developers tend to hand-code parsers even for tasks to which parser generators are ideally suited. Hand-coded parsers lead to code that is more verbose, less maintainable, and more likely to create security vulnerabilities when applied to untrusted input [26, 27]. Developers may also compromise the language syntax in order to simplify parsing, or avoid domain-specific languages and data formats altogether.

Despite the intrinsic limitations of LL grammars, top-down parser generators such as ANTLR [89, 90] are popular perhaps because their error messages are less inscrutable. It is surprising that there does not seem to have been much effort to improve debugging of conflicts in the more powerful LR grammars. Generalized LR parsers [114] enable programmers to resolve ambiguities programmatically, but even with GLR parsers, ambiguities could be better understood and avoided. Moving towards this goal, Elkhound [69] reports parse trees but only when the user provides a counterexample illustrating the ambiguity. Some LALR parser generators attempt to report counterexamples [78, 125] but can produce misleading counterexamples because their algorithms fail to take lookahead symbols into account. Existing tools that do construct correct counterexamples [12, 103] use a brute-force search over all possible grammar derivations. This approach is impractically slow and does not help diagnose unambiguous grammars that are not LALR.

We improve the standard error messages produced by LR parser generators by giving short, illustrative counterexamples that identify ambiguities in a grammar and show how conflicts arise. For ambiguous grammars, we seek a *unifying counterexample*, a string of symbols having two distinct parses. Determining whether a context-free grammar is ambiguous is undecidable, however [47], so the search for a unifying counterexample cannot be guaranteed to terminate. When a unifying counterexample cannot be found in a reasonable time, we seek a *nonunifying counterexample*, a pair of derivable strings of symbols sharing a common prefix up to the point of conflict. Nonunifying counterexamples are also reported when the grammar is determined to be unambiguous but not LALR.

Our main contribution is a search algorithm that exploits the LR state machine to construct both kinds of counterexamples. Our evaluation shows that the algorithm is efficient in practice. A key insight behind this efficiency is to expand the search frontier from the *conflict state* instead of the *start state*.

The remainder of the chapter is organized as follows. Section 3.2 reviews how LR parser generators work and how parsing conflicts arise. Section 3.3 outlines properties of good counterexamples. Sections 3.4 and 3.5 explore algorithms for finding nonunifying and unifying counterexamples. An implementation of the algorithm that works well in practice is discussed in Section 3.6. Using various grammars, we evaluate the effectiveness, efficiency, and scalability of the algorithm in Section 3.7. Section 3.8 discusses related work, and Section 3.9 concludes.

3.2 Background

We assume the reader has some familiarity with LR grammars and parser generators. This section briefly reviews the construction of an LR parser and shows how LR parsing conflicts arise.

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt} \\
& | & \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{stmt} \\
& | & \mathit{expr}\ \texttt{?}\ \mathit{stmt}\ \mathit{stmt} \\
& | & \texttt{arr [}\ \mathit{expr}\ \texttt{] :=}\ \mathit{expr} \\
\mathit{expr} & \rightarrow & \mathit{num} \ \mid\ \mathit{expr}\ \texttt{+}\ \mathit{expr} \\
\mathit{num} & \rightarrow & \langle \mathit{digit} \rangle \ \mid\ \mathit{num}\ \langle \mathit{digit} \rangle
\end{array}
$$

Figure 3.1: An ambiguous CFG

3.2.1 Parser state machine

Starting from a context-free grammar like the one in Figure 3.1, the first step in generating an LR(1) parser is the construction of a parser state machine for the grammar. Each state contains a collection of *transitions* on symbols and a collection of *production items*. Each transition is either a *shift action* on a terminal symbol or a *goto* on a nonterminal symbol. A production item (abbreviated *item*) tracks the progress on completing the right-hand side of a production. Each item contains a *dot* (•) indicating transitions that have already been made on symbols within the production, and a *lookahead set* of possible terminals that can follow the production.

The items within a state include those that result from taking transitions from a predecessor state, and also those generated by the *closure* of all the productions of any nonterminal that follows a dot. For the start state, the items include those of productions of the start symbol and their closure[1]. Figure 3.2 shows a partial parser state diagram for the example grammar.

A parser maintains a stack of symbols during parsing. A shift action on the next input symbol t is performed when a transition on t is available in the current state; t is pushed onto the stack. A reduction is performed when the current state contains an item

[1]The actual parser construction adds a special start symbol and production, which are omitted in this section.

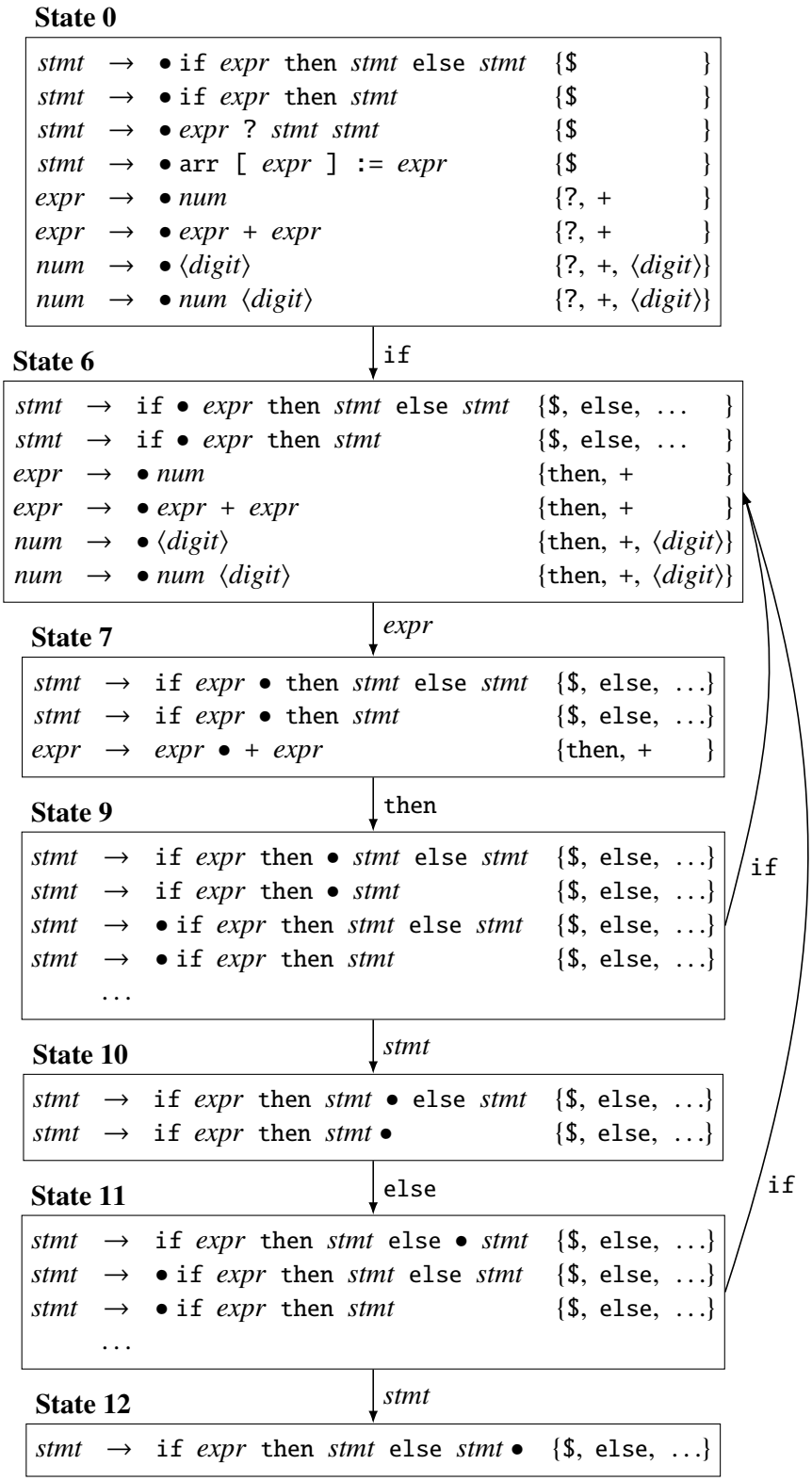


Figure 3.2: Selected parser states for the ambiguous CFG. Symbol `$` indicates the end of input.

of the form $A \rightarrow X_1\ X_2\ \cdots\ X_m\ \bullet$, whose lookahead set contains the next input symbol; m symbols are popped from the stack, and the nonterminal A is then pushed onto the stack. If neither a shift action nor a reduction is possible, a syntax error occurs.

3.2.2 Shift/reduce conflicts

For LR(1) grammars, actions on parser state machines are deterministic: given a state and the next input symbol, either a shift action or a reduction is executed. Otherwise, a state may contain a pair of items that create a *shift/reduce conflict* on a terminal symbol t:

- a *shift item* of the form $A \rightarrow X_1\ X_2\ \cdots\ X_k\ \bullet\ X_{k+1}\ \cdots\ X_m$, where $X_{k+1} = t$ for some $k \geq 0$ and $m \geq 1$, and
- a *reduce item* of the form $B \rightarrow Y_1\ Y_2\ \cdots\ Y_n\ \bullet$, whose lookahead set contains t.

The example grammar has a shift/reduce conflict, because the two items in State 10 match the criteria above on lookahead `else`. This is the classic *dangling else* problem. The grammar is ambiguous because there are two ways to parse this statement:

`if` *expr* `then if` *expr* `then` *stmt* • `else` *stmt*

Even though the grammar is ambiguous, not every conflict must contribute to the ambiguity. Conflicts may also occur even if the grammar is not ambiguous. For instance, the grammar in Figure 3.3 has a shift/reduce conflict between shift action $Y \rightarrow$ `a` • `a b` and reduction $X \rightarrow$ `a` • under symbol `a`. Nevertheless, this grammar is LR(2) and hence unambiguous. In fact, any LR(k) grammar can be transformed to an LR(1) grammar [54]. The transformation procedure is orthogonal to our work, however.

3.2.3 Reduce/reduce conflicts

A state may also contain a pair of distinct reduce items that create a *reduce/reduce conflict* because their lookahead sets intersect:

$$
\begin{array}{lcl}
S & \rightarrow & T \;\mid\; S\ T \\
T & \rightarrow & X \;\mid\; Y \\
X & \rightarrow & \texttt{a} \\
Y & \rightarrow & \texttt{a a b}
\end{array}
$$

Figure 3.3: An unambiguous CFG with a shift/reduce conflict

- $A \rightarrow X_1\ X_2\ \cdots\ X_m\ \bullet$, with lookahead set $\mathcal{L}_A$, and
- $B \rightarrow Y_1\ Y_2\ \cdots\ Y_n\ \bullet$, with lookahead set $\mathcal{L}_B$ such that $\mathcal{L}_A \cap \mathcal{L}_B \neq \emptyset$.

3.2.4 Precedence

To simplify grammar writing, precedence and associativity declarations can be used to resolve shift/reduce conflicts. For example, the grammar in Figure 3.1 has a shift/reduce conflict between shift item $expr \rightarrow expr\ \bullet\ +\ expr$ and reduce item $expr \rightarrow expr + expr\ \bullet$ under symbol +, exhibited by the counterexample $expr + expr\ \bullet\ +\ expr$. Declaring operator + left-associative causes the reduction to win.

3.3 Counterexamples

The familiar shift/reduce conflicts in the previous section are easily diagnosed by experienced programming language designers. In general, the source of conflicts can be more difficult to find.

3.3.1 A challenging conflict

The example grammar in Figure 3.1 has another shift/reduce conflict in State 1 (not shown in Figure 3.2) between

- shift item $num \rightarrow num\ \bullet\ \langle digit \rangle$, and

- reduce item $expr \rightarrow num \bullet$

under terminal symbol $\langle digit \rangle$. It is probably not immediately clear why this conflict is possible, let alone what counterexample explains the conflict. In fact, an experienced language designer in our research group spent some time to discover this counterexample by hand[2]:

$$expr \texttt{ ? arr [} expr \texttt{] :=} num \bullet \langle digit \rangle \langle digit \rangle \texttt{ ? } stmt\ stmt.$$

This statement can be derived in two ways from the production $stmt \rightarrow expr \texttt{ ? } stmt_1\ stmt_2$. First, we can use the reduce item:

- $stmt_1 \rightarrow^* \texttt{arr [} expr \texttt{] :=} num$
- $stmt_2 \rightarrow^* \langle digit \rangle \langle digit \rangle \texttt{ ? } stmt\ stmt$

Second, we use the shift item:

- $stmt_1 \rightarrow^* \texttt{arr [} expr \texttt{] :=} num \langle digit \rangle$
- $stmt_2 \rightarrow^* \langle digit \rangle \texttt{ ? } stmt\ stmt$

This counterexample, along with its two possible derivations, immediately clarifies why there is an ambiguity and helps guide the designer towards a better syntax, e.g., demarcating $stmt_1$ and $stmt_2$. Our goal is to generate such useful counterexamples automatically.

3.3.2 Properties of good counterexamples

Useful counterexamples should be concise and simple enough to help the user understand parsing conflicts effortlessly. This principle leads us to prefer counterexamples that are no more concrete than necessary. Although a sequence of terminal symbols that takes

[2]This conflict was originally part of a larger grammar.

the parser from the start state to the conflict state through a series of shift actions and reductions might be considered a good counterexample, some of these terminals may distract the user from diagnosing the real conflict. For example, the following input takes the parser from State 0 to State 10 in Figure 3.2:

```
if 2 + 5 then arr[4] := 7
```

But the expression `2 + 5` could be replaced with any other expression, and the statement `arr[4] := 7` with any other statement. Good counterexamples should use nonterminal symbols whenever the corresponding terminals are not germane to the conflict.

As discussed earlier, LALR parsing conflicts may or may not be associated with an ambiguity in a grammar. Counterexamples should be tailored to each kind of conflict.

Unifying counterexamples When possible, we prefer a *unifying counterexample*: a string of symbols (terminals or nonterminals) having two distinct parses. A unifying counterexample is a clear demonstration that a grammar is ambiguous. The counterexample given for the challenging conflict above is unifying, for example.

Good unifying counterexamples should be derivations of the innermost nonterminal that causes the ambiguity, rather than full sentential forms, to avoid distracting the user with extraneous symbols. For instance, a good unifying counterexample for the conflict in Section 3.2.4 is *expr* + *expr* • + *expr*, a derivation of the nonterminal *expr*, rather than *expr* + *expr* • + *expr* ? *stmt stmt*, a derivation of the start symbol.

Nonunifying counterexamples When a unifying counterexample cannot be found, there is still value in a *nonunifying counterexample*: a pair of derivable strings of symbols sharing a common prefix up to the point of conflict but diverging thereafter. The common prefix shows that the conflict state is reachable by deriving some nonterminal in the grammar. For example, Figure 3.4 shows a possible nonunifying counterexample for the challenging conflict.

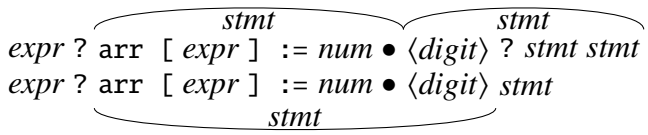


Figure 3.4: A nonunifying counterexample for the challenging conflict. Each bracket groups symbols derived from the nonterminal *stmt*.

Like unifying counterexamples, good nonunifying counterexamples should be derivations of the innermost nonterminal that can reach the conflict state.

Nonunifying counterexamples are produced for unambiguous grammars that are not LALR. Additionally, since ambiguity detection is undecidable, no algorithm can always provide a unifying counterexample for every ambiguous grammar. In this case, providing a nonunifying counterexample is a suitable fallback strategy.

3.4 Constructing nonunifying counterexamples

We first describe an algorithm for constructing nonunifying counterexamples that are derivations of the start symbol. The algorithm for constructing unifying counterexamples, described in Section 3.5, identifies the innermost nonterminal that can reach the conflict state.

Recall that certain terminals in a counterexample can be replaced with a nonterminal without invalidating the counterexample. Such terminals must have been part of a reduction. Therefore, a counterexample can be constructed from a walk along transition edges in the parser state diagram from the start state to the conflict state. Not all such walks constitute valid counterexamples, however. In particular, the shortest path is often invalid. For example, the input `if` *expr* `then` *stmt* forms the shortest path to State 10 in Figure 3.2, but a conflict does not arise at this point. If the next input symbol is `else`, the shift action is performed; if the end of input is reached, the reduction occurs. For a

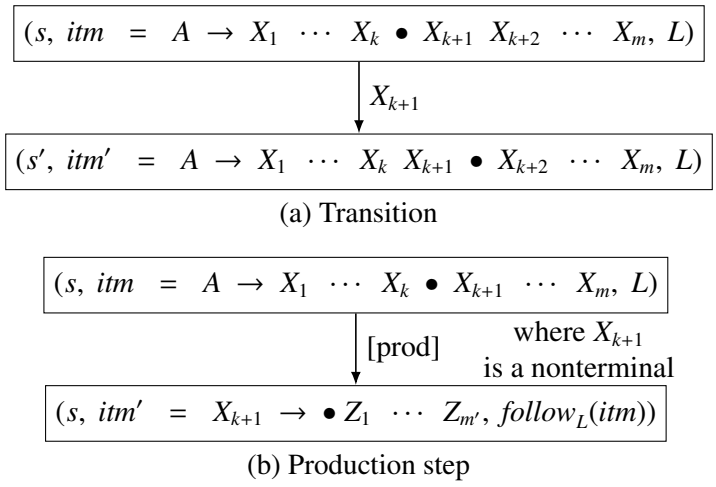


Figure 3.5: Edges of a lookahead-sensitive graph

counterexample to be valid, the lookahead sets of parser items must be considered as well.

Instead of finding the shortest path in the state diagram, our algorithm finds the shortest *lookahead-sensitive path* to the conflict state. Intuitively, a lookahead-sensitive path is a sequence of transitions and production steps[3] between parser states that also keeps track of terminals that actually can follow the current production.

To define lookahead-sensitive paths formally, we first define a *lookahead-sensitive graph*, an extension of an LR(1) parser state diagram in which production steps are represented explicitly. Each vertex is a triple (s, itm, L), where s is a state number, itm is an item within s, and L is a *precise lookahead set*. The edges in this graph are defined as follows:

- *transition* (Figure 3.5(a)): For every transition in the parser, there is an edge between appropriate parser states and items, preserving the precise lookahead set between the vertices.

[3] A production step picks a specific production of a nonterminal to work on. These steps are implicit in an LR closure.

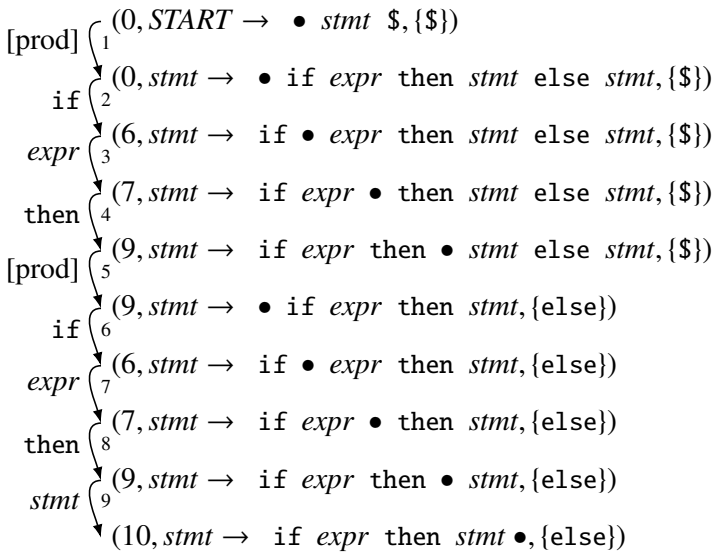
- *production step* (Figure 3.5(b)): For every item whose symbol after • is a nonterminal, there is an edge from this item to each item associated with a production of the nonterminal within the same state. The precise lookahead set changes to the set of terminals that actually can follow the production. Denoted $follow_L(itm)$, the *precise follow set* for *itm* in L's context is defined as follows:
  - $follow_L(A \rightarrow X_1 \cdots X_{n-1} \bullet X_n) \triangleq L$.
  - $follow_L(A \rightarrow X_1 \cdots X_k \bullet X_{k+1}\ X_{k+2} \cdots X_n) \triangleq \{X_{k+2}\}$ if X_{k+2} is a terminal.
  - $follow_L(A \rightarrow X_1 \cdots X_k \bullet X_{k+1}\ X_{k+2} \cdots X_n) \triangleq \text{FIRST}(X_{k+2})$ if X_{k+2} is a nonnullable nonterminal, i.e., a nonterminal that cannot derive ε. FIRST(N) is the set of terminals that can begin a derivation of N.
  - $follow_L(A \rightarrow X_1 \cdots X_k \bullet X_{k+1}\ X_{k+2} \cdots X_n) \triangleq \text{FIRST}(X_{k+2}) \cup follow_L(A \rightarrow X_1 \cdots X_{k+1} \bullet X_{k+2} \cdots X_n)$ if X_{k+2} is a nullable nonterminal.

A shortest lookahead-sensitive path is a shortest path in the lookahead-sensitive graph. To construct a counterexample, the algorithm starts by finding a shortest lookahead-sensitive path from $(s_0, itm_0, \{\$\})$ to (s', itm', L'), where s_0 is the start state, itm_0 is the start item, s' is the conflict state, itm' is the conflict reduce item[4], and L' contains the conflict symbol. The symbols associated with the transition edges form the first part of a counterexample. For instance, Figure 3.6(a) shows the shortest lookahead-sensitive path to the conflict reduce item in State 10 of Figure 3.2. This path gives the prefix of the expected counterexample:

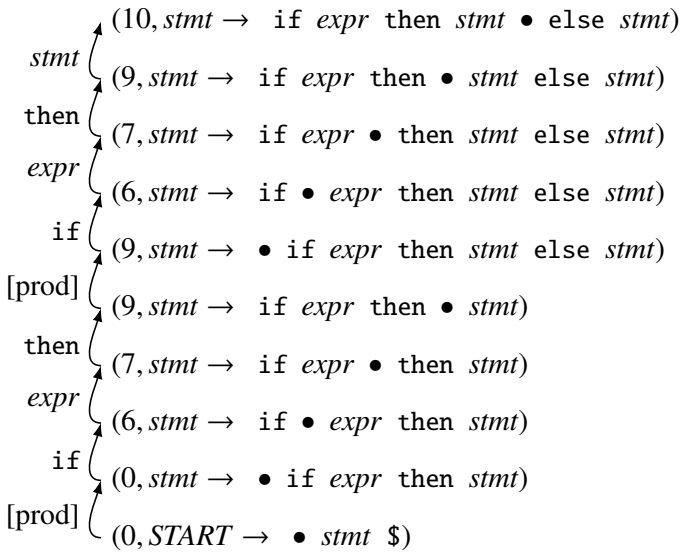
`if` *expr* `then if` *expr* `then` *stmt* •

To avoid excessive memory consumption, our algorithm does not construct the lookahead-sensitive graph in its entirety. Rather, vertices and edges are created as they are discovered.

[4]The conflict shift item cannot be used because we have no information about the lookahead symbol after the completion of the shift item.



(a) The shortest lookahead-sensitive path to the conflict reduce item



(b) The path to the conflict shift item obtained from the shortest lookahead-sensitive path

Figure 3.6: Paths to the dangling-else shift/reduce conflict

The partial counterexample constructed so far takes the parser to the conflict state. Counterexamples can be constructed in full by completing all the productions made on the shortest lookahead-sensitive path. Since the conflict terminal is a vital part of counterexamples, this terminal must immediately follow •. In the example above, a production step was made in State 9 (step 5 in Figure 3.6(a)), where the next symbol to be parsed is the conflict terminal `else`. In this case, the production can be completed immediately, yielding the counterexample

`if` *expr* `then if` *expr* `then` *stmt* • `else` *stmt*

On the other hand, if the symbol immediately after • is a nonterminal, a derivation of that nonterminal beginning with the conflict symbol is required. Consider once again the conflict between *expr* → *num* • and *num* → *num* • ⟨*digit*⟩ under lookahead ⟨*digit*⟩. The shortest lookahead-sensitive path to the reduce item gives the prefix

expr `?` `arr [` *expr* `] :=` *num* •

but the next symbol to be parsed is *stmt*. In this case, we must find a statement that starts with a digit, e.g., ⟨*digit*⟩ `?` *stmt stmt*, yielding the counterexample

expr `?` `arr [` *expr* `] :=` *num* • ⟨*digit*⟩ `?` *stmt stmt*

The shortest lookahead-sensitive path only reveals a counterexample that uses the conflict reduce item. A counterexample that uses the conflict shift item can be discovered by exploring the states on the shortest lookahead-sensitive path as follows. Since transitions on input symbols must be between the same states for both counterexamples, the only difference is that the derivation using the shift item may take different production steps within such states. To determine these production steps, our algorithm starts at the conflict shift item and explores backward all the productions that may be used in the states along the shortest lookahead-sentitive path, until an item used in the derivation using the reduce item is found. For example, Figure 3.6(b) shows the reverse sequence

leading to the shift item of the dangling-else conflict. Observe that this sequence follows the same states as in the shortest lookahead-sensitive path when making transitions, namely, $[0, 6, 7, 9, 6, 7, 9, 10]$. Even though this sequence yields the same counterexample as above, the derivation is different.

