UNIVERSITY OF CALIFORNIA SAN DIEGO

GAZELLE: A Framework for Compositional Programming-Language Semantics and Reasoning

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Mario M. Alvarez

Committee in charge:

Professor Ranjit Jhala, Chair
Professor Philip Guo
Professor Jim Hollan
Professor Sorin Lerner
Professor Victor Vianu

2022

The dissertation of Mario M. Alvarez is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2022

DEDICATION

To the pursuit of Truth,

and to everyone engaged in it

EPIGRAPH

If there is no struggle,

there is no progress.

—Frederick Douglass

TABLE OF CONTENTS

Dissertation Approval Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures ix

Acknowledgements x

Vita xii

Abstract of the Dissertation xiii

Chapter 1 Introduction 1
1.1 Background 1
1.2 Thesis Statement 2
1.3 The GAZELLE Approach: An Overview 3
1.4 Related Work: Other Approaches to Language Composition and Flexibility 4
1.4.1 The Expression Problem 4
1.4.2 *Datatypes á la Carte* 5
1.4.3 Automation for Adapting Datatypes and Proofs 8
1.4.4 Compositional Verified Compilation 8
1.5 The Rest of This Paper 9

Chapter 2 GAZELLE: A Framework for Language Composition 12
2.1 GAZELLE Example - Composing Programming Languages 13
2.1.1 The Languages 13
2.1.2 Composition 15
2.2 A Lightweight Approach to Instruction-Language Composition . . . 24
2.2.1 State Composition: Examples and Intuitions 26
2.2.2 Information-Ordering In GAZELLE 31
2.2.3 Wrapper Types for Mergeable States 41
2.2.4 Composition and Preservation of Suprema 50
2.2.5 Summary 53
2.3 Extending Composition to Multi-Step Executions 55
2.3.1 Syntax for GAZELLE's Control-Flow Interpreter 55
2.3.2 GAZELLE Control-Flow Semantics 60
2.4 Summary 66

Chapter 3 Lifting: Using Partial Orders with Existing Languages . . . 68
3.1 The Lifting Abstraction . . . 69
3.1.1 Lenses . . . 69
3.1.2 A Lens-Inspired Lifter Abstraction . . . 70
3.2 Lifter Instances . . . 75
3.2.1 Identity Lifting . . . 75
3.2.2 Trivial Lifting . . . 76
3.2.3 Option Lifting . . . 77
3.2.4 Priority Lifting . . . 78
3.2.5 Tuple Liftings . . . 80
3.2.6 Merge Lifting and Orthogonality . . . 81
3.2.7 Discussion . . . 84
3.3 Automating Lifter Instance Generation . . . 84
3.3.1 The Automated Lifting Generator - An Example . . . 84
3.3.2 Ad-Hoc Polymorphism in ISABELLE . . . 87
3.3.3 Typeclasses for Lifter Inference . . . 91
3.3.4 Automated Lifting Generation Internals - The `schem_lift` Polymorphic Constant . . . 95
3.3.5 Priority Functions in Liftings . . . 100
3.3.6 Automated Lifting Generation - Scalability . . . 100
3.3.7 Proof Automation for Lifting Correctness . . . 101
3.4 Discussion . . . 104

Chapter 4 Reasoning About GAZELLE . . . 106
4.1 Hoare Logic: A Primer . . . 107
4.2 Hoare Logic for Single-Step Semantics . . . 109
4.3 CPS-Flavored Hoare Logic . . . 112
4.4 CPS-Flavored Hoare Logic with Step-Counts . . . 115
4.5 Soundness of Step-Counting Hoare Logic . . . 118
4.6 Reasoning about Multi-Step Composition . . . 120
4.6.1 Dominance and Toggling . . . 120
4.7 Discussion . . . 124

Chapter 5 IMP: An Extended Example of GAZELLE in Practice . . . 126
5.1 IMP's Sub-Languages . . . 127
5.1.1 Arithmetic Language . . . 128
5.1.2 Boolean Language . . . 129
5.1.3 Variable-Store Language . . . 130
5.1.4 Sequencing Language . . . 132
5.1.5 IMP-Control Language . . . 134
5.1.6 Discussion . . . 137
5.2 Liftings for Constructing IMP . . . 137
5.2.1 An Overview of IMP State . . . 138

5.2.2 Priority Protocol for IMP . . . 141
5.2.3 Lifting Languages Without Control-Flow . . . 142
5.2.4 Arithmetic Language . . . 144
5.2.5 Boolean Language . . . 146
5.2.6 Variable-Store Language . . . 146
5.2.7 Sequencing Language . . . 148
5.2.8 IMP-Control Language . . . 149
5.2.9 IMP Semantics Definition . . . 150
5.3 Hoare Rules for IMP . . . 151
5.3.1 Control-Flow-Free Instructions . . . 151
5.3.2 Sequencing Rule . . . 153
5.3.3 If Rule . . . 153
5.3.4 While Rule . . . 155
5.4 IMP Example: Multiplication as Repeated Addition . . . 157
5.4.1 Multiplication Program . . . 157
5.4.2 Multiplication Specification . . . 158
5.4.3 Multiplication Proof . . . 159

Chapter 6 Conclusion . . . 163
6.1 Summary . . . 163
6.2 Ideas for Future Work . . . 164
6.2.1 Evaluating GAZELLE on More Case Studies . . . 165
6.2.2 Library of Language Components . . . 165
6.2.3 Porting GAZELLE to Other Proof Assistants . . . 165
6.2.4 Fully Leveraging `Bsup` . . . 166
6.2.5 Reconciling Categorical Approach to Lenses with Gazelle Liftings . . . 166

Bibliography . . . 168

LIST OF FIGURES

Figure 1.1: Visual depiction of the expression problem 5

Figure 2.1: Visual depiction of lifting into the example language's combined state; arrows denote injection of data 21
Figure 2.2: Definition of `pcomps`, which implements composition of instruction semantics 24
Figure 2.3: Visual depiction of `is_sup`; arrows denote `pleq` 34
Figure 2.4: Visual depiction of `is_bub`; arrows denote `pleq` 36
Figure 2.5: Visual depiction of `is_bsup`; arrows denote `pleq` 36
Figure 2.6: Visual depiction of GAZELLE ordering typeclasses; arrows indicate inheritance 42
Figure 2.7: Definition of `sem_run`, which implements multi-step execution for GAZELLE 56

Figure 3.1: Visual depiction of lifting example (arrows represent injection of data) . . . 85

Figure 5.1: Visual depiction of lifting into IMP's combined state; arrows denote injection of data 140

ACKNOWLEDGEMENTS

Thanks to my parents, for bringing me into this world and providing me with immeasurable love and support during my time in it thus far. Your emotional and financial support during the final months as I wrote this dissertation helped make it possible.

And to my brothers Gabe and Julian, whom I'm happy and proud to count as two of my closest friends. You've been with me (at various times in person, remotely, and in spirit) throughout this entire process, and I greatly appreciate that.

And to Gautam Mohan, who's been like another brother to me, and another source of so much support.

Thanks to Andrew Appel and Dave Walker, my research advisors at Princeton, who inspired me to pursue programming languages research. Thanks as well to Joey Dodds, who worked with me closely as I learned the ropes of using COQ during my senior year at Princeton.

Thanks to Gregory Malecha, whom I had the pleasure of working with in both my undergraduate and graduate research, and who was always a source of encouragement, knowledge, and patience.

Thanks to Sorin Lerner, my first Ph.D advisor, who influenced me to choose UCSD for graduate school, and helped me get my footing in graduate research.

Thanks to Victor Vianu, my second Ph.D advisor, for being willing to take me on as an advisee as I explored database verification and experimented with new (to me) research directions.

Thanks to Ranjit Jhala, my third Ph.D advisor, for taking me on as an advisee after I returned from a year away from the Ph.D. I came to him with crazy idea that derived from my experience in industry and was not particularly well aligned with his research program. He advised me on the project anyway, and the result is this dissertation.

I'm deeply grateful to have gotten a second chance (or two, depending on how one counts) to finish this degree. This is not something I take for granted. All three professors who have advised me at different times in my Ph.D were taking a chance on me. I hope, in the end, that I

have done all of you proud.

I'd also like to thank the rest of my committee for helping to make this happen.

Credit is due also to ConsenSys, who offered me an internship when I was lost and struggling in the middle of my Ph.D, funded a substantial part of my degree after I returned, and provided me with a source of inspiration and research ideas driven by practical needs in the blockchain industry. Bill Gleim, Gonçalo Sá, John Mardlin, Robert Drost, Joseph Chow, and Joe Lubin deserve special mention.

And thanks to my peers at UCSD, who were with me through the good times as well as the bad. To me, the Ph.D students and the community around them are what makes CS at UCSD special. At various times you have been friends, research partners, colleagues, bandmates, traveling companions, confidantes, roommates, and allies. These past eight years have probably been the craziest years of my life. I've had experiences I hadn't imagined I would have, and been tested in ways I had never been before. I'm profoundly grateful that I got to share this journey with you all. In no particular order (and not strictly limiting myself to UCSD students), I'd specifically like to express gratitude to Dimo Bounov, Alex Sanchez-Stern, Anish Tondwalkar, Elizabeth Hilbert, Erik Moyer, Shravan Narayan, John Renner, Valentin Robert, Peter Edge, Dan Ricketts, Arjun Roy, Danilo Gasques, Akshay Balsubramani, Ariel Weingarten, Ming Kawaguchi, and Alexander Bakst.

There are almost too many to name. If I've left your name off the list, or if you don't think your name belongs on the list but we interacted during our time at UCSD, know that you had an impact on my time here and that I'm grateful.

Finally, thanks to the friends of Bill W., especially Daniel G., without whose wisdom and support this dissertation would surely not exist.

VITA

2014	A. B. in Computer Science *magna cum laude*, *P*rinceton University
2015-2017	Graduate Teaching Assistant, University of California San Diego
2018	Master of Computer Science, University of California San Diego
2017-	Researcher at ConsenSys Mesh (ConsenSys AG)
2022	Ph. D. in Computer Science, University of California San Diego

PUBLICATIONS

Mario M. Alvarez, "Using Reflective Separation-Entailment Solvers for Reasoning Formally about C: Integrating the Verified Software Toolchain with the MirrorShard Solver" Princeton undergraduate thesis, 2014

Daniel Ricketts, Gregory Malecha, Mario M. Alvarez, Vignesh Gowda, and Sorin Lerner "Towards Verification of Hybrid Systems in a Foundational Proof Assistant", ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2015

ABSTRACT OF THE DISSERTATION

GAZELLE: A Framework for Compositional Programming-Language Semantics and Reasoning

by

Mario M. Alvarez

Doctor of Philosophy in Computer Science

University of California San Diego, 2022

Professor Ranjit Jhala, Chair

Formalizing the semantics of a programming language enables powerful techniques for understanding the correctness of tooling related to that language (e.g. compilers) as well as for understanding the correctness of programs written in that language. Traditional approaches to formalizing semantics lead to the production of monolithic systems whose components are difficult to reuse. In this dissertation we describe GAZELLE, a system for composing programming-language semantics out of reusable fragments. GAZELLE demonstrates an approach to surmounting these obstacles, enabling greater reuse and thus potentially more efficient development of formalized language semantics. Through an extended example, we show the usage of GAZELLE in practice

to formalize the IMP language and prove the correctness of an iterative IMP program. The entire development of GAZELLE is done in ISABELLE, and all key results are foundationally verified with machine-checked proofs.

Chapter 1

Introduction

1.1 Background

Formalization of programming language semantics has long been of interest to computer scientists. If we can say with precision what our programs *mean*, in the sense of having a mathematical or logical model of programs' behavior, we can hope to better understand their correctness. In the past few decades, the programming languages community has made significant strides toward formalizing programming-language semantics within *proof assistants*, software built for constructing and reasoning about mathematics. Importantly, proof assistants enable *machine-checked proofs*: as long as we trust the proof assistant itself, we can know that any purported proof checked by a proof assistant is mathematically valid.

As the scalability of proof assistants and commodity CPUs (in terms of compute performance) has improved, the formal semantics community has produced a number of striking machine-checked results, using a formalized language semantics to build machine-checked proofs of correctness [1] for increasingly large and complicated software systems. For example, the Coq [Tea22] theorem prover was used to prove the correctness of a compiler for the C language,

[1] "Correctness" here simply means "conforms to its (formal) specification". Determining whether a program's specification corresponds to our intuitive notion of correctness for that particular program can, of course, be difficult.

CompCert [LBK+16]. Coq was also used to formally verify an operating system kernel, CertiKOS [GKR+15]. Another proof assistant, HOL4, was used to verify an implementation of a Standard ML-like language, CakeML [KMNO14].

As impressive as these results are, they all share a significant drawback: each of these results took person-years' worth of expert effort, and resulted in systems that are difficult to build on or reuse components from. Researchers or developers undertaking a project to build, for instance, a formally verified compiler for a new language will likely find that there is little code or proofs they can reuse from prior efforts in compiler verification. This leads to duplication of developer effort in separate verification efforts, and represents part of why, even today, developing verified systems of any meaningful size remains a daunting task.

One important part of reducing the burden on developers of formally verified software is enabling reuse. In this dissertation, we focus specifically on the problem of enabling reuse of elements of formal programming language semantics. We do so by building a system, Gazelle, which has the aim of making it possible to construct a programming language semantics *by composition*. Gazelle is designed with an eye toward supporting reuse of existing formalizations, enabling the user to build semantics by composing pieces that were developed separately, without awareness of each other or of Gazelle itself. Gazelle also supports constructing *program logics* consisting of proof-rules for reasoning about the semantics of languages defined by composition; thus, Gazelle enables reuse of reasoning principles as well as semantic definitions. Gazelle is developed using the Isabelle [Wen] proof assistant, and all its associated proofs are machine-checked.

1.2 Thesis Statement

By enabling construction of formal programming language semantics out of reusable parts, the Gazelle system facilitates building and reasoning about formal models of programming

languages.

1.3 The GAZELLE Approach: An Overview

To achieve its goal of enabling compositional programming language semantics, GAZELLE makes use of elements drawn from several well-known approaches, combined together in a novel way. GAZELLE breaks down language components to be composed to the granularity of *instructions* operating over that component's *machine state*. Each instruction has a *denotational semantics* in the form of an ISABELLE function that describes how that instruction acts to update the state. At this level, GAZELLE gives a formal account of *merging* together results obtained by executing instructions from different language components, giving us a denotation for how the combined language acts on a combined state, even in the presence of nontrivial interactions between different components.

To compose these instructions together to form larger programs, GAZELLE makes use of *operational semantics*, describing program execution by means of a simple, control-flow-based virtual machine. This machine makes use of instructions' denotational semantics at each execution step and enables us to capture the meaning of full programs, not just single instructions. In doing so, it also provides us a with an executable interpreter for running such programs.

Finally, GAZELLE enables reasoning about these compositions using *axiomatic semantics* (reasoning rules for doing proofs about programs' behavior) in the form of Hoare logic. GAZELLE enables compositional axiomatic semantics, enabling reuse of individual language components' proof rules when reasoning about the behavior of the combined language. Taken together, these three pieces - denotational, operational, and axiomatic - complement each other to create a powerful tool for compositional reasoning.

1.4 Related Work: Other Approaches to Language Composition and Flexibility

GAZELLE sits within or adjacent to several well-studied sub-fields of computer science. Accordingly, there is a great deal of related work, far more than we could discuss in full detail here. In this section, we will discuss work exemplifying several different research directions related to GAZELLE's problem and approach. Though not exhaustive, this will give a good sense of the shape of the space of existing work, and GAZELLE's place within it.

1.4.1 The Expression Problem

One way to understand the design space GAZELLE inhabits is through the lens of a classic tradeoff in programming-language design: the *expression problem*. The expression problem is a term coined by Wadler [Wad98] to capture the fact that most major programming languages and paradigms did (and still do) allow extensibility along only one of two axes, but not both. Specifically, some programming paradigms (notably functional programming with abstract datatypes) make it easy to extend existing programs by adding new functions over existing datatypes, but do not offer a way to change the datatype's representation without refactoring all the functions that use it. On the other hand, in other paradigms (notably object-oriented programming), data representations used inside of an object can be easily changed, but changing the set of primitive operations supported by an object requires refactoring (or at least recompiling) code that uses the object to work with the new, changed interface.

The expression problem is to design a mechanism that allows for both kinds of extensibility. We represent the problem space visually in figure 1.1.

Wadler's original formulation of the problem had two other desiderata: namely, the solution be type-safe (without runtime casts), and that recompiling "client" code not be necessary when extending a datatype (along either axis). For our purposes, since we are concerned with

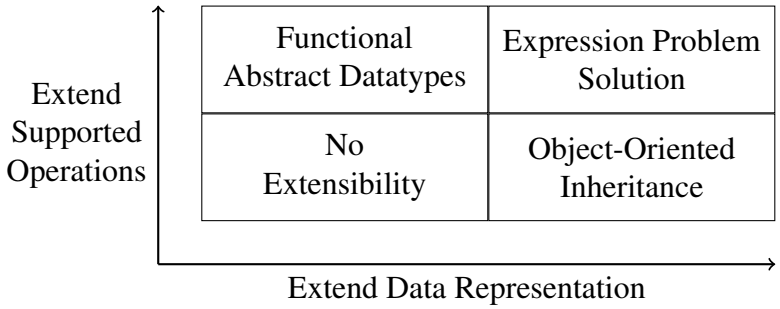

Figure 1.1: Visual depiction of the expression problem

reasoning about datatypes being extended, we want two analogous (though not identical) conditions: that there should be no loss of precision when reasoning about the extended version of the datatype as compared to the original, and that existing theorems about the datatype are trivial to adapt to the extended datatype. In [Wad98], Wadler also gave a solution to the expression problem in Java, but it was later found to not be type-safe [ZO05].

Having set the stage, we'll now take a look at a few families of approaches to variants of the expression problem, as well as some other related approaches to language composition.

1.4.2 *Datatypes á la Carte*

One approach to solving the expression problem is what is sometimes called the *á la carte* approach, after the paper that first proposed it, Swierstra's *Datatypes á la Carte* (*DTC*) [Swi08]. This approach is usually deployed in a functional language ([Swi08] used HASKELL); its goal is to enable flexibility in data representations (as we discussed earlier in this section, functional data abstraction such as Haskell offers already allows for adding new functions over existing data types). In this approach, the user expresses different language components as *algebras*. Each algebra corresponds to a type constructor `F` which must support an evaluation function: `eval :: F x => x` for result type `x`. Relying on the fact that the coproduct of two algebras is an algebra, the user of DTC constructs an algebra corresponding to the composition of all desired language features by taking the coproduct of the algebras corresponding to each feature. In DTC,

each language component represents a particular kind of syntax-tree node (which may include sub-trees of the entire language, rooted at that node). The result is an interpreter that is able to call the correct language component's interpreter based on the syntax-tree (i.e. program) provided as an argument, making recursive calls to handle sub-trees (if any are present).

DTC also offers some automation to enhance its usability. By making clever use of HASKELL's *typeclass* system, DTC enables the inference of injection and projection functions that enable the user to avoid having to manually pack and unpack nested coproducts when defining a combined language. It is worth noting that GAZELLE in fact uses something similar in its *lifting* infrastructure (see chapter 3; specifically section 3.3.7). While the specifics of the approaches differ, GAZELLE is certainly indebted to DTC for the basic idea of using typeclasses for this purpose.

Unfortunately, the *á la carte* approach described in [Swi08], while convenient for building interpreters, is not suitable for formal semantics: as Delaware et al. observe in [DdSOS13], Swierstra's solution relies on an unsound type-level fixed-point operator; a direct translation of the original *á la carte* technique in a proof assistant would be rejected by the proof assistant because of this unsoundness. The solution taken by [DdSOS13] - called *Metatheory á la Carte* (*MTC*) - replaces this unsound fixed-point operator with a sound one. However, this soundness comes at a price: in order to define functions using the sound type-level fixed-point operator, the functions need to be expressed as *Mendler-style folds*, and operate over *Church-encoded data*. These are rather significant restrictions on the structure of the formalizations that can be developed and reasoned about using the *Metatheory á la Carte* approach. As with DTC, some of these constraints are alleviated through the use of automation, but others - particularly the need to use Mendler-encodings for recursion - are difficult not to expose to the user.

Nonetheless, the formal development of MTC is an impressive result, and is probably the most similar in spirit to GAZELLE of existing work. Both MTC and GAZELLE aim to ease the expression of formal semantics by enabling a compositional approach, and both support reasoning

about such compositional definitions. The primary differences are:

- MTC and GAZELLE impose different restrictions on the structure of programs: GAZELLE does this by requiring a particular encoding of control flow (see section 2.3.2); MTC requires Church and Mendler encodings.

- GAZELLE supports a more general class of compositions. MTC supports mixing together existing language components, which may make use of each other's results; however, different components in MTC cannot override each other's results (e.g. in the case study presented in [DdSOS13], if a language contains the `Arith` component, then `Arith` is the only way to assign values to arithmetic expressions). GAZELLE, in contrast, allows the addition of new components that override or modify the behaviors of existing ones.

Another related approach that bears mentioning here is the construction of extensible syntax trees described in [NJ17]. This work focuses on the problem of reducing code reuse in building compilers, which often have multiple similar - but not identical - versions of the same datatype, corresponding to different intermediate representations used within the compiler. The solution proposed in [NJ17] is to define a single, extensible syntax tree type, with type parameters that can be used to add or remove constructors (hence, add or remove different kinds of syntax nodes), or change the parameters to existing nodes. On top of this core insight - that making syntax types polymorphic in type parameters representing extensions can allow for generic code that works over different extensions - [NJ17] also presents some typeclass and type-family based automation that greatly simplifies the process of working with such extensible syntax in HASKELL. The idea of using type parameters to generically extend syntax trees greatly influenced GAZELLE's approach to generic syntax (see section 2.3.2).

1.4.3 Automation for Adapting Datatypes and Proofs

Another important approach to the expression problem as it relates to improving reusability of formal developments is to automate the process of proof repair. It commonly happens in proof-assistant-based formalizations that changes to one datatype or definition will break a large number of proofs in fairly predictable ways. [RYLG18] presents PUMPKIN PATCH, a system for finding general patches for fixing proofs broken by such changes based on a user-supplied fix for any one proof broken by a particular change. Similarly, [RYLG19] presents a type-theory-based approach to automatically deriving new versions of functions and proofs based on certain classes of changes to datatypes (those which can be captured as *algebraic ornaments*).

1.4.4 Compositional Verified Compilation

Finally, another line of work worth mentioning is [Ahm15]. This work represents an approach to *compositional compiler correctness*: that is, developing a methodology for reasoning about the correctness of compiled code, in a context where that code is being *linked* with some other code that may not be written in the same language. It is relevant to the expression problem, as well as to building extensible and interoperable language formalizations: the basic principle behind [Ahm15] (along with other work in the same vein) is to "*view compiler correctness as a language interoperability problem*" ([Ahm15, pg.15]; emphasis in original).

When composing two programs `P1` and `P2`, respectively compiled by compilers for languages `L1` and `L2`, this approach first proposes identifying a suitable target language in which to do reasoning, into which `L1` and `L2` must compile. *Boundary terms* are added to source languages `L1` and `L2`, which give a semantics to `L2` programs from the point of view of `L1` (and vice versa). Reduction rules are defined for these boundary terms, describing, for instance, how `L1` terms wrapped in the boundary construct are mapped into `L2`. Having given a source-level account of the semantics of the composition of `L1` and `L2`, this approach next defines a *logical*

relation, relating (combined) source-language types to target-language terms. This logical relation is then used to show the final result (that the compiler for `L1` or `L2` is compositionally correct; that is, semantics preserving with respect to this logical relation).