3.5 Constructing unifying counterexamples

The algorithm for constructing nonunifying counterexamples does not guarantee that the resulting counterexamples will be ambiguous if the grammar is. To aid the diagnosis of an ambiguity, the symbols beyond the conflict terminal must agree so that the entire string can be parsed in two different ways using the two conflict items. Since these conflict items force parser actions to diverge after the conflict state, the algorithm must keep track of both parses simultaneously.

3.5.1 Product parser

The idea of keeping track of two parses is similar to the intuition behind generalized LR parsing [114], but instead of running the parser on actual inputs, our approach simulates possible parser actions and constructs counterexamples at parser generation time. Two copies of the parser are simulated in parallel. One copy is required to take the reduction and the other to take the shift action of the conflict. If both copies accept an input at the same time, then this input is a unifying counterexample. A distinct sequence of parser actions taken by each copy describes one possible derivation of the counterexample.

More formally, the parallel simulation can be represented by actions on a *product parser*, whose states are the Cartesian product of the original parser items. Two stacks are used, one for each original parser. This construction resembles that of a direct product of nondeterministic pushdown automata [2], but here the states are more tightly coupled

to make parser actions easier to understand. Like a lookahead-sensitive graph, a product parser represents production steps explicitly. Actions on a product parser are defined as follows:

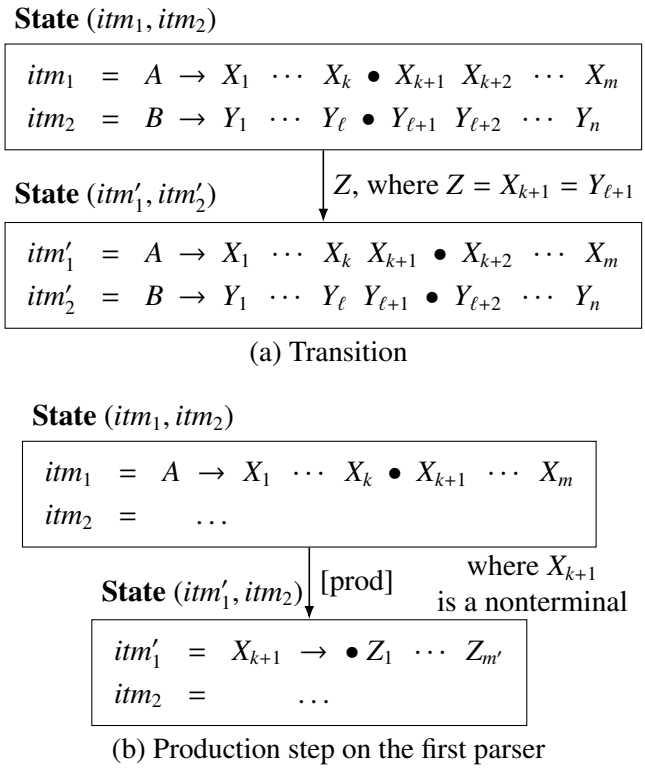
- *transition*: If both items in a state of the product parser have a transition on symbol Z in the original parser, there is a corresponding transition on Z in the product parser (Figure 3.7(a)). When this transition is taken, Z is pushed onto both stacks.
- *production step*: If an item in a state of the product parser has a nonterminal after $\bullet$, there is a production step on this nonterminal in the product parser (Figure 3.7(b)). Both stacks remain unchanged when a production step is taken.
- *reduction*: If an item in a state of the product parser is a reduce item, a reduction can be performed on the original parser associated with this item, respecting its loookahead set, while leaving the other item and its associated stack unchanged.

For a conflict between items itm_1 and itm_2, a string accepted by the product parser that also takes the parser through state (itm_1, itm_2) is a unifying counterexample for the conflict.

Although any algorithms that simulate the product parser through the conflict state are theoretically sufficient to exhibit a unifying counterexample, many are impractical because of the size of the search space. The remainder of this section describes an algorithm that explores no more search states than necessary.

3.5.2 Outward search from the conflict state

The strategy of using shortest lookahead-sensitive paths to avoid exploring too many states does not work in general, because symbols required after $\bullet$ might be incompatible with the productions already made on these paths. For example, the grammar in Figure 3.8 has two shift/reduce conflicts in the same state, between reduce item $A \rightarrow \mathtt{a} \bullet$ and two



(a) Transition

(b) Production step on the first parser

Figure 3.7: Components of the state machine for a product parser

$$
\begin{array}{lcl}
S & \rightarrow & N \;\mid\; N \;\texttt{c} \\
N & \rightarrow & \texttt{n}\; N\; \texttt{d} \;\mid\; \texttt{n}\; N\; \texttt{c} \;\mid\; \texttt{n}\; A\; \texttt{b} \;\mid\; \texttt{n}\; B \\
A & \rightarrow & \texttt{a} \\
B & \rightarrow & \texttt{a}\; \texttt{b}\; \texttt{c} \;\mid\; \texttt{a}\; \texttt{b}\; \texttt{d}
\end{array}
$$

Figure 3.8: An ambiguous grammar where the shortest lookahead-sensitive path does not yield a unifying counterexample

$$
\begin{array}{l}
I_1 = [itm_1^1, \ldots, itm_1^{m_1}] \\
D_1 = [d_1^1, \ldots, d_1^{n_1}] \\
\hline
I_2 = [itm_2^1, \ldots, itm_2^{m_2}] \\
D_2 = [d_2^1, \ldots, d_2^{n_2}]
\end{array}
$$

(a) General form

$$
\begin{array}{l}
I_1 = [\text{conflict-item}_1] \\
D_1 = [] \\
\hline
I_2 = [\text{conflict-item}_2] \\
D_2 = []
\end{array}
$$

(b) Initial configuration

Figure 3.9: Configurations. Each *itm* is an item in the original parser, and each *d* is a derivation associated with a transition between items.

shift items $B \rightarrow \texttt{a} \bullet \texttt{b}\ \texttt{c}$ and $B \rightarrow \texttt{a} \bullet \texttt{b}\ \texttt{d}$ under symbol `b`. The shortest lookahead-sensitive path gives prefix `n a` •, which is compatible with a unifying counterexample for the first shift item, namely, `n a` • `b c`. Still, no unifying counterexamples that use the second shift item can begin with `n a` •. An extra `n` is required before •, as in `n n a` • `b d c`. This example suggests that deciding on the productions to use before reaching the conflict state is inimical to discovering unifying counterexamples.

To avoid making such decisions, our search algorithm starts from the conflict state and completes derivations outward. Each search state, denoted *configuration* henceforth, contains two pairs of (1) a sequence of items representing valid transitions and production steps in the original parser, and (2) partial derivations associated with transitions between items, as shown in Figure 3.9(a). The initial configuration contains (1) singleton sequences of the conflict items and (2) empty derivations, as shown in Figure 3.9(b). As partial derivations are expanded, configurations progress through four stages, which are

illustrated in Figure 3.10 for the challenging conflict from Section 3.3.1. The four stages are as follows:

1. Completion of the conflict reduce item: the counterexample must contain derivations of all symbols in the reduce item. For shift/reduce conflicts, some of the symbols before • in the shift item will also be derived in this stage. For reduce/reduce conflicts, both conflict items are completed in this stage.
2. Completion of the conflict shift item: the counterexample must also contain derivations of all symbols in the shift item after •, and of any remaining symbols before • that were not derived in Stage 1. This stage is not needed for reduce/reduce conflicts.
3. Discovery of the unifying nonterminal: the counterexample must be a derivation of a single nonterminal. This stage is completed when the derivations on both paths of the parser originate from the same nonterminal. This stage also identifies the innermost nonterminal for nonunifying counterexamples.
4. Completion of the entire unifying counterexample: the final counterexample must complete all the unfinished productions. This stage attempts to find the remaining symbols so that the derivation of the nonterminal found in Stage 3 can be completed at the same time on both copies of the parser.

3.5.3 Successor configurations

We now present a strategy for computing successor configurations. Figure 3.11 pictures some of the possible successor configurations that can be reached from the configuration shown in Figure 3.9(a) via various actions in the product parser:

- *transition* (Figure 3.11(a)): If the product parser has a transition on symbol Z from the last item in the current configuration, append the current configuration with appropriate items and symbols.

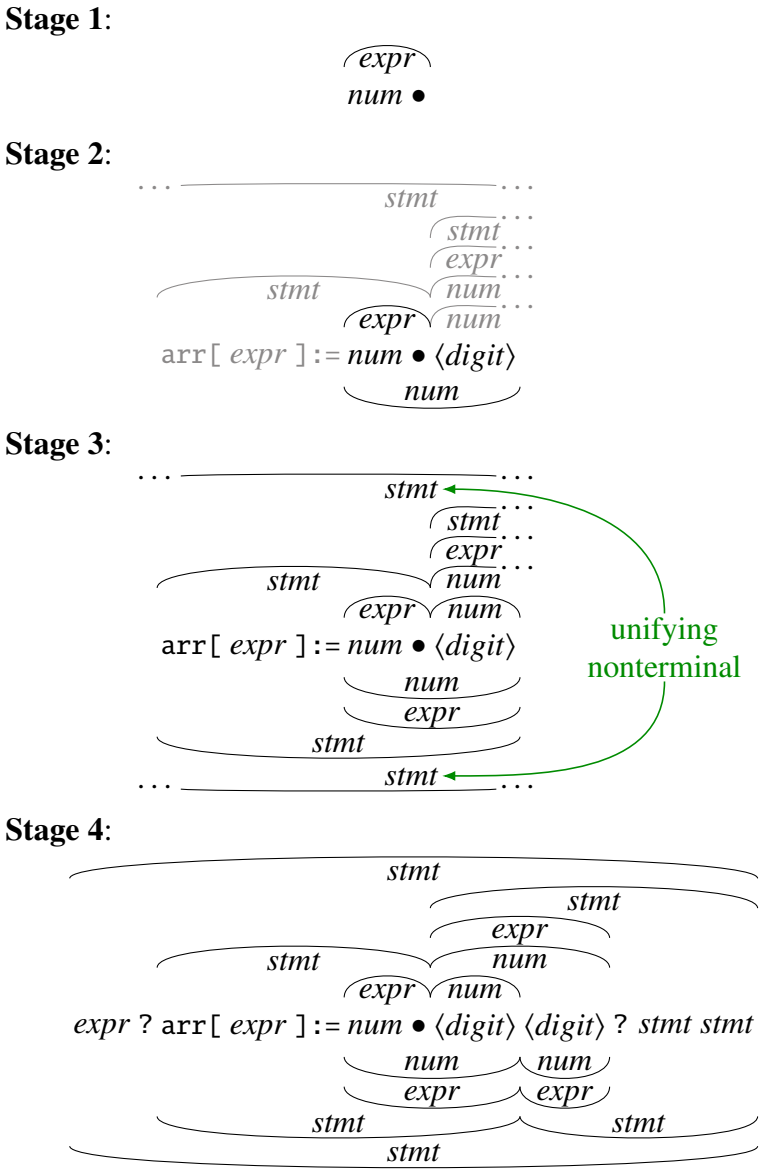
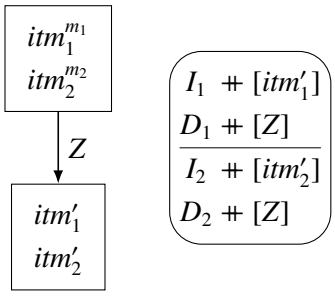
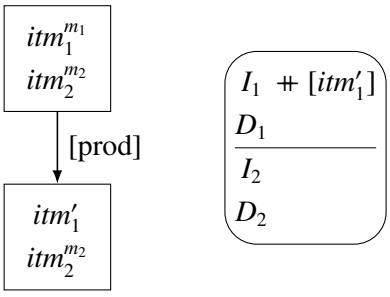


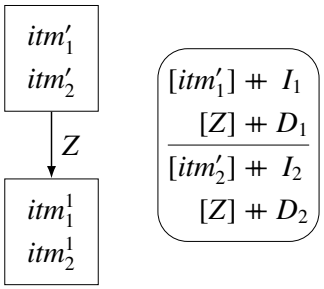
Figure 3.10: Counterexamples and derivations associated with configurations after finishing each stage for the challenging conflict. The derivation above each counterexample uses the reduce item; the one below uses the shift item. The gray portion of the configuration is not required for completing the stage.



(a) Transition

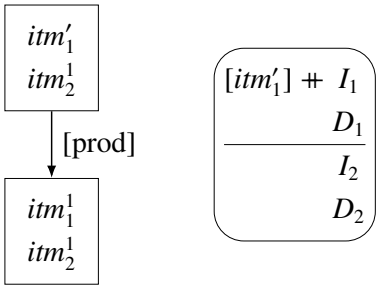


(b) Production step on the first parser



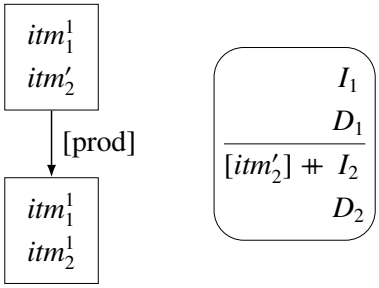
where $itm_1^{m_1}$ is a reduce item
and itm'_1, itm'_2 are in same state

(c) Reverse transition



where $itm_1^{m_1}$ is a reduce item of the form
$A \to X_1 \ \cdots \ X_\ell \bullet$, and $m_1 = \ell + 1$

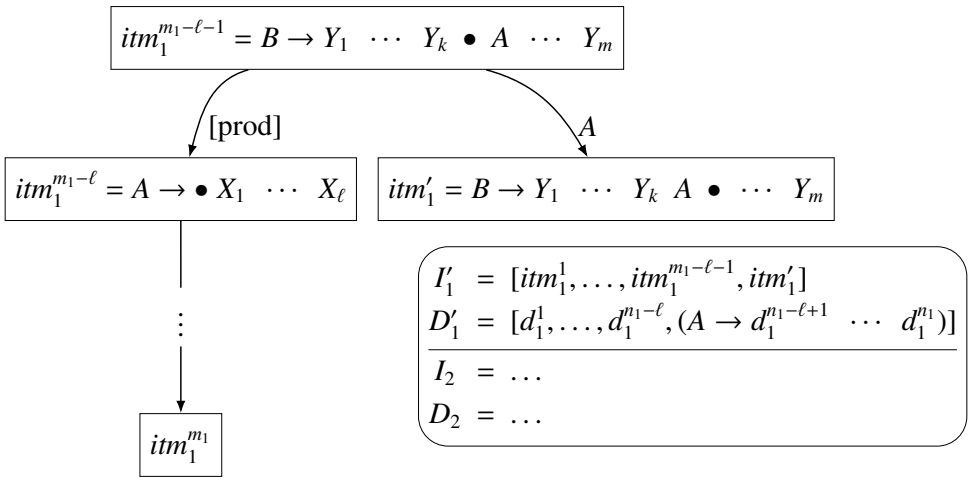
(d) Reverse production step on the first parser



where $itm_1^{m_1}$ is a reduce item of the form
$A \to X_1 \ \cdots \ X_\ell \bullet$, and $m_1 < \ell + 1$

(e) Reverse production step on the second parser

Figure 3.11: Successor configurations. Each kind of edge in the product parser corresponds to a particular successor configuration. Operator $\mathbin{+\!\!+}$ denotes list concatenation.



where $itm_1^{m_1}$ is a reduce item of the form
$A \to X_1 \ \cdots \ X_\ell \bullet$, and $m_1 > \ell + 1$

(f) Reduction on the first parser

Figure 3.11: Successor configurations (continued). Each kind of edge in the product parser corresponds to a particular successor configuration.

- *production step on the first parser* (Figure 3.11(b)): If the product parser has a production step on the first parser from the last item in the current configuration, append the item resulting from taking the production step (itm'_1) to the sequence of items for the first parser (I_1). A production step on the second parser is symmetric.

- *preparation of the first parser for a reduction*: If the last item for the first parser is a reduce item, but there are not enough items to simulate a reduction moving forward, then more items must be prepended to the configuration. That is, we must work backward to ready the reduction. Preparing the second parser for a reduction is symmetric. Successor configurations depend on the first item in the current configuration:

  – *reverse transition* (Figure 3.11(c)): If the product parser has a transition on symbol Z to the first item in the current configuration, prepend the current configuration with appropriate items and symbols. The prepended items must belong to the same state in the original parser. Additionally, the lookahead set of the item prepended to the first parser (itm'_1) must contain the conflict symbol if the current configuration

is yet to complete Stage 1 so that the derivation of the conflict symbol remains possible.

- *reverse production step on the first parser* (Figure 3.11(d)): If the product parser has a production step on the first parser to the first item in the current configuration, prepend the item prior to taking the production step (itm'_1). to the sequence of items for the first parser (I_1).
- *reverse production step on the second parser* (Figure 3.11(e)): Occasionally, the second parser will require a reverse production step so that further reverse transitions can be made. In this case, prepend the item prior to taking the production step (itm'_2) to the sequence of items for the second parser (I_2).

◆ *reduction on the first parser* (Figure 3.11(f)): If the last item for the first parser is a reduce item of the form $A \rightarrow X_1 \ \cdots \ X_\ell \bullet$, and the configuration has enough items, then the first parser is ready for a reduction. A successor configuration is obtained by (1) removing the last $\ell + 1$ items that are part of the reduction from I_1, which simulates popping the parser stack, (2) appending the result of taking the goto on A (itm'_1) to I_1, and (3) rearranging the partial derivations (D_1) to complete the derivation for A. The second parser remains unchanged throughout the reduction. A reduction on the second parser is symmetric.

3.5.4 Completing the search

The search algorithm computes successor configurations until it encounters a configuration C_f that has completed Stages 1 and 2, where both sequences of items in C_f are of the form

$$[? \rightarrow \cdots \ \bullet \ A \ \cdots, ? \rightarrow \cdots \ A \ \bullet \ \cdots]$$

for some nonterminal A. The partial derivations associated with these sequences, which must be of the form $[A \rightarrow \cdots]$, show that nonterminal A is ambiguous. The unifying counterexample is the sequence of the leaf symbols within these derivations.

Several observations can be made about the algorithm. First, the algorithm maintains an invariant that the head of both sequences of items in any configuration belong to the same parser state, as the sequence of states prior to the conflict must be identical for different derivations of the unifying counterexample. Second, a configuration generates multiple successor configurations only when a production step (forward or backward) or a reverse transition is taken. Therefore, the branching factor of the search is proportional to the ratio of the number of these actions to the number of items in the parser.

The third observation is that a production step may be taken repeatedly within the same state, such as one for items of the form $A \rightarrow \ \bullet\ A\ \cdots$. To avoid infinite expansions on one configuration without making progress on others, the search algorithm must postpone such an expansion until other configurations have been considered. The algorithm imposes different costs on different kinds of actions and considers configurations in order of increasing cost.

Finally, the algorithm is guaranteed to find a unifying counterexample for every ambiguous grammar, but the search will not terminate when infinite expansions are possible on unambiguous grammars. In other words, this semi-decision procedure for determining ambiguity is sound and complete. Since a naive implementation of this algorithm is too slow for practical use, the next section discusses techniques that speed up the search but still maintain the quality of counterexamples.

```
Warning : *** Shift/Reduce conflict found in state #13
  between reduction on expr ::= expr PLUS expr •
  and shift on         expr ::= expr • PLUS expr
  under symbol PLUS
  Ambiguity detected for nonterminal expr
  Example: expr PLUS expr • PLUS expr
  Derivation using reduction:
    expr ::= [expr ::= [expr PLUS expr •] PLUS expr]
  Derivation using shift    :
    expr ::= [expr PLUS expr ::= [expr • PLUS expr]]
```

Figure 3.12: A sample error message reported by the implementation. The first four lines are original to CUP.

3.6 Implementation

Our counterexample finder has been implemented in Java as a module extending the CUP LALR parser generator [48] version 0.11b 20150326[5]. The module contains 1478 non-comment, nonempty lines of code. Whenever a conflict is detected, the counterexample finder is run by default. The option `-noexamples` can be used to turn off the search. Figure 3.12 shows an error message reported by our implementation for the shift/reduce conflict in Section 3.2.4. Whenever a unifying counterexample is found, the nonterminal that permits two different derivations using the conflict items is reported along with the counterexample discovered. The actual derivations are then reported to let the user pinpoint the location of the conflict. One interesting design choice was the tradeoff between finding unifying counterexamples when they exist, and avoiding long, possibly fruitless searches when a nonunifying counterexample might suffice.

Data structures The search algorithm requires many queries on possible parser actions, but parser generators usually do not provide an infrastructure for fast lookups. In particular, reverse transitions and production steps are not represented directly. Before

[5] Available at `https://github.com/polyglot-compiler/polyglot/tree/master/tools/java_cup`.

working on the first conflict within a grammar, our implementation generates these lookup tables:

- transition map of type $\mathit{Item} \times \mathit{Symbol} \to \mathit{Item}$
- reverse transition map of type $\mathit{Item} \times \mathit{Symbol} \to \mathcal{P}(\mathit{Item})$, as many items can make a transition on a given symbol to a given item
- production step map of type $\mathit{Item} \to \mathcal{P}(\mathit{Item})$, as the nonterminal immediately after $\bullet$ may have multiple productions
- reverse production step map of type $\mathit{State} \times \mathit{NonTerminal} \to \mathcal{P}(\mathit{Item})$, i.e., given a parser state and a nonterminal, this map returns a set of items that can make a production step on the nonterminal.

A search configuration is an object containing two lists of items and two lists of derivations for the two parser paths, a *complexity* associated with the cost of parser actions that have already been made, and boolean flags indicating whether the two conflict items have been completed, i.e., completion of Stages 1 and 2. A derivation is a tree of symbols, where a leaf may be a terminal or a nonterminal whose derivations are not important to the counterexample.

The search algorithm uses a priority queue to consider configurations in increasing order of complexity. Instead of holding configurations directly, the priority queue maintains sets of configurations having the same complexity. We choose this representation because many configurations have the same complexity, but inserting an element into a priority queue has a running time logarithmic in size. This design choice reduces the size of the priority queue to the number of distinct configuration complexities. Also, the computation of successor configurations could result in duplicates, e.g., when reductions on the two paths are made in different orders. Using sets prevents the algorithm from considering identical configurations more than once.

Finding shortest lookahead-sensitive path When our tool receives a conflict, it finds the shortest lookahead-sensitive path to the reduce conflict item as the first step. This path can be used in the main search algorithm as a tradeoff between performance and discovery of all unifying counterexamples, discussed below. The shortest lookahead-sensitive path is also used occasionally to construct nonunifying counterexamples when the main search algorithm fails.

Blindly searching for the shortest path from the start state might explore all parser states. As an optimization, only states that can reach the reduce conflict item need be considered. These states can be found quickly using the lookup tables for reverse transitions and reverse production steps.

Constructing unifying counterexamples Next, the main search algorithm is invoked. Our implementation uses a variant of the aforementioned computation of successor configurations. First, transitions on nullable nonterminals should not incur additional costs, as they do not alter unifying counterexamples. For this reason, such transitions are made immediately after any other transitions and after any reductions. Making costless transitions along with other actions that incur cost ensures that the configuration complexity is strictly increasing. As a result, all configurations of any given cost will have been discovered when they are removed from the priority queue, guaranteeing that the algorithm will make progress.

Second, this search algorithm is unguided, like one for finding the shortest lookahead-sensitive path. As a tradeoff, the algorithm only considers states on the shortest lookahead-sensitive path when making reverse transitions. This restriction makes the algorithm incomplete, causing it to miss unifying counterexamples that use parser states outside the shortest path. Nevertheless, a counterexample that follows the shortest lookahead-sensitive path will take the parser to the conflict state as quickly as possible. These

Action	Cost
transition	1
production step	50
reverse transition	1
reverse production	50
reduction	1
extended search	10000

Table 3.1: Cost model used in the implementation of the search algorithm

compact counterexamples seem as helpful as unifying ones, so our tool does report them. The option `-extendedsearch` can be used to force a full search.

As stated earlier, the algorithm requires costs associated with different parser actions to ensure progress. The current implementation uses the cost model shown in Table 3.1. The cost for the extended search applies when a reverse transition is made to a state not on the shortest path.

Constructing nonunifying counterexamples The search for unifying counterexamples may fail in two cases: first, when eligible configurations run out; second, when a production step in an unambiguous grammar is taken repeatedly, resulting in nontermination. Therefore, our implementation imposes a 5-second time limit on the main search algorithm. When the search fails, a nonunifying counterexample is constructed and reported instead.

How nonunifying counterexamples are constructed depends on the configuration stages the main search has encountered at failure. If a configuration that have completed Stage 3 was encountered, the nonunifying counterexamples are two derivations of the unifying nonterminal in that configuration. Otherwise, they are derivations of the start symbol.

The implementation also imposes a 2-minute time limit on the cumulative running time of the unifying counterexample finder. After two minutes, at least 20 conflicts must

have been accompanied with counterexamples, so the user is likely to prefer resolving them first. Our tool seeks only nonunifying counterexamples thereafter.

Exploiting precedence Precedence and associativity are not part of the parser state diagram, and hence are not part of the generated lookup tables. Therefore, our implementation inspects precedence declared with relevant terminals and productions during the search. Alternatively, precedence and associativity could be considered when generating the lookup tables. We choose not to do so because these properties are inexpensive to check and because the lookup tables are simpler to generate when they are not taken into account.

3.7 Evaluation

Our evaluation aims to answer three questions:

- Is our implementation effective on different kinds of grammars?
- Is our implementation efficient compared to existing ambiguity detection tools?
- Does our implementation scale to reasonably large grammars?

3.7.1 Grammar examples

We have evaluated our implementation on a variety of grammars. For each grammar, Table 3.2 lists the complexity (the numbers of nonterminals and productions, and the number of states in the parser state machine) and the number of conflicts. The grammars are partitioned into the following categories:

Our grammars All grammars shown in this paper are evaluated. Other grammars that motivated the development of our tool, and a few grammars in previous software projects that pose challenging parsing conflicts are also part of the evaluation.

Grammar	# nonterms	# prods	# states	# conflicts	Amb?
figure3.1	3	9	24	3	✔
figure3.3	4	7	10	1	✗
figure3.8	4	10	16	2	✔
ambfailed01	6	10	17	1	✔
abcd	5	11	22	3	✔
simp2	10	41	70	1	✔
xi	16	41	82	6	✔
eqn	14	67	133	1	✔
java-ext1	185	445	767	2	✗
java-ext2	234	599	1255	1	✗
stackexc01	2	7	13	3	✔
stackexc02	6	11	15	1	✗
stackovf01	2	5	9	1	✗
stackovf02	2	5	9	4	✔
stackovf03	2	6	10	1	✔
stackovf04	5	9	13	1	✗
stackovf05	5	10	14	1	✔
stackovf06	6	10	15	2	✗
stackovf07	7	12	17	3	✔
stackovf08	3	13	21	8	✗
stackovf09	6	12	27	1	✗
stackovf10	9	20	53	19	✔
SQL.1	8	23	46	1	✔
SQL.2	29	81	151	1	✔
SQL.3	29	81	149	1	✔
SQL.4	29	81	151	1	✔
SQL.5	29	81	151	1	✔
Pascal.1	79	177	323	3	✔
Pascal.2	79	177	324	5	✔
Pascal.3	79	177	321	1	✔
Pascal.4	79	177	322	1	✔
Pascal.5	79	177	322	1	✔
C.1	64	214	369	1	✔
C.2	64	214	368	1	✔
C.3	64	214	368	4	✔
C.4	64	214	369	1	✔
C.5	64	214	370	1	✔
Java.1	152	351	607	1	✔
Java.2	152	351	606	1133	✔
Java.3	152	351	608	2	✔
Java.4	152	351	608	14	✔
Java.5	152	351	607	3	✔

 # nonterms
Number of nonterminals in the grammars.

 # prods
Number of productions in the grammars.

 # states
Number of states in the parser state machine.

 # conflicts
Number of conflicts in the parser state machine.

Amb?
Whether the grammar is ambiguous.

Table 3.2: Characterization of grammars used in the evaluation.

Grammar	# unif	# nonunif	# time out	Time (seconds) Total	Average
figure3.1	3	0	0	0.072	0.024
figure3.3	0	1	0	0.010	0.010
figure3.8	2	0	0	0.016	0.008
ambfailed01	0	1	0	0.010	0.010
abcd	3	0	0	0.024	0.008
simp2	1	0	0	0.548	0.548
xi	6	0	0	0.155	0.026
eqn	1	0	0	0.169	0.169
java-ext1	0	0	2	T/L	T/L
java-ext2	0	0	1	T/L	T/L
stackexc01	3	0	0	0.023	0.008
stackexc02	0	1	0	0.008	0.008
stackovf01	0	1	0	0.009	0.009
stackovf02	4	0	0	0.043	0.011
stackovf03	1	0	0	0.017	0.017
stackovf04	0	1	0	0.009	0.009
stackovf05	1	0	0	0.010	0.010
stackovf06	0	2	0	0.012	0.006
stackovf07	3	0	0	0.028	0.009
stackovf08	0	8	0	0.025	0.003
stackovf09	0	1	0	0.017	0.017
stackovf10	19	0	0	0.140	0.007
SQL.1	1	0	0	0.024	0.024 (1.8s)
SQL.2	1	0	0	0.060	0.060 (0.1s)
SQL.3	1	0	0	0.024	0.024 (0.1s)
SQL.4	1	0	0	0.031	0.031 (0.0s)
SQL.5	1	0	0	0.030	0.030 (0.4s)
Pascal.1	2	0	1	0.196	0.098 (0.3s)
Pascal.2	5	0	0	0.296	0.059 (0.1s)
Pascal.3	1	0	0	0.070	0.070 (1.2s)
Pascal.4	1	0	0	0.081	0.081 (0.3s)
Pascal.5	1	0	0	0.113	0.113 (0.3s)
C.1	1	0	0	0.327	0.327 (1.3s)
C.2	1	0	0	0.219	0.219 (1.11h)
C.3	4	0	0	1.015	0.254 (0.5s)
C.4	0	0	1	T/L	T/L (1.3s)
C.5	1	0	0	0.212	0.212 (4.9s)
Java.1	1	0	0	0.569	0.569 (32.4s)
Java.2	141	0	9 (983)	35.384	0.251 (0.4s)
Java.3	2	0	0	0.435	0.218 (35.1s)
Java.4	6	2	6	2.042	0.255 (6.5s)
Java.5	3	0	0	0.526	0.175 (3.3s)

 # unif
Number of conflicts for which unifying counterexamples are found within the time limit.

 # nonunif
Number of conflicts for which nonunifying counterexamples are found within the time limit.

 # time out
Number of conflicts for which the tool times out. Nonunifying counterexamples are reported for these conflicts.

Total time
Time used when counterexamples are found within the time limit. (Average of 15 runs, with a standard deviation of at most 15%, so the margin of error is at most 9% at 95% confidence.)

Average time
$$\frac{\text{Total time}}{\text{\# unif} + \text{\# nonunif}}$$

T/L
5-second time limit exceeded on all conflicts.

Times in parentheses indicate running time for the state-of-the-art ambiguity detector [3, 7].

Table 3.3: Evaluation results on finding counterexamples.

Grammars from StackOverflow and StackExchange We evaluate our tool against grammars posted on StackOverflow and StackExchange by developers who had difficulty understanding the conflicts. This section of Table 3.2 links to the corresponding web pages.

Grammars from existing tool To compare our implementation with the state of the art, we run our tool against the grammars used to evaluate the grammar filtering technique [7]. These grammars, which we call the *BV10 grammars* hereafter, were constructed by injecting conflicts into correct grammars for mainstream programming languages. In some grammars (e.g., Java.2), the addition of a nullable production generates a large number of conflicts.

3.7.2 Effectiveness

Our tool always gives a counterexample for each conflict in every grammar. Table 3.3 reports the numbers of conflicts for which our tool successfully finds a unifying counterexample (if the grammar is ambiguous), for which our tool determines that no unifying counterexample exists, and for which our tool times out and hence reports a nonunifying counterexample. For grammars requiring more than two minutes of the main search algorithm, the number of remaining conflicts is shown in parentheses. Our implementation finishes within the time limit on 92% of the conflicts.

The main search algorithm may fail to find a unifying counterexample even if the grammar is ambiguous. One reason is the tradeoff used to reduce the number of configurations, as explained in Section 3.6. Grammar ambfailed01 illustrates this problem. Another reason is that the configuration describing the unifying counterexample has a cost too high for the algorithm to reach within the time limit. For instance, the ambiguous counterexample for grammar C.4 requires a long sequence of production steps. For these failures, nonunifying counterexamples are reported instead.

We also compare effectiveness against prior versions of the Polyglot Parser Generator (PPG) [78], which attempt to report only nonunifying counterexamples. PPG produces misleading results on ten benchmark grammars: figure3.1, figure3.8, abcd, simp2, SQL.5, Pascal.3, C.2, Java.1, Java.3, and Java.4. Incorrect counterexamples are generated because PPG's algorithm ignores conflict lookahead symbols. For instance, PPG reports this invalid counterexample for the dangling-else conflict:

`if` *expr* `then` *stmt* • `else`

`if` *expr* `then` *stmt* • `else` *stmt*

For grammar SQL.5, the reported counterexample is

`delete from` *y_table* `where` *y_boolean* • `order`

`delete from` *y_table* `where` *y_boolean* • `order by` *y_order*

which does not even constitute a valid SQL syntax allowed by the grammar, as opposed to the unifying counterexample reported by our tool:

`select` *y_columns* `from` *y_table* `where` *y_boolean* • `order by` *y_order*

The unifying counterexamples given by our algorithm provide a more accurate explanation of how parsing conflicts arise. Our algorithm has been integrated into a new version of PPG.

3.7.3 Efficiency

We have measured the running time of the algorithm on the conflicts that our tool runs within the time limit. These measurements were performed on an Intel Core2 Duo E8500 3.16GHz, 4GB RAM, Windows 7 64-bit machine. The results are shown in the last two columns of Table 3.3. For the BV10 grammars, we also include in parentheses the time used on a similar machine by a grammar-filtering variant of CFGAnalyzer [3, 7],

which is the fastest, on average, among the ambiguity detection tools we have found. This state-of-the-art ambiguity detector terminates as soon as it finds one ambiguous counterexample, whereas our tool finds a counterexample for every conflict. Hence, the running time of the state-of-the-art tool is compared against the average time taken per conflict in our implementation.

On average, when the time limit is not exceeded, the algorithm spends 0.18 seconds per conflict to construct a counterexample. For grammars taken from StackOverflow and StackExchange, the average is 8 milliseconds. Instead of posting these grammars on the Internet and waiting for others to respond, programmers can use our tool to diagnose the grammars almost instantly, thereby increasing productivity.

For the BV10 grammars, our algorithm outperforms the filtering technique. Based on a geometric average, our tool is 10.7 times faster than the variant of CFGAnalyzer, which takes more than 30 seconds to find a counterexample for certain grammars. (One grammar takes 0.0s for both tools and therefore dropped from the average.) For most of these grammars, the time our implementation takes to find counterexamples for all conflicts is less than that of the state-of-the-art tool trying to find just one counterexample. For grammar C.4, the CFGAnalyzer variant finds a unifying counterexample, but our tool fails to do so within the time limit. This result suggests that grammar filtering would be a useful addition to our approach.

3.7.4 Scalability

The evaluation results show that the running time of our algorithm only increases marginally on larger grammars, such as those for mainstream programming languages. The performance shown here demonstrate that, unlike prior tools, our counterexample finder is practical and suitable for inclusion in LALR parser generators.

3.8 Related work

Generating counterexamples is just one way to help address parsing conflicts. In general, several lines of work address ways to deal with such problems. We discuss each of them in turn.

Ambiguity detection Several semi-decision procedures have been devised to detect ambiguity. Pandey provides a survey [88] on these methods, some of which we discuss below.

One way to avoid undecidability is to approximate input CFGs. The Noncanonical Unambiguity (NU) test [101] uses equivalence relations to reduce the number of distinguishable derivations of a grammar, reducing the size of the search space but overapproximating the language. Its mutual accessibility relations are analogous to actions in our product parser. Basten extends the NU test to identify a nonterminal that is the root cause of ambiguity [5]. One challenge of the NU test is choosing appropriate equivalence relations.

A brute-force way to test ambiguity is to enumerate all strings derivable from a given grammar and check for duplicates. This approach, used by AMBER [103], is accurate but prohibitively slow. Grammar filtering [7] combines this exhaustive approach with the approximative approach from the NU test to speed up discovery of ambiguities. AmbiDexter [6] uses parallel simulation similar to our approach, but on the state machine of an LR(0), grammar-filtered approximation that accepts a superset of the actual language. This allows false positives.

CFGAnalyzer [3] converts CFGs into constraints in propositional logic that are satisfiable if any nonterminal can derive an ambiguous phrase whose length is within a given bound. This bound is incremented until a SAT solver finds the constraints satisfiable.

CFGAnalyzer does report counterexamples, but never terminates on unambiguous input grammars even if there is a parsing conflict.

Schmitz's experimental ambiguity detection tool [102] for Bison constructs a nondeterministic automaton (NFA) of pairs of parser items similar to our product parser states. Its reports of detected and potential ambiguities remain similar to parsing conflict reports and hence difficult to interpret. Counterexample generation remains future work for Schmitz's tool. To obtain precise ambiguity reports for LALR(1) construction, this tool must resort to constructing NFAs for LR(1) item pairs.

SinBAD [119] randomly picks a production of a nonterminal to expand when generating sentences, increasing the chance of discovering ambiguity without exhaustively exploring the grammar. SinBAD's search still begins at the start symbol, so reported counterexamples might not identify the ambiguous nonterminal.

Counterexample generation Some additional attempts have been made to generate counterexamples that illustrate ambiguities or parsing conflicts in a grammar.

Methods for finding counterexamples for LALR grammars can be traced back to the work of DeRemer and Pennello [29], who show how to generate nonunifying counterexamples using relations used to compute LALR(1) lookahead sets. Unfortunately, modern implementations of parser generators do not compute these relations. Our method provides an alternative for finding nonunifying counterexamples without requiring such relations, and offers a bonus of finding unifying counterexamples when possible.

DMS [12] is a program analysis and transformation system whose embedded parser generator allows users to write grammars directly within the system. When a conflict is encountered, DMS uses an iterative-deepening [55] brute-force search on all grammar rules to find an ambiguous sentence [11]. This strategy can only discover counterexamples of limited length in an acceptably short time.

CUP2 [125] reports the shortest path to the conflict state, while prior versions of PPG [78] attempt to report nonunifying counterexamples. These parser generators often produce invalid counterexamples because they fail to consider lookahead symbols.

Menhir [94] is an LR(1) parser generator for the OCaml programming language that explains conflicts in terms of counterexamples like our approach, but only with nonunifying counterexamples. In other words, there is no guarantee that a unifying counterexample will be produced even if the grammar is ambiguous. Although Menhir does factor common derivation contexts when reporting counterexamples, it does not identify the innermost nonterminal that causes an ambiguity like in our approach. Furthermore, counterexamples in Menhir may be unclear if conflicts are caused by built-in nonterminals that shorthand common production descriptions, such as nullable and list productions. The lack of clarity is caused by the underlying implementation of these built-in nonterminals left unexplained in the counterexamples.

While less powerful than LR grammars, LL grammars can also produce conflicts. The ANTLR 3 parser generator [89] constructs counterexamples, but they can be difficult to interpret. For instance, ANTLR 3 provides the counterexample ⟨*digit*⟩⟨*digit*⟩ for the challenging conflict in Section 3.3.1[6]. Our technique describes the ambiguity more clearly.

Conflict resolution Generalized LR parsing [114] keeps track of all possible interpretations of the input seen so far by forking the parse stack. This technique avoids LR conflicts associated with having too few lookahead symbols but requires users to merge the outcomes of ambiguous parses at parse time. Our approach, which pinpoints ambiguities at parser construction time, is complementary and applicable to GLR parsing.

The GLR parsing algorithm is asymptotically efficient for typical grammars, but its constant factor is impractically high. Elkhound [69] is a more practical hybrid between

[6]The example grammar was modified to eliminate left recursion.

GLR and LALR parsing. The GLR algorithm is used only when forking parse stacks is necessary; otherwise, the usual LALR algorithm is used. Elkhound is almost as fast as LALR parsers yet supports a more general class of grammars. It can display different derivations of ambiguous sentences, but the user must provide these sentences.

The eyapp tool [97], a yacc-like parser generator for Perl, postpones conflict resolution until actual parsing. Users can write code that inspects parser states and provides an appropriate resolution.

SAIDE [91, 92] is an LALR parser generator that automatically removes conflicts arising from insufficient number of lookaheads, and attempts to detect ambiguities by matching conflicts with predefined patterns of known cases. Although this approach guarantees termination, conflicts could be miscategorized.

Dr. Ambiguity [8] provides diagnostics explaining causes of ambiguities as an Eclipse [30] plugin, but a collection of parse trees demonstrating ambiguities must be provided as input.

ANTLR 4 [90] uses textual ordering of productions as precedence and abandons static detection of conflicts. Textual ordering makes grammars less declarative, but ambiguous inputs can still exist; any ambiguities are discovered only at parse time.

3.9 Conclusion

Better tools that help language designers quickly find potential flaws within language syntax can accelerate the design and implementation of programming languages and promote the use of parser generators for problems involving custom data formats. Our method finds useful counterexamples for faulty grammars, and evaluation of the implementation shows that the method is effective and practical. This paper suggests that the undecidability of ambiguity for context-free grammars should not be an excuse for parser generators to give poor feedback to their users.

CHAPTER 4

RECONCILING EXHAUSTIVE PATTERN MATCHING WITH OBJECTS

In Chapter 2, we identified pattern matching as a useful language feature that would make the translations of language features easier to implement, and hence could simplify how compilers are written. While pattern matching is predominant in functional languages, it struggles to make way into object-oriented languages. This is because functional languages can exploit the availability of information on the concrete representations of instances of algebraic data types. On the contrary, abstraction in object-oriented programming hides this information from being inspected freely.

One important mechanism for ensuring the correctness of pattern matching is *exhaustiveness* verification—proving that every instance of a data type will be handled by some case when pattern-matched. Again, abstraction in object-oriented programming poses a challenge for checking exhaustiveness if pattern matching were made available.

In this chapter, we explore a language-design approach based on *modal abstraction* that reconciles safe pattern matching with data abstraction. This joint work with Andrew Myers appeared in the Proceedings of the 34$^{\text{th}}$ ACM SIGPLAN Conference on Programming Language Design and Implementation [49].

4.1 Introduction

Despite being an important feature of modern functional programming languages, pattern matching has not been adopted by most object-oriented languages. Data abstraction and extensibility, both primary goals of object-oriented languages, conflict with pattern matching. This work explores a language design for integrating pattern matching with object-oriented programming.

The following is a simple implementation of natural numbers in OCaml. The algebraic data type `nat`, with two constructors `Zero` and `Succ`, represents a natural number; the recursive `plus` function adds two naturals by matching them with one of three patterns.

```
type nat = Zero | Succ of nat

let rec plus m n =
  match (m, n) with
    (Zero, x)
  | (x, Zero) -> x
  | (Succ m', _) -> plus m' (Succ n)
```

This example illustrates two benefits of pattern matching in ML and other functional programming languages such as Haskell.

The first benefit is that patterns serve a dual role that enables algebraic reasoning and results in concise, intuitive code. A constructor such as `Succ` is also a pattern that matches the values produced by that constructor. Patterns can be nested to match complex values in a natural way, so a pattern like `Succ(Succ(n))` matches exactly the values constructed by expressions using the same syntax.

The second benefit is that pattern matching helps catch common programming errors. Patterns in a `match` expression can be checked to ensure that they are *exhaustive* and not *redundant*: that all possible values are matched by some pattern, and that every pattern can match some value. Without such checks, if the programmer forgot the first of the three cases above, the program could crash with an exception. With such checks, the compiler would warn that no cases match values of the form `(Zero, Succ _)`.

Relying on access to the concrete representation of data, however, makes the ML-style pattern matching inimical to data abstraction [121]. A value produced by one module can only be matched by patterns in another module if the second module knows the underlying representation of the value. Agreement on the concrete representation tightly

couples the two modules in a way usually considered undesirable for large software systems. For example, we might initially implement natural numbers as above, then later want to change the representation to be an `int`. This change is not possible in ML without breaking client code.

To make pattern matching compatible with data abstraction, prior work has developed pattern-matching constructs that can be implemented by arbitrary code. Examples of this approach include views [121], extractors [33], and active patterns [111]. These mechanisms permit matching on deep patterns over abstract data, but sacrifice other benefits of algebraic pattern matching. There is no check that patterns are consistent with their corresponding constructors, so algebraic reasoning is weakened. Further, data abstraction interferes with checking exhaustiveness and redundancy.

The JMatch language [61] introduced another way to harmoniously integrate pattern matching into object-oriented languages, through *modal abstractions* that support multiple directions of computation. Modal abstractions allow a constructor and its pattern to be implemented by the same invertible computation, ensuring that they are inverses. Determining whether patterns are exhaustive or redundant, however, remained impossible under the data abstraction provided by JMatch. Furthermore, the added expressive power of patterns implemented by complex computations means that programmers can accidentally omit patterns more easily than with algebraic data types.

The challenge for analysis of exhaustiveness and redundancy is to reason statically without violating data abstraction. The main contribution of this paper is, therefore, a way to extend modal abstractions with concise specifications that enable static reasoning about exhaustiveness and redundancy of pattern matching and, more generally, about the totality of computations.

Object-oriented programming involves more than just data abstraction; subtyping and inheritance are key ingredients supporting extensibility. For extensibility, different

implementations of (subtypes of) the same interface should support the same patterns without clients knowing which implementation has been used. We therefore introduce *named constructors* that can be used as patterns in this way. We also introduce two first-class *or-patterns* that generalize both data-type constructors and or-patterns in ML.

We proceed as follows. Section 4.2 reviews modal abstraction in JMatch. Section 4.3 introduces mechanisms that improve the expressive power of pattern matching and its integration with objects. Section 4.4 describes new static annotations that support reasoning about exhaustiveness and redundancy. The verification procedure is explained in Section 4.5. Section 4.6 describes our implementation of an extended version of JMatch. Using various code examples, we evaluate its expressiveness, analytic power, and efficiency in Section 4.7. Section 4.8 discusses related work, and Section 4.9 concludes.

4.2 Background

Some background will be helpful on JMatch [61, 62], an extension to Java 1.4 that supports pattern matching and iteration through modal abstraction.

4.2.1 Modal abstraction

Section 4.1 observed that in OCaml, natural numbers cannot support pattern matching while being represented internally with an `int`. Figure 4.1 shows how this can be done in JMatch. The key idea is that JMatch methods may declare multiple *modes* that correspond to different "directions" of evaluation, analogously to predicate mode declarations in the logic programming language Mercury [107]. In addition to the ordinary *forward mode*, which acts like a Java method, a JMatch method may also provide *backward modes* which, given a desired result, compute corresponding argument values. Backward modes

```
1  class Nat {
2    private int value;
3    private Nat(int n) returns(n)
4      ( value = n )
5    public static Nat zero() returns()
6      ( result = Nat(0) )
7    public static Nat succ(Nat n) returns(n)
8      ( result = Nat(n.value + 1) )
9  }
10 ...
11 static Nat plus(Nat m, Nat n) {
12   switch (m, n) {
13     case (zero(), Nat x):
14     case (x, zero()):
15       return x;
16     case (succ(Nat k), _):
17       return plus(k, Nat.succ(n));
18   }
19 }
```

Figure 4.1: Natural numbers with data abstraction in JMatch.

support pattern matching. For example, the method `succ` may be used in the forward mode to compute the successor of a number. As indicated by the clause `returns(n)` on line 7, it also has a backward mode that computes the number `n` for which the value given in `result` is the successor.