A more recent example of work along this same line is [Ahm21].

While the relationship to the expression problem is somewhat less direct than in the other research directions we have discussed, this work represents another important point in the design space of building programming language semantics compositionally. Like GAZELLE, being able to reason about programs in this compositional context is an explicit goal. While [Ahm15] and similar work focus on correctness of compositions of outputs of different compilers, the fact that this approach does so by building a semantics of the combined language for use when specifying the correctness of the compiler makes it highly relevant. Investigating whether GAZELLE could benefit from making use of boundary terms is an interesting potential direction for future work.

1.5 The Rest of This Paper

In the remainder of this dissertation, we describe in detail the GAZELLE system's design and implementation. We then demonstrate its utility by using it to develop a formalization of a simple imperative language from separate, reusable components.

In chapter 2, we discuss the core of GAZELLE, which supports merging at the granularity of individual instructions and states. We begin with a motivating example (section 2.1) of defining a simple language by composition, introducing the primitives we'll need to define in order to compose language components. We proceed (in section (2.2) to discuss how we define composition at the level of individual instructions by merging together their output states. In doing so, we describe a series of domain-theory inspired abstractions (section 2.2.2) that enable a precise definition of this merging operation in terms of least upper bounds in an information ordering. We conclude the chapter with a discussion of GAZELLE's general control-flow machine,

which allows us to extend our definition of merging from single instructions to entire programs (section 2.3.2).

Next, in chapter 3, we discuss GAZELLE's *lifting* subsystem, which enables the notions of composition defined in chapter (2) to work on languages that are not aware of GAZELLE's information-ordering abstractions. Liftings allow the user to specify exactly *how* a language component's state fits into the combined state for the full language we are defining. In the process, we discuss (section 3.3) how we leverage ISABELLE's existing inference algorithms to allow users to express these liftings an an intuitive way.

Having described how GAZELLE enables the construction of semantics by composition, in chapter 4 we discuss how GAZELLE permits the definition of axiomatic semantics for reasoning about programs whose semantics are given by composition. We define and show sound (with respect to the semantics of the languages being composed) an unusual twist on Hoare logic, which enables us to lift existing proof rules about language components in much the same way as we were able to lift those components' semantics in chapter 3. The result is a remarkably flexible system for reasoning about languages defined by composition, enabling Hoare-style reasoning without the assumption of a "closed world" of commands typical of a Hoare-style approach.

Finally, in chapter 5, we demonstrate an extended example of the usage of GAZELLE that brings together all these pieces. We demonstrate the utility of GAZELLE by showing that it is up to the task of formalizing a small, classic imperative language without sacrificing compositionality. We then define, specify, and verify an iterative program in this language, showing that formalizations created using GAZELLE can be reasoned about conveniently, with the ability to reuse proof rules from individual language components' formalizations when doing proofs about the combined semantics.

Throughout this document, we give code snippets to illustrate various points. Much of the code is drawn directly from the real implementation of GAZELLE. Unless otherwise noted, these code snippets correspond to ISABELLE code meant to run in `Isabelle2021`. All code can

be found on GitHub at

`https://github.com/mmalvarez/gazelle/tree/dissertation-latex`.

Chapter 2

GAZELLE: A Framework for Language Composition

In this chapter we describe the core implementation of the GAZELLE framework that support GAZELLE's goal of allowing users to express and reason about the meaning of composition of programming languages. The key ingredients are a general-purpose syntax, a domain-theory inspired approach to describing composition of program states, and a minimalistic operational semantics for expressing control-flow. Together, these pieces enable us to give formal meaning to the notions of *when* the composition of two languages is well-defined, as well as *what* exactly that composition looks like (including an executable interpreter for the combined language, if we have such interpreters for the component languages). This formalization enables us to reason formally about programs expressed in this system, as described in chapter 4.

2.1 GAZELLE Example - Composing Programming Languages

In this section, we walk through an example of using GAZELLE to construct a formal programming language semantics by composition of smaller pieces. This example helps serve both to motivate the GAZELLE project and to showcase its features.

2.1.1 The Languages

Suppose we have an existing implementation of an arithmetic language, as well as an existing implementation of a memory-store. Perhaps these implementations come from different codebases, and we would prefer not to modify their implementations. We want to use these existing implementations - which we will call *sub-languages* or *language components* - to define a combined language. The combined language will have an instruction set that is essentially the *union* of the instruction sets of the components from which it is constructed.

Concretely, suppose we have a language of instructions for arithmetic on a three-register machine (two inputs, one output):

```
datatype calc =
  Add
  | Sub
  | Mul
  | Div
  | Const int
  | Skip_Calc

type_synonym calc_state = "(int * int * int)"

fun calc_sem :: "calc ⇒ calc_state ⇒ calc_state" where
"calc_sem Add (x, y, z) = (x, y, x + y)"
| "calc_sem Sub (x, y, z) = (x, y, x - y)"
| "calc_sem Mul (x, y, z) = (x, y, x * y)"
| "calc_sem Div (x, y, z) =
  (x, y, divide_int_inst.divide_int x y)"
| "calc_sem (Const i) (x, y, z) = (x, y, i)"
| "calc_sem (Skip_Calc) t = t"
```

Suppose we also have a language implementing a memory-store mapping string names

to integer values[1]. To simplify the presentation, we assume that this memory language also has access to three registers:

```
datatype reg_id =
  Reg_a
  | Reg_b
  | Reg_c

datatype mem =
  Read "String.literal" "reg_id"
  | Write "String.literal" "reg_id"
  | Skip_Mem

type_synonym mem_state = "(int * int * int * (String.literal, int)
oalist)"

fun mem_sem :: "mem ⇒ mem_state ⇒ mem_state" where
"mem_sem (Read s r) (ra, rb, rc, mem) =
  (case get mem s of
    Some v ⇒
    (case r of
      Reg_a ⇒ (v, rb, rc, mem)
      | Reg_b ⇒ (ra, v, rc, mem)
      | Reg_c ⇒ (ra, rb, v, mem))
    | None ⇒ (ra, rb, rc, mem))"
| "mem_sem (Write s r) (ra, rb, rc, mem) =
   (case r of
    Reg_a ⇒ (ra, rb, rc, update s ra mem)
    | Reg_b ⇒ (ra, rb, rc, update s rb mem)
    | Reg_c ⇒ (ra, rb, rc, update s rc mem))"
| "mem_sem (Skip_Mem) t = t"
```

We want to construct programs as sequences of instructions drawn from these two instruction sets. The operations will interact with each other by means of "overlapping" state, in which parts of `Calc`'s state are mapped to parts of `Mem`'s state, and vice versa. To construct such sequential programs, we of course need a notion of sequencing. This is provided by the `Seq` language, which is provided as a standard language component in GAZELLE. We omit details on GAZELLE's `Seq` language-component for now; these details can be found in section 5.1.4.

[1] `oalist` is simply a datatype implementing such a store as an ordered association-list; `update` is used to insert a new value

```
datatype seq =
  Seq
  | Skip_Seq
```

Finally, suppose we have a "language" that simply counts the number of operations it has executed (having only one instruction, `Op`). We'd also like our combined language to have this instruction-counting behavior.

```
datatype count =
  Op
  | Skip_Count

type_synonym count_state = "int"

fun count_sem :: "count ⇒ count_state ⇒ count_state" where
"count_sem Op x = (x + 1)"
| "count_sem _ x = x"
```

We add this language component to the example to demonstrate GAZELLE's generality: GAZELLE supports more interesting notions of composition beyond simply sequencing operations from different language-components.

2.1.2 Composition

Intuitively, we want to compose these languages so that

- The instruction-set of the combined language is the union of `Calc`, `Mem`, and `Seq`
- The semantics of sequencing operations is given by `Seq` (for brevity and clarity of exposition, we do not define `Seq` precisely here; details can be found in section 5.1.4)
- `Calc` and `Mem` share register state (allowing calculations to be written to memory, and values from memory to be used in calculations)
- *all* instructions trigger `Op`'s logic for counting number of instructions executed (this count is stored a register in the machine state separate from those used by `Calc` and `Mem`)

2.1.2.1 Manual Composition

We could achieve this composition without using GAZELLE, by manually writing a composed semantics. This looks like the following:

```
type_synonym composed_state =
  "(int * int * int * (String.literal, int) oalist * int)"

datatype composed =
  Calc calc
  | Mem mem
  | Sq "composed list"

fun composed_sem ::
  "composed ⇒ composed_state ⇒ composed_state" where
"composed_sem (Calc i) (ra, rb, rc, mem, ct) =
  (case calc_sem i (ra, rb, rc) of
    (ra', rb', rc') ⇒ (ra', rb', rc', mem, count_sem Op ct))"
| "composed_sem (Mem i) (ra, rb, rc, mem, ct) =
  (case mem_sem i (ra, rb, rc, mem) of
    (ra', rb', rc', mem') ⇒
      (ra', rb', rc', mem', count_sem Op ct))"
| "composed_sem (Sq []) st = st"
| "composed_sem (Sq (h#t)) st =
   composed_sem (Sq t) (composed_sem h st)"
```

In this small example, we can already notice several downsides to the manual approach:

- `count_sem` must be invoked separately for each case; ideally we would like to specify in a single place each instruction `Count` should be executed for.
- `Seq` is treated very differently from the non-control-flow languages `Calc` and `Mem`. In a sense we have cheated here: we would like `Seq` to be treated more like `Count`, since it describes a behavior (go to the next instruction) to be executed along with each instruction. By handling control-flow implicitly (through recursive calls to `comp_sem`), we make it much harder to accommodate composition with other language-components expressing control-flow behavior.

- Relatedly, we are combining all of these semantics into a single ISABELLE function, meaning we cannot naturally express and reason about nonterminating programs. [2]

Running the manually composed code on an example input looks like the following. First we define an example program and an initial state to begin executing:

```
definition example_prog :: "composed" where
"example_prog =
  Sq
  [ Calc (Const 1)
  , Mem (Write (STR ''x'') Reg_c)
  , Calc (Const 2)
  , Mem (Write (STR ''y'') Reg_c)
  , Mem (Read (STR ''x'') Reg_a)
  , Mem (Read (STR ''y'') Reg_b)
  , Calc Add
  , Mem (Write (STR ''result'') Reg_c)
  ]"

definition init_state :: "composed_state" where
"init_state =
  (0, 0, 0, empty, 0)"
```

Then we invoke ISABELLE's `value` command to compute the result.

```
value "composed_sem example_prog init_state"
— Result:
definition result where
"result =
(1, 2, 3,
  Oalist
   [(STR ''result'', 3), (STR ''x'', 1),
    (STR ''y'', 2)],
  8)"
```

This is as we would expect: `x` gets assigned 1; `y` gets assigned 2; and `result` gets assigned 3, the result of the addition.

[2] While ISABELLE does allow nonterminating functions in its logic, they are treated as partially defined [Bre17]. This is unacceptable if, for instance, we want to distinguish between different nonterminating executions.

2.1.2.2 Composition Using GAZELLE

Implementing this composition in GAZELLE requires several pieces. First, we define a combined syntax for the composed language:

```
datatype composed =
  Calc calc
  | Mem mem
  | Sq
```

Next we define translation functions defining how the combined syntax corresponds with the syntax of each language component. Notice how `Cond` *overlaps* with the syntaxes for `Calc` and `Mem`, denoting that their executions take place *at the same time, in parallel*[3]:

```
fun calc_trans :: "composed ⇒ calc" where
"calc_trans (Calc x ) = x"
| "calc_trans _ = Skip_Calc"

fun mem_trans :: "composed ⇒ mem" where
"mem_trans (Mem m) = m"
| "mem_trans _ = Skip_Mem"

fun seq_trans :: "composed ⇒ Seq.syn" where
"seq_trans Sq = Seq.Sseq"
| "seq_trans _ = Seq.Sskip"

fun count_trans :: "composed ⇒ count" where
"count_trans Sq = Skip_Count"
| "count_trans _ = Op"
```

After this, we define *priority functions* denoting how "collisions" between simultaneous writes to different state components are to be resolved (writes at higher priorities supersede those at lower priorities). For more information on how GAZELLE handles priorities, see sections 2.2.3.4 and 5.2.2. Note that `Seq` does not have its own priority function, as it is handled separately (for details, see section 5.1.4).

```
fun calc_prio :: "(calc ⇒ nat)" where
"calc_prio Skip_Calc = 1"
```

[3] `Seq` overlaps as well, but this is less obvious - as we will see when discussing `Seq` in more detail (section 5.1.4), this is because `Sskip` has nontrivial behavior rather than being a no-op

```
| "calc_prio _ = 2"

fun mem_prio :: "mem ⇒ nat" where
"mem_prio (Skip_Mem) = 1"
| "mem_prio _ = 2"

fun count_prio :: "count ⇒ nat" where
"count_prio (Skip_Count) = 1"
| "count_prio Op = 2"
```

Then, for `Calc` and `Mem`, we define *toggle* functions describing which instructions those languages are enabled for. This is helpful when reasoning about the combined semantics, as it gives us stronger properties relating the behavior of these language-components in isolation to the behavior of the combined language. (For more details, see section 4.6.1).

```
fun calc_toggle :: "composed ⇒ bool" where
"calc_toggle (Calc _) = True"
| "calc_toggle _ = False"

fun mem_toggle :: "composed ⇒ bool" where
"mem_toggle (Mem _) = True"
| "mem_toggle _ = False"
```

We can now begin defining the combined semantics. First, we give the combined state. The state uses *wrapper types* (described in section 2.2.3) in order to facilitate the composition of the different language-components' semantics into a single, unambiguous behavior. The state type looks like the following (note that the type parameter `'x` in `composed_state'` represents an *extension field* leaving open the possibility of further extensions to the combined-language state later):

```
type_synonym 'x swr =
  "'x md_triv option md_prio"

definition Swr :: "'x ⇒ 'x swr" where
"Swr x = (mdp 0 (Some (mdt x)))"

type_synonym ('x) composed_state' =
  "(int swr * int swr * int swr *
   (String.literal, int) oalist swr * int swr * 'x)"
```

```
type_synonym ('s, 'x) composed_state =
  "('s, 'x composed_state') control"
```

To define the composed semantics, we need one more ingredient: *liftings*, which describe how the states of each language-component are mapped into the state of the combined language. Liftings are described in more detail in chapter 3; the `schem_lift` construct makes use of the *automated lifting generator* described in section 3.3. Briefly, the first argument to `schem_lift` assigns names each part (tuple-component) of the language-component's state, and the second argument describes how those parts correspond to parts of the combined state (including details such as what wrappers are used, what priority-functions to use when updating the priority attached to state pieces, etc).

```
definition calc_lift' ::
  "(calc, calc_state, _ composed_state') lifting" where
"calc_lift' =
  schem_lift (SP NA (SP NB NC))
             (SP (SPRC calc_prio (SO NA))
                 (SP (SPRC calc_prio (SO NB))
                 (SP (SPRC calc_prio (SO NC)) NX)))"

definition calc_lift ::
  "(calc, calc_state,
    (composed, _) composed_state) lifting" where
"calc_lift = no_control_lifting calc_lift'"

definition mem_lift' ::
  "(mem, mem_state, _ composed_state') lifting"
  where
"mem_lift' =
  schem_lift
    (SP NA (SP NB (SP NC ND)))
    (SP (SPRC mem_prio (SO NA))
        (SP (SPRC mem_prio (SO NB))
        (SP (SPRC mem_prio (SO NC))
        (SP (SPRC mem_prio (SO ND)) NX))))"

definition mem_lift ::
  "(mem, mem_state,
    (composed, _) composed_state) lifting" where
```

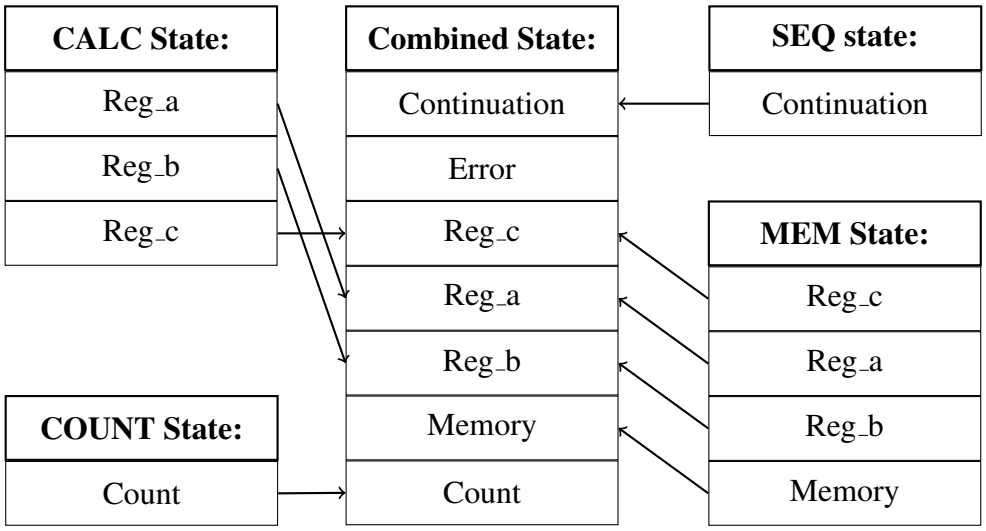

Figure 2.1: Visual depiction of lifting into the example language's combined state; arrows denote injection of data

```
"mem_lift = no_control_lifting mem_lift'"

definition count_lift' ::
  "(count, count_state, _ composed_state') lifting" where
"count_lift' =
  schem_lift NA
    (SP NX (SP NX (SP NX (SP NX (SP (SPRI (SO NA)) NX)))))"

definition count_lift ::
  "(count, count_state,
    (composed, _) composed_state) lifting" where
"count_lift = no_control_lifting count_lift'"
```

The overlaps between the different sub-language states, and how each contributes to the combined state, can be visualized in figure 2.1.

Having defined all this, we can now put together a semantics for our combined language by using GAZELLE's `pcomps` operator to compose the lifted versions of each language-component's semantics. (For more information on `pcomps`, see section 2.2; specifically section 2.2.4). The composition follows:

```
definition composed_sem ::
  "composed ⇒
```

```
   (composed, _) composed_state ⇒
   (composed, _) composed_state" where
"composed_sem =
  pcomps
    [ lift_map_t_s calc_trans calc_lift calc_toggle calc_sem
    , lift_map_t_s mem_trans mem_lift mem_toggle mem_sem
    , lift_map_s count_trans count_lift count_sem
    , seq_sem_l_gen seq_trans]"
```

Running this program on an example input looks like the following. Again, we proceed by defining the example program and its initial state:

```
definition example_prog :: "composed gensyn" where
"example_prog =
  ◊ Sq
  [ † Calc (Const 1)
  , † Mem (Write (STR ''x'') Reg_c)
  , † Calc (Const 2)
  , † Mem (Write (STR ''y'') Reg_c)
  , † Mem (Read (STR ''x'') Reg_a)
  , † Mem (Read (STR ''y'') Reg_b)
  , † Calc Add
  , † Mem (Write (STR ''result'') Reg_c)
  ]"

definition init_state :: "(composed, unit) composed_state" where
"init_state =
   (Swr [example_prog], Swr None, Swr 0,
    Swr 0, Swr 0, Swr empty, Swr 0, ())"
```

There are two primary differences to notice here. First, we use special syntactic sugar († and ◊) to conveniently express the program's syntax using GAZELLE's syntax-tree datatype, *gensyn*. For more on this, see section 2.3.1. We also need to take into account the wrapper datatypes we are using when constructing the state - hence the use of *Swr*, which applies these wrappings. For more about the wrappings used by GAZELLE, see section 2.2.3. As before, we invoke ISABELLE's *value* command to compute the result.

```
value "sem_run composed_sem 99 init_state"
— Result:
definition result where
"result =
```

```
Inl (mdp 18 (Some (mdt [])),
     mdp 0 (Some (mdt None)),
     mdp 16 (Some (mdt 1)),
     mdp 16 (Some (mdt 2)),
     mdp 16 (Some (mdt 3)),
     mdp 10 (Some
       (mdt (Oalist
             [(STR ''result'', 3),
              (STR ''x'', 1),
              (STR ''y'', 2)]))),
     mdp 9 (Some (mdt 8)), ())"
```

Further details of composition in GAZELLE will be explained later on (see in particular section 2.2 and chapter 3). What is important to notice now is that the composition `pcomps` is able to uniformly handle all of these language components, and that we are able to compose them by *wrapping* the existing definitions; that is, without needing to duplicate or modify the code for the language components (as was needed in the manual version for `Seq` and `Count`).

The GAZELLE version addresses each of the drawbacks we just listed in section 2.1.2.1. We specify the semantics of each sub-language separately, and compose them in a uniform way. Our approach to control-flow allows for handling of nonterminating programs. The implementation details of the `Seq` language, along with the GAZELLE control-flow interpreter (`sem_run`), will be discussed later in section 2.3.2.

Other than the `Count` language-component, which we include to emphasize the generality of the composition GAZELLE allows, the `Comp` language described here is actually a subset of the IMP language defined later (chapter 5) in GAZELLE. Before discussing the full IMP language, however, we will need to understand how GAZELLE renders possible this smaller example. In particular, we need to discuss the definitions of `pcomps` (section 2.2) and `sem_run`) (section 2.3). We also need to discuss liftings (chapter 3). We will turn our attention to these definitions next.

```
type_synonym ('syntax, 'state) langcomps =
  "('syntax ⇒ 'state ⇒ 'state) list"

fun pcomps :: "('syntax, 'state :: Mergeable) langcomps ⇒
('syntax ⇒ 'state ⇒ 'state)" where
"pcomps [] a b = b"
| "pcomps [lh] a b = lh a b"
| "pcomps (lh#lt) a b =
   [^ lh a b, pcomps lt a b ^]"
```

Figure 2.2: Definition of `pcomps`, which implements composition of instruction semantics

2.2 A Lightweight Approach to Instruction-Language Composition

We just saw (in section 2.1) an intuitive example of how GAZELLE can be used to develop a formalization of a programming language by composing smaller pieces together. In this section we give a more precise characterization of what we mean by language composition, and in the process give a mechanism by which we achieve it in the context of Gazelle. Our goal is to provide a definition for the `pcomps` function we encountered in our introductory example (section 2.1). Recall that the role of `pcomps` is to take a list of language-component semantics (`('a →'b →'b) list`, for syntax type `'a` and state type `'b`) and combine them into a single function giving the semantics of the combined language (`'a →'b →'b`).

The definition of `pcomps` is actually quite simple - it is essentially a *fold* of another function `bsup` over the results of the list of language-component semantics being composed, given as functions `'syntax →'state →'state` (`[^x, y^]` is simply syntactic sugar for `bsup x y`). This definition is given in figure 2.2.

Thus, in order to understand the definition of `pcomps`, it is necessary to understand this `bsup` operator: how it is defined, and the intuitions behind it. The definition of `bsup` draws on concepts from the world of denotational semantics and domain theory: in particular, the notion of an *information ordering*. With `pcomps`, we can define composition for instructions

whose semantics can be given as a total function - that is, they always have a defined result, and always terminate. We address issues related to nontermination in section 2.3.2 when discussing control-flow. This stratification of language semantics is a common pattern used in language formalizations [PAdAG$^+$21b, ch.12].

For our purposes in this section, a language is anything that can be expressed as a total function in Isabelle, `f :: syntax ⇒state ⇒state`, for some types `syntax` and `state`. This function gives the execution of an instruction (of the `syntax` type) in terms of how it operates on input program-states to produce output states (both of type `state`). To begin defining composition, we restrict ourselves to compositions of languages that *share both syntax and state types*. This is quite a severe restriction, but we show how to work around it through the use of *liftings* in chapter 3.

When composing such languages, we have a situation that looks something like the following:

```
definition f1 :: "syn ⇒ state ⇒ state"
  where
"f1 x st =
  — Actual implementation goes here
  undefined"

definition f2 :: "syn ⇒ state ⇒ state"
  where
"f2 x st =
  — Actual implementation goes here
  undefined"

definition composed :: "syn ⇒ state ⇒ state" where
"composed x st = (compose_states (f1 x st) (f2 x st))"
```

Intuitively, we want the function `composed` to do the following: take a `syn` and a `state`, and return some notion of the "composition" of the resulting states (represented by `compose_states` in the listing above). Now, of course, we need to actually implement `compose_states`. Before giving the implementation, however, it will be helpful to have some intuitions about what we *expect* from our state-composition operator - an *informal specification*

for what such composition should mean. In the next section (2.2.1), we explore by way of examples what we want out of composition, which will lead us to an approach to what state-composition should look like more generally, along with what other structure we should expect the `state` type to have in order for such a composition to make sense. We will end up instantiating `compose_states` using the `bsup` primitive.

2.2.1 State Composition: Examples and Intuitions

To simplify the exposition further, we for now set aside the issue of syntax, taking `f1` and `f2` to have types `state ⇒state`. In other words, we are looking the the compositional behavior of `f1` and `f2` *at some particular instruction*. The generalization to syntax-types with multiple instructions is somewhat straightforward, but we can give a more satisfying account of it after introducing GAZELLE's lifting machinery (see chapter 3).

These examples may seem overly simplistic, but they in fact sketch the outline of the three most important constructs used in GAZELLE's approach to composition. One thing worth noting is that the compositions described here work by "wrapping" around the `f1` and `f2`; they do not require changing the internals of the functions themselves. This is an important property of GAZELLE: when composing denotational semantics of instructions from multiple languages, we never need to modify the implementation of those languages.

2.2.1.1 Languages Operating on Distinct Tuple Components

We begin with an example where the intuitive meaning of composition is straightforward. Suppose we have `f1` and `f2` of type `syntax ⇒state1 * state2 ⇒state1 * state2`. Suppose further that `f1` only reads from and writes to the `state1` element of the tuple, and that `f2` only reads from and writes to `state2`. Equivalently (and foreshadowing a more general approach that will be explored later on when we discuss lifting in chapter 3) we could view `f1` as `fst o f1'` and f2 as `snd o f2'` for appropriate `f1'`, `f2'` (`o` here stands for function

composition).

In this case, we probably want the following for the composition of the two functions: *composed (x1, x2) = (f1' x1, f2' x2)*; that is, applying *f1'* and *f2'* componentwise to the piece of the tuple to which they are applicable. Concretely, suppose our state type is a pair *(int * int)*, where *f1* adds *1* to the first component and *f2* subtracts *1* from the second component. We would write this as:

```
type_synonym state = "(int * int)"

definition f1 :: "state ⇒ state"
  where
"f1 st =
  (case st of
    (x1, x2) ⇒ (x1 + 1, x2))"

definition f2 :: " state ⇒ state"
  where
"f2 st =
  (case st of
    (x1, x2) ⇒ (x1, x2 - 1))"

definition composed_f1_f2 :: "state ⇒ state" where
"composed_f1_f2 x =
 (case f1 x of (x1, _) ⇒
  (case f2 x of (_, x2) ⇒
   (x1, x2)))"
```

2.2.1.2 Languages Returning Optional Data

Suppose now we have two functions giving semantics to languages, *f1* and *f2*, each of which returns some optional data. More specifically, suppose they take a pair of a natural number and an optional result flag. Intuitively, the natural number corresponds to a shared input and the result flag corresponds to a shared output. However, to keep this exposition consistent with the approach actually taken in GAZELLE, which treats the state uniformly and does not distinguish between parts of the state used as inputs and parts of the state used as outputs, we simply treat this as a pair of state-elements. The first element - the input - is always left unchanged, and the

second - the output - never affects the result of the function. Suppose, further, that `f1` and `f2` will never *both* return distinct non-`None` data (or, at least, that we only require composition to be well-defined for cases where at least one result is `None`). That is, $\forall$ `x1 x2 . x1 = None ∨ x2 = None ∨ x1 = x2` for return values `x1` from `f1` and `x2` from `f2`.

Concretely, suppose `f1` and `f2` update the result flag to signal the presence of particular natural numbers in their inputs (as before, we ignore the syntax argument to simplify the presentation):

```
datatype flag =
 IS_ONE
 | IS_TWO

definition f1 :: "(nat * flag option) ⇒ (nat * flag option)" where
 "f1 p = (case p of (x, _) ⇒
    (x, (if x = 1 then Some IS_ONE
                  else None)))"
definition f2 :: "(nat * flag option) ⇒ (nat * flag option)" where
 "f2 p = (case p of (x, _) ⇒
    (x, (if x = 2 then Some IS_TWO
                  else None)))"
```

Here again, the intuitive meaning of composing `f1` and `f2` is fairly clear: to arrive at a single result, we want `x` (the unchanged input) as our first component; we want `Some IS_ONE` if `x = 1`; we want `Some IS_TWO` if `x = 2`; and we want `None` otherwise. One way of looking at this composition is using the concept of *information ordering* appearing frequently in lattice theory, domain theory, and other related formalisms [Sch88, pg.79]. If we consider `None` to be "less informative" than `Some x` (for all choices of `x`), then what we are doing is running f1 and f2 on the same state, allowing `f1` and `f2` to overwrite each other's output if their own output is "strictly more informative" than the output they are overwriting.

We could write the composition this way:

```
definition composed_f1_f2 ::
   "(nat * flag option ⇒ nat * flag option)" where
"composed_f1_f2 x =
(case x of
```

```
(x1, x2) ⇒
(case f1 (x1, x2) of
 (x1', None) ⇒ f2 (x1, x2)
 | (x1', Some x2') ⇒
  (case f2 (x1, x2) of
       (x1'', None) ⇒ (x1', Some x2')
   | (x1'', Some x2'') ⇒
          — can't occur for f1, f2 as defined above
          undefined)))"
```

2.2.1.3 Prioritized Outputs

Finally, and with the general concept of information-ordering in mind, let's look a third example of composition. Suppose again we have two functions giving language semantics, `f1` and `f2`. We'll need to consider syntax for this example to be motivated, so (unlike the previous examples) we present this one using a syntax of commands. Our setup is the following:

```
datatype syn =
 op1
 | op2

type_synonym state = "(nat * nat * nat)"

definition f1 :: "syn ⇒ state ⇒ state" where
"f1 s x =
(case x of
 (x1, x2, x3) ⇒
  (case s of
   op1 ⇒ (x1, x2, x1 + x2)
   | op2 ⇒ (x1, x2, x1 - x2)))"

definition f2 :: "syn ⇒ state ⇒ state" where
"f2 s x =
(case x of
 (x1, x2, x3) ⇒
  (case s of
   op1 ⇒ (x1, x2, x1 * x2)
   | op2 ⇒ (x1, x2,
              divide_nat_inst.divide_nat x1 x2)))"
```

That is, `f1` and `f2` each implement two different arithmetic operations; in `f1`, `op1` corresponds to addition and `op2` to subtraction; in `f2`, `op1` corresponds to multiplication and

`op2` to division. Without knowing more about what exactly we want out of our composition, we can't really say what it means to compose `f1` and `f2`; there is no clear way to impose an information-ordering on the data they produce.

However, suppose we further know that we want to define a composed language wherein the meaning of `op1` is given by `f1` and the meaning of `op2` is given by `f2`. A more general way to view this sort of composition is that for each syntax element `s`, we assign natural-number *priorities* to the results of `f1 s x` and `f2 s x` (for arbitrary state input `x`). These priorities can be seen as inducing an explicit information-ordering: when composing `f1` and `f2`, we take whichever of the two outputs has strictly higher priority (as with the previous example, we assume for now that one or the other result will be strictly greater).

If we want the assignment described above (`op1` corresponds to `f1`, `op2` corresponds to `f2`) we might write this as follows:

```
definition priority_f1 :: "syn ⇒ nat" where
"priority_f1 x =
 (case x of
  op1 ⇒ 2
  | op2 ⇒ 1)"

definition priority_f2 :: "syn ⇒ nat" where
"priority_f2 x =
 (case x of
  op1 ⇒ 1
  | op2 ⇒ 2)"

definition composed_f1_f2 :: "syn ⇒ state ⇒ state" where
"composed_f1_f2 s x =
 (if priority_f1 s > priority_f2 s then f1 s x
  else (if priority_f2 s > priority_f1 s then f2 s x
   else undefined — can't happen for priorities as defined above))"
```

In the next section, we will see how we can generalize the intuitions behind these three examples into a general framework for inducing information-orderings on program state types, enabling us to define a general notion of state-composition (and, thus, language-semantics-composition).

2.2.2 Information-Ordering In GAZELLE

GAZELLE uses several *typeclasses* to capture the behavior of types whose data satisfy different notions of *ordering* [4]. Gazelle also contains extensions of these typeclasses to implement a notion we call *mergeability* - essentially, types where least upper bounds (if they exist) of finite sets of elements can be computed. We can then give a concrete meaning to the notions of state-composition sketched out above (section 2.2.1). Composing two states means computing this least upper bound, and is only well-defined if such a least upper bound exists. In this section, we describe how these typeclasses work to achieve this goal.

2.2.2.1 Typeclasses in Isabelle: A Primer

Here we give a brief introduction to typeclasses in general, as well as some details about ISABELLE's implementation of typeclasses that set it apart from more well-known typeclass systems, such as the one found in HASKELL. For more detail about ISABELLE typeclasses, the reader should refer to [Haf21]. Typeclasses in ISABELLE allow specification of a set of operations on an abstract (i.e., parameterized) type, as well as facts about those operations. Instances for different types can then be given, which requires giving concrete implementations of the operations for that type, as well as proofs that the properties specified in the typeclass hold for that type.

Typeclasses are used to restrict polymorphism. Isabelle/HOL primarily makes use of parametric polymorphism: a function taking a parameter of a polymorphic type `'a`, for instance, must be able to handle a parameter of any type. Thus, such a function can make only very few assumptions about how its parameter can be used. When implementing a function of type `'a` $\Rightarrow$ `'a`, for instance, the only available choice is the identity function [Wad89]. [5] We know nothing

[4] ISABELLE contains its own implementation of orderings, including related typeclasses; they are largely not used in this development. Primarily this is to enable flexibility around certain notions, such as completeness of partial orders, of which GAZELLE only uses a rather weak form; and to avoid other complexities stemming from the more general notions developed in the standard library.

[5] In fact, this is not quite true in ISABELLE. ISABELLE requires all types be inhabited, and defines a polymorphic

about the structure of a datum of type `'a` that would let us modify it or create a new one, so all we can do is return back the same input we were given.

This can be a useful approach to dealing with polymorphic data in many cases. Lists, for instance, can operate uniformly no matter what type of data is stored in their elements; the behavior of the `cons` and `nil` constructors, and of the case-analysis functions that let us discriminate on list inputs, do not need information about the structure of the data within the list. However, in many cases - such as in GAZELLE - we need to know something more about the data we are working with. Consider, for example, a type of *sorted* lists - in order to be able to implement basic operations such as insertion and deletion, the implementor must be able to rely on the existence of an ordering function for list elements. As we will see shortly, we are in a similar situation when trying to implement operations enabling us to meaningfully compose together multiple states in GAZELLE. For these operations, in fact, we also need a notion of ordering.

In the remainder of this section, we will discuss several notions of ordering used in GAZELLE, along with their associated typeclasses. We will then discuss how this ordering infrastructure enables us to merge together states while maintaining the kinds of guarantees intuitively sketched out in section 2.2.1

2.2.2.2 Weak Partial Order Typeclass

We begin with the most basic notion of (information) ordering used in GAZELLE, a *weak partial order* `Pord_Weak`. A partial order on a data type `'a` consists of a binary relation `pleq`, written as `<[`, [6] on data of that type. While not strictly necessary for a binary relation to be valid as a partial order, we will tend to assume `pleq` is *computable*. That is, it can meaningfully be viewed

constant `undefined :: 'a`. So the function `(λx . undefined)` is also a function from `'a ⇒'a`, at least in a strict sense. However it is not computationally meaningful; any attempt to use the `undefined` result leads to a crash. See [Bre17] for more on this.

[6] We use this notation to avoid conflict with the notation < for the built-in less-than-or-equal-to operator in Isabelle's standard library.

as an executable function `'a ⇒'a ⇒bool`, and we will often refer to `pleq` implementations as functions for this reason. Another way to say this is that (in this development) we will tend to work with datatypes having a *decidable* less-than-or-equal-to operator.

Weak partial orders are the most basic (having the weakest axioms) notion of ordering we work with here. For a datatype with a `pleq` to be partially ordered, we require:

- Reflexivity: ∀ `(x :: 'a) . pleq x x`
- Transitivity: ∀ `(x, y, z :: 'a)` if `pleq x y` and `pleq y z`, then `pleq x z`.

We can write this in ISABELLE as follows:

```
class Pord_Weak =
  fixes pleq :: "'a ⇒ 'a ⇒ bool" (infixl ⟨<[⟩ 71)
  assumes
    leq_refl : "pleq a a"
  assumes
    leq_trans : "pleq a b ⟹ pleq b c ⟹ pleq a c"
```

Orderings satisfying `Pord_Weak` are missing an important property that makes them less useful for our purposes - namely, they do not require *antisymmetry*, the property that ∀ `(x, y :: 'a)`, if `pleq x y` and `pleq y x` then `x = y`. We generally need this property in order to be able to convert abstract facts about the relative order of pieces of data to concrete facts about their exact contents. Nonetheless, many basic facts and utilities related to orderings do not depend on antisymmetry; `Pord_Weak` is a useful abstraction on which to prove these.

In particular, with `Pord_Weak` we can define notions of *upper bound* and *least upper bound* (also called *supremum* or *sup*) of sets in a standard way.

```
definition is_ub :: "('a :: Pord_Weak) set ⇒ 'a ⇒ bool" where
"is_ub A a =
  (∀ x ∈ A . pleq x a)"
```

An *upper bound* of a set of type `S :: 'a` is any datum of type `'a` that is greater than or equal to (according to `pleq`) all elements of the set `S`.

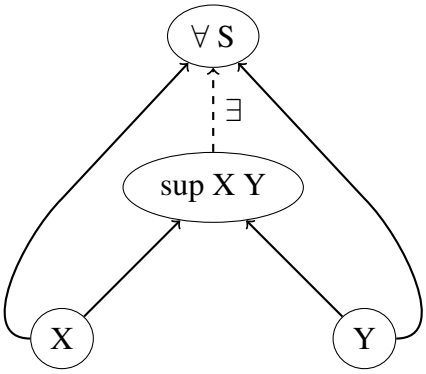

Figure 2.3: Visual depiction of `is_sup`; arrows denote `pleq`

```
definition is_least :: "(('a :: Pord_Weak) ⇒ bool) ⇒ 'a ⇒ bool"
where
"is_least P a =
   (P a ∧
    (∀ a' . P a' ⟶ pleq a a'))"
```

A *least* datum `x :: 'a` satisfying a predicate `P :: 'a → bool` is any `x` satisfying `P` that is also less than or equal to all other data satisfying `P`.

```
definition is_sup :: "('a :: Pord_Weak) set ⇒ 'a ⇒ bool" where
"is_sup A a =
  is_least (is_ub A) a"
```

The supremum or least upper bound of a set `S :: 'a` is simply that: the least upper bound of the set, according to the definitions just given. For convenience, we also define predicates `has_ub` and `has_sup`, which existentially quantify the (least) upper bound rather than giving it explicitly.

```
definition has_ub :: "('a :: Pord_Weak) set ⇒ bool" where
"has_ub A = (∃ s . is_ub A s)"
```

```
definition has_sup :: "('a :: Pord_Weak) set ⇒ bool" where
"has_sup A = (∃ s . is_sup A s)"
```

We can graphically represent the supremum of two pieces of data as follows (figure 2.3):

At this point we define another notion on weak partial orders - *biased supremum* or *bsup* - that will be useful later when defining merging of states.

2.2.2.3 Biased Supremum

Intuitively, suppose we are merging two states `x` and `y` (of type `'a :: Pord_Weak`, equipped with an information ordering). When computing `merge x y`, it is clear that if `x` and `y` have a least upper bound, then `merge x y` should be equal to this least upper bound. In other words, if possible we'd like to return a result that is consistent with - that is, at least as informative as - both `x` and `y` (upper bound). We'd also like to return the "informationally minimal" such result (then least upper bound). However, `x` and `y` are not guaranteed to have a least upper bound; we need to decide what to do in that case.

One option would be to have `merge` return a result wrapped in an `option` type; that is, make it a partial function. We instead choose an approach that allows `merge` to be total and always return a meaningful result that coincides with the least upper bound when one exists. When no least upper bound exists, we "do the best we can" to return meaningful data that is intuitively "as close as possible" to being a least upper bound. This is the role of `bsup`, which we now define.

```
definition is_bsup :: "('a :: Pord_Weak) ⇒ 'a ⇒ 'a ⇒ bool" where
"is_bsup a b s =
  is_least (is_bub a b) s"

definition is_bub :: "('a :: Pord_Weak) ⇒ 'a ⇒ 'a ⇒ bool" where
"is_bub a b s =
  (pleq a s ∧
    ((∀ bd sd . pleq bd (b) ⟶
                is_sup {a, bd} sd ⟶
                pleq sd (s))))"
```

Similar to how we defined least upper bound above, we first define *biased upper bound* (bub), and then define `bsup` as the least such biased upper bound. The definition of biased upper bound - `is_bub` - is more interesting. If `is_bub a b s` holds, we say that `s` is a *biased upper bound* of `a` and `b`, *biased toward* `a`. If this is the case, we know `a <[ s`. That is, we know the result will be consistent with `a` (this is why we say that we are biased toward `a`). However, we also require that, `s` is consistent with "as much of the data in `b` as possible". Formally, for any

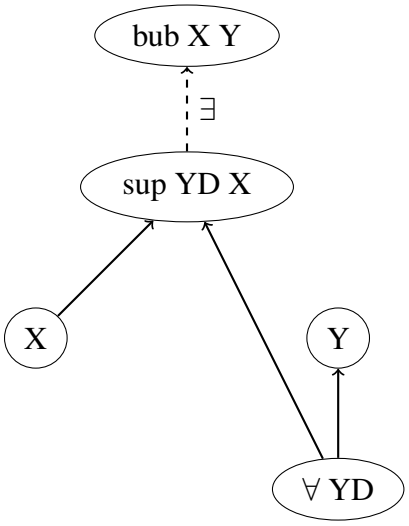

Figure 2.4: Visual depiction of `is_bub`; arrows denote `pleq`

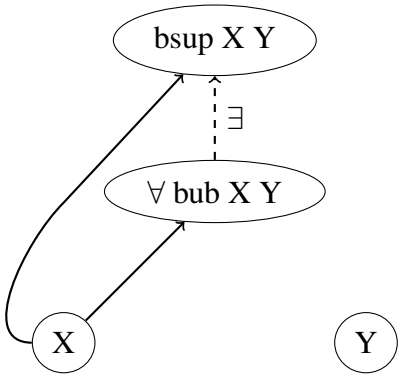

Figure 2.5: Visual depiction of `is_bsup`; arrows denote `pleq`

less-informative data consistent with `b` (`bd` in the listing above) having a supremum with `a` (`sd` in the listing above being that supremum), then that supremum is less than the biased upper bound.

We graphically represent biased upper bound and biased supremum in figures 2.3 and 2.5.

Though not obvious from this definition, it turns out that `is_bsup a b s` satisfies the property that if a least upper bound of `a` and `b` exists, then `s` will equal this least upper bound (we prove this in ISABELLE). Additionally, even when a least upper bound does not exist, we can still rely on several properties that will hold on `bsup` no matter what.

Making more aggressive use of the full generality of `bsup` is future work.

2.2.2.4 Partial Order and Complete Partial Order

If we add antisymmetry to *Pord_Weak*, we get a typeclass corresponding to data with a partial ordering - *Pord*. In ISABELLE we implement this by extending the *Pord_Weak* typeclass:

```
class Pord =
    Pord_Weak +
    assumes leq_antisym : "pleq a b ⟹ pleq b a ⟹ a = b"
```

As mentioned above, without antisymmetry it is difficult to prove facts about concrete program states (most naturally expressed in terms of equalities) from information-ordering inequalities. Notably, antisymmetry allows us to prove the uniqueness of least and greatest set elements (thus, uniqueness of least upper bound as well as bsup).

Continuing to extend our partial-order abstraction, we next add an assumption of completeness. In its most general form, [7] completeness of a partial order states that any set which has an upper bound also has a least upper bound. For our purposes here, it suffices and is convenient to have a somewhat weaker formulation: namely, that finite sets of elements of a complete, partially-ordered type have a supremum. This is because we intend to use the partially-ordered datatypes as a means by which to merge program states corresponding to different language fragments. In most reasonable use-cases (e.g. composing two or more existing languages), there will only be a finite number of such fragments, so we need not worry about the existence of suprema of infinite sets. (Indeed, in general, there is no reason to expect we could compute such suprema even if they could be shown to always exist, and being able to produce an executable interpreter by means of calculating these suprema is our goal here).

The typeclass for complete partial orders - *Pordc* - looks like this:

```
class Pordc =
  Pord +
  assumes complete2: "has_ub {a, b} ⟹ has_sup {a, b}"
```

[7] Several different definitions of completeness can be found in the literature; the one used here corresponds most closely to directed-completeness, albeit in a weaker form restricted only to finite sets.

Because we are restricting ourselves to finite subsets, it suffices to show an even weaker completeness axiom applying only to sets with two elements. We can then prove the more general (finite)-completeness statement as a theorem by induction on the size of the finite set of elements having an upper bound to show that it indeed has a least upper bound.

2.2.2.5 Additional Partial-Order Extensions

There are a few more related notions that need to be developed on top of the partial-ordering infrastructure described above.