This implementation of `Nat` *is* more complex than in OCaml because the abstract view that supports pattern matching must be related to its concrete representation as an `int`. The methods of `Nat` demonstrates that JMatch programs can define patterns that both preserve data abstraction, because the field `value` is private, and are also usable outside the module that defines them. Lines 11–19 show how the backward modes of these methods can be used to implement the method `plus` similarly to the earlier OCaml code.

In general, a JMatch method implements a relation over its arguments and its result. Each of its modes is a different way of exploring the relation. For example, the `succ` operation is a binary relation on `Nat`, a subset of `Nat` $\times$ `Nat`. In each mode, some of

the arguments or the return value are *knowns* supplied by the caller, and the others are *unknowns* to be solved for.

Individual method modes may be implemented by imperative, Java-like code, but one single declarative-style implementation of multiple modes is often more concise. As in each of `Nat`'s methods in Figure 4.1, a declarative method implementation is a boolean formula placed inside parentheses, directly expressing the implemented relation. For example, the equation `result = Nat(n.value + 1)` at line 8 exactly captures the `succ` relation.[1] For each mode of a method, the compiler generates an imperative algorithm that, given values for knowns, finds values of all unknowns that satisfy the formula. Thus, the backward mode often comes nearly for free, unlike with related approaches such as extractors [33].

Not only user-defined abstractions but also built-in types such as primitive types support modal abstractions. For example, integer operations such as `+` and `-` can solve for either of their arguments, given a result to match against.

4.2.2 Iterative modes

Modes need not be functions; viewed as relations, they may be one-to-many or many-to-many. A mode is *iterative* when there may be more than one solution to the unknowns for given knowns; the keyword `iterates` is used in place of `returns` to indicate such a mode. For example, the `contains` method of the `Collection` class has the signature

```
boolean contains(Object x) iterates(x)
```

meaning that its backward mode can be used to iterate over all contained objects. Using iterative modes, the Java collections framework could be made 35% more concise by implementing its operations, including iterators, as modes of relatively few methods [62].

[1] Note that the operator = is an equality test, which unambiguously *subsumes* its usual Java role as imperative assignment.

As another simple example of implementing an iterator in this style, the following method could be added to class `Nat`.

```
boolean greater(Nat x) iterates(x)
  (this = succ(Nat y) && (y = x || y.greater(x)))
...
Nat n = ...;
foreach (n.greater(Nat x)) {
  ... // x in scope here
}
```

In the forward mode, the method tests whether `this` is greater than `x`. In the backward mode, it iterates over all numbers smaller than `this`, allowing code like the `foreach` loop that follows. The forward mode of `greater` is also its predicate mode because its return type is `boolean`.

In the body of `greater`, only one boolean formula needs to be expressed to define how both the forward and backward computations are carried out. In the forward mode, the JMatch compiler first generates the code that solves for `y`, and the subsequent boolean formula can then be evaluated directly; in the backward mode, the JMatch compiler generates the obvious recursive algorithm for finding all satisfying assignments to the output `x`.

The JMatch type system also checks that the multiplicity of solved unknowns matches their use in the mode declaration. In non-iterative modes, unknowns can only have one solution. For example, the use of disjunction in `greater` is permitted only because the backward mode is explicitly iterative.

Some built-in types have iterative modes. For example, array indexing can be inverted to obtain an iterator over the array elements.

4.2.3 Semantics and solving

The semantics of JMatch is defined as a syntax-directed, type-preserving translation to Java$_{\texttt{yield}}$ [60], which extends Java with *coroutine methods* in which control is yielded to the caller via the `yield` statement, much as in languages like C#, Ruby, and, originally, CLU [59]. The semantics of Java$_{\texttt{yield}}$ is defined similarly as a syntax-directed translation to Java. This translation is analogous to the CPS conversion. These translations also describe the JMatch compiler implementation, modulo some optimizations.

JMatch supports imperative Java code, the translation of which is relatively straightforward. The interesting parts of the translation involve the solving of boolean formulas and pattern expressions. JMatch considers a formula or pattern *solvable* when the compiler can generate an algorithm that either finds satisfying assignments to unknowns or determines that there are none. In the latter case, the formula or pattern is not *satisfiable* but is still solvable. A formula or pattern may also be satisfiable but not solvable if the compiler does not know how to generate an appropriate algorithm for determining satisfying assignments.

As an extension to Java, JMatch allows side effects, although its new features encourage a declarative programming style. With side effects, programmers need to reason about the order in which computations occur. The JMatch solver therefore solves formulas in a well-defined order that is left-to-right as much as possible.

After the JMatch solver determines the order of the unknowns to be solved, it generates the algorithm for solving formulas using the following inductively defined functions, which translate to Java$_{\texttt{yield}}$:

- $\mathcal{F}[\![f]\!]\ U\ s$ is the translation of a formula f. It is a sequence of Java$_{\texttt{yield}}$ statements that solve the formula f to find bindings for the unknowns in the set U. The translation executes statement s for each solution found.

- $\mathcal{M}[\![p]\!]\ U\ x\ s$ generates code to match a pattern p against a known value x and find bindings for the unknowns in the set U. The translation produces a statement that solves for the unknowns in U satisfying the formula $p = x$, and then executes statement s.
- $\mathcal{P}[\![p]\!]\ U\ w\ s$ is a pattern translation that solves for the value of the pattern p and its unknowns in the set U without a value to match against. The output code executes s for every solution. The statement s may refer to the unknowns in U, which are assigned a binding to produce the desired value for p, and the variable w, which is assigned the value of the pattern p itself.

Given these translation specifications, their definitions for the various language constructs are fairly straightforward and are available in full in [60].

For example, Figure 4.2 shows the translation of the formula `x - 2 = 1 + y`, where `x` is known and `y` is unknown. Figure 4.3(a) shows a pretty-printed version of the translation result. To confirm that the translation is correct, Figure 4.3(b) shows a hand-optimized version of the result. If s were the statement `System.out.println(y);`, then the value of `y` would be displayed to the console.

4.3 Pattern-matching extensions

We extend JMatch, adding new pattern-matching constructs to better support object-oriented programming and data abstraction and to increase expressive power in other ways.

4.3.1 Named constructors

In JMatch, pattern matching using procedures is successful only if the value being matched is either their result or one of their arguments. Therefore, a JMatch procedure

$$
\begin{aligned}
\mathcal{F}[\![\texttt{x - 2 = 1 + y}]\!]\ \{\texttt{y}\}\ s \;&= \texttt{int z;}\ \mathcal{P}[\![\texttt{x - 2}]\!]\ \emptyset\ \texttt{z}\ (\mathcal{M}[\![\texttt{1 + y}]\!]\ \{\texttt{y}\}\ \texttt{z}\ s) \\
&= \texttt{int z;}\ \mathcal{P}[\![\texttt{x - 2}]\!]\ \emptyset\ \texttt{z}\ (\\
&\quad\ \texttt{int y1, y2;}\ \mathcal{P}[\![\texttt{1}]\!]\ \emptyset\ \texttt{y1}\ (\\
&\quad\quad\ \texttt{\{ y2 = z - y1;}\ (\mathcal{M}[\![\texttt{y}]\!]\ \{\texttt{y}\}\ \texttt{y2}\ s)\ \texttt{\}}))) \\
&= \texttt{int z;}\ \mathcal{P}[\![\texttt{x - 2}]\!]\ \emptyset\ \texttt{z}\ (\\
&\quad\ \texttt{int y1, y2;}\ \mathcal{P}[\![\texttt{1}]\!]\ \emptyset\ \texttt{y1}\ (\\
&\quad\quad\ \texttt{\{ y2 = z - y1; y = y2;}\ \ s\ \ \texttt{\}})) \\
&= \texttt{int z;}\ \mathcal{P}[\![\texttt{x - 2}]\!]\ \emptyset\ \texttt{z}\ (\\
&\quad\ \texttt{int y1, y2; y1 = 1;} \\
&\quad\ \texttt{\{ y2 = z - y1; y = y2;}\ \ s\ \ \texttt{\})} \\
&= \texttt{int z; int x1, x2;}\ \mathcal{P}[\![\texttt{x}]\!]\ \emptyset\ \texttt{x1}\ (\\
&\quad\ \mathcal{P}[\![\texttt{2}]\!]\ \emptyset\ \texttt{x2}\ (\texttt{\{ z = x1 - x2;} \\
&\quad\quad\ \texttt{int y1, y2; y1 = 1;} \\
&\quad\quad\ \texttt{\{ y2 = z - y1; y = y2;}\ \ s\ \ \texttt{\}\}))} \\
&= \texttt{int z; int x1, x2; x1 = x; x2 = 2; \{} \\
&\quad\ \texttt{z = x1 - x2; int y1, y2; y1 = 1; \{} \\
&\quad\quad\ \texttt{y2 = z - y1; y = y2;}\ \ s\ \ \texttt{\}\}}
\end{aligned}
$$

Figure 4.2: The translation of the formula `x - 2 = 1 + y`, where `x` is known and `y` is unknown

```
int z;
int x1, x2;
x1 = x;
x2 = 2;
{
  z = x1 - x2;
  int y1, y2;
  y1 = 1;
  {
    y2 = z - y1;
    y = y2;
    s
  }
}
```

(a) Pretty-printed result

```
y = x - 2 - 1;
s
```

(b) Hand-optimized result

Figure 4.3: Pretty-printed and optimized versions of the translation result in Figure 4.2

```
interface Nat {
  constructor zero() returns();
  constructor succ(Nat n) returns(n);
  ...
}
```

Figure 4.4: Natural number interface with named constructors.

can successfully match on its own receiver object (`this`) only if the procedure is a constructor or happens to return its receiver object as the result. Since a constructor belongs to a particular class, code using a constructor pattern is tightly coupled to that particular implementation. This tight coupling interferes with extensibility and code reuse.

To support implementation-oblivious pattern matching, we extend JMatch with *named constructors* that can pattern-match an object whose run-time class is unknown. Named constructors have an explicit name different from that of their class, and they can be declared in interfaces.

For example, Figure 4.4 shows a `Nat` interface exposing two named constructors, `zero` and `succ`. Figure 4.5 shows two partial implementations of `Nat`. The first (`ZNat`) corresponds to the implementation of Figure 4.1. The second is analogous to the OCaml version and consists of two classes: `PZero`, representing zero, and `PSucc`, representing the successor of its field `pred` at line 18.

Named constructors can be invoked as if it is a static method to construct new objects of their class, as in the expression `ZNat.zero()`. In the forward mode, the fields of `this` are in scope as unknowns to be solved for either directly in the formula or via another constructor. For example, `val` in the equation `val = 0` at line 6 is solved directly by assigning zero to it. In addition, unlike ordinary constructors, a named constructor can also be invoked as if it is an instance method. When an object of type `Nat` or of any subtype of `Nat` is passed as a receiver object to the named constructor `zero`, `zero` acts as a boolean predicate. For example, `ZNat(0).zero()` evaluates to `true` because `ZNat(0)`

```
1  class ZNat implements Nat {
2    int val;
3    private ZNat(int n) returns(n)
4      ( val = n && n >= 0 )
5    constructor zero() returns()
6      ( val = 0 )
7    constructor succ(Nat n) returns(n)
8      ( val >= 1 && ZNat(val - 1) = n )
9    ...
10 }
11
12 class PZero implements Nat {
13   constructor zero() returns() ( true )
14   constructor succ(Nat n) returns(n) ( false )
15   ...
16 }
17 class PSucc implements Nat {
18   Nat pred;
19   constructor zero() returns() ( false )
20   constructor succ(Nat n) returns(n) ( pred = n )
21   ...
22 }
```

Figure 4.5: Three implementations of `Nat`.

```
1  class ZNat extends Nat {
2    ...
3    constructor equals(Nat n)
4      ( zero() && n.zero() | succ(Nat y) && n.succ(y) )
5  }
6
7  class PZero extends Nat {
8    ...
9    constructor equals(Nat n)
10     ( n.zero() )
11 }
12 class PSucc extends Nat {
13   ...
14   constructor equals(Nat n)
15     ( n.succ(pred) )
16 }
```

Figure 4.6: Equality constructors.

is zero; its implementation tests the equation `val = 0`. Finally, a named constructor declared in type T may be invoked without an explicit receiver object when it is used to pattern-match a value of type T. In this case, the receiver object is the value being matched against. For instance, we can use named constructors to write code like `plus` in Figure 4.1. The parameter `m` of the `plus` function becomes the implicit receiver of the pattern `succ(Nat k)` on line 16.

4.3.2 Equality constructors

As written, the implementations of `Nat` in Figure 4.5 are incomplete. The problem is that the forward mode of `succ` in `ZNat` promises to construct a `ZNat` from an arbitrary `Nat` predecessor `n`. If `n` is not a `ZNat`, the equality test at line 8 between `ZNat(val - 1)` and `n` will fail. We fix this by adding an operation to `Nat` that allows solving for equality between objects of different classes:

```
constructor equals(Nat n);
```

Because `equals` is used for creating objects of the class in which it is implemented, `equals` becomes a special named constructor—an *equality constructor*—rather than an ordinary boolean method as in Java. If defined, `equals` is used for solving equality in addition to JMatch's default strategy of direct assignment. The code of `equals` for the classes implementing `Nat` is given in Figure 4.6.

Using `equals`, the equality `ZNat(val - 1) = n` is solved for non-`ZNat` objects `n` by invoking `ZNat.equals`, defined at lines 3–4. This method tests whether `n` is zero or the successor of some number. If the former, it returns `ZNat.zero`; if the latter, it invokes `ZNat.succ` recursively to retrieve the predecessor of `n`, which is bound to `y` by the constructor invocation `n.succ(y)`. Operationally, `ZNat`'s `equals` and `succ` interoperate to find successive predecessors until either zero or a `ZNat` representation (as in `PSucc.succ(ZNat(3))`, which is legal!) is encountered. Once `equals` converts `n` to a `ZNat` object, `succ` matches the internal representation of this `ZNat` object with `val - 1`, solving for `val`, which internally represents the desired successor.

4.3.3 Other extensions

A complete overview of the existing patterns in JMatch can be found in Section 2.2 of the JMatch technical report [60]. We extend the language with additional operators and a new pattern that increase expressive power:

- JMatch already has a pattern conjunction operator called `as`, which generalizes ML's pattern operator of the same name by requiring two arbitrary patterns to match the same value. We add a pattern *disjunction* operator, `#`, that combines two patterns into a single pattern that matches either or both of the two, and solves for the same unknowns. For example, the formula `int x = y-1 # y+1` (which should be read as `int x = (y-1 # y+1)`) generates the two solutions `x = y-1` and `x = y+1` when solving for `x`, and `y`

```
1  public Expr CPS(Expr e) returns(e) (
2    Var k = freshVar("k", e) &&
3    (e, result) =
4      (Var(_),                          // ⟦v⟧ k ≜ k v
5         Lambda(k, Apply(k, e)))
6    | (Lambda(Var vl, Expr body),     // ⟦λx.e'⟧ k ≜ k (λxk'.⟦e⟧ k')
7         Lambda(k,
8           Apply(k, Lambda(vl, CPS(body)))))
9    | (Apply(Expr fn, Expr arg),      // ⟦e1 e2⟧ k ≜ ⟦e1⟧ (λf.⟦e2⟧ (λv.f v k))
10        Lambda(k, Apply(CPS(fn),
11          Lambda(f, Apply(CPS(arg),
12            Lambda(Var("v") as Var v,
13              Apply(Apply(f, v), k))))))
14          where Var f = freshVar("f", arg))
15 )
```

Figure 4.7: Invertible CPS conversion.

`= x+1` and `y = x-1` when solving for `y`. Unlike Icon's alternation expression [43], a match is attempted against all alternatives even if one of them fails.

- We also add a *disjoint* disjunction operator, `|`, that behaves like `#` except that the patterns must be disjoint. A pattern constructed with this operator produces at most one solution when a value is matched against it, unlike `#`. The number of solutions is important because the pattern-matching statements require that there be only a single solution. The compiler verifies that patterns combined via `|` are disjoint. The formula `x = 1 | 2` would therefore be legal but `x = y-1 | y+1` would not if used to solve for `y`.
- A *tuple pattern*, written `(`p_1`, ..., `p_n`)`, may be used to match multiple values at once. Tuples are not first-class values; uses of tuple patterns are equivalent to, but often more concise than, a set of equations expressed over the tuple components. Tuples are most helpful when used in conjunction with the `#` and `|` operators.

These new constructs add expressiveness. For example, the JMatch 1.1.6 release [60] includes an example of invertible conversion to continuation-passing style (CPS). The same two computations, CPS conversion and its inverse, are both expressed even more

concisely in Figure 4.7 using the new pattern operators. With the `CPS` method we can invert the CPS conversion by writing `let CPS(Expr source) = target` to obtain non-CPS source code corresponding to CPS code `target`. In this code, the use of tuples enables the translation rules to be expressed essentially as inference rules. The pattern (p `where` f) on line 14 refines pattern p to succeed only when formula f is also satisfiable. The use of `|` ensures that `CPS` is one-to-one, though not total in its backward mode. Without `|`, the JMatch compiler would be unable to conclude that the three cases are disjoint and would raise the error that `CPS` is not one-to-one.

4.4 Static annotations for exhaustiveness reasoning

Several pattern-matching forms in JMatch can benefit from verification of exhaustiveness. As we saw in Figure 4.1, `switch` statements are one such form. Whether `switch (`e`) {`$\overrightarrow{\texttt{case } p_i\texttt{: } s_i}$`}` is exhaustive corresponds to (roughly) whether

$$\bigvee_{i=1}^{n} e = p_i \tag{4.1}$$

is a tautology. A second such form is the JMatch statement `cond {`$\overrightarrow{\texttt{(}f_i\texttt{) \{}s_i\texttt{\}}}$`}`, which executes the first statement s_i such that its corresponding formula f_i is true. For exhaustiveness, at least one such formula must be true. A third pattern-matching form is `let` f, which is analogous to `cond {(`f`) {}}`, except that variable bindings made in f are in scope for the remainder of the statement's block. For example, the declaration `int x = 2` is syntactic sugar for `let int x = 2`. Since the values of these variables may be used later in the scope, the formula f in a `let` statement should always be satisfiable.

In principle, exhaustiveness checking seems simple. Reasoning about exhaustiveness while preserving data abstraction, however, is challenging because the client code performing pattern matching is oblivious to the concrete representation (e.g., private fields) of objects. For example, given the code in Figure 4.8, the compiler does not

```
Nat n;
...
switch (n) {
  case succ(Nat p): ...
  case succ(succ(Nat pp)):  ...
  case zero(): ...
}
```

Figure 4.8: Redundant switch statement.

know the implementation of `succ` and `zero` with which `n` will be matched. Even if it did know, using this knowledge would violate modularity, coupling correctness of this code to implementation choices internal to `Nat`. Moreover, given a value of type `Nat`, the compiler may not assume that `succ` and `zero` are the only ways to construct the value; there could be another constructor defined in `Nat` that could produce the same value. As a result, the compiler does not have enough information about the patterns to show that disjunction (4.1) is a tautology.

To enable the compiler to reason modularly about exhaustiveness, we must expose enough information to the client about the relation implemented by a method without exposing implementation details. Supplied with this information for the code in Figure 4.8, the compiler should be able to determine that all values of type `Nat` will be matched by some case. If that were not true (e.g., if the first case were omitted), the compiler should issue a warning. Also, in the code as written, the second case is redundant because anything matching `succ(succ(Nat pp))` must have matched `succ(Nat p)`. Redundant code often indicates errors in programmer's reasoning; the compiler ought to report this too. At the same time, the exposed information should let the compiler know that `zero` and `succ` are indeed disjoint and conclude that the third case is not redundant with the first two. Without such information, the compiler could generate a false redundancy warning.

To support static verification of exhaustiveness and other properties, three new kinds of concise and intuitive specifications provide the missing information: *class invariants*, *matches clauses*, and *ensures clauses*. As an orthogonal benefit, all of these specifications can exploit the new pattern operator | to prove patterns disjoint. We now explore these new features in more detail.

4.4.1 Class and interface invariants

One way to provide the information needed to determine exhaustiveness is as a *class* or *interface invariant*. For example, we can express that all instances of `Nat` match either `zero()` or `succ()` by adding the following invariant to the `Nat` interface, using | to assert that the two patterns are disjoint:

```
interface Nat {
  invariant(this = zero() | succ(_));
  ...
}
```

As another example, suppose we wanted to verify exhaustiveness of a `switch` statement like the following:

```
Nat n;
...
switch (n) {
  case ZNat z: ...
  case PZero _: ...
  case PSucc p: ...
}
```

Again, an invariant on `Nat` suffices (underscores here are wild cards on variables or patterns):

```
1 class ZNat implements Nat {
2   int val;
3   private invariant(val >= 0);
4   private ZNat(int n) matches(n >= 0) returns(n)
5     ( val = n && n >= 0 )
6   ...
7 }
```

Figure 4.9: Private invariant and `matches` clause.

```
invariant(this = ZNat _ | PZero _ | PSucc _);
```

These example invariants show how to can obtain the exhaustiveness analysis provided by algebraic data types, while preserving data abstraction and allowing extensibility. The first example permits new implementations of the `Nat` interface without invalidating the invariant. The second one prevents the definition of new classes that directly implement `Nat`; however, new subclasses of the three listed classes are permitted.

Class and interface invariants can be thought of as a kind of boolean-valued method whose value is always asserted to be true and whose implementation is visible to callers. Invariants may be given visibility modifiers (`public`, `protected`, or `private`). To maintain modularity, an invariant may only mention methods and fields that are at least as visible as the invariant itself.

Invariants not publicly visible may be useful for verifying the implementation of a class, such as the totality of the implementation of its methods. For instance, in the `ZNat` code of Figure 4.5, the field `val` cannot be negative. We can add a private invariant asserting this constraint, as in Figure 4.9. This invariant supports successful verification of the backward mode of the implementation of the constructor `ZNat()`, which should be total among all `ZNat` values. The invariant plus the first conjunct imply the second conjunct, `n >= 0`. The private invariant also helps verify both modes of `succ()`.

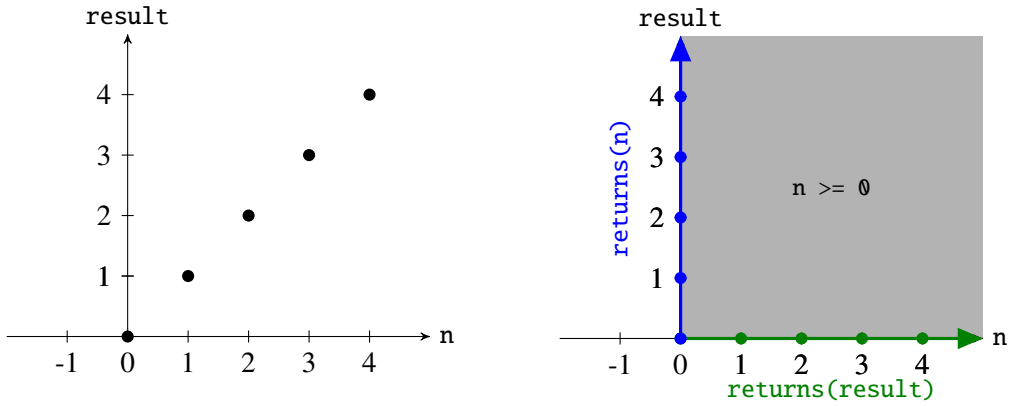
4.4.2 Matches clauses

One impediment to checking exhaustiveness is that a method mode may implement partial functions: on some inputs, its body might be unsatisfiable, in which case the method will fail rather than returning values for its unknowns. We extend the JMatch language with a way to specify when a method will successfully produce a result. This "matching precondition" is analogous to a precondition, but rather than specifying when a method call is legal, it specifies when pattern matching is guaranteed to succeed. The specification is conservative in that matching could succeed even when the condition does not hold.