Often we will want to restrict ourselves to working with partial orders with a least element. We call these *Pordb* ("partial order with base"); in the literature they are often called *pointed* partial orders. They are defined as follows:

```
class Pord_Weakb = Pord_Weak +
fixes bot :: "'a" ("⊥")
assumes bot_spec :
  "⋀ (a :: 'a ) .  pleq bot a"

class Pordb =  Pord + Pord_Weakb
```

As we will see later in (section 2.2.4.1), when reasoning about merged states, we will sometimes find it useful to be able to prove that the union of two finite sets each having a least upper bound has its own least upper bound. *Pordps* ("partial order with pairwise suprema") captures this.

```
class Pordps =
  Pord +
  assumes pairwise_sup :
    "has_sup {a, b} ⟹ has_sup {b, c} ⟹ has_sup {a, c} ⟹
     has_sup {a, b, c}"
```

Some of the *pordc* datatypes we work with obey a particularly strong property: that any finite set has an upper bound (hence, a least upper bound due to completeness):

```
class Pordc_all = Pordc +
  assumes ub2_all : "⋀ a b . has_ub {a, b}"
```

Finally, we have typeclasses capturing two concepts that are somewhat orthogonal to the notion of ordering but worth mentioning here because they are frequently combined with the ordering typeclasses defined above. The more interesting of these, `Okay`, marks types having a defined subset `ok_S` that we consider to be suitable for projecting valid data out of.

```
class Okay =
  fixes ok_S :: "('a) set"
```

For example, the instantiation for `option` is

```
instantiation option :: (Okay) Okay
begin
definition option_ok_S : "(ok_S :: 'a option set) = (Some ` ok_S)"
instance proof qed
end
```

In other words, for any type `'a`, the set `ok_S :: 'a option` is just the set of all elements constructed using `Some x` for `x :: 'a`. Note that unlike the other typeclasses we've seen thus far, `Okay` does not come with any theorems; the fact that `ok_S` corresponds to data that can "validly be projected out" is entirely a matter of convention (indeed, it's not fully clear what this would mean in general). We will see `Okay` used later on (in section 4.3), when discussing how we handle reasoning about merged program states and merged semantics functions.

Another typeclass of a more purely technical nature is `Bogus`, which assigns to a data type a "default" element that can be returned when an operation has no valid result (for instance, when trying to project a value of type `'a` out of `None :: 'a option`). The reason this is necessary is due to a quirk in Isabelle's code generator: while all datatypes are inhabited [Bre17], and so `undefined :: 'a` can always be returned instead of a valid result, we have observed that code generation using `undefined` values is very brittle. When `undefined` is encountered in executing code, it usually leads to a crash (typically a pattern-match error); even in code that seems that it should avoid these cases, it will often happen that if an `undefined` is lurking somewhere in the code it will get executed anyway, due to the fact that ISABELLE's code generator

works with an eager evaluation order. [8]

The `Bogus` typeclass definition is quite simple:

```
class Bogus =
  fixes bogus :: "'a"
```

We give instances for all the datatypes we work with in GAZELLE. For example, the following are the instances for natural numbers `nat` and `'a option`.

```
instantiation nat :: Bogus begin
definition nat_bogus : "bogus = (0 :: nat)"
instance proof qed
end

instantiation option :: (Bogus) Bogus begin
definition option_bogus : "bogus = Some bogus"
instance proof qed
end
```

Note that the instance for `'a option` requires that the parameter `'a` itself implement `Bogus`. While we could have chosen `bogus :: 'a option` to be `None`, this would violate the spirit of the `Bogus` typeclass, since we generally want to regard `None :: 'a option` as representing the lack of a valid `'a`, whereas `Bogus` needs to return an arbitrary (but valid) `'a`.

2.2.2.6 The Mergeable Typeclass

We now have the terminology and primitives needed to define exactly what we want to require of a datatype for it to be suitable for supporting GAZELLE's merging operations. We want a complete partial order (possibly with a least element) for which the biased supremum of any two pieces of data can be computed. This is captured in the `Mergeable` typeclass, which is defined as follows:

[8] ISABELLE does support code generation into the lazy language HASKELL, but this is not as well integrated as Isabelle's default, ML-based code generator, which can be used easily from within Isabelle itself (for instance, when trying to quickly compute the value of a constant using the `value` command.) `Bogus` allows us to avoid this problem by instead using data that will not cause pattern-match errors that crash the program prematurely.

```
class Mergeable =
  Pordc +
  fixes bsup :: "('a :: Pordc) ⇒ 'a ⇒ 'a" ("[^ _, _ ^]")
assumes bsup_spec :
  "⋀ a b . is_bsup a b (bsup a b)"
```

That is, a `Mergeable` is any `Pordc` that additionally provides a function `bsup`, for which `bsup a b` (also written as `[^a, b^]`) is the biased supremum of `a` and `b` as defined above (section 2.2.2.3). `Mergeable` also has variants corresponding to complete partial orders having least elements, pairwise suprema, and/or all suprema. For example, `Mergeableb` captures `Mergeable` datatypes having a least element:

```
class Mergeableb = Mergeable +
  Pordbc
```

2.2.2.7 Summary: GAZELLE's Ordering Typeclasses

This concludes our discussion of the hierarchy of typeclasses used in GAZELLE to characterize types supporting a useful merge operation (namely, `bsup`). In the next section, we will see how we instantiate these typeclasses to create types suitable for representing program-states that can be meaningfully merged. Figure 2.6 gives a schematic diagram of the typeclasses described above, and how they inherit from each other.

2.2.3 Wrapper Types for Mergeable States

The ordering abstractions we've just defined (section 2.2.2) allow us to define what it means to merge states, as long as the types of those states come equipped with a partial order. However, most data types do not come with an obvious ordering, and for those that do, the "native" ordering is generally not suitable as an information ordering. (Consider integers, for instance: `1 <= 2`, but both represent distinct data; we want to consider neither to be more informative than the other). The solution adopted by GAZELLE is to provide a set of "wrapper" types that can be used to impose a partial-ordering structure on arbitrary data. In this section, we discuss the

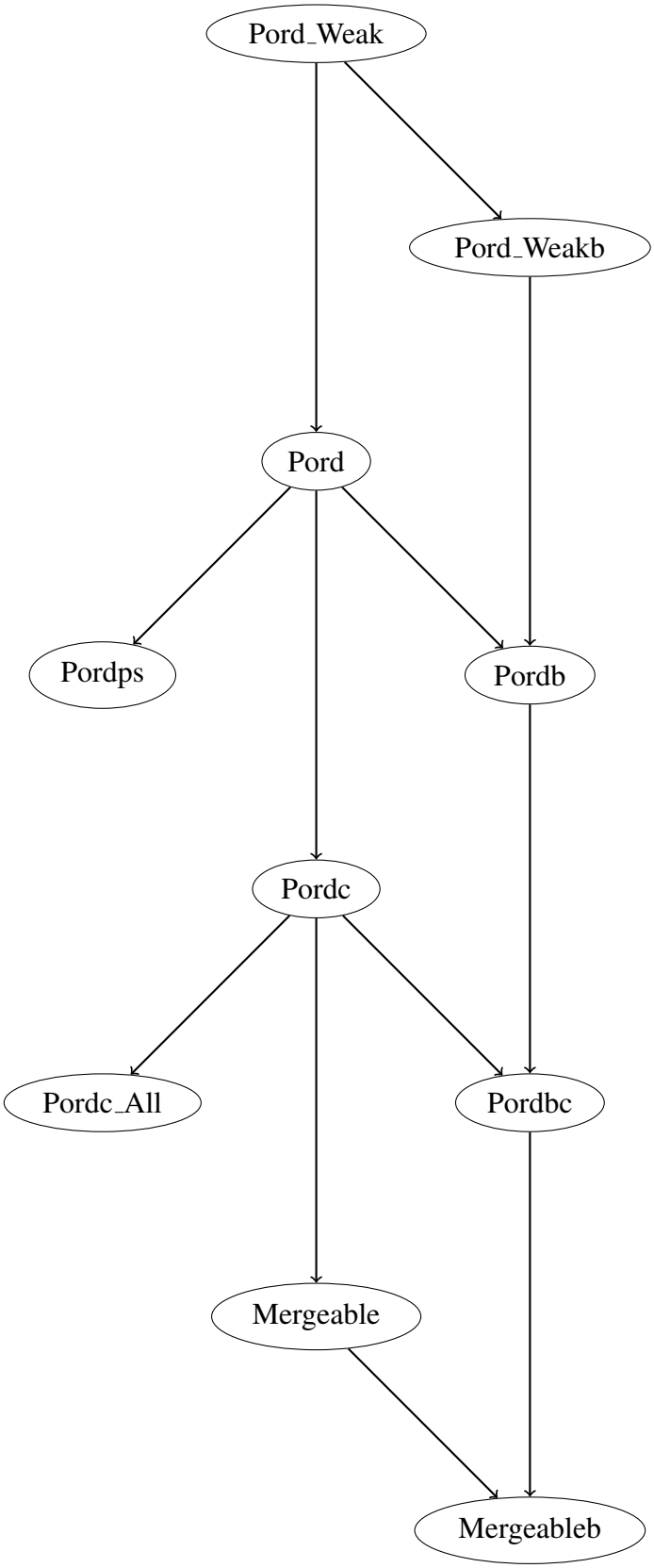

Figure 2.6: Visual depiction of GAZELLE ordering typeclasses; arrows indicate inheritance

most important of these wrapper-types and how they implement different pieces of the typeclass hierarchy described in section 2.2.2.7.

It is worth emphasizing that the user need not use these wrapper types to take advantage of GAZELLE's tools for defining and reasoning about merged semantics. Any `Mergeable` datatype will also work, including datatypes with more highly customized information orderings. These can be defined by the user of GAZELLE.

2.2.3.1 Trivial Ordering

When faced with arbitrary datatype (that does not come with its own information ordering), we have few options when trying to impose a useful ordering on that type. Because the datatype represents some kind of program state, however, it is reasonable to assume that we can compute an equality check on it. Leveraging this fact gives us the *trivial ordering*, in which we define `<[` to be simply = (i.e., the equality on the underlying type). We use a wrapper datatype in case the inner data already has another `Pord` ordering defined on it, since typeclasses in ISABELLE need to have a single, unambiguous instance for a given type in order for typeclass inference on that type to succeed. The wrapper datatype, as well as the instances for it, are given here:

```
datatype 'a md_triv =
  mdt 'a

instantiation md_triv :: (_) Pord_Weak
begin
definition triv_pleq : "(a :: 'a md_triv) <[ b = (a = b)"
```

The proofs are omitted here, but in our development we show that the trivial ordering satisfies the laws of `Pord` (and, thus, also `Pord_Weak`), by virtue of the fact that equality is also transitive and reflexive. It also satisfies the `Pordc` completeness law: if s is an upper bound of some set S, then $S = \{s\}$, and so s is also the least upper bound of S. Additionally, it obeys the pairwise-supremum law (`Pordps`) as well as having a `Mergeable` instance:

```
instantiation md_triv :: (_) Mergeable
begin
```

definition `triv_bsup : "[^(a :: 'a md_triv), b^] = a"`

However, the trivial ordering is not guaranteed to have a least element (indeed, will not have a least element unless the type in question is isomorphic to the unit type; that is, has only a single distinct element). Additionally, a trivially-ordered type on its own is not especially useful for merging states - there are no least upper bounds for non-equal data, and `bsup a b = a` for all a. Fortunately, we can do better than this in many cases. The remainder of the instances described below make use of additional information to impose a more interesting ordering more amenable to merging.

2.2.3.2 Ordering on Optional Data

When working with partial orders, one common trick [Sch88, pg.104] is to induce a least element by fiat.[9] Namely, we can take an existing partially ordered set and add a new element to it defined to be less than or equal to all other elements in the set. We do precisely the same thing in GAZELLE by creating a `Pordb` instance for the `option` type. In our context, the new (least) element being added to the order is simply `None`, while `Some x <[Some y` iff `x <[y`. This corresponds to the intuition sketched out in section 2.2.1.2. The implementation is as follows:

```
instantiation option :: (Pord_Weak) Pord_Weak
begin
definition option_pleq : "(x :: 'a option) <[ y =
(case x of
      None ⇒ True
      | Some x' ⇒
        (case y of
          None ⇒ False
          | Some y' ⇒ (pleq x' y')))"
```

Unlike with the trivial ordering, we require that the `'a` data inside the `'a option` type be equipped with a `Pord_Weak` ordering. This is the ordering used to compare two elements

[9] [Sch88] refers to this construction as *lifting*. It is related to but distinct from the notion of lifting defined in this paper in chapter 3.

wrapped in *Some* constructors. The purpose of *option* as a wrapper type is to enable the following *Pord_Weakb* instance:

```
instantiation option :: (Pord_Weak) Pord_Weakb
begin

definition option_bot : "bot = (None :: 'a option)"
```

By using *None* as the least element, we are able to wrap any *Pord_Weak* ordering to get a new ordering with a base element (*Pord_Weakb*). We define a *Mergeable* instance for *'a option* as follows:

```
instantiation option :: (Mergeable) Mergeableb
begin
definition option_bsup: "[^(x :: 'a option), y^] =
  (case x of
    None ⇒ y
    | Some x' ⇒ (case y of
                    None ⇒ Some x'
                    | Some y' ⇒ Some (bsup x' y')))"
```

option also has instances for *Pordc*, *Pordps*, and the others listed above in section 2.2.2, provided the base type (*'a*) satisfies the laws for those typeclasses.

2.2.3.3 Ordering on Tuples

When modeling programming-language states, we will need a treatment of compound data consisting of multiple fields, each with possibly different types. We saw an example of this in section 2.2.1.1. In GAZELLE, we find it convenient to model such data as tuples (product types) [10]. Naturally, this means we will also want to have partial-ordering typeclass instances for tuple types. N-tuples in ISABELLE are represented as nested pairs (for instance, the triple *(a, b, c)* is just a shorthand way to write *(a, (b, c))*. So we turn our attention to defining orderings on pairs of (ordered) data.

[10] While we do not explicitly address other ways of structuring data, such as records, the approaches described here can be extended easily to other compound datatypes - especially in ISABELLE, where a record type is little more than a fancy tuple with some extra syntactic sugar for field accesses and record construction [NPW21, pg.152].

When considering how to impose a partial order on a product type `'a * 'b`, where `'a` and `'b` each come equipped with a partial order, it might seem like there are a few natural options. One option (the one we implement) is to say that `(x1, y1) <[(x2, y2)` only when both components on the left-hand side are less than or equal to their corresponding components on the right-hand side (that is, `x1 <[ x2` and `y1 <[ y2`). Another seemingly reasonable choice would be to use a lexicographic order: `(x1, y1) <[ (x2, y2)` when either `x1 <[ x2` (with `x1 ≠x2`), or `x1 = x2` and `y1 <[ y2`. We choose the former for a few reasons:

- Since our goal is to establish an information ordering, it does not really make sense to privilege one component of the tuple over the other when determining whether two tuples are informationally compatible. Intuitively, for two tuples to be compatible, all their components must respectively be compatible.

- In a lexicographic ordering, we cannot prove completeness without additional assumptions on the first component - even if we assume the existence of a least element. To see this, consider an ordered datatype where `a <[b`, `a <[c`, but `b` and `c` are not comparable. Then the pairs `(a, b)` and `(a, c)` share as upper bounds `(b, ⊥)`, `(c, ⊥)`, but neither is less than or equal to the other.

- Most of the benefits of lexicographic ordering for our purposes can be achieved using the specialized `md_prio` datatype, described below (section 2.2.3.4). The aforementioned requirements on the first component impose so many requirements that we end would up assuming that the first component look a lot like natural numbers anyway.

Here are our instances for products:

```
instantiation prod :: (Pord_Weak, Pord_Weak) Pord_Weak
begin
  definition prod_pleq :
  "(x :: 'a * 'b) <[ y =
    (case x of
        (x1, x2) ⇒ (case y of
                      (y1, y2) ⇒ (pleq x1 y1 ∧ pleq x2 y2)))"
```

```
instantiation prod :: (Pord_Weakb, Pord_Weakb) Pord_Weakb
begin
definition prod_bot :
  "(bot :: 'a * 'b) = (bot, bot)"
```

We show that product types `('a * 'b)` additionally obey the laws of the other typeclasses listed above in section 2.2.2, assuming their components `'a` and `'b` do. The `Mergeable` implementation is the following; we simply merge the tuple elements componentwise.

```
instantiation prod :: (Mergeableb, Mergeableb) Mergeableb
begin

definition prod_bsup :
"[^ a, b ^] =
  (case a of
    (a1, a2) ⇒ (case b of
                  (b1, b2) ⇒ (bsup a1 b1, bsup a2 b2)))"
```

One notable limitation here is that we require the existence of a least element (i.e., that the elements of the tuple implement the `Mergeableb` typeclass rather than just `Mergeable`). It turns out that without such a least element, we are unable to prove that the `bsup` implementation satisfies the `is_bsup` specification. This is one of many reasons the `option` instance (described in section 2.2.3.2) is so useful.

2.2.3.4 Priority Ordering

The final important `Pord` and `Mergeable` instances are for the following type:

```
datatype 'a md_prio =
  mdp nat 'a
```

In other words, `'a md_prio` is a pair of a natural number - the *priority* - and a piece of data of type `'a`. The goal of the `'a md_prio` wrapper type is to sort out conflicts when merging data (of type `'a`) that cannot be resolved simply by consulting the partial order defined for `'a`. This corresponds, for example, to the case we already saw in section 2.2.1.3, in which we have two languages, each with the same two instructions, both writing the same piece of data. `md_prio`

is used in such a situation: we assign (just as in the example) a priority to each language's output, and our final (merged) result is whichever of the two has greater priority.

We give a *Pord_Weak* instance for *'a md_prio* (provided that we are dealing with a *'a* which itself has a *Pord_Weak* instance):

```
instantiation md_prio :: (Pord_Weak) Pord_Weak
begin
definition prio_pleq :
"x <[ y =
  (case x of
      mdp xi x' ⇒
      (case y of
        mdp yi y' ⇒
        (if (xi ≤ yi) then
          (if (yi ≤ xi) then
            pleq x' y'
            else True)
          else False)))"
```

Likewise, if *'a* has a *Pord_Weakb* instance, so does *'a md_prio*:

```
instantiation md_prio :: (Pord_Weakb) Pord_Weakb
begin

definition prio_bot :
"⊥ = mdp 0 bot"
```

This is simply (a special case of) a lexicographic approach: when comparing two data of type *'a md_prio*, we first compare their natural-number priorities, then (if the priorities are equal) we compare the data (of type *'a*). Note that <= above is Isabelle's built-in comparison operator for natural numbers, not *pleq*. In order to prove the instances we need for *'a md_prio*, we need to rely on further properties of comparisons on natural numbers - including *linearity*. Linearity says that natural-number comparison admits a trichotomy law: for any *n :: nat, n' :: nat*, *n < n' ∨n = n' ∨n > n'*. We also rely on the fact that natural numbers are not dense; i.e., there is no natural number between *1* and *2*, for example.

We provide instances for our other ordering typeclasses: *'a md_prio* is a *Pord* if *'a* is; is *Pord_Weakb* if *'a* is *Pord_Weakb*; and is a *Pordbc* if *'a* is a *Pordbc*. As with the instance

for product, we require the existence of a least element in order to show completeness.

We also show that if *'a* is a *Pordbc* (that is, partial order with a least element and completeness), *'a md_prio* is a *Pordc_all* - that is, *any* two elements of type *'a md_prio* have a least upper bound. For example, if we have *x, y :: 'a* incomparable, then the supremum of *mdp 1 x* and *mdp 1 y* is *mdp 2* $\bot$. This is convenient but can also be dangerous, since $\bot$ is not likely to correspond to a meaningful datum (e.g. $\bot$*:: 'a option* is *None*).

'a md_prio also has a *Mergeable* instance, which is as follows (since *Mergeable* depends on *Pordc*, we require a least element *bot :: 'a md_prio*):

```
instantiation md_prio :: (Mergeableb) Mergeableb
begin

definition prio_bsup :
"bsup a b =
  (case a of
    mdp ai a' ⇒
      (case b of
        mdp bi b' ⇒
          (if ai ≤ bi then
            (if bi ≤ ai then
              (if pleq b' (bsup a' b') then
                   mdp ai (bsup a' b')
                   else mdp (1 + ai) bot)
                          else mdp bi b')
       else mdp ai a')))"
```

This captures the intuition given in the examples above: for unequal priorities, we return the datum with the larger priority, at that priority. For equal priorities, compute *bsup* of the contained data *a'* and *b'*, and check to see if it is the least upper bound (this will happen if and only if *b' <[[^a', b'^]*). If it is, we return this least upper bound of the data, at a priority equal to the input priorities. If *[^a', b'^]* does not give a least upper bound for *a'* and *b'*, then no such least upper bound exists. In this case we return the least element, at a priority one step higher.

2.2.3.5 Summary

At this point, we have described the key typeclasses and instances used by GAZELLE to enable the merging behavior described in our motivating examples (from section 2.2.1). The `Mergeable` instance for products allows us to handle our first example; the instance for `option` handles the second; and `md_prio` handles the third. Taken together, these provide a solid foundation for formally defining merging of states (via `bsup`) in a meaningful way.

However, we still need to develop abstractions for reasoning about such merged data: one of our goals, after all, is to enable composition and reuse not just of language semantics but of tools for reasoning about them. Additionally, we still need to describe a way to make the transition from the types used to represent state in existing language developments - which may not be equipped with an information order - into the information-ordering framework we've just described.

In section 2.2.4 and chapter 3, respectively, we will describe how we solve these problems, completing the picture of the denotational fragment of GAZELLE. Before doing so, we must define a few more notions related to information ordering in GAZELLE.

2.2.4 Composition and Preservation of Suprema

Now that we have defined biased supremum (`bsup`), (section 2.2.2.3), we have a means of composing two states, as well as a way of saying whether that composition is meaningful. Namely, the composition of states `a` and `b` is `bsup a b`, and the composition is well-defined in the event that `a` and `b` have a least upper bound. This also gives us a way to describe the composition of two semantics functions sharing a state and syntax type: `compose f1 f2 = (λsyn st . [^f1 syn st, f2 syn st^])` We can naturally extend this to compositions of more than two functions, arriving at the definition we from figure 2.2.

As mentioned above, we essentially fold the binary composition `bsup` over the outputs

of the functions (i.e., states) to be composed. We know that, when `a` and `b` have a least upper bound, the composition *on states* given by `[^a, b^]` is commutative. `[^a, b^]` is the least upper bound of the set {`a, b`}, which equals the set {`b, a`}, whose least upper bound is `[^b, a^]`; since least upper bounds are by definition unique, we therefore know `[^a, b^] = [^b, a^]`.

By a similar token, if we know the partial order we are working with is complete (`Pordc`) we can also show `bsup` to be associative on states: `[^[^a, b^], c^] = [^a, [^b, c^]^]`. The proof of this fact is slightly nontrivial, but can be found in the formal development[11].

When composing semantics *functions* using `pcomps` - rather than single states using `bsup` - we also wish to show that the composition is commutative and associative under some appropriate assumption. That assumption - the analogue of the existence of a least upper bound when composing states - is defined as follows:

```
definition sups_pres ::
  "('a ⇒ ('b :: Pord) ⇒ 'b) set ⇒ ('a ⇒ 'b  set) ⇒ bool" where
"sups_pres Fs S =
  (∀ x syn Fs' f .
    x ∈ S syn ⟶
    Fs' ⊆ Fs ⟶
    f ∈ Fs' ⟶
    (has_sup ((λ f . f syn x) ` Fs')))"
```

We define this as a predicate - `sups_pres`, short for "suprema are preserved" - that takes a set of semantics functions, as well as a function assigning to each syntax element (of type `'a`) a set of "valid" states [12]. The predicate says that, if

- we start in a valid state, `x` (valid for some piece of syntax `syn`), then
- for any nonempty subset `Fs'` of our set of input functions `Fs`,

[11] Like all the other theorems discussed in this dissertation, this proof is machine-checked in ISABELLE; this one can be found in `Gazelle/Mergeable/Mergeable.thy`

[12] The use of this parameter is to allow us to restrict ourselves to reasoning about states that can have data projected out of them in a valid way (see section 2.2.2.5); without this added assumption the conclusion does not hold for state types that have `option` or similar in them.

- the set of results after applying each `f ∈Fs` to `syn` and `x` has a least upper bound.

Under the assumption that the functions being composed using `pcomps` obey `sups_pres`, we can show that if two such lists of functions have the same set-representation, the result of `pcomps` on those lists is the same. This implies, among other things, the commutativity and associativity facts that we want. Formally, the theorem is:

```
lemma pcomps_set_eq :
  assumes H : "sups_pres Fs S"
  assumes Hf : "f ∈ Fs"
  assumes Hl1 : "set l1 = Fs"
  assumes Hl2 : "set l2 = Fs"
  assumes Hx : "x ∈ S syn"
  shows "pcomps l1 syn x = pcomps l2 syn x"
```

2.2.4.1 Proving `sups_pres`

Proving `sups_pres` directly using its definition is challenging, since it requires reasoning about all subsets of functions in the set `S` of functions in question; in other words, in general we might expect proofs of `sups_pres` to grow exponentially with the size of `S` (i.e., linear in the size of the power-set of `S`). Fortunately, it turns out that in many realistic cases we can get away with a much simpler proof technique, provided we know a bit more about the functions in `S`.

First, we notice that proving `sups_pres` is trivial for singleton sets and relatively straightforward for pairs.

```
lemma sups_pres_singletonI :
  "⋀ S f .
    sups_pres {f} S"
lemma sups_pres_pairI :
  fixes Fs
  fixes f
  fixes S :: "'a ⇒ ('b :: Pord) set"
  assumes Sups : "⋀ x s . x ∈ S s ⟹ has_sup {f1 s x, f2 s x}"
  shows "sups_pres {f1, f2} S"
```

When working with state types that implement the `Pordps` typeclass (see section 2.2.2.5), we can prove `sups_pres` for finite sets of functions by proving that all pairs of functions obey

sups_pres. This enables us to show *sups_pres* with a polynomial (squared) rather than an exponential amount of effort. One way to express this approach is using the following theorem, which enables us to add elements to existing sets satisfying *sups_pres*:

```
lemma sups_pres_insert :
  fixes Fs
  fixes f
  fixes S :: "'syn ⇒ ('x :: Mergeableps) set"
  assumes Hf : "sups_pres {f} S"
  assumes HFs : "sups_pres (set fs) S"
  assumes Pairwise : "⋀ g . g ∈ set fs ⟹ sups_pres {g, f} S"
  shows "sups_pres (set (f#fs) ) S"
```

Another important case where we can more easily prove *sups_pres* is the case where we are reasoning about datatypes that belong to the *Pordc_all* typeclass (that is, types for which completeness holds and for which all suprema are guaranteed to exist). This case arises frequently in the extended IMP example in chapter 5, as well as in other contexts in which *md_prio* is used to wrap all state elements in priorities. In such cases, we can immediately derive *sups_pres* for all nonempty, finite sets of functions *'a ⇒'b ⇒'b*, where *'b :: Pordc_all*:

```
lemma sups_pres_finite_all:
  fixes Fs :: "('a ⇒ ('b :: Pordc_all) ⇒ 'b) set"
  assumes Nemp : "f ∈ Fs"
  assumes Fin : "finite Fs"
  shows "sups_pres Fs S"
```

While these special cases are quite useful due to how often they arise in practice (indeed, these cases suffice to show all the *sups_pres* proofs required in our formalization of IMP in chapter 5), it is important to remember that *sups_pres* is a more general notion and can handle compositions of functions that are valid for more interesting reasons - albeit at the cost of requiring additional proof effort.

2.2.5 Summary

In this section, we have covered how we use an information-ordering-based approach to define composition of programming-language semantics in a precise way. In the following

sections, we will see how we extend this approach to handle control-flow by adding an operational perspective to our notion of evaluation (section 2.3.2). Then, in chapter 3, we will cover how we use *liftings* to adapt this approach to work with existing semantics functions and state types.

2.3 Extending Composition to Multi-Step Executions

Thus far (section 2.2), we have described how GAZELLE assigns meaning to individual instructions in a way that enables a formal definition of merging of machine states (and, thus, merging of instruction-language semantics). However, this still does not provide a complete picture of composition of programming languages - real programs, after all, generally have multiple instructions and some notion of *control-flow* providing an order to their execution. In this section, we describe how we build on the framework developed so far to enable a formal account of control-flow that still provides for state-level merging. This enables us to define full executions of programs in GAZELLE-based languages, via the function `sem_run` (which we saw previewed in the introductory example, section 2.1.2.2). Additionally, we define propositional notions of multi-step execution that are more useful for formal reasoning.

We give the definition of `sem_run` in figure 2.7.

Recall from our introductory example (section 2.1.2.2) that program execution is defined by applying `sem_run` to the instruction semantics for the combined language being executed. In order to explain `sem_run`, we will begin by describing the syntactic structure used by `sem_run` to represent multi-instruction programs (section 2.3.1), and then provide different versions of semantics defining execution (section 2.3.2), including `sem_run`.

2.3.1 Syntax for GAZELLE's Control-Flow Interpreter

When defining a general syntax representation for GAZELLE, we cannot know *a priori* which languages are to be composed. Therefore we need to develop a syntactic and semantic approach that is agnostic to the specifics of the languages being composed; in other words, *extensible*. As already mentioned (section 1.4), *Trees That Grow* [NJ17] describes one approach to making programming language syntax representations more extensible. The paper essentially describes "decorating" an existing language's syntax tree with additional type-parameters. These parameters

```
type_synonym ('full, 'mstate) control =
  "('full gensyn list md_triv option md_prio *
    String.literal option md_triv option md_prio * 'mstate)"


type_synonym ('syn, 'full, 'mstate) sem =
  "'syn ⇒ ('full, 'mstate) control ⇒ ('full, 'mstate)
control"

type_synonym 'x orerror =
  "('x + String.literal)"

type_synonym ('syn, 'mstate) semc = "('syn, 'syn, 'mstate)
sem"

fun sem_run :: "('syn, 'mstate) semc ⇒ nat ⇒
('syn, 'mstate) control ⇒
   (('syn, 'mstate) control orerror)" where
"sem_run gs 0 m =
  (case cont m of
    Inr msg ⇒ Inr msg
    | Inl [] ⇒ Inl m
    | _ ⇒ Inl m)"
| "sem_run gs (Suc n) m =
   (case cont m of
    Inr msg ⇒ Inr msg
    | Inl [] ⇒ Inl m
    | Inl ((G x l)#tt) ⇒ sem_run gs n (gs x m))"
```

Figure 2.7: Definition of sem_run, which implements multi-step execution for GAZELLE

can be later instantiated with datatypes describing extensions to the original (parameterized) type, enabling the type to be extended without requiring modification.

Building on these insights, we begin by noticing that one need not start with an existing programming language in order to take advantage of the use of type parameters to add new syntax-tree nodes, or to enrich a syntax tree with further information. Instead, we can begin with a datatype capturing trees indexed by a type parameter holding all node-level information. In Isabelle, we can write that type as follows:

```
datatype ('x) gensyn =
  G "'x" "(('x) gensyn) list"
```

`'x gensyn` is a type of trees of nodes, where each node contains a piece of data drawn from an arbitrary data type `'x`. At each node, we can have zero or more subnodes, which are simply other `gensyn`s of type same type (i.e., using the same type parameter `'w`.) Using this general tree, we can express a large subset of inductive, nested datatypes, a subset that intuitively corresponds to useful notions of syntax trees.

For convenience, we define some "syntactic sugar" in the form of notations that make `gensyn` syntax trees look more like syntax trees as we would write them without the GAZELLE framework (we saw this notation used in the introductory example in section 2.1.2.2):

```
notation G ("◊ _ _" [15, 16] )

abbreviation G0 ::
  "'a ⇒ 'a gensyn" ("(† _)" [80] 81) where
"G0 x ≡ G x []"
```

◊ is essentially a synonym for `G`, the constructor for the `'x gensyn` datatype. It constructs a `'x gensyn` node from a label (of type `'x`) and a list of children (of type `'x gensyn list`). Generally, many nodes in a program's syntax tree will be leaf nodes (i.e., their list of children will be the empty list). For this common special case, we have also defined the † notation, which allows us to elide the empty-list argument when constructing such nodes. Together, these two notations help reduce syntactic noise and render GAZELLE syntax trees more readable.

To make this more concrete, let's discuss a small example of using *gensyn* in defining syntax trees for a simple arithmetic language supporting addition, subtraction, multiplication, and integer literals. Without using *gensyn*, we can define it as a typical inductive datatype:

```
datatype arith_manual =
  AmLit int
  | AmPlus arith_manual arith_manual
  | AmMinus arith_manual arith_manual
```

We can write a nested expression for $(4+3)-(2+1)$ as follows:

```
definition example_manual :: arith_manual where
"example_manual =
  AmMinus (AmPlus (AmLit 4) (AmLit 3))
          (AmPlus (AmLit 2) (AmLit 1))"
```

Using *gensyn*, we can express this same language as follows:

```
datatype arith_label =
  ALit int
  | APlus
  | AMinus

type_synonym arith = "arith_label gensyn"
```

That is, we define a type of syntax-node labels (*arith_label*), and then apply the *gensyn* type constructor to it to obtain *arith_label gensyn*, the type of *gensyn* trees labeled with *arith_label* labels. The syntax for $(4+3)-(2+1)$ then looks like the following:

```
definition example :: arith where
"example =
  G AMinus [G APlus [G (ALit 4) [], G (ALit 3) []],
            G APlus [G (ALit 2) [], G (ALit 1) []]]"
```

Using the syntactic sugar for *gensyn*, we can express the same program a bit more readably:

```
definition example_sugared :: arith where
"example_sugared =
  ◊ AMinus [◊ APlus [† ALit 4, † ALit 3]
           , ◊ APlus [† ALit 2, † ALit 1]]"
```

There is a drawback to this approach to syntax. Namely, because `G` nodes are always allowed to take arbitrary lists of child nodes, we have no way of enforcing that particular nodes have particular numbers of children; in our example, for instance, nothing at the type level prevents us from having `ALit` nodes with children, or from having `APlus` or `AMinus` nodes with zero children. So the following syntax, unlikely to be meaningful given the intent of this syntax type, is nonetheless permitted by Isabelle's typesystem:

```
definition bad_arith :: arith where
"bad_arith =
  ◊ (ALit 1) [† APlus, † AMinus]"
```

This is an annoyance, but we are willing to tolerate it. For one thing, it is easy to rule out such badly-formed syntax trees with further checks, defined in isabelle as functions e.g. `example_lang ⇒bool`. When reasoning about such programs, we then just need an additional assumption that the program obeys this well-formedness predicate. Since ISABELLE's typesystem is rather limited in its expressive power compared to theorem-provers such as COQ [Tea22] - ISABELLE does not have a built-in notion of dependent types, nor even of certain weaker constructs such as generalized algebraic datatypes (GADTs) - when trying to express more interesting constraints on the shape of syntax trees, we quickly run up against the limits of what ISABELLE's typesystem can capture. In situations where we need such constraints, we will have no choice but to use well-formedness checks of the sort just described.

2.3.1.1 `gensyn` Type Parameters as Sub-Language Syntax

Setting aside for a moment the question of how exactly we handle control-flow (that is, determining what instruction to execute next after executing an instruction), we can now bridge the gap between instruction semantics of the sort we discussed previously (section 2.2) and the behavior of GAZELLE's virtual machine. Each node in the `'x gensyn` syntax tree has a `'x` datum attached - this datum corresponds to a piece of syntax in an instruction language of the sort we had discussed previously. That is, to execute a step of a GAZELLE program, we need simply

to invoke the function corresponding to the semantics of the instruction language (which is likely to represent a merging of several sub-languages).

After executing a step, we now have a new state, and we are almost ready to continue executing. The only remaining issue is selecting the next instruction. Next (in section 2.3.2), we describe how we handle this issue (i.e., the issue of control-flow in GAZELLE), in the process explaining the definition of the general interpreter `sem_run` given in figure 2.7.

2.3.2 GAZELLE Control-Flow Semantics

To describe control flow in GAZELLE, we take an *operational* approach, describing the execution of GAZELLE programs in terms of steps taken by a very simple virtual machine. This is as opposed to the approach we took when defining the semantics of execution of single instructions (see section 2.2, which had a *denotational* character. Operational semantics has a significant advantage over denotational semantics for our purposes, when it comes to expressing control flow. By taking an operational approach to control flow, we can express the execution of non-terminating programs in a simple and succinct manner. The operational approach also makes it straightforward to derive an executable interpreter for GAZELLE programs. In contrast, with a pure-denotational approach, extra work will generally need to be done to derive a concrete means of executing the program, as the mathematical objects denoting the programs may not be trivial to translate into instructions a computer can execute [Sch88, pg.199]. Since, as mentioned in section 1.3, our goals include producing an executable interpreter, an operational approach to control-flow semantics is a clear win.

We assign an operational semantics to Gazelle programs in two closely-related forms: as an executable interpreter, and as a predicate describing executions. First, we give the interpreter for GAZELLE programs. This interpreter works on the `gensyn` syntax-tree datatype described in section 2.3.1. The insight here is that all that is really needed is a protocol or convention by which the "inner" single-step instruction language can signal to the "outer" interpreter what it

should execute next.

2.3.2.1 Single-Step GAZELLE Interpreter

We begin with some helpful type abbreviations.

```
type_synonym ('full, 'mstate) control =
  "('full gensyn list md_triv option md_prio *
    String.literal option md_triv option md_prio * 'mstate)"


type_synonym ('syn, 'full, 'mstate) sem =
  "'syn ⇒ ('full, 'mstate) control ⇒ ('full, 'mstate) control"

type_synonym 'x orerror =
  "('x + String.literal)"

type_synonym ('syn, 'mstate) semc = "('syn, 'syn, 'mstate) sem"
```

The `control` type synonym corresponds to states that conform to GAZELLE's control-flow protocol. Such states must be a triple of

- A list of syntax trees to execute next (a sort of *defunctionalized continuation* [Dan08])
- A field for signaling errors (a `String.literal`)
- A field for "everything else" (which can, of course, be a further-nested tuple, or some other compound data-structure).

The first two fields of `control` are each wrapped in `md_triv option md_prio`, which enables them to be treated as ordered data for the purposes of merging. This formulation of the `control` type imposes only the minimum amount of structure necessary for the control-flow interpreter to know where to find the data corresponding to the next instruction to execute. Otherwise, the interpreter does not need to know or care about the details of the representation of the remainder of the state (more formally, the interpreter is parametrically polymorphic in the type `'mstate` representing the remainder of the state). Another benefit to this representation is that

it allows us to treat the data corresponding to control-flow as "just another piece of data": from the point of view of GAZELLE's ordering infrastructure, it does not really matter that the *'full gensyn list md_triv option md_prio* happens to represent a continuation. As long as the sub-languages that manipulate control-flow state, and the control-flow interpreter, all agree on where this data should be (i.e., the first component of the triple), no other special treatment is needed.

The *sem* type is the type of instruction-language semantics to be used with the interpreter. *sem* allows for a distinction between the type of syntax stored in the *gensyn* nodes in the *control* argument (the type parameter *'full*) and the type of syntax actually used by the semantics function (the type parameter *'syn*). In practice these may be the same type, in which case we can use the *semc* type synonym. Separating these types allows for greater extensibility. *'syn* allows individual sub-languages to pattern-match on syntactic elements belonging to that sub-language without needing to know about the syntax representation of the entire combined language, which may contain syntactic elements belonging to other sub-languages. On the other hand, *'full* allows the control-flow interpreter access to the entire syntax representation for the language, which it can use to dispatch control to the correct sub-language (while still allowing for further extension of the full language's syntax). In section 2.1.2.2, saw how *translation functions* are used to convert between these two syntax types when composing sub-languages; we will see more of this in section 5.1.

```
definition cont :: "('full, 'mstate) control ⇒ ('full gensyn list
orerror)" where
"cont m ≡
  (case m of
    ((mdp _ (Some (mdt x))), (mdp _ (Some (mdt msg))), _) ⇒
      (case msg of
        None ⇒ Inl x
        | Some msg ⇒ Inr msg)
    | ((mdp _ None), _, _) ⇒
      Inr (STR ''Hit bottom in continuation field'')
    | ((mdp _ _), (mdp _ None), _) ⇒
      Inr (STR ''Hit bottom in message field''))"
```

`cont` projects out the contents of the continuation field of a GAZELLE state (`('full, 'mstate) control`). If the projection succeeds, we return the data (or error message) contained in the state; otherwise, we return one of two special error messages indicating that the projection failed due to absence of data (i.e., presence of $\perp$) in one of the relevant fields.

For convenience, we also define a function `payload`, that gives the remaining fields of a `('full, 'mstate) control` that are not relevant to the GAZELLE control-flow interpreter (i.e., are exclusively the domain of the language-component step semantics):

```
definition payload :: "('full, 'mstate) control ⇒ 'mstate" where
"payload c =
  (case c of
    (_, _, m) ⇒ m)"
```

Next, we define an execution step for GAZELLE:

```
definition sem_step ::
  "('syn, 'mstate) semc ⇒
   ('syn, 'mstate) control ⇒
   ('syn, 'mstate) control orerror" where
"sem_step gs m =
  (case cont m of
    Inr msg ⇒ Inr msg
    | Inl [] ⇒ Inr (STR ''Halted'')
    | Inl ((G x l)#tt) ⇒ Inl (gs x m))"
```

A single step of GAZELLE consists of extracting the control-flow data from the provided state. If that succeeds, we take the label from the first element of the list of continuations and use it as the syntax argument for the instruction-language semantics (if the list is empty, we are done executing). We then return the result of running the instruction-language semantics.

2.3.2.2 Multi-Step GAZELLE Interpreters

We give three definitions of multi-step execution in GAZELLE. The first is an interpreter, convenient for executing programs (and allowing us to make the case that we have truly defined an executable semantics). The latter two are predicates giving a propositional account of GAZELLE

execution. The predicates are more useful for reasoning and establishing proof rules about GAZELLE execution. We prove suitable notions of equivalence relating all four semantics; these proofs give us the flexibility to use whichever is most suitable for a given purpose without worrying about potential discrepancies between these definitions.

2.3.2.2.1 Executable Interpreter The GAZELLE interpreter, *sem_run*, takes a natural- number parameter (*fuel*) bounding the number of steps the interpreter takes before halting. This is a standard technique [PAdAG+21a, ch.14] from the theorem-proving literature, allowing us to express (potentially nonterminating, hence nontotal) interpreters as *total* functions from inputs to outputs. Its definition was given in figure 2.7.

If *sem_run* runs out of fuel before the program being run has halted, it simply returns the current state of the program at the point where the fuel ran out.

2.3.2.2.2 Propositional Semantics We begin by defining a propositional notion of *sem_step*:

```
inductive sem_step_p ::
  "('syn, 'mstate) semc  ⇒ ('syn, 'mstate) control ⇒
  ('syn, 'mstate) control ⇒ bool"
  where
"⋀ gs m x l  tt .
 cont m = Inl ((G x l)#tt) ⟹
 sem_step_p gs m (gs x m)"
```

We show that this is equivalent to *sem_step* for execution steps that do not produce an error:

```
lemma sem_step_p_eq :
  "(sem_step_p gs m m') = (sem_step gs m = Inl m') "
```

With our propositional step construct, we can now describe multi-step execution propositionally. Our first semantics for multi-step execution is the reflexive-transitive closure of the propositional step relation:

```
definition sem_exec_p ::
  "('syn, 'mstate) semc  ⇒ ('syn, 'mstate) control ⇒
  ('syn, 'mstate) control ⇒ bool" where
"sem_exec_p gs ≡
  (rtranclp (sem_step_p gs))"
```

`rtranclp` is the reflexive-transitive closure operator for predicates that comes with ISABELLE's standard library. `sem_exec_p` has the advantage of simplicity (especially in that it directly captures the intuition that multi-step execution is simply chaining together zero or more steps of execution), but it has the disadvantage of not tracking the number of steps taken. This makes it more difficult to relate directly to the interpreter we just defined, as well as creating friction when attempting to formalize the step-indexed Hoare logic we later define in section 4.4. Therefore, we define the following alternate predicate that takes an explicit step count:

```
inductive sem_exec_c_p ::
  "('syn, 'mstate) semc  ⇒
  ('syn, 'mstate) control ⇒ nat ⇒
  ('syn, 'mstate) control ⇒ bool"
  for gs :: "('syn, 'mstate) semc"
  where
Excp_0 :"sem_exec_c_p gs m 0 m"
| Excp_Suc :
  "sem_step_p gs m1 m2 ⟹
   sem_exec_c_p gs m2 n m3 ⟹
   sem_exec_c_p gs m1 (Suc n) m3"
```

We show that `sem_exec_p` and `sem_exec_c_p` are equivalent in the following sense:

```
lemma exec_c_p_imp_exec_p :
  assumes H : "sem_exec_c_p gs m n m'"
  shows "sem_exec_p gs m m'" using H

lemma exec_p_imp_exec_c_p :
  assumes H : "sem_exec_p gs m m'"
  shows "∃ n . sem_exec_c_p gs m n m'" using H
```

That is, if `sem_exec_c_p` says that we can get from `m` to `m'` for any `n`, then `sem_exec_p` will also agree we can get from `m` to `m'`. Conversely, if `sem_exec_p` holds for `m` and `m'`, then we can get from `m` to `m'` using `sem_exec_c_p` in `n` steps, for *some* value of `n`.

Finally, we relate the *sem_run* interpreter to *sem_exec_c_p* (and, hence, to *sem_exec_p* due to the lemmas just described).

```
lemma sem_exec_c_p_run:
  assumes "sem_exec_c_p gs m n m'"
  assumes "cont m' = Inl l"
  shows "sem_run gs n m = Inl m'"
lemma sem_exec_c_p_run' :
  assumes "sem_run gs n m = Inl m'"
  shows "∃ nmin . nmin ≤ n ∧ sem_exec_c_p gs m nmin m'"
```

The first lemma states that if we can get from *m* to *m'* in *n* steps, and the final state *m'* does not correspond to a crash (that is, its continuation field is *Inl* of a continuation rather than *Inr* of an error message), then *sem_run* will also take us from *m* to *m'* in the same number of steps.

Conversely, if *sem_run* executes from *m* to *m'* in *n* steps, the for some *smaller or equal* number *n'* steps, *sem_exec_c_p* can get us from *m* to *m'* in *n'* steps. (We need to relax the statement to talk about *n'* rather than *n*, since *sem_exec_c_p* will stop executing early if the program halts before fuel runs out).

2.4 Summary

This completes our description of the core GAZELLE system for specifying and composing language syntax and semantics. In this chapter we have discussed the following:

- An information-ordering framework for precisely specifying the meaning of merged (single-step, denotational) language semantics (section 2.2)
- A generalized syntax language *gensyn* enabling syntax trees to be constructed using an arbitrary labeling set (section 2.3.1)
- A general, operational system for dealing with control-flow in multi-step GAZELLE programs (section 2.3.2)

In the next chapter (chapter 3), we will discuss the lifting system adapting language developments not aware of the information ordering systems described earlier in this chapter (specifically in section 2.2).

In the subsequent chapter (chapter 4), we will discuss GAZELLE's Hoare logic, which enables reasoning about multi-step programs while enabling reuse of properties proven on sub-languages. We will then (in chapter 5) examine an extended case-study detailing a realistic use of GAZELLE to built an imperative language out of small, self-contained parts via composition. In the process, we will fully flesh out the relationship between merging and control-flow in GAZELLE.