For example, consider the constructor `ZNat()` in Figure 4.5. Any `ZNat` object must have a representation as a nonnegative integer. The corresponding matching precondition for the forward mode is `n >= 0`, meaning that for any nonnegative `n`, there exists a `ZNat` object matching that `n`. This matching precondition implies the constructor body, allowing successful verification of the forward mode. The backward mode of `ZNat()`, on the other hand, is total, corresponding to the matching precondition `true`.

Asking the programmer to specify matching preconditions for each mode would be verbose and repetitive, since different modes may share knowns (i.e., inputs). Our insight is that the programmer can write a single condition that captures when matching will succeed for the entire relation implemented by a method. We call this condition the *matches clause* for the method. Methods having no `matches` clause defaults to `matches(false)`, meaning that matching is not guaranteed to succeed for any input. The JMatch 2.0 compiler must extract the matching precondition for each mode from the consolidated `matches` clause. The extraction is described formally in Section 4.4.3; the rest of this section illustrates how extraction works for `ZNat()`.

The `matches` clause for `ZNat()` is shown in Figure 4.9. Figure 4.10(a) shows the actual relation implemented by `ZNat`; Figure 4.10(b) shows the `matches` clause, describing the



(a) actual relation (b) matching preconditions

Figure 4.10: The `ZNat` relation.

relation consisting of integral points in the shaded region. This relation can be viewed as an approximation to the true `ZNat` relation. Informally, the extraction obtains the matching precondition by projecting this relation onto the axis corresponding to an appropriate mode, obtaining matching preconditions shown as thick arrows. For the forward mode (`returns(result)`), the relation is protected onto the `n` axis, obtaining `n >= 0`. This corresponds to the fact that the creation of a `ZNat` object succeeds whenever `n` is nonnegative. For the backward mode (`returns(n)`), it is projected onto the `result` axis, obtaining `true`. This corresponds to the fact that the decomposition of a `ZNat` object always succeeds.

4.4.3 Extracting matching precondition from matches clause

In general, the body of a method implements some relation B and the `matches` clause specifies another relation M. Suppose that the method is a relation over a set of variables $\{\vec{x}\}$. For each mode $\mathcal{M}$ of the method, this set is partitioned into disjoint sets of knowns (inputs) $\{\vec{k}\}$ and unknowns (outputs) $\{\vec{u}\}$. We can then view the relations M and B as predicates over knowns and unknowns, $M(\vec{k}, \vec{u})$ and $B(\vec{k}, \vec{u})$, respectively. Given $\vec{k}$, the

precise condition in which the body guarantees success is therefore $\exists \vec{u}.\ B(\vec{k}, \vec{u})$. We call this formula the *precise matching precondition*.

For brevity, we define a function $\pi_{\mathcal{M}}$ that constructs the precise matching precondition for mode $\mathcal{M}$ by projecting an arbitrary predicate B onto the knowns:

$$(\pi_{\mathcal{M}} B)(\vec{k}) \stackrel{\Delta}{\Longleftrightarrow} \exists \vec{u}.\ B(\vec{k}, \vec{u})$$

Given $\mathcal{M}$ and B, $\pi_{\mathcal{M}} B$ is a predicate on $\vec{k}$ that holds when $\vec{k}$ provides some way to satisfy B in mode $\mathcal{M}$ and hence to successfully pattern-match.

To preserve abstraction, reasoning about exhaustiveness must be done using the `matches` clause M, not B. Intuitively, if $M \Rightarrow B$, the body will be satisfiable whenever the `matches` clause holds; however, to require this implication would be unnecessarily restrictive. In the case of `ZNat` relation, for example, `n >= 0` does *not* imply the actual relation, and the only `matches` clause that does so while preserving abstraction is `false`. A more useful correctness condition is $\bigwedge_{\mathcal{M}'} \pi_{\mathcal{M}'} M \Rightarrow \pi_{\mathcal{M}'} B$, where $\mathcal{M}'$ is ranges over declared modes. In other words, we only need the matching specification to imply the precise matching precondition for each mode actually available.

This suggests that given the mode $\mathcal{M} = \left(\left\{\vec{k}\right\}, \{\vec{u}\}\right)$, we should verify exhaustiveness by using $(\pi_{\mathcal{M}} M)(\vec{k})$ as the matching precondition. Unfortunately, the existential quantifiers in this formula make it ill-suited to automated reasoning. Instead, we construct a weakening of $\pi_{\mathcal{M}} M$ that does not mention existential quantifiers. Let us denote this weakened predicate on $\vec{k}$ as $\mathit{Extract}_{\mathcal{M}} M$.

The construction of $\mathit{Extract}_{\mathcal{M}} M$ proceeds as follows. We first convert the `matches` clause into negation normal form (NNF) so that the formula uses only positive logical operators over atomic formulas. We then use a variant of the usual JMatch algorithm for generating solutions to a formula. The first step is to reorder the atoms so that as many unknowns as possible can be solved from left to right. After this reordering, atoms that do not mention unknowns are left unchanged, as are atoms in which all unknowns

are solvable in the left-to-right order. Atoms mentioning any unsolvable unknowns are dropped; that is, they are replaced with `true`. Any remaining occurrences of unknowns can be thought of as existentially quantified, but because each remaining unknown is solvable, it represents a solution expressed entirely in terms of knowns.

For instance, in the backward mode of the `ZNat` example, the specified matching precondition `n >= 0` is existentially quantified as $\exists$`n.n >= 0` because `n` is an unknown. That is, the matching precondition is equivalent to `true`. Correspondingly, the extraction algorithm drops the atomic formula `n >= 0` because `n` is unsolvable, leaving only `true`. As another example, consider extracting a matching precondition for `x > 0 && y >= 0 && x+1 = y`, where `x` is unknown and `y` known. The formula is first reordered to allow solving for `x`, yielding `y >= 0 && x+1 = y && x > 0`. The first atom is left unchanged because it only mentions `y`. The second is also kept because it solves for `x`, allowing the third atom to be retained as well. Because `x` is solved by the value `y-1`, the extracted precondition is `y >= 0 && (y-1)+1 = y && (y-1) > 0`, which is equivalent to `y > 1`. In general, `x` might be solved by a user-defined method. Section 4.5 explains how atoms containing such unknowns are handled.

Dropping unsolvable atoms is a heuristic, but it seems effective because such atoms are typically satisfiable for all possible values of the knowns. In general, however, dropped atoms might not be satisfiable, in which case $Extract_{\mathcal{M}}M$ may not be conservative. For example, if the `matches` clause were instead `y >= 0 && x < y && x > 0`, dropping the atoms `x < y` and `x > 0` would result in the extracted precondition `y >= 0`. The precise matching precondition $(\pi_{\mathcal{M}}M)$(`y`) is rather `y >= 2`, since there is no satisfying assignment for `x` when `y` < 2.

$Extract_{\mathcal{M}}M$ can be used not only for analyzing exhaustiveness, but also for verifying that the method body implements its extracted precondition in mode $\mathcal{M}$. That is, when the method body is implemented as a formula, the compiler verifies in each mode $\mathcal{M}$ that

for all inputs $\vec{k}$, $(Extract_{\mathcal{M}}M)(\vec{k}) \Rightarrow (\pi_{\mathcal{M}}B)(\vec{k})$. This ensures soundness for exhaustiveness analysis done using $Extract_{\mathcal{M}}M$. Verification is done using an SMT solver, as described in Section 4.6. For imperative method implementations, this verification is left to the programmer, though existing program logics might be used to obtain a verifiable logical interpretation in many cases. Also left to future work is the extraction of more precise preconditions and conservative detection of unsoundness from the `matches` clause alone.

4.4.4 Opaquely refining matches

In general, we may want to support modes in which precondition extraction fails because the `matches` clause does not or cannot capture the relationship among the arguments. For example, consider adding to `ZNat()` a *predicate mode* `returns()`, in which there are no unknowns. In this mode, the `matches` clause `n >= 0` does not correctly capture the matching precondition, yet the existing implementation is correct. To support such modes, `matches` clauses may be refined using the special opaque predicate `notall`. During precondition extraction, an atom `notall`$(\overrightarrow{x_i})$ is treated as unsolvable if any of the variables x_i is unknown, and is therefore dropped; if all of the variables are known or already solved, however, the predicate is treated as `false`.

Thus, to support a predicate mode for `ZNat()`, the predicate `notall(result, n)` is conjoined with `n >= 0` to indicate that pattern matching is not guaranteed to succeed when both `result` and `n` are known. This `notall` predicate corresponds to the refinement that converts the gray area in Figure 4.10(b) into just the black dots in Figure 4.10(a). The opaque `notall` is needed because this refinement cannot be characterized abstractly.

4.4.5 Ensures clauses

Matches clauses are a kind of multimodal precondition. To improve the precision of verification and exhaustiveness reasoning in JMatch, we add the *ensures clause*, a

multimodal postcondition whose syntax resembles previous, unimodal postcondition specifications (e.g., [98, 109]). The `ensures` clause for a method is an abstraction of the relation implemented by the method, expressed in terms that client code can understand; that is, it only mentions names a legal caller could name, similar to the specifications proposed in JML by Leavens and Müller [57].

Unlike the `matches` clause, the `ensures` clause must define an overapproximation (a superset) of the implemented relation. Thus, in any context where a method call is known to have succeeded, the `ensures` clause can be assumed to hold with respect to the values supplied as knowns and the values returned as unknowns.

Because the clauses for both `matches` and `ensures` are often identical, the syntax `matches ensures(`f`)` may be used as a shorthand for `matches(`f`) ensures(`f`)`. For example, the constructor `ZNat()` from earlier might declare `matches ensures(n >= 0)`. Using `matches ensures` might cause the opaque predicate `notall` to appear in an `ensures` clause. Because the `ensures` clause overapproximates the implemented relation, treating both `notall` and its negation as true is sound when the clause is in NNF.

4.5 Checking exhaustiveness and totality

JMatch 2.0 must show the exhaustiveness of various pattern-matching statements (`switch`, `cond`, and `let`). Similar verification is required for methods with a `matches` or `ensures` clause, since they promise to succeed in each mode when the extracted matching precondition is true, and since the postcondition must hold if the methods succeed. In addition, both arms of `|` must be verified as disjoint. In each case, the analysis constructs quantifier-free formulas that can be satisfied only if some cases are not handled by the appropriate patterns or formulas.

Section 4.4 described verification informally while pretending that formulas can be verified directly, e.g., by an SMT solver. This is not true in general, because formulas may contain user-defined predicates that must be treated abstractly.

To aid in constructing formulas to be verified the SMT solver, we introduce an intermediate representation language $\mathbb{F}$ that is similar to the language of quantifier-free logical formulas. $\mathbb{F}$ is different from that of quantifier-free formulas in two respects. First, negations can only appear at the atomic level. This property is enforced by defining a function negate to negate formulas in $\mathbb{F}$. The operator $\neg$ is introduced and eliminated only by this function. Second, an additional right-associative *assume* operator $\triangleright$ can be used. Using the metavariable F to represent formulas in $\mathbb{F}$, $F_1 \triangleright F_2$ is a formula in which F_1 captures knowledge about the environment in which F_2 is being evaluated. Therefore, F_1 remains true even when $F_1 \triangleright F_2$ is negated, i.e.,

$$\mathsf{negate}(F_1 \triangleright F_2) \triangleq F_1 \triangleright \mathsf{negate}(F_2)$$

Intuitively, F_1 solves for some variable v appearing in F_2. $F_1 \triangleright F_2$ acts like substituting v appearing in F_2 with the solution from F_1. F_1 is usually an assignment to v, though in general it could solve for v via user-defined methods.

We need a little more notation to define the translations. We assume the function $\mathsf{fresh}(F)$ renames all unknown variables declared in $F \in \mathbb{F}$ to fresh ones. For a JMatch formula f and a JMatch pattern p, let μf and μp denote the set of variables declared in f and p, respectively. Furthermore, let $\mathsf{type}(x, \tau)$ be an $\mathbb{F}$-predicate representing the instantiation of the invariant associated with type τ on variable x.

Each formula is transformed to an $\mathbb{F}$-formula using three functions defined inductively on the syntax of formulas and patterns:

- $\mathcal{V}_{\mathcal{F}}$ is the transformation of a JMatch formula. It takes a formula f to be transformed, along with the set U of unknowns to be solved in f, and an additional $\mathbb{F}$ formula F

that represents the rest of the constraint. $\mathcal{V}_{\mathcal{F}}[\![f]\!]\ U\ F$ is a formula in $\mathbb{F}$ that holds if both of the following hold:

- f is satisfiable.
- F also holds under any solution to all $u \in U$ satisfying f.

- $\mathcal{V}_{\mathcal{M}}$ is the transformation of a JMatch pattern with a known value to match against. It takes a pattern p to be transformed, along with the set U of unknowns to be solved in p, a variable x representing the known value, and an additional $\mathbb{F}$ formula F that represents the rest of the constraint. $\mathcal{V}_{\mathcal{M}}[\![p]\!]\ U\ x\ F$ is a formula in $\mathbb{F}$ that holds if both of the following hold:
  - The formula $p = x$ is satisfiable.
  - F also holds under any solution to all $u \in U$ satisfying $p = x$.

- $\mathcal{V}_{\mathcal{P}}$ is the transformation of a JMatch pattern without a value to match against. It takes a pattern p to be transformed, along with the set U of unknowns to be solved in p, a variable w that will store the value of p, and an additional $\mathbb{F}$ formula F that represents the rest of the constraint and may mention w. $\mathcal{V}_{\mathcal{P}}[\![p]\!]\ U\ w\ F$ is a formula in $\mathbb{F}$ that holds if both of the following hold:
  - A solution to p exists.
  - F also holds under any solution to p, which is assigned to w, and to all $u \in U$ that are solved when p is solved.

These functions are similar in structure to the syntax-directed translation from JMatch to $\text{Java}_{\texttt{yield}}$ [60]. However, that prior translation only ensures that JMatch formulas are solvable, whereas the new translations, given solvability, also ensure satisfiability.

The types of $\mathbb{F}$-related functions are summarized in Figure 4.11, and their definitions are partially shown in Figure 4.12. For example, the translation $\mathcal{V}_{\mathcal{P}}[\![\texttt{x}]\!]\ \emptyset\ w\ F$ has the

$$
\begin{aligned}
\mathsf{negate} &: \mathbb{F} \to \mathbb{F} \\
\mathsf{fresh} &: \mathbb{F} \to \mathbb{F} \\
\mathsf{type} &: \mathsf{Var} \to \mathsf{Type} \to \mathbb{F} \\
\mathcal{V}_{\mathcal{F}} &: J_F \to \mathbb{U} \to \mathbb{F} \to \mathbb{F} \\
\mathcal{V}_{\mathcal{M}} &: J_P \to \mathbb{U} \to \mathsf{Var} \to \mathbb{F} \to \mathbb{F} \\
\mathcal{V}_{\mathcal{P}} &: J_P \to \mathbb{U} \to \mathsf{Var} \to \mathbb{F} \to \mathbb{F}
\end{aligned}
$$

where

$$
\begin{aligned}
J_F &= \text{set of JMatch formulas} \\
J_P &= \text{set of JMatch patterns} \\
\mathsf{Var} &= \text{set of variable names} \\
\mathbb{U} &= \mathcal{P}(\mathsf{Var}) \\
\mathsf{Type} &= \text{set of Java types}
\end{aligned}
$$

Figure 4.11: The types of $\mathbb{F}$-related functions.

following interpretation. Because w is yet to have a value to match against, it is assumed to have the same value as `x`. Then, assuming that w has the same type as that of `x`, F holds. As another example, consider the translation $\mathcal{V}_{\mathcal{F}}[\![p_1 \texttt{ = } p_2]\!]\ U\ F$. First, a solution to p_1 that solves for all the variables in $U \cap \mu p_1$ must exist. Such a solution is then assigned to a fresh variable `y`. Finally, `y` must equals p_2, which solves for the remaining variables in $U \setminus p_1$, and F holds for such solutions to p_2.

To see the transformation in action, Figure 4.13(a) shows how a constraint for the formula `x - 2 = 1 + y`, where `x` is known and `y` is unknown, is generated. Figure 4.13(b) shows a hand-optimized version of the transformation result, under the assumption that the predicate $\mathsf{type}(x, T_x)$ is already valid in the scope of the transformation and can be omitted. The transformation has the following interpretation: under the assumption that `y2` has the same value as $\texttt{x} - 2 - 1$, there are two proof obligations:

- `y2` satisfies the invariant associated with the type of `y`.
- F, under the further assumption that `y2` has the same value as `y`.

For the appropriate mode $\mathcal{M} = \left(\left\{\vec{k}\right\}, \{\vec{u}\}\right)$ of method $\texttt{m}(\vec{x})$, let $\mathsf{matches}(\texttt{m}, \vec{v})$ be an $\mathbb{F}$-predicate representing the instantiation of the `matches` clause in mode $\mathcal{M}$ of `m` with

Formula translations

$$
\begin{array}{rcl}
\mathcal{V}_{\mathcal{F}}[\![f_1 \texttt{ \&\& } f_2]\!]\ U\ F & \triangleq & \mathcal{V}_{\mathcal{F}}[\![f_1]\!]\ (U \cap \mu f_1)\ (\mathcal{V}_{\mathcal{F}}[\![f_2]\!]\ (U \setminus \mu f_1)\ F) \\
\mathcal{V}_{\mathcal{F}}[\![f_1 \texttt{ || } f_2]\!]\ U\ F & \triangleq & \mathcal{V}_{\mathcal{F}}[\![f_1]\!]\ U\ F \vee \mathcal{V}_{\mathcal{F}}[\![f_2]\!]\ U\ F \\
\mathcal{V}_{\mathcal{F}}[\![p_1 \texttt{ = } p_2]\!]\ U\ F & \triangleq & \mathcal{V}_{\mathcal{P}}[\![p_1]\!]\ (U \cap \mu p_1)\ \texttt{y}\ (\mathcal{V}_{\mathcal{M}}[\![p_2]\!]\ (U \setminus \mu p_1)\ \texttt{y}\ F) \\
 & & \quad \texttt{y fresh} \\
\mathcal{V}_{\mathcal{F}}[\![p_1 \texttt{ != } p_2]\!]\ U\ F & \triangleq & \mathcal{V}_{\mathcal{P}}[\![p_1]\!]\ (U \cap \mu p_1)\ \texttt{y1} \\
 & & \quad (\mathcal{V}_{\mathcal{P}}[\![p_2]\!]\ (U \setminus \mu p_1)\ \texttt{y2}\ (\texttt{y1} \neq \texttt{y2} \wedge F)) \\
 & & \quad \texttt{y1, y2 fresh} \\
\mathcal{V}_{\mathcal{F}}[\![p_1 \texttt{ <= } p_2]\!]\ U\ F & \triangleq & \mathcal{V}_{\mathcal{P}}[\![p_1]\!]\ (U \cap \mu p_1)\ \texttt{y1} \\
 & & \quad (\mathcal{V}_{\mathcal{P}}[\![p_2]\!]\ (U \setminus \mu p_1)\ \texttt{y2}\ (\texttt{y1} \leq \texttt{y2} \wedge F)) \\
 & & \quad \texttt{y1, y2 fresh} \\
\mathcal{V}_{\mathcal{F}}[\![\mathtt{m}(p_1, \ldots, p_\ell)]\!]\ U\ F & \triangleq & \mathsf{m}_{\mathcal{P}}(p_{k_1}, \ldots, p_{k_m}, U, \mathsf{matches}(\mathtt{m}, y_{k_1}, \ldots, y_{k_m}) \wedge \\
 & & \quad (\mathsf{ensures}(\mathtt{m}, y_1, \ldots, y_\ell) \triangleright \mathsf{m}_{\mathcal{M}}(p_{u_1}, \ldots, p_{u_n}, U', F))) \\
 & & \quad\quad y_{u_j}\text{'s fresh} \\
 & & \quad\quad y_{k_i}\text{'s and } U' \text{ are as defined} \\
 & & \quad\quad\quad \text{in the auxiliary functions}
\end{array}
$$

Auxiliary functions

$$
\begin{array}{rcl}
\mathsf{m}_{\mathcal{P}}(p_{k_1}, \ldots, p_{k_m}, U, F) & \triangleq & \mathcal{V}_{\mathcal{P}}[\![p_{k_1}]\!]\ U_1\ y_{k_1}\ (\mathcal{V}_{\mathcal{P}}[\![p_{k_2}]\!]\ U_2\ y_{k_2}\ (\ \ldots \\
 & & \quad \mathcal{V}_{\mathcal{P}}[\![p_{k_m}]\!]\ U_m\ y_{k_m}\ F \ldots\ )) \\
 & & \quad\quad y_{k_i}\text{'s fresh} \\
\mathsf{m}_{\mathcal{M}}(p_{u_1}, \ldots, p_{u_n}, U', F) & \triangleq & \mathcal{V}_{\mathcal{M}}[\![p_{u_1}]\!]\ U'_1\ y_{u_1}\ (\mathcal{V}_{\mathcal{M}}[\![p_{u_2}]\!]\ U'_2\ y_{u_2}\ (\ \ldots \\
 & & \quad \mathcal{V}_{\mathcal{M}}[\![p_{u_n}]\!]\ U'_n\ y_{u_n}\ F \ldots\ )) \\
 & & \quad\quad y_{u_j}\text{'s defined prior to } \mathsf{m}_{\mathcal{M}}\text{'s invocation}
\end{array}
$$

where

$$
\begin{array}{rrll}
V_1 = U, & V_i = V_{i-1} \setminus \mu p_{k_{i-1}} & & \text{for } 1 < i \leq m, \\
 & U_i = V_i \cap \mu p_{k_i} & & \text{for } 1 \leq i \leq m, \\
 & U' = V_m \setminus \mu p_{k_m}, & & \\
V'_1 = U', & V'_j = V'_{j-1} \setminus \mu p_{u_{j-1}} & & \text{for } 1 < j \leq n, \\
 & U'_j = V'_j \cap \mu p_{u_j} & & \text{for } 1 \leq j \leq n.
\end{array}
$$

Figure 4.12: Selected definitions of $\mathbb{F}$-related functions.

Pattern translations Let $T_{\mathtt{x}}$ be the type of $\mathtt{x}$.

$$
\begin{array}{rcl}
\mathcal{V}_{\mathcal{M}}[\![\mathtt{x}]\!]\ \emptyset\ x\ F & \triangleq & \mathsf{type}(x, T_{\mathtt{x}}) \wedge x = \mathtt{x} \wedge F \\
\mathcal{V}_{\mathcal{P}}[\![\mathtt{x}]\!]\ \emptyset\ w\ F & \triangleq & w = \mathtt{x} \triangleright \mathsf{type}(w, T_{\mathtt{x}}) \triangleright F \\
\mathcal{V}_{\mathcal{M}}[\![\_]\!]\ \emptyset\ x\ F & \triangleq & F \\
\mathcal{V}_{\mathcal{M}}[\![\mathtt{x}]\!]\ \{\mathtt{x}\}\ x\ F & \triangleq & \mathsf{type}(x, T_{\mathtt{x}}) \wedge (x = \mathtt{x} \triangleright F) \\
\mathcal{V}_{\mathcal{M}}[\![p_1\ \mathtt{as}\ p_2]\!]\ U\ x\ F & \triangleq & \mathcal{V}_{\mathcal{M}}[\![p_1]\!]\ (U \cap \mu p_1)\ x\ (\mathcal{V}_{\mathcal{M}}[\![p_2]\!]\ (U \setminus \mu p_1)\ x\ F) \\
\mathcal{V}_{\mathcal{P}}[\![p_1\ \mathtt{as}\ p_2]\!]\ U\ w\ F & \triangleq & \mathcal{V}_{\mathcal{P}}[\![p_1]\!]\ (U \cap \mu p_1)\ w\ (\mathcal{V}_{\mathcal{M}}[\![p_2]\!]\ (U \setminus \mu p_1)\ w\ F) \\
\mathcal{V}_{\mathcal{M}}[\![p_1\ +\ p_2]\!]\ U\ x\ F & \triangleq & \mathcal{V}_{\mathcal{P}}[\![p_1]\!]\ (U \cap \mu p_1)\ \mathtt{y1}\ (\\
 & & \quad \mathtt{y2} = x - \mathtt{y1} \triangleright (\mathcal{V}_{\mathcal{M}}[\![p_2]\!]\ (U \setminus \mu p_1)\ \mathtt{y2}\ F)) \\
 & & \qquad \mathtt{y1}, \mathtt{y2} \text{ fresh} \\
\mathcal{V}_{\mathcal{P}}[\![p_1\ +\ p_2]\!]\ U\ w\ F & \triangleq & \mathcal{V}_{\mathcal{P}}[\![p_1]\!]\ (U \cap \mu p_1)\ \mathtt{y1}\ (\\
 & & \quad \mathcal{V}_{\mathcal{P}}[\![p_2]\!]\ (U \setminus \mu p_1)\ \mathtt{y2}\ (w = \mathtt{y1} + \mathtt{y2} \triangleright F)) \\
 & & \qquad \mathtt{y1}, \mathtt{y2} \text{ fresh} \\
\mathcal{V}_{\mathcal{M}}[\![\mathtt{m}(p_1, \ldots, p_\ell)]\!]\ U\ x\ F & \triangleq & \mathsf{type}(x, T_{\mathtt{result}}) \wedge x = y_0 \wedge \\
 & & \quad \mathsf{m}_{\mathcal{P}}(p_{k_1}, \ldots, p_{k_m}, U, \\
 & & \qquad \mathsf{matches}(\mathtt{m}, y_0, y_{k_1}, \ldots, y_{k_m}) \wedge \\
 & & \qquad\quad (\mathsf{ensures}(\mathtt{m}, y_0, y_1, \ldots, y_\ell) \triangleright \\
 & & \qquad\qquad \mathsf{m}_{\mathcal{M}}(p_{u_1}, \ldots, p_{u_n}, U', F))) \\
 & & \qquad\qquad y_{u_j}\text{'s fresh} \\
\mathcal{V}_{\mathcal{P}}[\![\mathtt{m}(p_1, \ldots, p_\ell)]\!]\ U\ w\ F & \triangleq & \mathsf{m}_{\mathcal{P}}(p_{k_1}, \ldots, p_{k_m}, U, \mathsf{matches}(\mathtt{m}, y_{k_1}, \ldots, y_{k_m}) \wedge \\
 & & \quad (\mathsf{ensures}(\mathtt{m}, y_0, y_1, \ldots, y_\ell) \triangleright w = y_0 \triangleright \\
 & & \qquad \mathsf{type}(w, T_{\mathtt{result}}) \triangleright \mathsf{m}_{\mathcal{M}}(p_{u_1}, \ldots, p_{u_n}, U', F))) \\
 & & \qquad\qquad y_{u_j}\text{'s fresh} \\
 & & \qquad\qquad y_{k_i}\text{'s and } U' \text{ are as defined} \\
 & & \qquad\qquad\quad \text{in the auxiliary functions}
\end{array}
$$

Figure 4.12: Selected definitions of $\mathbb{F}$-related functions (continued).

$$
\begin{aligned}
\mathcal{V}_{\mathcal{F}}[\![\texttt{x - 2 = 1 + y}]\!]\ \{\texttt{y}\}\ F \;&= \mathcal{V}_{\mathcal{P}}[\![\texttt{x - 2}]\!]\ \emptyset\ \texttt{z}\ (\mathcal{V}_{\mathcal{M}}[\![\texttt{1 + y}]\!]\ \{\texttt{y}\}\ \texttt{z}\ F) \\
&= \mathcal{V}_{\mathcal{P}}[\![\texttt{x - 2}]\!]\ \emptyset\ \texttt{z}\ (\mathcal{V}_{\mathcal{P}}[\![\texttt{1}]\!]\ \emptyset\ \texttt{y1}\ (\\
&\qquad \texttt{y2} = \texttt{z} - \texttt{y1} \rhd (\mathcal{V}_{\mathcal{M}}[\![\texttt{y}]\!]\ \{\texttt{y}\}\ \texttt{y2}\ F)) \\
&= \mathcal{V}_{\mathcal{P}}[\![\texttt{x - 2}]\!]\ \emptyset\ \texttt{z}\ (\mathcal{V}_{\mathcal{P}}[\![\texttt{1}]\!]\ \emptyset\ \texttt{y1}\ (\\
&\qquad \texttt{y2} = \texttt{z} - \texttt{y1} \rhd (\mathsf{type}(\texttt{y2}, T_{\texttt{y}}) \wedge (\texttt{y2} = \texttt{y} \rhd F)))) \\
&= \mathcal{V}_{\mathcal{P}}[\![\texttt{x - 2}]\!]\ \emptyset\ \texttt{z}\ (\texttt{y1} = 1 \rhd \mathsf{type}(\texttt{y1}, T_1) \rhd (\\
&\qquad \texttt{y2} = \texttt{z} - \texttt{y1} \rhd (\mathsf{type}(\texttt{y2}, T_{\texttt{y}}) \wedge (\texttt{y2} = \texttt{y} \rhd F)))) \\
&= \mathcal{V}_{\mathcal{P}}[\![\texttt{x}]\!]\ \emptyset\ \texttt{x1}\ (\mathcal{V}_{\mathcal{P}}[\![\texttt{2}]\!]\ \emptyset\ \texttt{x2}\ (\\
&\qquad \texttt{z} = \texttt{x1} - \texttt{x2} \rhd (\texttt{y1} = 1 \rhd \mathsf{type}(\texttt{y1}, T_1) \rhd (\\
&\qquad\quad \texttt{y2} = \texttt{z} - \texttt{y1} \rhd (\\
&\qquad\qquad \mathsf{type}(\texttt{y2}, T_{\texttt{y}}) \wedge (\texttt{y2} = \texttt{y} \rhd F)))))) \\
&= \mathcal{V}_{\mathcal{P}}[\![\texttt{x}]\!]\ \emptyset\ \texttt{x1}\ (\texttt{x2} = 2 \rhd \mathsf{type}(\texttt{x2}, T_2) \rhd (\\
&\qquad \texttt{z} = \texttt{x1} - \texttt{x2} \rhd (\texttt{y1} = 1 \rhd \mathsf{type}(\texttt{y1}, T_1) \rhd (\\
&\qquad\quad \texttt{y2} = \texttt{z} - \texttt{y1} \rhd (\\
&\qquad\qquad \mathsf{type}(\texttt{y2}, T_{\texttt{y}}) \wedge (\texttt{y2} = \texttt{y} \rhd F)))))) \\
&= \texttt{x1} = \texttt{x} \rhd \mathsf{type}(\texttt{x1}, T_{\texttt{x}}) \rhd (\texttt{x2} = 2 \rhd \mathsf{type}(\texttt{x2}, T_2) \rhd (\\
&\qquad \texttt{z} = \texttt{x1} - \texttt{x2} \rhd (\texttt{y1} = 1 \rhd \mathsf{type}(\texttt{y1}, T_1) \rhd (\\
&\qquad\quad \texttt{y2} = \texttt{z} - \texttt{y1} \rhd (\\
&\qquad\qquad \mathsf{type}(\texttt{y2}, T_{\texttt{y}}) \wedge (\texttt{y2} = \texttt{y} \rhd F))))))
\end{aligned}
$$

(a) Transformation

$$\texttt{y2} = \texttt{x} - 2 - 1 \rhd (\mathsf{type}(\texttt{y2}, T_{\texttt{y}}) \wedge (\texttt{y2} = \texttt{y} \rhd F))$$

(b) Hand-optimized result

Figure 4.13: Constraint generation for the formula `x - 2 = 1 + y`, where `x` is known and `y` is unknown

values $\vec{v}$ in place of the known arguments $\vec{k}$, and let $\mathsf{ensures}(\mathtt{m}, \vec{v})$ be an $\mathbb{F}$-predicate representing the instantiation of the `ensures` clause of `m` with values $\vec{v}$ replacing *all* formal arguments $\vec{x} = \{\vec{k}, \vec{u}\}$. Denoting the `matches` and `ensures` clauses of `m` by M and E, and recalling that $Extract_{\mathcal{M}}(M)$ is a predicate over $\vec{k}$, we have:

$$\mathsf{matches}(\mathtt{m}, \vec{v}) \triangleq Extract_{\mathcal{M}}(M)(\vec{v})$$

$$\mathsf{ensures}(\mathtt{m}, \vec{v}) \triangleq (\mathcal{V}_{\mathcal{F}}[\![E]\!]\ \mu E\ \mathsf{true})\{\vec{v}/\vec{x}\}$$

If `m` requires a receiver object and has `result`, they are added to the definitions appropriately.

The Z3 theorem prover [28] is used to find a model satisfying these $\mathbb{F}$-formulas. This model can be used to construct a counterexample to explain the failure of exhaustiveness or totality to the user. The verification done by Z3 does not affect the dynamic semantics of JMatch 2.0; it only affects warnings given to the programmer.

4.5.1 Verifying exhaustiveness

Each `switch` statement

```
switch (v) { case p_i: s_i default: s }
```

is converted into a `cond` statement

```
T_v y = v;
cond { (y = p_i) {s_i} else s }
```

where T_v is the type of v and `y` is fresh. Thus, verification of `switch` statements reduces to that of `cond` statements.

To verify a `cond` statement `cond { (f_i) {s_i} else s }`, we begin by asserting the invariants of all the known variables in the context. We then proceed case by case. Let I_i be the invariant prior to the verification of the ith `cond` arm. The algorithm first checks

whether f_i yields a solution to its unknowns; that is, $I_i \wedge \mathcal{V}_{\mathcal{F}}[\![f_i]\!]\ \mu f_i$ true is satisfiable. If not, the compiler issues a warning that this arm is redundant. In either case, I_{i+1} is defined as $I_i \wedge$ negate(fresh($\mathcal{V}_{\mathcal{F}}[\![f_i]\!]\ \mu f_i$ true)). The updated invariant rules out patterns matched up to the current arm. The `else` arm, if present, generates the formula true.

Let I' be the invariant after all arms are checked. The `cond` statement is exhaustive if I' is unsatisfiable. If not, a counterexample is generated from a satisfying assignment, and a nonexhaustive warning is reported.

A `cond` statement can be used to refine patterns in the same way as a `where` pattern. Since both `switch` and `if` are syntactic sugar for `cond`, so can they. Let I be the invariant prior to the verification of a conditional case `(`f`) {`s`}`. We verify s with the stronger invariant $I' \triangleq I \wedge (\mathcal{V}_{\mathcal{F}}[\![f]\!]\ \mu f$ true).

To verify `let` f, we check whether negate($\mathcal{V}_{\mathcal{F}}[\![f]\!]\ \mu f$ true) is satisfiable. If so, a warning that the `let` statement may not always be total is reported to the programmer.

4.5.2 Verifying matching specifications

As described in Section 4.4.3, the bodies of methods are checked against the `matches` clause of the method to ensure that the body succeeds whenever the `matches` clause is true. Recall that this entails verifying the proposition $Extract_{\mathcal{M}} M \Rightarrow \pi_{\mathcal{M}} B$.

One complication is that the `matches` clause M of a method may refer to other methods. These method references may solve for unknown variables in M. In turn, these unknowns may be further referenced by other atoms in M, imposing additional matching preconditions.

The `matches` and `ensures` clauses of the referenced methods are used to resolve this complication. The `matches` clause imposes additional matching precondition to M, and the `ensures` clause constrains the values of unknowns that may be referenced later in M.

In the following example, the `matches` clause of method `bar` refers to `foo`:

```
int foo(int x)
  matches(x > 2) ensures(result >= x);
int bar(int y)
  matches(y > 0 && result = foo(y) && result < 4);
```

Now, suppose we want to extract `bar`'s matching precondition for the forward mode, i.e., when `y` is known. The reordering and atom-dropping procedure does not alter the `matches` clause. This means `bar(y)` succeeds if `y > 0`, `foo(y)` returns a result, and `foo(y) < 4`. The invocation of `foo` in `bar`'s `matches` clause succeeds if `y` > 2, and `foo`'s `ensures` clause says `result` $\geq$ `y`. Therefore, `bar(y)` is guaranteed to succeed if `y` $> 0 \wedge$ `y` $> 2 \wedge$ `result` $\geq$ `y` $\wedge$ `result` < 4, which is equivalent to `y` $= 3$.

We now give the formal translation for $Extract_{\mathcal{M}}M$, where $\mathcal{M} = \left(\left\{\vec{k}\right\}, \{\vec{u}\}\right)$. If $\hat{M}$ is the result of reordering and dropping atoms in M, and $\left\{\vec{\hat{u}}\right\} \subseteq \{\vec{u}\}$ is the set of unknowns remaining in $\hat{M}$, then we have

$$Extract_{\mathcal{M}}M \triangleq \mathcal{V}_{\mathcal{F}}[\![\hat{M}]\!] \left(\left\{\vec{\hat{u}}\right\} \cup \mu\hat{M}\right) \text{ true}$$

That is, we translate $\hat{M}$, where the variables to be solved for are those in $\left\{\vec{\hat{u}}\right\}$ and those declared in $\hat{M}$ itself. Similarly, the precise matching precondition is defined as

$$\pi_{\mathcal{M}}B \triangleq \mathcal{V}_{\mathcal{F}}[\![B]\!] \left(\{\vec{u}\} \cup \mu B\right) \text{ true}$$

With the above definitions, we are ready to formally define the verification conditions for JMatch methods. To verify a method

T_r `foo(`$\overrightarrow{T_i\ x_i}$`)` **matches**`(`$M$`)` **ensures**`(`$E$`)`

in mode $\mathcal{M}$ with body B, we prove these two assertions:

$$Extract_{\mathcal{M}}M \Rightarrow \pi_{\mathcal{M}}B \tag{4.2}$$

$$\pi_{\mathcal{M}}B \Rightarrow \mathcal{V}_{\mathcal{F}}[\![E]\!]\ \mu E \text{ true} \tag{4.3}$$

Assertion (4.2) says that if the extracted matching precondition for $\mathcal{M}$ holds, then B succeeds in generating a solution to all of its unknowns, which can be part of the arguments or declared in B itself. Assertion (4.3) says that if B succeeds in generating a solution, then the postcondition of the method holds. Equivalently, we show that

$$Extract_{\mathcal{M}}M \wedge \mathsf{negate}(\pi_{\mathcal{M}}B) \tag{4.4}$$

$$\pi_{\mathcal{M}}B \wedge \mathsf{negate}(\mathcal{V}_{\mathcal{F}}[\![E]\!]\ \mu E\ \mathsf{true}) \tag{4.5}$$

are unsatisfiable. A satisfying assignment for Assertion (4.4) means that the method body may not generate a solution to all of the unknowns when the matching precondition holds. A satisfying assignment for Assertion (4.5) means that the postcondition of the method may not hold when the body successfully generates a solution.

For a method declared in an interface or declared abstract, and for each mode $\mathcal{M}$ declared in that method, the assertion

$$Extract_{\mathcal{M}}M \Rightarrow Extract_{\mathcal{M}}E$$

is proven instead. Since the `matches` clause must specify an underapproximation of the (unimplemented) relation and the `ensures` clause an overapproximation, this assertion says that by transitivity, if the matching precondition holds, then the postcondition should hold as well. A satisfying assignment for $Extract_{\mathcal{M}}M \wedge \mathsf{negate}(Extract_{\mathcal{M}}E)$ means that the postcondition of the method may not hold when the matching precondition is true.

When a satisfying assignment is available in these verifications, a totality warning, that the method does not respect its specifications, is reported to the programmer.

4.5.3 Verifying disjoint patterns

JMatch 2.0 verifies multiplicity of formulas and patterns, ensuring that they generate at most one solution in non-iterative modes. Disjoint pattern disjunctions allow disjunctions

to be expressed without generating multiple solutions, but this property must be verified. We also overload the | symbol as a logical operator; the formula f_1 | f_2 is a disjunction that may be used only if at most one of f_1 or f_2 is satisfiable. Let U be the set of unsolved unknowns in p_1 | p_2, and let p_1' be the result of substituting each unsolved unknown in p_1 with a fresh variable, and similarly for p_2'. Patterns p_1 and p_2 are disjoint if $(\mathcal{V}_{\mathcal{F}}[\![x = p_1']\!]\ U\ \mathsf{true}) \wedge (\mathcal{V}_{\mathcal{F}}[\![x = p_2']\!]\ U\ \mathsf{true})$, where x is a fresh variable, is unsatisfiable. Similiarly, when | is used as a logical operator, formulas f_1 and f_2 are disjoint if $(\mathcal{V}_{\mathcal{F}}[\![f_1']\!]\ U\ \mathsf{true}) \wedge (\mathcal{V}_{\mathcal{F}}[\![f_2']\!]\ U\ \mathsf{true})$ is unsatisfiable.

Consider the examples in Section 4.3.3. The pattern `1 | 2` is disjoint because $x = 1 \wedge x = 2$ is unsatisfiable. The disjunction `y-1 | y+1` is disjoint when `y` is known. When `y` is unknown, the verification procedure renames `y` in each arm to a fresh variable, yielding $x = \mathtt{y}_1 - 1 \wedge x = \mathtt{y}_2 + 1$, which is satisfiable, so the compiler generates a warning.

4.5.4 Soundness

As in most functional programming languages, we consider failures of exhaustiveness not as errors but rather as a reason to warn the programmer. Our goal is to help programmers be *effective*. Therefore, some unsoundness or incompleteness may be tolerable or even desirable if it rarely limits or annoys the programmer. Our verification procedures establish two main sources of unsoundness, possibly leading to erroneous warnings or lack of warnings. An obvious source is that JMatch is an imperative language, yet the reasoning procedures described here do not take side effects into account. We do not consider this a serious problem because JMatch encourages a programming style in which side effects are used sparingly and are encapsulated inside data abstractions. A second source of unsoundness arises from recursively defined methods, which are discussed in Section 4.6.2. In some cases, the compiler may report that it cannot prove exhaustiveness or lack of redundancy. This does not seem to be a problem in practice.

4.6 Implementation

We have built a prototype implementation of JMatch 2.0 by extending the JMatch 1.1.6 compiler [60] to add the new pattern matching features in Section 4.3 and the static annotations in Section 4.4, and to use the Z3 theorem prover [28] to verify exhaustiveness, totality, and multiplicity.

4.6.1 Translating new features

Each named constructor `foo(...)` defined in class `C` is translated into two JMatch methods having the same visibility as that of `foo`. The first method is `boolean foo(...)` and handles all the modes where `result` is known. The other method is `static C create$foo(...)` and handles the remaining modes where a fresh `result` object needs to be created. For named constructors defined in an interface, the latter translation is omitted. An invocation of a named constructor is also transformed to use one of the translated methods accordingly, with the exception of invocations appearing in invariants and `matches` and `ensures` clauses. The invocations appearing as part of specifications are not part of the dynamic semantics, and can be used directly during verifications. An example translation of `Nat` and `PSucc` is shown in Figure 4.14.

In the JMatch implementation, when a variable w of type T_w is matched against a value x of type T_x, only an `instanceof` check is introduced if T_w is not a supertype of T_x. To use the equality constructor, JMatch 2.0 further checks whether an equality constructor accepting one argument of type T_x exists in the implementation of T_w and invokes it on x if the `instanceof` check fails.

```
interface Nat {
  boolean zero() returns();
  boolean succ(Nat n) returns(n);
  ...
}
class PSucc implements Nat {
  Nat pred;
  boolean zero() returns() ( false )
  static PSucc create$zero() ( false )
  boolean succ(Nat n) returns(n)
    ( pred = n )
  static PSucc create$succ(Nat n)
    ( result = PSucc() && result.pred = n )
  ...
}
```

Figure 4.14: Translation of named constructors.

4.6.2 Handling recursion

The verification functions defined in Section 4.5 unwind all method invocations appearing in a formula being translated into assertions expressed in terms of the `matches` and `ensures` clauses of the methods. In general, these translations may not be well-founded when the `matches` and `ensures` clauses of methods are mutually dependent, or in invariants of mutually recursive types. Nevertheless, the verification may be successful without fully unrolling all facts about method calls and types. We use Z3's external theory plugin to implement lazy assertions by introducing interpreted theory predicates and functions. Our external theory for Z3 expands facts about type invariants and about matching preconditions and postconditions only when instances of the theory predicates are assigned a truth value. For example, if an instance of the predicate on procedure invocation is assigned true, the `ensures` clause of the associated procedure is asserted on the procedure inputs, expanding the verification context with facts about the successful execution of the procedure. If the instance is assigned false, the negation of the `matches` clause is asserted, expanding the verification context with facts about the failure of the

procedure. An interpreted theory function is used to enforce the uniqueness of procedure outputs when the procedure is a (partial) function.

Because Z3 treats each asserted axiom as global, every instantiated axiom is asserted as an implication whose premise is the assigned predicate. Z3 also keeps track of every asserted theory predicate in its logical context, which allows proving exhaustiveness using class invariants without unrolling them entirely. To prevent unbounded unrolling, iterative deepening [55] is used to unroll as deeply as possible within a time budget. When our external theory plugin to Z3 does not further expand facts beyond the maximum depth, Z3 concludes that no satisfying assignment exists. If this happens when checking exhaustiveness, the compiler warns that it did not find a counterexample to exhaustiveness, but that there might be one.

4.7 Evaluation

Our evaluation of JMatch 2.0 aims to answer three kinds of questions:

- Is the extended language expressive? In particular, does it permit concise implementations? What annotation burden is incurred by programmers using the new verification procedures?
- Is the verification performed by our implementation effective on different kinds of code?
- What is the compile-time overhead of verification?

4.7.1 Code examples

We have evaluated our prototype JMatch 2.0 implementation on a variety of different coding problems. For each of these code examples, we have shown that the compiler correctly performs the three verification tasks described above, and we have measured

the time taken by verification and compared it to total compilation time. To evaluate expressiveness, we have also implemented each example as concisely as we could using Java.

Natural numbers The implementations of natural numbers shown earlier in the paper are also used for our evaluation.

Lists A JMatch 2.0 interface `List` for immutable lists is shown in Figure 4.15. We implement this interface in four very different ways: the empty list (`EmptyList`), regular cons lists (`ConsList`), snoc lists (`SnocList`) in which elements are appended to the end, and lists with an array representation (`ArrList`) in which the underlying array object can be shared among lists having the same suffix. More concretely, each `ArrList` has an index indicating the first element of the list in the underlying array; changing this index gives different views of the array as different lists. The underlying array is copied in the forward mode of `cons`, as the resulting list may break the shared-suffix invariant. To give the flavor of these implementations, the figure shows how the multimodal named constructor `snoc` is defined for `ConsList`. As the remaining code in the figure shows, these four list implementations interoperate smoothly, and list operations, even including `reverse`, can be used as patterns.

CPS We implement CPS conversion of a simple abstract syntax tree (AST) for lambda calculus; though Figure 4.7 shows only the key code, the implementation also includes AST classes.

Type inference We implement unification-based type inference over the same ASTs, augmented with type declarations. The code for type inference is placed within the AST node classes.

Trees A JMatch 2.0 interface `Tree` for binary trees is shown in Figure 4.16. We implement the AVL tree based on this interface. The `rebalance` method, also shown in the

```
interface List {
  invariant(this = nil() | cons(_, _));
  constructor nil() matches(notall(result));
  constructor cons(Object hd, List tl)
    matches(notall(result)) returns(hd, tl);
  constructor snoc(List hd, Object tl)
    matches ensures(cons(_, _)) returns(hd, tl);
  constructor equals(List l);
  constructor reverse(List l) matches(true) returns(l);
  boolean contains(Object elem) iterates(elem);
  int size();
}
constructor snoc(List h, Object t) // in ConsList
  matches ensures(cons(_, _)) returns(h, t) (
    h = EmptyList.nil() && cons(t, h)
  | h = cons(Object hh, List ht) && cons(hh, snoc(ht, t))
)
static int length(List l) {
  switch (l) {
    case nil(): return 0;
    case snoc(List t, _): return length(t) + 1;
    case cons(_, List t): return length(t) + 1;
      // detected as redundant
  }
}
List l = EmptyList.nil();  // l = []
l = SnocList.cons(0, l);   // l = [[], 0]
l = ConsList.snoc(l, 1);   // l = [0, [1, []]]
l =  ArrList.snoc(l, 2);   // l = [0, 1, 2]
l = ConsList.cons(3, l);   // l = [3, [0, 1, 2]]
let l = reverse(List r1);  //r1 = [2, [1, [0, [3, []]]]]
l =  ArrList.cons(4, l);   // l = [4, 3, 0, 1, 2]
let l = reverse(List r2);  //r2 = [2, 1, 0, 3, 4]
```

Figure 4.15: List interface and sample usage.

figure, returns the balanced version of the input subtree having `v` as the value at the root and `l` and `r` as its children. The invariant of `Tree` and the `ensures` clause of `branch` are crucial for the JMatch 2.0 compiler to verify that the formula in `rebalance` covers all the possible input subtrees. Checking the disjoint disjunctions also ensures that there is only one way to match each tree.