Chapter 3

Lifting: Using Partial Orders with Existing Languages

Previously (in section 2.2), we've discussed our framework for merging state types and semantics functions in GAZELLE. The framework as described thus far, though, does not meet the goals of GAZELLE: namely, to enable reuse of language components, including "off the shelf" reuse of existing language developments with no (or minimal) changes. In particular, in order to be able to use notions the notions we've developed around merging of states, it is necessary to be working with state types that implement one of the `Mergeable` family of typeclasses (see section 2.2.2). Additionally, primitives such as `pcomps` (see section 2.2.4) require the languages being composed to share the same state and syntax representations. In this section, we describe a *lifting* framework that enables us to overcome these restrictions by adapting existing formal developments, with the support of automation to make such adaptation as painless as possible.

3.1 The Lifting Abstraction

3.1.1 Lenses

In order to define a suitable abstraction for adapting existing functions to work with GAZELLE's ordered state types, we draw inspiration here from existing work on *lenses* including [FGM+05] and [PGW17], which capture the notion of a container with contents that can be "put in" (i.e., updated) or taken out. A lens defines two primitive operations over two types, `'a` (the "contents") and `'b` (the "container").

We would like to use a typeclass to express these operations and specifications, but this is not possible in ISABELLE due to the restriction that typeclasses cannot have more than one type parameter. Instead, we use a *locale* [Bal], a construct in ISABELLE that is similar to a typeclass, [1] but does not provide inference. Locales still allow us to package together a set of primitive operations and axioms about them. As an ISABELLE locale, the definition of the lens abstraction looks like the following:

```
locale lens =
 fixes get :: "'b ⇒ 'a"
 fixes put :: "'a ⇒ 'b ⇒ 'b"

locale lens_valid = lens +
 assumes get_put :
  "⋀ (a :: 'a) (b :: 'b) . get (put a b) = a"
 assumes put_get :
  "⋀ (b :: 'b) . put (get b) b = b"
 assumes put_put :
  "⋀ (a1 :: 'a) (a2 :: 'a) (b :: 'b) .
   put a2 (put a1 b) = put a2 b"
```

The first law, `get_put`, states that updating a container with its current contents leaves the container unchanged. The second law, `put_get`, states that, after updating the contents of a container, the result of projecting out the contents will equal what was put in. These two laws define what we call *weak lenses*. The third law `put_put`, holds only for *proper* or *strong* lenses,

[1] indeed, Isabelle's typeclasses are implemented as locales "under the hood"; see [Haf21]

and states that updating the contents of a container multiple times has the same result as applying only the final update.

A lens-like abstraction is useful to us, because we can use it to "wrap" functions defined on data *inside* the container to get new functions over the container type, without modifying the implementation of the function itself. For instance, with this lens abstraction, we might write:

```
definition (in lens)
 lens_lift :: "('syn ⇒ 'a ⇒ 'a) ⇒ ('syn ⇒ 'b ⇒ 'b)" where
"lens_lift f syn st =
 (put (f syn (get st)) st)"
```

That is, we project (`get`) out the contents of the full state type to obtain a sub-state of a type that `f` can operate on, and then `put` that result back into the initial state to get a new, updated state with `f` "mapped" over it (in the sense of mapping a functor). In fact, it should be noted that in this example we could get away with using a functor - that is an abstraction providing only a `fmap` operator equivalent to `lens_lift` - it ends up being useful to have the additional structure provided by a lens-like abstraction, especially when we begin adapting this abstraction to work with the ordered datatypes we defined in section 2.2.2.

3.1.2 A Lens-Inspired Lifter Abstraction

The lens abstraction, as just described, is not quite appropriate for our use in GAZELLE. One problem is that the `get_put` law requires that updating a container with its current contents will leave the container unchanged, but often we do not want this to be the case. For instance, when updating the data in a priority type, we will often want to increment the priority, but this of course means that updating a piece of priority data (`'a md_prio`) with its contents will not be equal to the original value as the priority will have changed. (`put_put` has a similar problem.)

Another issue with `get_put` is that it does not handle the case where the data inside the container is not present - for instance, when trying to project out of a `'a option` whose value is `None`. The choice made in GAZELLE is to return a piece of `bogus` (see section 2.2.2.5) data in

such a case, but that means (ignoring the fact that we have yet to precisely define the `lifter` instance for `'a option`): `get None = bogus`,

`put (get None) None = put bogus None = Some bogus`,

so it is not the case that `put (get x) x = x`.

Nonetheless, we want to define an abstraction that intuitively captures the "spirit" of `lens`, while allowing enough flexibility to permit the additional manipulation needed by GAZELLE's ordering mechanism. We begin by defining an analogue to weak lenses:

```
datatype ('syn, 'a, 'b) lifting =
  LMake  (LUpd : "('syn ⇒ 'a ⇒ 'b ⇒ 'b)")
         (LOut : "('syn ⇒ 'b ⇒ 'a)")
         (LBase : "('syn ⇒ 'b)")
```

A *lifting* in GAZELLE consists of an update function `LUpd` (analogous to `put` above); a projection function `LOut` (analogous to `get` above); and a "base" element `LBase`, used when constructing new elements of the container type `'b`. All of these are parameterized over a syntax type `'syn`, which enables their behavior (hence, the behavior of the lifting) to depend on the specific command (i.e., piece of syntax) with which the lifting is being used. `LMake` is the name of the function used to construct liftings.

`LMap`, which maps a function over a container type `'b` using a lifting, provides an example of how the syntax is passed through the lifting primitives when wrapping existing functions. It implements a notion of mapping similar to a functor's `fmap` [PGW17, pg.33] function. [2] We can give the definition of `LMap` as:

```
definition LMap :: "('syn, 'a, 'b) lifting ⇒
  ('syn ⇒ 'a ⇒ 'a) ⇒
  ('syn ⇒ 'b ⇒ 'b)"
  where
"LMap l f s b =
  LUpd l s (f s (LOut l s b)) b"
definition LNew :: "('syn, 'a, 'b) lifting ⇒
  'syn ⇒ 'a ⇒ 'b"  where
"LNew l s a = LUpd l s a (LBase l s)"
```

[2] However, `LMap` is not guaranteed to obey the functor laws; we will see some "non-functorial" examples of liftings shortly, in section 3.2.4.

Here are the axioms that define a valid (i.e. lawful) lifting:

```
locale lifting_sig =
  fixes l :: "('syn, 'a, 'b :: Pord_Weak) lifting"
  fixes S :: "('syn, 'b) valid_set"

locale lifting_putonly = lifting_sig +
  assumes put_S : "⋀ s a b . LUpd l s a b ∈ S s"

locale lifting_valid_weak =
  lifting_putonly +
  assumes put_get : "⋀ a . LOut l s (LUpd l s a b) = a"
  assumes get_put_weak :
    "⋀ s b . b ∈ S s ⟹
      b <[ LUpd l s (LOut l s b) b"
```

We find it convenient to split the specification of valid liftings into several separate locales that built on each other (similar to what was done for the `Pord` and `Mergeable` typeclasses in section 2.2.2). The most basic specification is the law `put_S`, which states that updating a container using `LUpd` always leads to a valid result. "Valid" here is really a shorthand for "has data that can be validly projected out, when syntax element `s` is used as a parameter for the projection". This is why `S` has type `'syn ⇒'b set` rather than just `'b set`: in general, whether there are valid contents in the container might depend on, for instance, which field is being accessed in a container with multiple fields.

The `put_get` law is essentially the same as the law for lenses we just saw, but `get_put_weak` is rather different. First, the law does not hold for all inputs, but only for ones that are valid in the sense just described (in `S s`, where `s` is the syntax parameter to the update and projection.) Just as important, we don't require that the result of updating a container with its own contents be *equal* to the original container (as in the lens laws above), but rather impose the weaker requirement that it be *informationally at least as large*. This enables us to accommodate, for instance, liftings that increment the priority field in a piece of `'a md_prio` data while updating the contents.

Often - especially in the case of the `md_prio` priority type - we care about a stronger version of the `get_put` law for liftings, which is as follows:

```
locale lifting_valid_ext = lifting_sig +
  assumes get_put : "⋀ s a b . b <[ LUpd l s a b"

locale lifting_valid = lifting_valid_weak + lifting_valid_ext
```

This law is (roughly) analogous to the `put_put` law for lenses; it says that when we update container `b :: 'b` with *any* data, regardless of the original contents of `b`, the result will be informationally greater. We have further extensions versions of the `lifting` abstraction corresponding to liftings that work sanely with the least element of types implementing the typeclass `Pordb`, as well as with the `ok_S` subsets of of types implementing `Okay` (section 2.2.2.5):

```
locale lifting_valid_base_ext = lifting_sig +
  assumes base : "⋀ s . LBase l s = ⊥"
locale lifting_valid_ok_ext =
  lifting_sig +
  assumes ok_S_valid : "⋀ s . ok_S ⊆ S s"
  assumes ok_S_put : "⋀ s a b . b ∈ ok_S ⟹ LUpd l s a b ∈ ok_S"
```

For `Pordb` types, we want `LBase` to coincide with the least element $\bot$. For `Okay` types, we require that, for any syntax `s`, the set of states valid according to the lifting (`S s`) is a superset of the set of states valid according to the set `ok_S` that comes with the `Okay` instance for that type. We have a lifting extension corresponding to liftings that preserve suprema in a sense similar to `sups_pres` (see section 2.2.4). (Note that `‘` here is ISABELLE's syntax for mapping a function over a set).

```
locale lifting_presonly = lifting_sig +
  assumes pres :
    "⋀ v V supr f s .
         v ∈ V ⟹
         V ⊆ S s ⟹
         is_sup V supr ⟹
         supr ∈ S s ⟹
         is_sup (LMap l f s ‘ V) (LMap l f s supr)"
```

Finally, we have another `lifting_valid` extension that is used with liftings involving types having the "pairwise-suprema" property described in section 2.2.4 (that is, implementing

the *Pordps* typeclass). For such liftings, we want to be sure that any supremum of three elements in *S s*, all pairs of which have a supremum in *S s*, also have a supremum in *S s*:

```
locale lifting_valid_pairwise_ext =
  fixes S :: "('syn, 'b :: {Pordc, Pordps}) valid_set"
  assumes pairwise_S :
    "⋀ x1 x2 x3 s s12 s23 s13 s123 .
      x1 ∈ S s ⟹
      x2 ∈ S s ⟹
      x3 ∈ S s ⟹
      is_sup {x1, x2} s12 ⟹
      s12 ∈ S s ⟹
      is_sup {x2, x3} s23 ⟹
      s23 ∈ S s ⟹
      is_sup {x1, x3} s13 ⟹
      s13 ∈ S s ⟹
      is_sup {x1, x2, x3} s123 ⟹
      s123 ∈ S s"
```

All of these extensions to the lifting abstraction are orthogonal in that all can be used separately. ISABELLE's locale system makes it easy to specify concatenations of existing locales. When concatenating locales, we create a new locale that inherits all parameters and assumptions from its parent locale(s). This enables us to easily define locales capturing exactly the properties we desire, while maintaining a separation between the different orthogonal features. For instance, we can define:

```
locale lifting_valid_ok_pres =
  lifting_valid + lifting_valid_ok_ext + lifting_valid_pres_ext
```

There is one exception to the orthogonality of these extensions. The *Okay* extension and *Pordb* extension have a particular interaction; namely, that the least element $\bot$ cannot be a valid state according to the lifting's valid-set (*S s*, regardless of syntax element *s*). This interaction that is captured with by the following locale:

```
locale lifting_valid_base_pres_ext = lifting_valid_pres_ext +
  assumes bot_bad : "⋀ s . ⊥ ∉ S s"
```

Having introduced these notions of lifting, we are now ready to discuss how they are implemented in GAZELLE for different ordered datatypes. These implementations (locale in-

stances) enable application of the machinery for operating on and merging partially-ordered states, described in section 2.2.2, to existing functions not aware of this infrastructure, enabling such functions to be adapted to work with richer or more complex state types than they were originally defined on.

3.2 Lifter Instances

Next, we examine the instances GAZELLE uses to implement the *lifting* abstractions just described.

3.2.1 Identity Lifting

We begin with a lifting that lifts from a type (which must already be equipped with an ordering) into itself.

```
definition id_l ::
  "('x, 'a :: {Pord, Bogus}, 'a) lifting" where
"id_l =
  LMake (λ s a a' . a) (λ s a . a) (λ s . bogus)"

definition id_l_S :: "'x ⇒ 'a md_triv set" where
"id_l_S = (λ _ . UNIV)"
```

This definition, while simple, is worth unpacking because it can serve as a template for understanding the more complex lifting definitions that follow. The first argument to *LMake*, the update function *LUpd*, replaces the entire datum of type *'a* with its data argument *a* (ignoring the syntax argument *s* as well as the argument corresponding to the current data, *a'*). The second argument to *LMake*, the projection function (*LOut*) simply returns the entire datum (again ignoring the syntax argument). Finally, the third parameter to *LMake* (*LBase*), returns a bogus value of type *'a*. Because of the need to return an arbitrary yet computationally sensible bogus value when implementing *LBase*, we require the *Bogus* typeclass constraint on the data type *'a*.

We consider all data to be valid (have a valid projection) under the identity lifting; hence the valid-set `id_l_S x` is simply the full set `UNIV :: 'a md_triv set` (that is, { `x :: 'a . True`}) for any syntax element `x`.

3.2.2 Trivial Lifting

We continue with the trivial lifting, which lifts elements of type `'a` into type `'a md_triv` (that is, a type isomorphic to `'a` but implementing the `Pord` typeclass using a trivial ordering - see section 2.2.3.1).

```
definition triv_l ::
  "('x, 'a :: Bogus, 'a md_triv) lifting" where
"triv_l =
  LMake (λ s a _ . mdt a) (λ s b . (case b of (mdt b') ⇒ b'))
(λ s . mdt bogus)"

definition triv_l_S :: "'x ⇒ 'a md_triv set" where
"triv_l_S = (λ   _ . UNIV)"
```

This instance is straightforward: we use the constructor `LMake` to construct a lifting instance that simply moves data into and out of the `mdt` constructor. The `LBase` (default element) used by `triv_l` returns `bogus` as a default element; for this reason, the data type being with which `triv_l` is being used needs to implement the `Bogus` typeclass. All data wrapped in `md_triv` are considered valid for the purposes of the lifting; hence, `triv_l_S x` is the full set `UNIV` for any syntax element `x`, similar to `id_l`.

`triv_l` fulfills the axioms of `lifting_valid_weak`, as well as all the other locales listed above, with the exception of `lifting_valid` (that is, it fulfills the requirements of the *pairwise* and *pres* locales, as well as the *Okay* locale if the underlying data type `'a` implements `Okay`). We cannot prove the axioms of `lifting_valid` because successive updates may be informationally incompatible. In order to implement these other instances, we will need to use `lifting` instances for GAZELLE's other wrapper types, in order to impose additional structure. We discuss these next.

3.2.3 Option Lifting

Next, we have our implementation for lifting data into an option type. This lifting is parameterized over another lifting: `option_l` transforms a lifting from `'a` to `'b` into a lifting from `'a` to `'b option`. In this sense, `option_l` is really a *lifting transformer* rather than a lifting in and of itself. This pattern of providing transformers that can be used to combine simpler liftings into more complex ones will used throughout the remainder of this section, and plays a key role in automatic inference of liftings (section 3.3).

```
definition option_l ::
  "('x, 'a, 'b) lifting ⇒ ('x, 'a, 'b option) lifting" where
"option_l t =
    LMake (λ s a b .
              (case b of
                Some b' ⇒ Some (LUpd t s a b')
                | None ⇒ Some (LUpd t s a (LBase t s))))
          (λ s b . (case b of Some b' ⇒ LOut t s b'
                            | None ⇒ LOut t s (LBase t s)))
          (λ s . None)"

definition option_l_S ::
  "('s, 'b) valid_set ⇒ ('s, 'b option) valid_set" where
"option_l_S S s = (Some ` S s)"
```

If data is present, `option_l` simply threads the lifting given as its parameter through the `Some` constructor. The more interesting case is when data is not present (i.e., we are working with `None :: 'b option` as a parameter to `LUpd` or `LOut`). In this case, we use the lifting parameter's `LBase` to come up with a default element and get around our lack of data. Of course, the result of this will only be as useful as `LBase` of the wrapped lifting itself. For `option_l`, `LBase` is simply `None`.

Only elements corresponding to uses of the `Some :: 'b option` constructor are considered valid under `option_l`. That is, unlike the previous liftings we discussed, we do not make "get-put" guarantees (see section 3.1.1) when dealing with `None` data. For `Some` data, the data must be valid under the lifting being wrapped; this explains the definition `option_l_S`.

option_l implements all the lifting instances given above, provided the lifting being wrapped in *option_l* also implements those instances. Additionally, it is able to produce a lifting satisfying *lifting_valid_base_ext* even if the lifting it wraps does not. This is as one would expect, given the role of the *option* type in both the lifting and ordering infrastructure of GAZELLE; namely, inducing a base element in an ordered type that lacks one.

3.2.4 Priority Lifting

The priority lifting *prio_l* plays an important role in GAZELLE's lifting system, as it allows us to transform liftings that are not aware of the existence of priorities into liftings that operate on prioritized data. In the process of adding priority behavior to an existing lifting, choices must be made about the specifics of this behavior. For this reason, we see some extra parameters in *prio_l*. Here is the definition:

```
definition prio_l ::
 "('x ⇒ nat) ⇒
  ('x ⇒ nat ⇒ nat) ⇒
  ('x, 'a, 'b) lifting⇒
  ('x, 'a, 'b md_prio) lifting" where
"prio_l f0 f1 t =
      LMake (λ s a b .
              (case b of
                mdp m b' ⇒ mdp (f1 s m) (LUpd t s a b')))
            (λ s p . (case p of
                       mdp m b ⇒ LOut t s b))
            (λ s . mdp (f0 s) (LBase t s))"

definition prio_l_S :: "('x, 'b) valid_set ⇒
  ('x, 'b md_prio) valid_set" where
"prio_l_S S s =
  { p . (case p of
          mdp n x ⇒ x ∈ S s) }"
```

In this definition, *'x* represents the syntax type. Unlike the liftings we saw previously, *prio_l* makes use of syntax in order to be able to modify the data's priority based on syntax. This ends up being useful when composing languages, since often an instruction in one language

corresponds to a "no-op" in another; in these cases, it is convenient to express the "no-op" effect at a low priority so that its result can be overwritten by the other language's result at a higher priority. (We saw an example of something similar in section 2.2.1.3).

`prio_l` takes, in order, the following arguments:

- A function `f0 :: ('x ⇒nat)`, describing how to set the priority when constructing a new piece of prioritized data (using `LBase`)
- A function `f1 :: ('x ⇒nat ⇒nat)`, describing how to update the priority when updating data (using `LUpd`)
- The lifting being adapted to work with a priority, of type `('x, 'a, 'b) lifting`

`prio_l` simply wraps `LBase`, `LUpd`, and `LOut` in an intuitive way - the operations of the provided lifting are used on the data inside the `'b md_prio`, and the provided functions are used to determine the priority of the output (with the ability to determine the priority based on syntax).

The validity of `prio_l` is conditioned on the provided priority-updating functions `f0` and `f1` being well-behaved (in addition to the standard requirement that the lifting being wrapped be valid). In order to satisfy the axioms of `lifting_valid_weak`, we require the following assumption:

```
locale prio_l_valid_weak' =
  fixes l :: "('syn, 'a, 'b) lifting"
  fixes f0 :: "'syn ⇒ nat"
  fixes f1 :: "'syn ⇒ nat ⇒ nat"
  assumes f1_nondecrease : "⋀ s p . p ≤ f1 s p"
```

We require that the priority not decrease when performing updates. Without this, we would not be able to show that `x <[ LUpd s (LOut s x) x` [3]. We also require that the lifting being wrapped by `prio_l` itself satisfy `lifting_valid_weak`. If we assume, instead, that

[3]We could get away with a weaker assumption here, but in practice our usage of the priority type always conforms to this law anyway, and using this version streamlines the proofs involved.

`f1` strictly increases the priority (`⋀s p . p < f1 s p`), we can do away with the assumption of that the wrapped lifting satisfies the *put-get* law (we still require it to fulfill *get-put*, however.)

```
locale prio_l_valid_ext_strong' =
  fixes l :: "('syn, 'a, ('b :: Pord_Weak)) lifting"
  fixes S :: "'syn ⇒ 'b set"
  fixes f0 :: "'syn ⇒ nat"
  fixes f1 :: "'syn ⇒ nat ⇒ nat"
  assumes f1_increase : "⋀ s p . p < f1 s p"
```

The other instances do not require additional assumptions, beyond the assumption that the lifting being wrapped satisfies the stronger properties we are trying to show on `prio_l`. Some of the proofs involved, however, are rather nontrivial; the motivated reader can refer to the formal development (in `Gazelle/Lifter/Instances/Lift_Prio`) for details.

3.2.5 Tuple Liftings

Moving on, we have two liftings corresponding to the components of a tuple type (`fst` and `snd`). We only discuss `fst_l` (lifting into the first component) here, as `snd_l` is symmetrical. First, we give the definition:

```
definition fst_l ::
  "('x, 'a, 'b1 :: Pord_Weak) lifting ⇒
   ('x, 'a, 'b1 * ('b2 :: Pord_Weakb)) lifting" where
"fst_l t =
  LMake (λ s a b . (case b of (b1, b2) ⇒ (LUpd t s a b1, b2)))
        (λ s x . (LOut t s (fst x)))
        (λ s . (LBase t s, ⊥))"

definition fst_l_S :: "('x, 'b1 :: Pord_Weak) valid_set ⇒ ('x, ('b1
* 'b2 :: Pord_Weakb)) valid_set" where
"fst_l_S S s =
  { b . case b of (b1, _) ⇒ (b1 ∈ S s) }"
```

`fst_l` updates the first component of a tuple without touching the second. We require the other (second) component of the tuple to have a least element (be of type `Pord_weakb`), so that we can construct a base element (`LBase`, the third parameter to `LMake` above) that guarantees the second element does not have any data in it. Likewise, if we assume the second component

implements *Pord_weakb*, we are able to implement each instance given in section 3.1.2 as long as the first component implements that instance.

fst_l and *snd_l* do not give us the ability to combine liftings on separate components of tuples into a single lifting - these liftings only allow us to "ignore" the other element in the pair. The next lifting we will explore, *merge_l* enables this, among other useful constructs.

3.2.6 Merge Lifting and Orthogonality

merge_l handles cases where we wish to combine two liftings, both of which target the same datatype, into a single lifting from a pair of elements representing the inner data of the liftings being combined. It is defined as follows:

```
definition merge_l ::
  "('x, 'a1, 'b) lifting ⇒
   ('x, 'a2, 'b) lifting ⇒
   ('x, 'a1 * 'a2, 'b) lifting" where
"merge_l t1 t2 =
    LMake
      (λ s a b .
        (case a of (a1, a2) ⇒
          LUpd t1 s a1 (LUpd t2 s a2 b )))
      (λ s b . (LOut t1 s b, LOut t2 s b))
      (λ s . LBase t1 s)"

definition merge_l_S ::
"('x, 'b :: Pord_Weak) valid_set ⇒
 ('x, 'b :: Pord_Weak) valid_set ⇒
 ('x, 'b) valid_set" where
"merge_l_S S1 S2 s = S1 s ∩ S2 s"
```

In a bit more detail: *LUpd* is defined as first applying one lifting's update, then the other (arbitrarily we choose to apply *t2* first, then *t1*); *LOut* simply pairs the results of *LOut* on the two liftings being merged; and *LBase* is chosen to be the base element of lifting *t1* (we could have just as easily chosen *t2*).