Collections We convert the prior JMatch reimplementation of the key collection classes from the Java collections framework [62] into JMatch 2.0. This code base includes implementations of various data structures: hash tables, red-black trees, and resizable arrays.

4.7.2 Expressiveness

We can assess the expressiveness of JMatch 2.0 by comparing the number of language tokens needed to implement each of the examples. The resulting token counts shown in Table 4.1 indicate that JMatch 2.0 code is considerably more concise than in Java: 42.9% shorter on average. This conciseness is largely due to the JMatch support for modal abstraction and for equality constructors.

4.7.3 Effectiveness

There are three new verification tasks. First, `switch` and related constructs (`let`, `cond`, etc.) should be exhaustive. Second, method implementations must be correct with respect to both their declared `matches` clause and their `ensures` clause. Third, disjoint disjunctions must indeed be disjoint, to verify multiplicity.

All of the examples shown in the table, and all prior examples shown in the paper, are successfully verified for exhaustiveness, (non-)redundancy, and multiplicity. The compiler caught several subtle exhaustiveness bugs during development of this code, such as

```
interface Tree {
  invariant(this = leaf() | branch(_,_,_));
  constructor leaf()
    matches(height() = 0)
    ensures(height() = 0);
  constructor branch(Tree l, int v, Tree r)
    matches(height() > 0)
    ensures(height() > 0 &&
      (height() = l.height() + 1 && height() > r.height()
        || height() > l.height() && height() = r.height() + 1))
    returns(l, v, r);
  int height()
    ensures(result >= 0);
}

static Tree rebalance(Tree l, int v, Tree r) // in AVLTree
    matches(true) (
  result = Branch(Branch(Tree a, int x, Tree b),
                  int y,
                  Branch(Tree c, int z, Tree d))
    && ( // rotation from left
         l.height() - r.height() > 1 && d = r && z = v
           && ( // case 1: single rotation
                l = branch(Tree ll, y, c) &&
                ll = branch(a, x, b) && ll.height() >= c.height()
              | // case 2: double rotation
                l = branch(a, x, Tree lr) &&
                lr = branch(b, y, c) && a.height() < lr.height())
       | // rotation from right
         r.height() - l.height() > 1 && a = l && x = v
           && ( // case 3: double rotation
                r = branch(Tree rl, z, d) &&
                rl = branch(b, y, c) && rl.height() > d.height()
              | // case 4: single rotation
                r = branch(b, y, Tree rr) &&
                rr = branch(c, z, d) && b.height() <= rr.height()))
  | abs(l.height() - r.height()) <= 1 && result = Branch(l, v, r)
)
```

Figure 4.16: Tree interface and the AVL tree `rebalance` method, which uses the interface to check for totality.

Implementation	JMatch	Java	w/o verif	w/ verif
`Nat`	41 (21)	29	0.100	0.104
`PZero`	85	189	0.258	0.331
`PSucc`	98	226	0.280	0.435
`ZNat`	161	319	0.377	0.459
`List`	114 (72)	91	0.129	0.123
`EmptyList`	164	455	0.416	0.510
`ConsList`	309	1007	0.807	2.47
`SnocList`	311	1006	1.05	3.36
`ArrList`	473	1208	0.864	1.90
`Expr`	96 (57)	80	0.710	0.846
`Variable`	192	434	0.689	0.852
`Lambda`	239	500	1.20	1.52
`TypedLambda`	86	92	1.38	1.57
`Apply`	232	506	1.15	2.31
`CPS`	325	1279	7.88	8.37
`Type`	154	187	0.218	0.307
`BaseType`	73	163	0.350	0.443
`ArrowType`	82	189	0.357	0.444
`UnknownType`	154	245	0.372	0.490
`Environment`	211	310	0.695	0.862
`Tree`	114 (44)	69	0.165	0.170
`Leaf`	124	351	0.420	0.510
`Branch`	202	553	0.529	0.682
`AVLTree`	439	720	2.84	9.01
`ArrayList`	773	1098*	1.67	1.81
`LinkedList`	886	1232*	2.00	2.20
`HashMap`	1082	1874*	3.41	3.66
`TreeMap`	3606	3955*	5.90	6.43

Table 4.1: The number of tokens for implementations in JMatch 2.0 versus Java. Interface token counts are reported both with and without (in parentheses) `matches` and `ensures` clauses. Verification overhead is given in seconds as the average of 24 runs, with a standard deviation of at most 15%. Some comparisons (*) are versus a PolyJ [75] implementation that is more concise than the Java one. For example, the PolyJ `TreeMap` is 20% shorter than the Java equivalent [62].

incorrect order of arguments to methods and invocation to an unexpected implementation of overloaded or overridden methods. In case of `TreeMap`, the absence of red-black tree invariants results in a nonexhaustive warning in the `balance` method.

4.7.4 Efficiency

Table 4.1 shows that verification time is reasonable for all of the code examples, even with our unoptimized prototype implementation. The reported numbers include compilation time of dependencies but exclude the overhead of initializing the compiler (689 ms) and the Z3 solver (680 ms). On average, the verification overhead on the evaluated code is 37.5% compared to the regular compilation time.

The speed of verification is not surprising, because verification is performed one method at a time. Verification is simple and tractable because the abstraction mechanisms we introduced to JMatch allow both programmers and the SMT solver to reason locally about code.

Because JMatch 2.0 does not significantly change the dynamic semantics of JMatch, the translation to Java is essentially unchanged. The performance of the compiled programs is therefore similar to one in the previous evaluation [62].

4.8 Related work

Integrating pattern matching with objects and data abstraction has been the subject of quite a few research efforts.

Case classes in the Scala programming language [84], as in Pizza [81], provide pattern matching by allowing case-class constructors in case arms. Scala uses *sealed classes* to limit the number of case classes that can inherit them. This makes exhaustiveness easy to verify, but sacrifices extensibility because only one implementation is allowed

per declaration of a sealed class. Our `invariant` declaration achieves the same level of exhaustiveness checking but allows programmers to extend classes freely. Closely related approaches include extensible algebraic data types [126] and polymorphic variants [41], which support some extensibility and deep pattern matching, but tie pattern matching to the data representation more than is ideal.

Wadler's views [121] were an early, influential generalization of pattern matching. Views require an explicitly defined bijection between the abstract view and the representation. Unlike in our language, views do not reconcile pattern matching with subtyping and do not allow matching without knowing the identity of the implementation.

Extractors are introduced in [33] as an alternative to case classes that is compatible with data abstraction. Each extractor contains `apply` and `unapply` methods, called implicitly during construction and pattern matching. There is no check that these methods are inverses, however. Modal abstractions in JMatch are less verbose and reduce the chance of such errors. No exhaustiveness checking was proposed for extractors. Dotta et al. [31] verify extractors by relying on sealed classes, and support user-defined constructor patterns. Their work does check for pattern disjointedness; abstraction prevents us from making this guarantee.

Active patterns in F# [111] are similar to extractors, but support exhaustiveness checking by allowing the declaration that a set of patterns is complete. Because they offer only a backward mode, they do not support algebraic reasoning in the same way as modal abstractions. They also do not support object-oriented extensibility.

The RINV language of Wang et al. [123] also uses invertible computation to implement pattern matching that is compatible with data abstraction. Rather than extracting computations from a logical characterization of the computation, RINV instead uses a restricted language for abstraction functions that guarantees invertibility. These functions are bidirectional rather than fully multimodal and do not support iterative modes. RINV

analyzes exhaustiveness via specifications of complete sets of constructors, but does not verify these specifications. RINV supports neither subtyping nor extensibility.

Suter et al. [109] also use abstraction functions to reduce algebraic data types to abstract values such as multisets, and use known theories of these abstract values to reason about data types. Methods may be annotated with a postcondition in terms of abstraction functions. Leon [110] extends this reasoning to recursive programs. Used in conjunction with sealed classes, these decision procedures assist in a more precise analysis of pattern exhaustiveness by taking type refinement into account. These decision procedures do not support modal abstraction.

An orthogonal approach to integrating pattern matching into object-oriented languages is predicate dispatch [36, 71], which extends multimethods with the ability to choose an implementation based on general predicates over the arguments. Predicate dispatch appears to be largely orthogonal and complementary to the pattern matching mechanisms described here. The predicates in prior work on predicate dispatch are, however, less expressive than those we have explored here. OOMatch [96] uses pattern matching in predicate dispatch. Its deconstructors are similar to the backward mode of JMatch constructors. OOMatch's pattern matching differs in that it can appear only in method headers as part of predicate dispatch, and no separation of specification and implementation is provided. HydroJ [58] uses predicate dispatch to express extensible communication patterns in distributed systems; however, pattern matching is done over concrete data structures called tagged trees.

Matchete [45] extends Java with pattern matching operators similar to extractors, but matches on regular expressions and other specialized expressions. It does not analyze exhaustiveness.

The Thorn language integrates patterns to make code more concise and robust [16]. Its rich set of patterns includes boolean combinations of patterns, general list patterns,

regular expressions, and first-class patterns. First-class patterns in Thorn provide pattern abstraction that supports evolution of the data structure used in pattern matching, but Thorn does not support multiple implementations. As a dynamic language, Thorn does not check exhaustiveness.

Harmony and the Boomerang language [18, 39] support bidirectional computations over trees and strings through domain-specific lens combinators. The types in these languages support reasoning about the totality of transformations in these domains, but data abstraction is not a feature of these languages.

JMatch uses a simple solver to convert logical formulas to algorithms that do pattern matching and iteration. Integrating more sophisticated solvers would be an interesting future direction. Examples of this approach include Squander [70], which uses an SMT solver to synthesize code, and Juno 2 [77], which integrates a numerical solver.

One focus of research on pattern matching has been on how to generate efficient code that shares computation across different patterns (e.g., [56]). Such optimizations are orthogonal to this work.

4.9 Conclusion

A clean integration of pattern matching into the object-oriented setting could simplify many programming tasks. Prior work has not managed to provide expressive pattern matching with strong data abstraction and subtyping, along with statically checked exhaustiveness. This is the first work that manages to combine these important features. We improved the integration of pattern matching with object-oriented programming, yet showed that even with this more powerful pattern matching, it is possible to reason statically about exhaustiveness, redundancy, totality, and multiplicity.

The most important insight was that programmers need to be able to specify the precondition for successful pattern matching in an abstract way. We showed that it is

possible to do this while keeping the annotation burden low, by automatically extracting matching preconditions. The specification techniques introduced may be helpful for other models of multidirectional computation.

Another insight was that pattern matching can integrate with object-oriented programming by treating constructors as methods that solve for the fields of the created object, and by viewing equality itself as a constructor that shifts views between different implementations of the same abstraction.

CHAPTER 5

CONCLUSION

The ability to add new features to programming languages is much in demand, for both design experimentation and domain-specific language extensions. However, traditional monolithic compilers make small language extensions hard to implement and to maintain. We argue that compilers should be implemented as a collection of modularly extensible and composable transformations, so application developers can choose the language features they need and language designers can experiment freely with different feature combinations. This dissertation explores new design patterns for constructing a compiler so that it can be extended in a modular way and its extensions can be merged with little effort. These design patterns are implementable in a mainstream programming language, Java. A new AST representation allows a single AST to represent programs in multiple programming languages. A new dispatch mechanism supports changes to the relationships between AST node types. Compiler passes are independent of their source language, making translations reusable and composable. Our experience with creating compilers for dozens of language variants, by applying both extension and composition, shows that the design pattern offers effective machinery for language design and implementation. The new design pattern may also guide development of future mainstream language constructs.

Meanwhile, writing a parser for the compiler front end remains remarkably painful. Automatic parser generators offer a powerful and systematic way to parse complex grammars, but debugging conflicts in grammars can be time-consuming even for experienced language designers. Better tools for diagnosing parsing conflicts will alleviate this difficulty. This dissertation proposes a practical algorithm that generates compact, helpful counterexamples for LALR grammars. For each parsing conflict in a grammar, a counterexample demonstrating the conflict is constructed. When the grammar in question

is ambiguous, the algorithm usually generates a compact counterexample illustrating the ambiguity. This algorithm has been implemented as an extension to the CUP parser generator. The results from applying this implementation to a diverse collection of faulty grammars show that the algorithm is practical, effective, and suitable for inclusion in other LALR parser generators.

Finally, pattern matching, an important feature of functional languages that proves useful for implementing translation passes in compilers, is in conflict with data abstraction and extensibility, which are central to object-oriented languages. Modal abstraction offers an integration of deep pattern matching and convenient iteration abstractions into an object-oriented setting; however, because of data abstraction, it is challenging for a compiler to statically verify properties such as exhaustiveness. We extends modal abstraction in the JMatch language to support static, modular reasoning about exhaustiveness and redundancy. New matching specifications allow these properties to be checked using an SMT solver. We also introduce expressive pattern-matching constructs. Our evaluation shows that these new features enable more concise code and that the performance of checking exhaustiveness and redundancy is acceptable.

We hope that better tools, more expressive language design, and new language mechanism introduced in this dissertation will bring composable compiler implementations closer to a practical reality.

5.1 Future directions

Contributions in this dissertation lay a foundation for the following possible areas of future research that could make writing compilers an even more enjoyable experience.

5.1.1 Composable integrated development environments

Integrated development environments (IDEs) play an important role in making large software projects manageable. IDEs such as Eclipse [30] and IntelliJ IDEA [87] offer syntax highlighting, project navigation, autocompletion of code, code refactoring, and debugging platform that speed up the implementation process and help with testing software. Not only are IDEs helpful for large projects, but the availability of IDEs can also make new programming languages easier to learn, as users can play with various language features and potentially discover more features with the help of various tools that IDEs provide.

Despite the importance of IDEs, they are usually not released with compilers. One reason is the high cost of implementing IDEs, so investing human hours into coding an environment that may or may not be used by the general programming community poses a risk. Rather, there is often a gap between the initial release of a new programming language, to gauge its popularity, before IDEs are designed, implemented, and released. But this means designers of new programming languages need to work even harder to make the new languages well known and widely used. A better way to simplify IDE implementations can encourage more domain-specific developments along the line of the language toolbox approach we have proposed.

Some effort has been made to improve this process. Eclipse provides the ability to attach plugins as a way to extend the base IDE [13]. These *extension points* work in the same way as extensible compilers, where plugins can extend on top of one another. Like the compiler composability problem, however, composing independently developed plugins can be painstaking. Our design patterns that address the composability problem would be a good starting point in tackling composability of IDEs.

One significant challenge when work with IDEs, unlike with compilers, is that the existing code for the base IDE poses additional constraints for applying our design-

pattern approach. In implementing our composable compilers, we had the freedom of structuring the compilers in any way we would like. That is, composable compilers can be implemented from scratch. On the contrary, given an IDE implementation, we no longer have full freedom to change the base IDE. A more restricted set of design patterns appears necessary for composition to work with existing developments. Nevertheless, our design-pattern proposal in a mainstream language suggests that existing languages for implementing plugins might already be enough to make this work.

5.1.2 Modal abstraction and parsing

Parsing is the first important step of the compilation process, where source programs are transformed into ASTs that can be processed by compilers. At the other end of the process, pretty-printing is a critical step if the resulting programs are to be inspected by humans, for example, in an IDE. Line breaks, indentations, and proper spacing play a role in determining how understandable the code is. Poorly formatted code, however simple, can be incomprehensible.

Parsing and pretty-printing are closely related. Ignoring whitespace, given an AST constructed from parsing a program, pretty-printing this AST should result in the identical program. In other words, parsing and pretty-printing should be an invertible process. Prior work such as invertible syntax descriptions [95], FliPpr [66], and object grammars [116] has proposed ways to make parsing and pretty-printing invertible. FliPpr even takes screen widths into account to produce different printouts. In these approaches, indentations and spacing are encoded so that pretty-printing outputs the "nicest" format whenever possible.

But the beauty of pretty-printing is subjective. Some programmers want an open brace immediately after the conditional expression in an `if` statement; others want it on the next line. Tabs or spaces? Tab size? How many spaces for a level of indentation?

Not everybody can agree on these questions, so it is best to leave these choices to individual programmers. What we need is an invertible parsing and pretty-printing process that accepts configurable preferences for desired formatting. The Eclipse IDE already provides such configurations to some extent, but parsing remains decoupled from pretty-printing. As a generalization of invertible computation, modal abstraction is a possible candidate for reconciling parsing with configurable pretty-printing.

APPENDIX A

CODE LISTINGS FOR COMPOSABLE COMPILER IMPLEMENTATION

A.1 Core implementation of the Lang interface

```
1  /**
2   * A {@code Lang} represents a programming language. Any programming language
3   * has zero or more parents as immediate superlanguages. A language without a
4   * parent is a base language that does not extend any other language. A
5   * language with more than one parent is a composition of its immediate
6   * superlanguages.
7   */
8  public interface Lang {
9
10     /** The name of this programming language. */
11     String name();
12
13     /** The immediate superlanguages of this language. */
14     Set<Lang> superLangs();
15
16     /** The superlanguages of this language. */
17     Set<Lang> allSuperLangs();
18
19     /**
20      * Return true if {@code lang} is a superlanguage of this language;
21      *         false otherwise
22      */
23     boolean isSublanguage(Lang lang);
24
25     /** Node class factory for this language. */
26     NodeClassFactory nodeClassFactory();
27
28     /** Operator factory factory for this language. */
29     default OperatorFactoryFactory opFactoryFactory() {
30        return null;
31     }
32
33     /**
34      * Available target languages that this languages can be
35      * directly translated to.
36      */
37     Collection<Lang> targetLangs();
38
39     /** partial order for comparing languages in the hierarchy */
40     PartialOrder<Lang> partialOrder = new PartialOrder<Lang>() {
41        @Override
42        public boolean comparable(Lang e, Object other) {
43           if (other instanceof Lang) {
```

```
44              Lang l = (Lang) other;
45              return e.isSublanguage(l) || l.isSublanguage(e);
46          }
47          return false;
48      }
49
50      @Override
51      public boolean moreSpecificThan(Lang e, Object other) {
52          if (!comparable(e, other))
53              throw new UnsupportedOperationException();
54          Lang l = (Lang) other;
55          return e.isSublanguage(l);
56      }
57  };
58
59  abstract class Class implements Lang {
60      protected final Set<Lang> superLangs;
61      /* cache of superlanguages */
62      protected Set<Lang> allSuperLangs;
63
64      protected Class(Lang... superlangs) {
65          Set<Lang> superLangs = new HashSet<>();
66          for (Lang superLang : superlangs)
67              superLangs.add(superLang);
68          this.superLangs = Collections.unmodifiableSet(superLangs);
69      }
70
71      @Override
72      public String name() {
73          ...
74      }
75
76      @Override
77      public final Set<Lang> superLangs() {
78          return superLangs;
79      }
80
81      @Override
82      public final Set<Lang> allSuperLangs() {
83          if (allSuperLangs == null) {
84              // initialize cache
85              Set<Lang> superLangs = superLangs();
86              Set<Lang> result = new HashSet<>(superLangs);
87              for (Lang lang : superLangs)
88                  result.addAll(lang.allSuperLangs());
89              allSuperLangs = Collections.unmodifiableSet(result);
90          }
91          return allSuperLangs;
92      }
93
94      @Override
95      public final boolean isSublanguage(Lang lang) {
```

```
96              return allSuperLangs().contains(lang);
97          }
98
99          @Override
100         public final Collection<Lang> targetLangs() {
101             ...
102         }
103     }
104 }
```

A.2 Core implementation of the `NodeClass` interface

```
1  public interface NodeClass {
2      /** Immediate superclasses of this node class. */
3      Set<NodeClass> superclasses(NodeClassFactory af);
4
5      /** All superclasses of this node class. */
6      Set<NodeClass> allSuperclasses(NodeClassFactory af);
7
8      /**
9       * Return true if this node class is a subclass of the given node class
10      * in the given node type hierarchy.
11      */
12     boolean isSubclass(NodeClassFactory af, NodeClass astClass);
13
14     /**
15      * Return an operator associated with this node class in the given
16      * operator factory, or null if undefined.
17      */
18     <O extends Operator> O operator(OperatorFactory<O> f);
19
20     /** partial order cache */
21     Map<NodeClassFactory, PartialOrder<NodeClass>> partialOrderMap =
22             new HashMap<>();
23
24     /** partial order for comparing node types in the given hierarchy */
25     static PartialOrder<NodeClass> partialOrder(NodeClassFactory af) {
26         if (!partialOrderMap.containsKey(af)) {
27             partialOrderMap.put(af, new PartialOrder<NodeClass>() {
28                 @Override
29                 public boolean comparable(NodeClass e, Object other) {
30                     if (other instanceof NodeClass) {
31                         NodeClass astClass = (NodeClass) other;
32                         return e.isSubclass(af, astClass)
33                                 || astClass.isSubclass(af, e);
34                     }
35                     return false;
36                 }
37
38                 @Override
```

```
39              public boolean moreSpecificThan(NodeClass e, Object other) {
40                  if (!comparable(e, other))
41                      throw new UnsupportedOperationException();
42                  NodeClass astClass = (NodeClass) other;
43                  return e.isSubclass(af, astClass);
44              }
45          });
46      }
47      return partialOrderMap.get(af);
48  }
49
50  abstract class Class implements NodeClass {
51      protected String name;
52      /** cache of immediate superclasses */
53      protected final Map<NodeClassFactory, Set<NodeClass>> superclasses =
54              new HashMap<>();
55      /** cache of all superclasses */
56      protected final Map<NodeClassFactory, Set<NodeClass>>
57              allSuperclasses = new HashMap<>();
58
59      public Class(String name) {
60          this.name = name;
61      }
62
63      @Override
64      public final Set<NodeClass> superclasses(NodeClassFactory af) {
65          if (!superclasses.containsKey(af))
66              superclasses.put(af,
67                              Collections.unmodifiableSet(
68                                      superclassesImpl(af)));
69          return superclasses.get(af);
70      }
71
72      protected abstract Set<NodeClass>
73              superclassesImpl(NodeClassFactory af);
74
75      @Override
76      public final Set<NodeClass> allSuperclasses(NodeClassFactory af) {
77          if (!allSuperclasses.containsKey(af)) {
78              Set<NodeClass> superclasses = superclasses(af);
79              Set<NodeClass> result = new HashSet<>(superclasses);
80              for (NodeClass superclass : superclasses)
81                  result.addAll(superclass.allSuperclasses(af));
82              allSuperclasses.put(af, Collections.unmodifiableSet(result));
83          }
84          return allSuperclasses.get(af);
85      }
86
87      @Override
88      public final boolean isSubclass(
89              NodeClassFactory af, NodeClass astClass) {
90          return allSuperclasses(af).contains(astClass);
```

```
91          }
92
93          @Override
94          public final String toString() {
95              ...
96          }
97      }
98  }
```

A.3 Implementation of operator factory explorer

First, we declare a utility class that directs the explorer to the correct operator factory for a given language:

```
1  class NodeOpUtil<Op extends Operator> {
2      /** resolver for operator factory factory for the given language */
3      private static final OpFactoryFactoryResolver offResolver =
4              (lang) -> lang.opFactoryFactory();
5      /** cache for previously requested implementations */
6      Map<NodeClass, Map<Lang, Map<Lang, Map<NodeClass, Op>>>> superOpCache =
7              new HashMap<>();
8      /** resolver for operator factory for the given operator */
9      protected OpFactoryResolver<Op> ofResolver;
10
11     public NodeOpUtil(OpFactoryResolver<Op> finder) {
12         ofResolver = finder;
13     }
14
15     /**
16      * Return the implementation defined in (lang, rep),
17      * or null if not exists.
18      */
19     public final Op op(Lang lang, NodeClass rep) {
20         // Obtain the desired operator factory.
21         OperatorFactory<Op> of =
22                 ofResolver.resolve(offResolver.resolve(lang));
23         // If operator factory is null, there is no implementation of
24         // this operator at all within the language, so return null.
25         // Otherwise, let the node type invoke the appropriate factory method
26         // in the operator factory for the desired implementation.
27         return of == null ? null : rep.operator(of);
28     }
29 }
30
31 interface OpFactoryFactoryResolver {
32     OperatorFactoryFactory resolve(Lang lang);
33 }
34
35 interface OpFactoryResolver<O extends Operator> {
36     OperatorFactory<O> resolve(OperatorFactoryFactory off);
```

```
37 }
```

Each language defines an *operator factory factory* containing methods that return operator factories for all implemented operators in the language. Given a language, its operator factory factory is determined. Then, given an operator, the appropriate operator factory can be determined from a method in the operator factory factory. Finally, given a node type, the appropriate implementation can be determined from a method in the operator factory.

To compute the most specific implementation efficiently, the explorer code uses a special implementation of `Map` in the Java Collections Framework that retains only the most specific keys added to the map so far. Keys for this specialized map are the most specific node types or languages that define an implementation of interest; the values are the implementations returned by operator factories.

The following explorer methods need to be implemented only once per dispatch ordering, to be shared by all operators. To implement a different dispatch ordering, only method `findSuperOperator` needs to be modified to correct the ordering.

```
1  /** Find the most specific implementation for (lang, rep). */
2  static <Op extends Operator> Op getOperator(Ast rep, Lang lang,
3          NodeOpUtil<Op> ou) {
4      return getOperator(rep, lang, ou, lang, rep);
5  }
6
7  /**
8   * Find the most specific implementation for (lang, rep),
9   * taking the original node type and language into account in case we need to
10  * explore the node type hierarchy in superlanguages.
11  * In that case, we need to start from the original node type all over again.
12  */
13 static <Op extends Operator> Op getOperator(NodeClass origRep, Lang origLang,
14         NodeOpUtil<Op> ou, Lang lang, NodeClass rep) {
15     // Try getting the implementation defined in (lang, rep) itself.
16     Op op = ou.op(lang, rep);
17     // If that implementation does not exist,
18     // try finding the superclass implementation.
19     if (op == null) op = getSuperOperator(origRep, origLang, ou, lang, rep);
20     return op;
21 }
22
23 /**
```

```
24  * cached version of findSuperOperator below to improve performance by
25  * avoiding querying factories repetitively
26  */
27 static <Op extends Operator> Op getSuperOperator(NodeClass origRep,
28        Lang origLang, NodeOpUtil<Op> ou, Lang lang, NodeClass rep) {
29    // Try cache first.
30    if (!ou.superOpCache.containsKey(origRep))
31       ou.superOpCache.put(origRep, new HashMap<>());
32    Map<Lang, Map<Lang, Map<NodeClass, Op>>> origRepMap =
33          ou.superOpCache.get(origRep);
34    if (!origRepMap.containsKey(origLang))
35       origRepMap.put(origLang, new HashMap<>());
36    Map<Lang, Map<NodeClass, Op>> origLangMap = origRepMap.get(origLang);
37    if (!origLangMap.containsKey(lang))
38       origLangMap.put(lang, new HashMap<>());
39    Map<NodeClass, Op> langMap = origLangMap.get(lang);
40    if (!langMap.containsKey(rep)) {
41       // If cache has no result, do the actual work of exploring
42       // operator factories in relevant languages.
43       Pair<Lang, NodeClass> spec =
44              findSuperOperator(origRep, origLang, ou, lang, rep);
45       if (spec == null) // No applicable implementation.
46          throw new InternalCompilerError("Message not understood");
47       // Cache the result.
48       langMap.put(rep, ou.op(spec.part1(), spec.part2()));
49    }
50    return langMap.get(rep);
51 }
52
53 /**
54  * Find the super implementation of the given current node type and language,
55  * taking the original node type and language into account in case we need to
56  * explore the node type hierarchy in superlanguages.
57  * In that case, we need to start from the original node type all over again.
58  */
59 static <Op extends Operator> Pair<Lang, NodeClass> findSuperOperator(
60       NodeClass origRep, Lang origLang, NodeOpUtil<Op> ou, Lang lang,
61       NodeClass rep) {
62    // Dispatch mechanism:
63    // - Go up the node type hierarchy within the same language first.
64    // - If no implementation found within the same language, explore
65    //  the superlanguages.
66
67    NodeClassFactory af = origLang.nodeClassFactory();
68    // Find candidates within the current language.
69    Map<NodeClass, Pair<Lang, NodeClass>> repCandidates =
70          new MostSpecificKeyMap<>(NodeClass.partialOrder(af));
71    for (NodeClass repCandidate : rep.allSuperclasses(af))
72       if (ou.op(lang, repCandidate) != null) {
73          repCandidates.put(repCandidate,
74                            new Pair<>(lang, repCandidate));
75       }
```

```
76      int numReps = repCandidates.size();
77      if (numReps == 1) // Unique, most specific implementation found.
78          return repCandidates.get(repCandidates.keySet().iterator().next());
79      else if (numReps > 1) {
80          throw new InternalCompilerError("Message ambiguous");
81      }
82      // No implementation within the current language.
83      // Find candidates in superlanguages.
84      Map<Lang, Pair<Lang, NodeClass>> langCandidates = new HashMap<>();
85              new MostSpecificKeyMap<>(Lang.partialOrder);
86      for (Lang superLang : lang.superLangs())
87          if (ou.op(superLang, origRep) != null) {
88              // Implementation for (superLang, T_0) found.
89              // No need to explore the node type hierarchy
90              // for this superlanguage.
91              langCandidates.put(superLang, new Pair<>(superLang, origRep));
92          }
93          else {
94              // Otherwise, find the implementation recursively.
95              Pair<Lang, NodeClass> superSpec =
96                      findSuperOperator(origRep,
97                                        origLang,
98                                        ou,
99                                        superLang,
100                                       origRep);
101             if (superSpec != null) {
102                 // A super implementation is found; record result.
103                 Lang langCandidate = superSpec.part1();
104                 langCandidates.put(langCandidate, superSpec);
105             }
106         }
107     // Most specific superlanguage needs to be unique,
108     // or ambiguity occurs.
109     int numLangs = langCandidates.size();
110     if (numLangs == 1) // Unique most specific superlanguage, OK.
111         return langCandidates.get(langCandidates.keySet().iterator().next());
112     else if (numLangs > 1) {
113         throw new InternalCompilerError("Message ambiguous");
114     }
115     return null;
116 }
```

Here is an example of implementing the dispatcher for type checking:

```
1  /** resolver for type checking */
2  NodeOpUtil<TypeCheckOperator> typeCheckOp =
3          new NodeOpUtil<>(off -> off instanceof JLOperatorFactoryFactory
4                  ? ((JLOperatorFactoryFactory) off).typeCheck() : null);
5
6  /** dispatcher to be invoked by client code */
7  static Node typeCheck(Node n, TypeChecker tc) throws SemanticException {
8      // Get the most specific operator for this node type in
```

```
9     // the language the type checker is working on, i.e.,
10    // L_0 = tc.lang() and T_0 = rep().
11    TypeCheckOperator op = Node.getOperator(n.rep(), tc.lang(), typeCheckOp);
12    return op.typeCheck(n, tc);
13 }
14
15 /** superclass dispatcher to be invoked by type-checking implementation */
16 static Node typeCheckSuper(Node n, TypeChecker tc, Lang lang, NodeClass rep)
17        throws SemanticException {
18    // Get the most specific superclass operator for node type "rep"
19    // in language "lang".
20    // The original node type and language are used when we need to
21    // move to another node type hierarchy in a different language,
22    // so we know where to start over.
23    TypeCheckOperator op =
24           Node.getSuperOperator(n.rep(), tc.lang(), typeCheckOp, lang, rep);
25    return op.typeCheck(n, tc);
26 }
27
28 /**
29  * direct superclass dispatcher to resolve conflicts;
30  * to be used sparingly by type-checking implementations
31  */
32 static Node typeCheck(Node n, TypeChecker tc, Lang lang, NodeClass rep)
33        throws SemanticException {
34    TypeCheckOperator op =
35           Node.getOperator(n.rep(), tc.lang(), typeCheckOp, lang, rep);
36    return op.typeCheck(n, tc);
37 }
```

BIBLIOGRAPHY

[1] Srikanth Adayapalam. Allow interface methods to be private. Java Bug System, March 2015. Retrieved April 17, 2017.

[2] Tadashi Ae. Direct or cascade product of pushdown automata. *Journal of Computer and System Sciences*, 14(2):257–263, 1977.

[3] Roland Axelsson, Keijo Heljanko, and Martin Lange. Analyzing context-free grammars using an incremental SAT solver. In *Automata, Languages and Programming*, volume 5126 of *Lecture Notes in Computer Science*, pp. 410–422. Springer Berlin Heidelberg, 2008.

[4] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. 2nd Working Conference on Reverse Engineering*, pp. 86–95, July 1995.

[5] H. J. S. Basten. Tracking down the origins of ambiguity in context-free grammars. In *Theoretical Aspects of Computing – ICTAC 2010*, volume 6255 of *Lecture Notes in Computer Science*, pp. 76–90. Springer Berlin Heidelberg, 2010.

[6] H. J. S. Basten and T. van der Storm. AmbiDexter: Practical ambiguity detection. In *Proc. 10th IEEE Int'l Workshop on Source Code Analysis and Manipulation (SCAM 2010)*, pp. 101–102, September 2010.

[7] H. J. S. Basten and J. J. Vinju. Faster ambiguity detection by grammar filtering. In *Proc. 10th Workshop on Language Descriptions, Tools and Applications*, pp. 5:1–5:9, 2010.

[8] Hendrikus J. S. Basten and Jurgen J. Vinju. Parse forest diagnostics with Dr. Ambiguity. In *Software Language Engineering*, volume 6940 of *Lecture Notes in Computer Science*, pp. 283–302. Springer Berlin Heidelberg, 2012.

[9] David Bau and David Anthony Bau. A preview of Pencil Code: A tool for developing mastery of programming. In *Proc. 2nd 2Nd Workshop on Programming for Mobile & Touch*, pp. 21–24, October 2014.

[10] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 368–377, November 1998.

[11] Ira Baxter. What is the easiest way of telling whether a BNF grammar is ambiguous or not? (answer). StackOverflow, July 2011. Retrieved November 11, 2014.

[12] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS®: Program transformations for practical scalable software evolution. In *Proc. 26th Int'l Conf. on Software Engineering (ICSE)*, pp. 625–634, May 2004.

[13] Dorian Birsan. On plug-ins and extensible architectures. *Queue*, 3(2):40–46, March 2005.

[14] Andrew P. Black. The expression problem, gracefully. In *Proc. MechAnisms on SPEcialization, Generalization and inHerItance*, pp. 1–7, July 2015.

[15] Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. Grace: The absence of (inessential) difficulty. In *Proc. ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 85–98, October 2012.

[16] Bard Bloom and Martin J. Hirzel. Robust scripting via patterns. In *Proc. 8th symposium on Dynamic languages (DLS)*, pp. 29–40, 2012.

[17] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In *Proc. 1st ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 17–29, November 1986.

[18] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *Proc. 35th ACM Symp. on Principles of Programming Languages (POPL)*, pp. 407–419, January 2008.

[19] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 302–315, 1997.

[20] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. 5th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 303–311, 1990.

[21] Martin Bravenboer and Eelco Visser. *Parse Table Composition*, pp. 74–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[22] Stephen Chang, Alex Knauth, and Ben Greenman. Type systems as macros. In *Proc. 44th ACM SIGPLAN Symposium on Principles of Programming Languages*,

pp. 694–705, 2017.

[23] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[24] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proc. 15th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 130–145, 2000.

[25] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. ISBN 0-13-451832-2.

[26] CVE. CVE-2013-0269. Common Vulnerabilities and Exposures, February 2013. Retrieved November 13, 2014.

[27] CVE. CVE-2013-4547. Common Vulnerabilities and Exposures, November 2013. Retrieved November 13, 2014.

[28] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[29] Frank DeRemer and Thomas Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Trans. on Programming Languages and Systems*, 4(4):615–649, October 1982.

[30] J. des Rivières and J. Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[31] Mirco Dotta, Philippe Suter, and Viktor Kuncak. On static analysis for expressive pattern matching. Technical report, École Polytechnique Fédérale de Lausanne, February 2008.

[32] Torbjörn Ekman and Görel Hedin. The JastAdd system — modular extensible compiler construction. *Science of Computer Programming*, 69(1–3):14–26, 2007. Special issue on Experimental Software and Toolkits.

[33] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *Proc. 21st European Conf. on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pp. 273–298. Springer Berlin Heidelberg, 2007.

[34] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proc. 12th Workshop on Language Descriptions, Tools, and Applications*, pp. 7:1–7:8, March 2012.

[35] Erik Ernst. The expression problem, Scandinavian style. In *Proc. 3rd International Workshop on Mechanism for Specialization, Generalization and Inheritance*, pp. 27–30, June 2004.

[36] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *Proc. 12th European Conf. on Object-Oriented Programming*, pp. 186–211, 1998.

[37] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 113–128, May 2015.

[38] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 237–247, 1993.

[39] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. on Programming Languages and Systems*, 29(3), May 2007.

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994. ISBN 0-201-63361-2.

[41] Jacques Garrigue. Typing deep pattern-matching in presence of polymorphic variants. In *JSSST Workshop on Programming and Programming Languages*, 2004.

[42] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014. ISBN 013390069X, 9780133900699.

[43] Ralph E. Griswold, David R. Hanson, and John T. Korb. Generators in Icon. *ACM Trans. on Programming Languages and Systems*, 3(2):144–161, April 1981.

[44] Görel Hedin and Eva Magnusson. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, November 2002.

[45] Martin Hirzel, Nathaniel Nystrom, Bard Bloom, and Jan Vitek. Matchete: Paths through the pattern matching jungle. In *Proc. 10th Int'l Conf. on Practical Aspects of Declarative Languages*, pp. 150–166, 2008.

[46] Michael Homer and James Noble. A tile-based editor for a textual programming language. In *1st IEEE Working Conf. on Software Visualization (VISSOFT)*, pp. 1–4, September 2013.

[47] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, 1979. ISBN 978-0-201-02988-8.

[48] Scott E. Hudson, Frank Flannery, C. Scott Ananian, Dan Wang, and Andrew Appel. CUP LALR parser generator for Java, 1996. Software release. At `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[49] Chinawat Isradisaikul and Andrew C. Myers. Reconciling exhaustive pattern matching with objects. In *Proc. 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 343–354, 2013.

[50] Chinawat Isradisaikul and Andrew C. Myers. Finding counterexamples from parsing conflicts. In *Proc. 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 555–564, 2015.

[51] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, July 1975.

[52] Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In *Proc. 18th ACM SIGPLAN International Conference on Functional Programming*, pp. 343–350, 2013.

[53] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. 11th European Conf. on Object-Oriented Programming*, pp. 220–242, June 1997.

[54] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[55] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, September 1985.

[56] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *Proc. 6th ACM SIGPLAN Int'l Conf. on Functional Programming*, pp. 26–37, 2001.

[57] Gary T. Leavens and Peter Müller. Information hiding and visibility in interface specifications. In *Proc. 29th Int'l Conf. on Software Engineering (ICSE)*, pp. 385–395, 2007.

[58] Keunwoo Lee, Anthony LaMarca, and Craig Chambers. HydroJ: object-oriented pattern matching for evolvable distributed systems. In *Proc. 18th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 205–223, 2003.

[59] Barbara Liskov, Russell Atkinson, Toby Bloom, J. Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Springer-Verlag, 1984. Also published as Lecture Notes in Computer Science 114, G. Goos and J. Hartmanis, Eds., Springer-Verlag, 1981.

[60] Jed Liu and Andrew C. Myers. JMatch: Java plus pattern matching. Technical Report 2002-1878, Computer Science Department, Cornell University, October 2002. Software release at `http://www.cs.cornell.edu/projects/jmatch`.

[61] Jed Liu and Andrew C. Myers. JMatch: Iterable abstract pattern matching for Java. In *Proc. 5th International Symposium on Practical Aspects of Declarative Languages*, pp. 110–127, January 2003.

[62] Jed Liu, Aaron Kimball, and Andrew C. Myers. Interruptible iterators. In *Proc. 33rd ACM Symp. on Principles of Programming Languages (POPL)*, pp. 283–294, January 2006.

[63] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proc. 22nd ACM Symp. on Operating System Principles (SOSP)*, pp. 321–334, 2009.

[64] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The Scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, November 2010.

[65] Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17(3):387–421, 2007.

[66] Kazutaka Matsuda and Meng Wang. *FliPpr: A Prettier Invertible Printing System*, pp. 101–120. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[67] P. M. Maurer. Metamorphic programming: unconventional high performance. *Computer*, 37(3):30–38, March 2004.

[68] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[69] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *Proc. 13th Int'l Conf. on Compiler Construction (CC'04)*, pp. 73–88, 2004.

[70] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. Unifying execution of imperative and declarative code. In *Proc. 33rd Int'l Conf. on Software Engineering (ICSE)*, pp. 511–520, May 2011.

[71] Todd Millstein. Practical predicate dispatch. In *Proc. 19th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 345–364, 2004.

[72] Todd Millstein, Mark Reay, and Craig Chambers. Relaxed MultiJava: Balancing extensibility and modular typechecking. In *Proc. 18th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 224–240, 2003.

[73] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990. ISBN 978-0262631327.

[74] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pp. 228–241, 1999.

[75] Andrew C. Myers, Barbara Liskov, and Nicholas Mathewson. PolyJ: Parameterized types for Java. Software release, at `http://www.cs.cornell.edu/polyj`, July 1998.

[76] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow, July 2006. Software release. At `http://www.cs.cornell.edu/jif`.

[77] Greg Nelson and Allen Heydon. Juno-2 language definition. SRC Technical Note 1997-007, Digital Systems Research Center, June 1997.

[78] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. 12th Int'l Conf. on Compiler Construction (CC'03)*, pp. 138–152, April 2003.

[79] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proc. 19th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 99–115, 2004.

[80] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software composition. In *Proc. 21st ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 21–36, October 2006.

[81] Martin Odersky and Philip Wadler. Pizza into Java: translating theory into practice. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pp. 146–159, 1997.

[82] Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In *Proc. 12th Intl. Workshop on Foundations of Object-Oriented Languages*, volume 12 of *FOOL '05*, 2005.

[83] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proc. 20th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 41–57, October 2005.

[84] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008. ISBN 0981531601, 9780981531601.

[85] Bruno C. d. S. Oliveira. Modular visitor components. In *Proc. 23rd European Conference on Object-Oriented Programming*, pp. 269–293, July 2009.

[86] Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses: practical extensibility with object algebras. In *Proc. 26th European conference on Object-Oriented Programming*, pp. 2–27, June 2012.

[87] JetBrains Self-regulatory organization. IntelliJ IDEA – capable and ergonomic Java IDE, 2016. Software release. At `https://www.jetbrains.com/idea/`.

[88] Hari Mohan Pandey. Advances in ambiguity detection methods for formal grammars. *Procedia Engineering*, 24(0):700–707, 2011. International Conference on Advances in Engineering 2011.

[89] Terence Parr and Kathleen Fisher. LL(*): The foundation of the ANTLR parser generator. In *Proc. 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 425–436, 2011.

[90] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) parsing: The power of dynamic analysis. In *Proc. 2014 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 579–598, 2014.

[91] Leonardo Teixeira Passos, Mariza A. S. Bigonha, and Roberto S. Bigonha. A methodology for removing LALR(k) conflicts. *Journal of Universal Computer Science*, 13(6):737–752, June 2007.

[92] Leonardo Teixeira Passos, Mariza A. S. Bigonha, and Roberto S. Bigonha. An LALR parser generator supporting conflict resolution. *Journal of Universal Computer Science*, 14(21):3447–3464, December 2008.

[93] Dmitry Petrashko, Ondřej Lhoták, and Martin Odersky. Miniphases: Compilation using modular and efficient tree transformations. In *Proc. 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 201–216, 2017.

[94] François Pottier and Yann Régis-Gianas. Menhir LR(1) parser generator for the OCaml programming language, 2017. Software release. At `http://gallium.inria.fr/~fpottier/menhir/`.

[95] Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: Unifying parsing and pretty printing. In *Proc. 3rd ACM Haskell Symposium on Haskell*, pp. 1–12, 2010.

[96] Adam Richard and Ondřej Lhoták. OOMatch: pattern matching as dispatch in Java. In *Companion 22nd ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 771–772, 2007.

[97] C. Rodriguez-Leon and L. Garcia-Forte. Solving difficult LR parsing conflicts by postponing them. *Comput. Sci. Inf. Syst.*, 8(2):517–531, 2011.

[98] Clyde Ruby and Gary T. Leavens. Safely creating correct subclasses without seeing superclass code. In *Proc. 15th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 208–228, 2000.

[99] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. In *Proc. 9th ACM SIGPLAN International Conference on Functional Programming*, pp. 201–212, 2004.

[100] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. In *Proc. 17th European Conf. on Object-Oriented Programming*, pp. 248–274, July 2003.

[101] Sylvain Schmitz. Conservative ambiguity detection in context-free grammars. In *Automata, Languages and Programming*, volume 4596 of *Lecture Notes in Computer Science*, pp. 692–703. Springer Berlin Heidelberg, 2007.

[102] Sylvain Schmitz. An experimental ambiguity detection tool. *Science of Computer Programming*, 75(1–2):71–84, 2010. Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07).

[103] Friedrich Wilhelm Schröer. AMBER, an ambiguity checker for context-free grammars. Technical report, Fraunhofer Institute for Computer Architecture and Software Technology, 2001.

[104] August Schwerdfeger and Eric Van Wyk. *Verifiable Parse Table Composition for Deterministic Parsing*, pp. 184–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[105] August C. Schwerdfeger and Eric R. Van Wyk. Verifiable composition of deterministic grammars. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 199–210, 2009.

[106] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.

[107] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1–3):17–64, October–December 1996.

[108] Lixin Su and Mikko H. Lipasti. Dynamic class hierarchy mutation. In *Proc. Int'l Symposium on Code Generation and Optimization*, pp. 98–110, March 2006.

[109] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. In *Proc. 37th ACM Symp. on Principles of Programming Languages (POPL)*, pp. 199–210, 2010.

[110] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *Proc. 18th Int'l Conf. on Static Analysis*, pp. 298–315, 2011.

[111] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *Proc. 12th ACM SIGPLAN Int'l Conf. on Functional Programming*, pp. 29–40, 2007.

[112] Michiaki Tatsubori, Shigeru Chiba, Marc-Oliver Killijian, and Kozo Itano. OpenJava: A class-based macro system for Java. In *Reflection and Software Engineering*, pp. 117–133, July 2000.

[113] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 132–141, June 2011.

[114] Masaru Tomita, editor. *Generalized LR Parsing*. Springer US, 1991. ISBN 978-1-4613-6804-5.

[115] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1–40, 2015.

[116] Tijs van der Storm, William R. Cook, and Alex Loh. Object grammars. In *5th Int'l Conference on Software Language Engineering*, pp. 4–23, September 2012.

[117] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[118] Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger. Attribute grammar-based language extensions for Java. In *Proc. 21st European Conf. on Object-Oriented Programming*, pp. 575–599, July 2007.

[119] Naveneetha Vasudevan and Laurence Tratt. Detecting ambiguity in programming language grammars. In *Proc. 6th Int'l Conf. on Software Language Engineering*, pp. 157–176, October 2013.

[120] Marcos Viera and S. Doaitse Swierstra. First class syntax, semantics, and their composition. In *Proc. 25th Symposium on Implementation and Application of Functional Languages*, pp. 73–84, August 2014.

[121] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proc. 14th ACM Symp. on Principles of Programming Languages (POPL)*, pp. 307–313, 1987.

[122] Philip Wadler et al. The expression problem, December 1998. Discussion on Java-Genericity mailing list.

[123] Meng Wang, Jeremy Gibbons, Kazutaka Matsuda, and Zhenjiang Hu. Refactoring pattern matching. *Science of Computer Programming*, 78(11):2216–2242, 2013.

[124] Yanlin Wang and Bruno C. d. S. Oliveira. The expression problem, trivially! In *Proc. 15th International Conference on Modularity*, pp. 37–41, March 2016.

[125] Andreas Wenger and Michael Petter. CUP2 LR parser generator for Java, 2014. Software beta release. At `http://www2.in.tum.de/cup2`.

[126] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proc. 6th ACM SIGPLAN Int'l Conf. on Functional Programming*, pp. 241–252, September 2001.