Naturally, it is not reasonable to expect that arbitrary pairs of liftings would be able to be merged in this way. In order to characterize when *merge_l t1 t2* is well defined for some

particular *t1* and *t2*, we define a notion we call *orthogonality*, which captures the intuition that the two liftings' updates commute with each other (and that their base elements are equal).

```
locale l_ortho' =
  fixes l1 :: "('a, 'b1, 'c :: Pord) lifting"
  fixes S1 :: "'a ⇒ 'c set"
  fixes l2 :: "('a, 'b2, 'c :: Pord) lifting"
  fixes S2 :: "'a ⇒ 'c set"

locale l_ortho =
  l_ortho' +

assumes eq_base : "⋀ s . LBase l1 s = LBase l2 s"
  assumes compat : "⋀ s a1 a2 . LUpd l1 s a1 (LUpd l2 s a2 b) =
LUpd l2 s a2 (LUpd l1 s a1 b)"
  assumes put1_get2 : "⋀ s a1 . LOut l2 s (LUpd l1 s a1 b) = LOut
l2 s b"
  assumes put2_get1 : "⋀ s a2 . LOut l1 s (LUpd l2 s a2 b) = LOut
l1 s b"
  assumes put1_S2 : "⋀ s a1 . b ∈ S2 s ⟹ LUpd l1 s a1 b ∈ S2
s"
  assumes put2_S1 : "⋀ s a2 . b ∈ S1 s ⟹ LUpd l2 s a2 b ∈ S1
s"
```

In words, we require the following in order for liftings *l1* and *l2* (with valid-sets *S1* and *S2*) to have a well-defined (i.e., valid as a lifting) composition using *merge_l*:

- The base elements are equal (for all syntax elements)
- The two update operations commute
- Updating using *l1* does not change the contents projected out by *l2* (and vice versa)
- Elements in *S2 s* remain in *S2 s* after applying *l1*'s update, for all syntax elements *s*. (And vice versa).

The intuition behind these restrictions - which are admittedly quite strong - is that they capture the behavior of liftings corresponding to independent elements of a tuple (that is, with no overlapping or shared state). This is our primary use of *l_ortho*, and it is abstract enough

to enable us to express complex liftings (e.g. permutations and reassociations) on tuples by composition. `merge_l` satisfies `lifting_valid_weak` if both liftings (`l1` and `l2`) individually satisfy `lifting_valid_weak`, and `l1` and `l2` are orthogonal. Likewise, under the assumption of orthogonality, `merge_l` of `l1` and `l2` satisfies the assumptions of each of the other lifting-validity locales defined in section 3.1.2, as long as `l1` and `l2` in turn both satisfy that locale's assumptions.

It is reasonable to ask why we don't use the `bsup` operator to define `LUpd` for `merge_l` (that is, replace the first parameter to `LMake` with `(λs a b . (case a of (a1, a2) ⇒[^LUpd t1 s a1, LUpd t2 s a2^])`). The answer is that this version of `merge_l`, while more general (in the sense that it allows a larger number of liftings to be merged), in turn requires a stronger set of properties to hold on the data types and liftings involved in the merge. Indeed, it was unclear whether such a property allowing for this formulation of `merge_l` to work while still being lax enough to apply to the other lifting and data-wrapper constructs used by GAZELLE could be found. Additionally, this stronger version of `merge_l` is not particularly useful, as most of the cases in which we want to apply merge liftings (e.g., composing the first and second components of a tuple) fit nicely into the formulation given here.

We need to instantiate `l_ortho` for the liftings that we hope to merge using `merge_l`. We prove the following instances:

- `option_l l1` and `option_l l2` are orthogonal if `l1` and `l2` are both valid and are orthogonal.
- `fst_l l1` and `fst_l l2` are orthogonal if `l1` and `l2` are both valid and are orthogonal.
- `snd_l l1` and `snd_l l2` are orthogonal if `l1` and `l2` are both valid and are orthogonal.
- `fst_l l1` and `snd_l l2` are orthogonal if `l1` and `l2` both satisfy `lifting_valid_base`

- If *l1*, *l2*, and *l3* are all (pairwise) orthogonal, then *merge_l l1 l2* is orthogonal to *l3* (this can be seen as a kind of associativity law governing *merge_l*).

Because orthogonality as we've defined it is commutative, we also have versions of all of these instances with the order of the arguments flipped (e.g. *snd_l l1* and *fst_l l2* are orthogonal).

3.2.7 Discussion

In this section, we have covered the abstractions used to construct the lifting functions used to adapt program semantics to work with GAZELLE's state types and their built-in orderings. For further details on the locales and instances corresponding to the GAZELLE's lifter subsystem, the interested reader can refer to the `Gazelle/Lifter` directory in the formal development.

As presented here, the system is not very user-friendly. Because we are using locales and do not have access to automated inference, we must instead rely on the user to construct the instance they want to use by hand (or through some other means). One advantage to this lack of inference is that we are no longer restricted to only being able to have a single instance at each type, since we can rely on the user to resolve any ambiguity by giving an instance explicitly). In GAZELLE we remediate this problem by providing a semi-automated approach to inferring *lifting* instances; we describe this automation next.

3.3 Automating Lifter Instance Generation

3.3.1 The Automated Lifting Generator - An Example

To understand the goal of automated lifter instance generation, it will help to look at an example. Suppose we have liftings

l1 :: ('x, 'p1, 'q1) lifting, *l2 :: ('x, 'p2, 'q2) lifting*, and

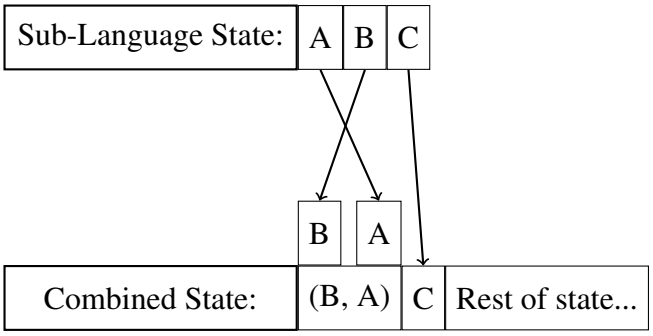

Figure 3.1: Visual depiction of lifting example (arrows represent injection of data)

l3 :: ('x, 'p3, 'q3) lifting. Suppose further that we start out with a language semantics that operates over states *('p1 * ('p2 * 'p3))*, and we want to lift this semantics to a state of type *(('q2 * 'q1) * ('q3 * 'q4))*. That is, we want to transform the *'p1, 'p2, 'p3* components using the liftings *l1, l2, l3*; at the same time, we want to commute the first two components, reassociate the components, and end up with a larger state containing a type *'q4* not present in the input (which will be ignored by the lifting we want to generate). This example may seem a bit contrived, but it is a reasonable representation of the kinds of liftings that will be required by our case study on the IMP language (see chapter 5). Additionally, it showcases the main kinds of structural transformations we need our liftings to perform. A schematic depiction of this lifting is shown in figure 3.1.

We could write this lifting by hand, using the constructs defined in section 3.2. Specifying liftings this way is inconvenient, however; such liftings are difficult both to read and to write. Here is what the manual construction of this lifting looks like: [4].

```
definition my_lift_manual :: "
('x, 'a1 :: Bogus, 'b1 :: Mergeableb) lifting ⇒
('x, 'a2 :: Bogus, 'b2 :: Mergeableb) lifting ⇒
('x, 'a3 :: Bogus, 'b3 :: Mergeableb) lifting ⇒
('x, ('a1 * ('a2 * 'a3)),
  (('b2 * 'b1) * ('b3 * ('b4 :: Mergeableb)))) lifting"  where
"my_lift_manual l1 l2 l3 =
```

[4] In fact, without *merge_l*, this would look far worse, since we would need liftings to explicitly handle the commutation and association of the products involved; relying on the Mergeability of the data frees us from having to work through such low-level details: we can determine each component's lifting individually, then merge them all together.

```
(merge_l (fst_l (snd_l l1))
         (merge_l (fst_l (fst_l l2))
                  (snd_l (fst_l l3))))"
```

Making use of GAZELLE's automation for lifter generation, we can implement this same example in a way that more directly captures the intuition expressed in figure 3.1. Using this automation, we can express the lifting in terms of which named fields in the input datatype map to which named fields in the output datatype. This looks like the following:

```
definition my_lift :: "
('x, 'a1 :: Bogus, 'b1 :: Mergeableb) lifting ⇒
('x, 'a2 :: Bogus, 'b2 :: Mergeableb) lifting ⇒
('x, 'a3 :: Bogus, 'b3 :: Mergeableb) lifting ⇒
('x, ('a1 * ('a2 * 'a3)),
  (('b2 * 'b1) * ('b3 * ('b4 :: Mergeableb)))) lifting"  where
"my_lift l1 l2 l3 =
 schem_lift (SP NA (SP NB NC))
  (SP (SP (SINJ l2 NB) (SINJ l1 NA)) (SP (SINJ l3 NC) NX))"
```

We remark that the `Bogus` and `Mergeableb` annotations are only required because of how general our setting is. If we were working with more concrete liftings or types already satisfying these annotations, they would not be necessary.

We use specialized types to describe the structure of the state being lifted from (the first argument to `schem_lift`) as well as the structure of the state being lifted into (the second argument). We call these structural descriptions *schemas*. The first argument ("left-hand side") schema is essentially an assignment of names to the different fields of the tuple. In this example, we are referring to the first component of the tuple as `NA`, the second as `NB`, and the third as `NC`. We require the user to assign names to all fields of the datatype being lifted *from* using such a descriptor (intuitively, we need to cover all the data in the type being lifted from in order to satisfy the lifting laws given in section 3.1.2). This is not as cumbersome as it might seem: since `('a * 'b * 'c)` is the same as `('a * ('b * 'c))`, we could assign a single name to the `'b * 'c` component if we didn't care about treating its components separately.

The second ("right-hand side") argument to `schem_lift` is more interesting. Here, we specify the structure of the type being lifted into, but we have some additional descriptors that describe how exactly to perform the lifting: in this case, we use `SINJ` - "injection" - to use our existing liftings `l1, l2, l3` with the appropriate components. We also have the ability to specify components left untouched by the lifting; by convention these are referred to using the name `NX`[5].

When we compute `my_lift`, we end up with the same result as when we wrote the lifting by hand; however, using the automated generator leads to code that more directly captures the intuition behind the desired lifting, and thus is easier to read, write, and modify. [6]

3.3.2 Ad-Hoc Polymorphism in ISABELLE

The automated lifting generation used by GAZELLE relies on a novel combination of typeclasses and *ad-hoc polymorphism* that allows us to work around the relative weakness of ISABELLE's typeclass system and (ab)use ISABELLE's built-in typeclass instance inference to construct the lifting we want. By ad-hoc polymorphism, we refer to a specific facility within ISABELLE that enables the declaration of *polymorphic constants*, which are allowed to have different definitions at different types.

For instance, we can define a constant `tyname` that returns a string containing the name of a type. To do so, we use the `'a itself` type, which can be thought of as a type isomorphic to `unit` (i.e., with a single element), that also carries around the type `'a` (at type-checking time, as types in ISABELLE are erased at runtime). We can construct an element of `'a itself` (for any `'a`) with the constructor `TYPE('a)`. Here are some basic instances:

consts `tyname :: "'a itself ⇒ char list"`

[5] In fact, any other name would do; however, it is convenient to have a special name that can never match an element from the left-hand side, and we use `NX` for this purpose

[6] Of course, this is a subjective judgment; in our experience working with constructing an IMP-like language out of smaller components (chapter 5), the ability to work with liftings in this way was invaluable.

```
definition tyn_unit :: "unit itself ⇒ char list" where
"tyn_unit _ = ''UNIT''"

definition tyn_nat :: "nat itself ⇒ char list" where
"tyn_nat _ = ''NAT''"


adhoc_overloading tyname
  tyn_unit
  tyn_nat

value "tyname (TYPE (nat))"
— Result:

''NAT''
```

We now have the desired polymorphic constant (of course, it only supports types we explicitly give overloading instances for; application to other data types would lead to a type error). Things get more interesting, however, if we decide we also want to print out datatypes with type parameters, such as `option` and tuple types. Suppose, say, we want the result of `tyname (TYPE(nat option))` to be the string "NAT OPTION". The problem is that, for such definitions, we need to invoke the constant we are currently defining, in order to print the datatype corresponding to the data inside the option or tuple. Naively, we might try to write:

```
definition tyn_option_bad ::
  "('a option itself ⇒ char list)" where
"tyn_option_bad _ =
  (tyname (TYPE('a))) @ '' OPTION''"

adhoc_overloading tyname
  tyn_unit
  tyn_nat
  tyn_option_bad

— The following produces a type error:
— value [nbe] "tyname (TYPE (unit option))"
```

(`@` here is list concatenation). Everything seems to be working until the last (commented) line, where we try to calculate the value of tyname for the `unit option` type. If we uncomment

this line, we get a type error stating that the instances of *tyname* are ambiguous. The problem is that the occurrence of *tyname* inside the definition of *tyname_option_bad* does not have enough type information to unambiguously choose an instance of *tyname*; due to the way ISABELLE resolves polymorphic constants this ambiguity never goes away, leading to problems when we try to execute *tyname*.

Fortunately, we can solve this problem by using a standard trick (seen for instance in [Sch88, pg.97]) for encoding recursive functions: we add a parameter corresponding to the function calling itself recursively, and use this parameter instead of explicit recursion. Then, we "tie the knot" later, plugging the polymorphic constant in for this new parameter when listing the instances. Concretely:

```
definition tyn_option ::
  "('a itself ⇒ char list) ⇒ ('a option itself ⇒ char list)"
where
"tyn_option t _ =
  (t (TYPE( 'a))) @ '' OPTION''"

adhoc_overloading tyname
  tyn_unit
  tyn_nat
 "tyn_option tyname"

value [nbe] "tyname (TYPE (unit option))"
```

— Result:

```
''UNIT OPTION''
```

```
value [nbe] "tyname (TYPE (unit option option))"
```

— Result:

```
''UNIT OPTION OPTION''
```

By "opening up" the recursive occurrence of the polymorphic constant in this way, we ensure that ISABELLE does not attempt to resolve the recursive occurrence of the constant until there is enough type information to do so. This enables us to get the desired result. This trick works for deeper levels of recursion, as can be seen with the second *value* line in the snippet above.

Continuing with this running example, suppose we want to print a "default" result for types that *don't* have an overloading instance (say, the string "UHOH"). Unfortunately, there is no way to do this without constructing overlapping instances. The naive approach - which leads to resolution errors - might look like this:

```
definition tyn_noname_bad :: "'a itself ⇒ char list" where
"tyn_noname_bad _ = ''UHOH''"

adhoc_overloading tyname
  tyn_unit
  tyn_nat
  tyn_noname_bad
 "tyn_option tyname"

— The following produces a type error:
— value [nbe] "tyname (TYPE (nat))"
```

If, however, we are willing to construct a typeclass capturing all the types that don't have an overloading instance, we can do something almost as good:

```
class noname

instantiation bool :: noname
begin
instance proof qed
end

instantiation char :: noname
begin
instance proof qed
end

definition tyn_noname :: "('a :: noname) itself ⇒ char list" where
"tyn_noname _ = ''UHOH''"

adhoc_overloading tyname
  tyn_unit
  tyn_nat
  tyn_noname
 "tyn_option tyname"

value [nbe] "tyname(TYPE (bool))"
— Result:
```

```
''UHOH''
```

value *[nbe] "tyname(TYPE (bool option))"*
— Result:

```
''UHOH OPTION''
```

While this is admittedly a rather contrived example, it illustrates another trick we make use of in the lifting generator: *typeclass constraints factor into ad-hoc polymorphic constant resolution, and can be used to disambiguate what would otherwise be ambiguous instances*.

3.3.3 Typeclasses for Lifter Inference

Our goal is to enable the definition of *schem_lift*, such that we can infer liftings from schemas of the sort we saw in section 3.3.1. In order to do this, we will define *schem_lift* as a polymorphic constant. We will use a combination of ad-hoc overloading and typeclass constraints similar to what we saw in section 3.3.2 in order to guide ISABELLE's built-in typeclass and instance inference to find the desired result, while avoiding ambiguous or overlapping instances.

To do this, we first need a number of type-definitions corresponding to schema-descriptor elements, as well as several typeclasses used to guide the generation process. It should be noted that these types and typeclasses are relatively trivial in nature; their use, as we will see shortly, is to help constrain the search performed by the typesystem when generating a lifter instance. Because we want to use typeclass inference to implement our lifting-instance generator, we need to make sure all the relevant information is accessible to the type-inference system.

We define the *schem* typeclass, which will be used to capture types corresponding to *schemas*. The typeclass has no data or properties attached to it; we simply use it to label datatypes used in this inference process. We also define a typeclass *basename* (also without data or properties) used to capture types corresponding to names.

class *schem*
class *basename*

Now we choose a fixed, finite set of names that will be used by the system (in GAZELLE currently, these correspond to the letters A through K; as we'll see later, this can be modified easily.) For each letter (using A as an example), we define:

- a datatype `nA`, consisting of a single element `NA` (i.e., isomorphic to unit)
- a typeclass `n_A`, containing only the type `nA`
- a typeclass `hasntA`, containing schema types that *do not* have the name `NA` in them
- an instantiation `nA :: hasntZ` for all *other* names `Z` in the set chosen above.
- an instantiation `nA :: schem`

We also declare a special name, `nX`, in exactly the same way as above (we assume `X` is not in the set of names we chose), with one exception: `nX` does not have a corresponding `hasntX` typeclass, but is still is a member of all `hasntZ` typeclasses for `Z` in the set of names. We use this name as a sort of wildcard capturing parts of terms we don't care about, as we will see shortly.

In code, this looks like the following (assuming our chosen names are `nA` and `nB`):

```
datatype nA =
  NA
datatype nB =
  NB

class n_A
class hasntA

class n_B
class hasntB

instantiation nA :: schem begin
instance proof qed
end

instantiation nB :: schem begin
instance proof qed
end
```

```
instantiation nA :: basename begin
instance proof qed
end

instantiation nB :: basename begin
instance proof qed
end
instantiation nA :: hasntB begin
instance proof qed
end
instantiation nB :: hasntA begin
instance proof qed
end
```

We define several datatypes capturing the structure of our schema:

```
datatype ('a, 'b) sprod =
  SP "'a" "'b"


datatype ('x, 'a) sprio =
  SPR "('x => nat)" "('x => nat => nat)" "'a"

datatype 'a soption =
  SO "'a"

datatype 'a sid =
  SID "'a"

datatype ('x, 'da, 'db, 'a) sinject =
  SINJ "('x, 'da, 'db) lifting" "'a"

datatype ('a, 'b) smerge =
  SM "'a" "'b"
```

`sprod` corresponds to product types (both on the left-hand and right-hand side of the schema-inference constant `schem_lift`); `sprio` corresponds to priority liftings on the right-hand side; `sid` corresponds to usages of the identity lifting (by default, names correspond to usages of the trivial lifting `triv_l`); `sinject` corresponds to injecting existing liftings; and `smerge` corresponds to right-hand side usages of the merge-lifting.

Each of these has a schema instance hand a `hasntZ` instance for each name `Z`. For example, for `sprod` and `soption`, and for name `nA` we have:

```
instantiation soption :: (schem) schem begin
instance proof qed
end

instantiation soption :: (hasntA) hasntA begin
instance proof qed
end

instantiation sprod :: (schem, schem) schem begin
instance proof qed
end

instantiation sprod :: (hasntA, hasntA) hasntA begin
instance proof qed
end
```

The *sprod* type illustrates the reasoning behind why we bother defining all of the *hasntZ* typeclasses. The intuition is that what we'd really like to have, for each name *Z*, is a typeclass *hasZ*, capturing precisely those types which contain the name *Z*. Unfortunately, we cannot define such a typeclass in ISABELLE. To see why, consider the following two instances:

```
instantiation sprod :: (hasZ, _) hasZ begin
instance proof qed end

instantiation sprod :: (_, hasZ) hasZ begin
instance proof qed end
```

ISABELLE will accept one or the other of these instance definitions, but not both, since they overlap. Another way to look at this is that ISABELLE's typeclass inference (at least, the fragment we use here) can be implemented without backtracking, but in order for these instances to be useful, backtracking would be necessary (first attempt to infer a *hasZ* instance for the first tuple component; then, if that fails, attempt to infer for the second one).

We get around this problem by fixing a set of names as defined above. This then enables us to construct *hasntZ* typeclasses for each *Z*, which now means we can turn the *disjunctive* requirement that *one* of the tuple components has a *Z* to prove the product has a *Z* (which requires backtracking) into a *conjunctive* requirement that *all* tuple components *not* have a *Z* in order to prove that the product *doesn't* have a *Z*. In other words, we simply apply de Morgan's Law:

$\neg(A \vee B) = (\neg A) \wedge (\neg B)$. As we will see, having a typeclass capturing the negation is sufficient for our purposes.

3.3.4 Automated Lifting Generation Internals - The `schem_lift` Polymorphic Constant

The typeclasses just described are used to constrain the (type-driven) inference of instances for a polymorphic constant, *schem_lift*. We now discuss the instances for *schem_lift*, and how their inference makes use of these typeclasses to construct our desired liftings. There are five primary cases we need to handle here: bare names on the right-hand side, unary operators on the right-hand side, products on the right-hand side, explicit merges on the right-hand side, and products on the left-hand side. We cover these five cases, then describe how they all fit together.

We define a type alias *('s1, 's2, 'x, 'a, 'b) schem_lift* to make the types of the recursive instances more legible:

```
type_synonym ('s1, 's2, 'x, 'a, 'b) schem_lift =
"('s1 ⇒ 's2 ⇒ ('x, 'a, 'b) lifting)"

consts schem_lift ::
  "('s1 :: schem, 's2 :: schem, 'x, 'a, 'b) schem_lift"
```

The type parameters to the *schem_lift* type are as follows:

- *'s1* is the left-hand side schema (representing the type being lifted from).
- *'s2* is the right-hand side schema (representing the type being lifted into).
- *'x* is the syntax type used in the generated lifting.
- *'a* is the type the generated lifting will lift from.
- *'b* is the type the generated lifting will lift into.

To handle bare names on the right-hand side, we for each name define an instance that emits the trivial lifting. For `NA`, we have:

```
definition schem_lift_base_trivA ::
  "('n :: n_A, 'n, 'x, 'a :: Bogus, 'a md_triv) schem_lift" where
"schem_lift_base_trivA _ _ =
  triv_l"
```

The interesting part here are the type constraints: we require the left- and right- hand schema to be the same type, which must implement `n_A` (i.e., both must be the type NA). Given such a schema, we construct a lifting from `'a` into `'a md_triv`.

Next, to handle unary operators on the right-hand side, we use `option_l` as an example. Its instance looks like the following:

```
definition schem_lift_option_recR ::
  "('n, 'ls, 'x, 'a, 'b2 :: Pord) schem_lift ⇒
   ('n, 'ls soption, 'x, 'a, 'b2 option) schem_lift" where
"schem_lift_option_recR rec n s =
  (case s of
    SO s' ⇒
      option_l (rec n s'))"
```

The `recR` in the name reflects the fact that this instance recurses over the right-hand schema. `'n` is the left-hand side schema, while `'ls` is the schema parameter to `SO` (the option schema). The idea is that if we can infer (recursively) a lifting from type `'a` to type `'b`, with `'n` on the left-hand side, then we can infer a lifting from `'a` to `'b2 option` by emitting `option_l` applied to the inferred lifting. This uses the same recursive-instance trick discussed above in section 3.3.2.

The next important case we deal with is products (tuples) on the right-hand side. This is where our `hasntZ` typeclasses come in handy. For each name (taking `A` as an example), we define two instances corresponding respectively to injecting into the left and right side of the product:

```
definition schem_lift_prod_recR_A_left ::
  "('n, 'ls, 'x, 'a, 'b2 :: Pord) schem_lift ⇒
```

```
    ('n :: n_A, ('ls, 'rs :: hasntA) sprod,
     'x, 'a, 'b2 * ('rest :: Pordb)) schem_lift" where
"schem_lift_prod_recR_A_left rec n s =
  (case s of
    SP ls rs ⇒
      fst_l (rec n ls))"

definition schem_lift_prod_recR_A_right ::
  "('n, 'rs, 'x, 'a, 'b2 :: Pord) schem_lift ⇒
   ('n :: n_A, ('ls :: hasntA, 'rs ) sprod,
    'x, 'a, ('rest :: Pordb) * ('b2)) schem_lift" where
"schem_lift_prod_recR_A_right rec n s =
  (case s of
    SP ls rs ⇒
      snd_l (rec n rs))"
```

Each of these is quite similar to the instance for `option`, except for the type constraints. With these instances, we require that either the first component of the tuple does not have an `A` anywhere in it (with `schem_lift_prod_recR_A_right`); or the second component does not have an `A`. By constructing these instances in this way, we make certain that these instances will not overlap and cause inference errors.

In order for these instances to work, we need to be able to recurse on the left-hand side in such a way that the LHS schema has been reduced down to a single name before we use the `prod_recR` instances on the RHS. This is done using the `schem_lift_recL` instance, which can be seen as the entry point for the entire inference procedure (in the typical case where the LHS is a product representing compound data [7]).

```
definition schem_lift_recL ::
  "('s1l, 's2, 'x, 'a1l, 'b2) schem_lift ⇒
   ('s1r, 's2, 'x, 'a1r, 'b2) schem_lift ⇒
   (('s1l :: schem, 's1r :: schem) sprod,
     's2 :: schem,
     'x,
     (('a1l :: Bogus) * ('a1r :: Bogus)),
     'b2 :: Mergeable) schem_lift" where
"schem_lift_recL recl recr s1 s2 =
```

[7] Tuples are the compound datatype of choice in GAZELLE, but supporting other datatypes with similar structure, such as records, would not be difficult.

```
  (case s1 of
    SP s1l s1r ⇒
      merge_l (recl s1l s2) (recr s1r s2))"
```

In this case, we need to recursively infer two liftings: one corresponding to *recl* (the lifting from the first component *s1l* of the LHS tuple into the entire RHS, *s2*); the other corresponding to *recr* (the lifting from the second component *s1r* of the LHS tuple into *s2*). Since we are using *merge_l* to combine these two liftings, the output type of the liftings *b2* need to be equal.

With this piece in place, we can now see the overall algorithm (run by ISABELLE's inference engine) for generating liftings: we recurse along the left-hand schema until we find a single name; we then try to find the sole occurrence of that name on the right-hand schema; we then use the path traversed to reach that name in order to construct a lifting from that name into the entire right-hand side. We do this for each name on the left-hand side, then use *merge_l* to combine them all together.

One final case of note is the handling of explicit merges in the RHS schema (as opposed to merges implicitly generated by the *schem_lift_recL* instance to handle products on the left-hand side). As with *schem_lift_prod_recR*, we need separate instances for each name. These instances look like the following:

```
definition schem_lift_merge_recR_A_left ::
  "('n, 'ls, 'x, 'a, 'b2) schem_lift ⇒
   ('n :: n_A, ('ls, 'rs :: hasntA) smerge,
    'x, 'a, 'b2) schem_lift" where
"schem_lift_merge_recR_A_left rec n s =
  (case s of
    SM ls rs ⇒
      (rec n ls))"

definition schem_lift_merge_recR_A_right ::
  "('n, 'rs, 'x, 'a, 'b2) schem_lift ⇒
   ('n :: n_A, ('ls :: hasntA, 'rs) smerge,
    'x, 'a, 'b2) schem_lift" where
"schem_lift_merge_recR_A_right rec n s =
  (case s of
```

```
  SM ls rs ⇒
    (rec n rs))"
```

It should be noted that *merge_l* occurs nowhere in these liftings; all the actual merging takes place through the recursion on the left-hand side. In fact, these instances are nearly identical to the instances for *schem_lift_prod_recR*, the only difference being that *fst_l* and *snd_l* are omitted (as well as removing the requirement that the "other" component not being iterated into have a least element). In other words, we handle arbitrarily nested products as simply a special case of merging!

In the end, we declare all of these definitions as *schem_lift* instances. These are only a subset of the instances actually declared - in GAZELLE we have a few (less useful) instances not discussed here, as well as more names than just *A* and *B*. The interested reader can consult the formal development
(in the directory `Gazelle/Lifter/Velocity`).

adhoc_overloading *schem_lift*

```
"schem_lift_base_trivA"
"schem_lift_base_trivB"

"schem_lift_prod_recR_A_left schem_lift"
"schem_lift_prod_recR_A_right schem_lift"
"schem_lift_prod_recR_B_left schem_lift"
"schem_lift_prod_recR_B_right schem_lift"

"schem_lift_option_recR schem_lift"
"schem_lift_merge_recR_A_left schem_lift"
"schem_lift_merge_recR_A_right schem_lift"
"schem_lift_merge_recR_B_left schem_lift"
"schem_lift_merge_recR_B_right schem_lift"


"schem_lift_recL schem_lift schem_lift"
```

3.3.5 Priority Functions in Liftings

When defining liftings into datatypes involving the `md_prio` wrapper, we often make use of *priority functions* that describe how the priority field of the `md_prio` is to be updated based on which instruction is being executed. As described in section 3.2.4, these functions take a syntax element and current priority, and return a new priority. When composing liftings using the lifter generator, this is usually done through the `SPRC` schema element. `SPRC` ("Schema PRiority Case") takes such a priority function and a schema corresponding to a lifting to be used on the contents of the `md_prio` and creates a lifting that updates contents according to that lifting and the priority according to the priority function. Another commonly-used schema primitive for constructing priorities is `SPRI` ("Schema PRiority Increment"), which increments the priority when updating data, ignoring the syntactic input.

3.3.6 Automated Lifting Generation - Scalability

A reasonable concern about the system just described is the number of extra definitions and instances required to make it work. In particular, we require a linear number of types and typeclasses; a linear number of instances of the polymorphic constant `schem_lift`; and a quadratic number of typeclass instances (since for each name `Z`, `hasntZ` needs a number of instances equal to the number of total names minus one).

In the current version of GAZELLE, names A-K, plus X (a "dummy" name not matching anything) are used. With this number of names, the inference-time performance of the system is not noticeable (i.e., typeclass and polymorphic-constant resolution do not add an overhead readily observable to the user). Investigating the precise limits of this approach in terms of number of names is future work, but we are reasonably confident that the palette of names can be expanded much further, certainly enough to enable the expression of liftings corresponding to complex, deeply nested program states.

These is also the matter of generating the ISABELLE code that implements this system. Since the fixed set of names are "baked-in" to the particular typeclasses and instances used by the system, changing this set of names requires rewriting a quadratic amount of code. ISABELLE lacks its own macro-system, so we make use of an external templating tool, VELOCITY [Dev06] to handle automating the generation of these typeclass, instance, and constant definitions. By doing so, we enable the user to modify the lifter-generation automation (including, most importantly, changing the set of names) without needing to touch a large number of lines of code. One benefit of VELOCITY is that it has an integration with ISABELLE's JEDIT-based editor.

For more details on the full implementation of this inference system, as well as the code generation, the interested reader can refer to the formal development
(in the directory `Gazelle/Lifter/Velocity`).

3.3.7 Proof Automation for Lifting Correctness

The system just described in 3.3 handles the generation of lifter instances, but provides no guarantee of their validity (i.e., that they meet the lifter laws discussed in 3.1). Fortunately, ISABELLE's built-in proof automation makes it relatively painless to construct these proofs without manual effort.

First, we define a system very similar to the one just described for inferring valid-sets for liftings [8]. We don't describe it in detail here, but this valid-set inference uses the same structures and patterns to infer the valid-sets corresponding to the lifting given by the same schema. Once we have the valid-sets, we still need to show correctness of the lifting we inferred with respect to the valid-set we inferred.

To do this, we observe that the structure of the proofs involved is rather straightforward: primarily, we need to apply introduction rules corresponding to implementing the assumptions of

[8] We refer to these as "valid-sets" as a shorthand, but recall that they are actually functions mapping syntax elements to their corresponding set of valid states; see the description above in section 3.1.

the locales corresponding to the lifting instances we inferred. For example, consider the locale *option_l_valid_weak* (defined above in section 3.2.3). After we define this locale, ISABELLE automatically generates several proof rules for working with *lifting_valid_weak* instances; the one relevant here is the following:

— option_l_valid_weak.intro:

```
lifting_valid_weak ?l ?S ⟹ option_l_valid_weak ?l ?S
```

Considered in terms of "backwards" proof search (i.e., applying introduction rules starting from the desired conclusion), this enables us to turn a proof search for *option_valid_weak l S* into a search for *lifting_valid_weak l S*. *lifting_valid_weak*, in turn, has its own *lifting_valid_weak.intro* theorem. So a large part of the proof search we need consists of repeatedly applying such rules.

To handle the remainder of the proof search, we define [9] what looks a bit like an elimination rule for *option_l_valid_weak*. Because of the particular structure of the proofs we are looking for, we can treat them as introduction rules in exactly the same way as the *intro* rules.

```
lemma (in option_l_valid_weak) ax :
  shows "lifting_valid_weak (option_l l) (option_l_S S)"
lemma (in option_l_valid_weak) ax_g :
  assumes "S' = option_l_S S"
  shows "lifting_valid_weak (option_l l) S'"
```

Both these rules express the fact that, to prove a lifting wrapped in *option_l* is valid, it suffices to prove (i.e., instantiate) *option_valid_weak* for that lifting. The difference between the two rules is that *ax* requires the valid-set in the goal and the valid-set in the instance proof for *option_l_valid_weak* to match syntactically, while *ax_g* generates a new variable *S'* to capture the valid-set. This allows more proofs to be found, but can result in a much slower proof search since it will cause ISABELLE's automation to apply set-based reasoning to try to show equalities between the relevant valid-sets.

[9] The following rules are not automatically generated by ISABELLE, unlike the *intro* rule.

ISABELLE provides built-in automation that is quite good at dealing with the kinds of proofs we need to construct when proving the validity of automatically-inferred liftings. In particular, it provides a family of tactics (i.e., proof-generating programs) for solving problems involving application of introduction and elimination rules as well as simplification by equality facts [Wen, pg.238]. Of these tactics, `fastforce` can solve our goals quickly and reliably, but has the drawback that it is "all-or-nothing"; if it fails, the user does not get helpful error messages or proof state to enable debugging of what went wrong.

Fortunately, because we hook into this built-in automation by just providing lists of theorems to the tactics, we can debug what is going on by applying less-aggressive tactics that produce an intermediate proof state if they don't fully solve the goal. The `auto` tactic is quite useful for this purpose, although it tends to be much less aggressive about applying introduction rules than `fastforce` (in cases where it cannot complete the proof, `auto` tends to return a proof state with fewer introduction rules applied than one might expect).

Nonetheless, this approach (applying `auto` to debug broken theorem-lists) was more than sufficient for debugging the generation of proofs for the instances inferred by the our automated lifter-generator (indeed, even for hand-constructed ones that use the same primitives `triv_l`, `option_l`, etc.) Such debugging should only be necessary when adding new types of liftings to the lifting generator (which in turn will require adding new theorems to the list for `fastforce` to apply), or when attempting to prove validity of an invalid lifting (an uncommon case, as the lifter-inference system is designed so that instances that type-check are in fact valid).

Proving the validity of a lifting constructed using our inference system then ends up looking like the following:

```
lemma lifting_valid_example :
  "lifting_valid
  (schem_lift (SP NC (SP NB NA))
    (SP (SPRI (SO NA))
        (SP (SPRC (λ _ . 1) (SO NB))
            (SPRI (SO NC)))))
  (schem_lift_S (SP NC (SP NB NA))
```

```
    (SP (SPRI (SO NA))
        (SP (SPRC (λ _ . 1) (SO NB))
        (SPRI (SO NC)))))  "
unfolding schem_lift_defs schem_lift_S_defs
by(fastforce intro: lifting_valid_standard
    lifting_ortho_standard )
```

That is, the proof of correctness boils down to:

- Unfold the instance definitions for *schem_lift* (and *schem_lift_S*, the equivalent for valid-sets). *schem_lift_defs* and *schem_lift_S_defs* are theorem-sets that simply collect all these instance definitions.
- Perform the standard *fastforce* automation, with all relevant lifting introduction rules (as described above) added to its standard rule-set[10]. We add orthogonality rules in a similar manner.

We define a few different rule sets; the primary difference is whether the *ax* or *ax_g* versions of the "elimination" rules for locales are included. We have found that *lifting_ortho_standard*, which uses the *ax_g* formulations of the rules for all liftings except *merge_l*, to be a good compromise between generality and speed. For more details, the reader can consult the formal development (`Gazelle/Lifter/Auto_Lifter_Proofs.thy`).

3.4 Discussion

In this chapter, we have discussed the lifting infrastructure of GAZELLE. Liftings in GAZELLE enable us to reuse existing denotational semantics (given as functions) that may not be aware of the partial-ordering typeclasses used by GAZELLE to define merging of program states. Additionally, we have described an automated approach to generating lifting instances as well

[10]The standard rules correspond, e.g., to introduction and elimination rules from classical logic; common arithmetic simplifications; and similar theorems of general-purpose use.

as correctness proofs for these instances, which helps to greatly ameliorate the lack of built-in inference for liftings (which are too complex to capture using ISABELLE's typeclasses).

Together with chapter 2, this chapter completes our tour of the components that enable GAZELLE to construct programming languages out of separately-defined components. In the following chapter (chapter 4), we will describe GAZELLE's facilities enabling formal reasoning about programming languages defined in this way. Having done so, we will have achieved the last of GAZELLE's remaining design goals. Then, in (chapter 5), we will show how all of these components fit together by demonstrating the usage of GAZELLE to formalize an imperative programming language, IMP.

Chapter 4

Reasoning About GAZELLE

In this chapter, we talk about how we enable support for reasoning on languages defined in GAZELLE (as described in chapters 2 and 3). Just as GAZELLE enables language components defined separately to be composed in a well-defined way, the goal of the reasoning infrastructure for GAZELLE is to enable *reasoning principles* for separately-defined languages to be used conveniently on the composition of those languages.

To support reasoning about multiple execution steps in the interpreter described in section 2.3.2, we define a *Hoare logic* for GAZELLE. Hoare logic ([PAdAG+21a, ch.2]) is a commonly-used framework for reasoning about imperative programs. In Hoare logic, rules are defined corresponding to statements in the language, which can then be composed in a structure mirroring that of the original program. As with the multi-step interpreter semantics of GAZELLE, we give two versions of Hoare logic, one of which carries with it an explicit step-count. The step-count version of Hoare logic is useful for proving rules about possibly nonterminating constructs, such as the `for` loop in the IMP language we will define later (see sections 5.1.5 and 5.3). The version without the step count is more standard; by relating it to the step-count definition, we can build confidence that the logic with step-counts is correct.

It should be noted that present a somewhat atypical formalization of Hoare logic, inspired

by the developments given in [ADH+14, ch.4] and [AB07]. In particular, we make use of their notion of "CPS-flavored" (continuation-passing-style) Hoare logic. [ADH+14, pg.32] asserts that such a formulation of Hoare logic is useful for reasoning about more complex control (such as `break` and `continue` statements in `C`-style `for`-loops in [AB07]). We have found this approach to Hoare logic to be a good match for GAZELLE's general approach to control flow.

4.1 Hoare Logic: A Primer

In Hoare Logic, we prove statements about programs in the form of *Hoare Triples*, usually written as {`P`} `Prog` {`Q`}. In such a triple, `Prog` is a program, `P` is a precondition on the program state (i.e., a predicate `state ⇒bool`, for program state type `state`), and `Q` is another state predicate representing a postcondition. The meaning of {`P`} `Prog` {`Q`} is the following: if `P` holds before we execute `Prog`, and `Prog` terminates, then `Q` holds after the execution finishes.

We call this a *partial correctness* Hoare Logic, which means that the postcondition is only guaranteed to hold *if* the program terminates, but we make no guarantee about termination. An alternative approach is *total correctness* Hoare Logic, in which the meaning of {`P`} `Prog` {`Q`} is instead that if `P` holds before executing `Prog`, then `Prog` terminates and `Q` holds afterwards. In practice, total correctness Hoare logic requires an intermixing of termination conditions (i.e., under what circumstances can we prove *Prog* terminates) with reasoning about pre- and postconditions, which can be inconvenient. For this reason, we choose the partial-correctness approach here, and consider termination proofs to be beyond the scope of this work.

The rules of Hoare logic generally correspond to statements in the language being reasoned about, describing the behavior of such statements in terms of their pre- and postconditions, often using other Hoare triples as assumptions. For instance, sequencing in an imperative language might look like the following:

```
lemma HSeq :
 assumes Hc1 : "{P1} c1 {P2}"
```

```
assumes Hc2 : "{P2} c2 {P3}"
shows "{P1} c1 ; c2 {P3}"
```

In words, this rule says that

- if we know that `P2` holds after executing `c1` as long as `P1` holds before executing `c1`
- and we know that `P3` holds after executing `c2` as long as `P2` holds before executing `c2`
- *then* we know that, if `P1` holds before we execute the sequenced operations `c1; c2` (that is, "run `c1`, then run `c2`"), `P3` holds after we are done

In addition to rules corresponding to statements in the language they are designed to reason about, Hoare logics have an additional rule that does not correspond to a program statement - this is the *rule of consequence*, stating that preconditions can be strengthened and postconditions weakened without affecting the validity of a triple:

```
lemma HConseq :
 assumes HP : "⋀ st . P' st ⟹ P st"
 assumes HQ : "⋀ st . Q st ⟹ Q' st"
 assumes H : "{P} c {Q}"
 shows "{P'} c {Q'}"
```

In our context, it is important that we formally tie our Hoare logic to our programming-language semantics, so that we can trust that our Hoare proofs are sound with respect to actual program executions. In other words, we need to supply a *definition* of the Hoare triple in terms of GAZELLE's semantics. One standard approach would be to adopt something like the following. Suppose for simplicity we have a type of programs (syntax) `prog` and a semantics `sem :: prog ⇒state ⇒state ⇒bool`, in which a nonterminating program is represented by the absence of an element in the relation (i.e., if program `p` does not terminate starting in state `s`, then there is no `s'` such that `sem p s s'` holds).

```
definition triple :: "(state ⇒ bool) ⇒ prog ⇒
  (state ⇒ bool) ⇒ bool"
  ("{_} _ {_}")
```

```
  where
"triple P c Q =
  (∀ (st :: state) (st' :: state) .
     P st ⟶
     sem c st st' ⟶
     Q st')"
```

This is a straightforward expression of the intuition just given: if `c` terminates under a state satisfying `P`, then `Q` holds afterwards. The disadvantage to this approach is that it makes it difficult to deal with "non-local" control flow (e.g., code that breaks out of loops): under this definition of Hoare triple, we are essentially committing to eventually ending up in a state satisfying `Q`. However, a non-local control-flow construct might, for instance, cause us to skip over the state where `Q` holds. Since Hoare rules mirror the structure of programs, this is a problem. Suppose `prog` is a piece of a larger program we are trying to reason about. If `prog` contains within it a statement that can *escape* `prog` (e.g., a `goto` statement), we don't have enough information, examining only `prog`, to say exactly where we will end up [1].

4.2 Hoare Logic for Single-Step Semantics

While the presentation of Hoare logic just given is not the solution we adopt for reasoning about multi-step programs with control flow in GAZELLE, it does end up being useful for stating properties about the single-step denotational semantics of language components. We give a definition of Hoare triples for such semantics [2]:

```
definition HT ::
   "('a ⇒ bool) ⇒ ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ bool"
   ("{{_}} _ {{_}}" [0,0,0] 61) where
"HT pre x post =
   (∀ a b . pre a ⟶ x a b ⟶ post b)"
```

[1] The ability to handle non-local control flow enabled a Hoare-style approach to reasoning about `break` and `continue` statements in [AB07].

[2] the numbers preceding the ***where*** statements are priorities given to ISABELLE's parser and are not relevant to this discussion.

```
definition HTS ::
  "('x ⇒ 'a ⇒ 'a) ⇒
   ('a ⇒ bool) ⇒ 'x ⇒ ('a ⇒ bool) ⇒ bool"
  ("_ % {{_}} _ {{_}}" [250,252,254,256])
  where
"HTS sem pre x post =
  HT pre (λ a b . sem x a = b) post"
```

`HT` captures a Hoare triple-style abstraction for *predicates* (`'a ⇒'a ⇒bool`) representing a language semantics; `HTS`, which is more useful, builds on `HT` to express Hoare triples for executable single-step semantics taking a syntax argument (i.e., functions `'x ⇒'a ⇒'a`, where `'x` is the type of syntax elements). Note that one major difference from conventional presentations of Hoare logic is that GAZELLE's Hoare triples take a semantics function as a parameter (i.e., are technically quadruples[3]). Most Hoare logics are designed to work with only one language, so the semantics is "hard-wired" rather than passed as a parameter. Because GAZELLE is designed for extensibility, we will usually need to be explicit about which semantics we are writing rules for; hence, the need for an additional parameter.

The primary use-case for these definitions is when lifting facts about single-step language components, expressed as Hoare triples using `HTS`, into facts about multi-step languages built out of these components. We will see this in detail when discussing the GAZELLE implementation of the IMP language (in chapter 5).

We define a some rules for characterizing the behavior of these Hoare triples under merging and lifting (see section 2.2 and chapter 3). We first define `lift_pred_noS_s`, which enables us to lift predicates using GAZELLE's `lifting` abstraction:

```
definition lift_pred_noS_s ::
  "('a1, 'b1) syn_lifting ⇒
   ('a1, 'a2, 'b2 :: Pord) lifting ⇒
   'b1 ⇒
   ('a2 ⇒ bool) ⇒
   ('b2 ⇒ bool)"
  where
```

[3]We still refer to these constructs as "Hoare Triples", in order to emphasize the similarity with the typical notion developed in the literature, for instance [PAdAG+21a, ch.2].

```
"lift_pred_noS_s l' l syn P st =
 P (LOut l (l' syn) st)"
```

`lift_pred_noS_s` (so named because it does not consider membership in the lifting `l`'s corresponding valid-set - hence "`noS`" - but does take a syntax-transforming function - hence "`_s`"). This is a rather naive notion of predicate-lifting: to lift a predicate `P` using a lifting `l`, we only check that `P` holds on the projection (`LOut l`) applied to the data in question.

With this definition, we can prove the rule `Vlift` for our single-step Hoare logic. This rule allows us to translate from statements about a step in a sub-language to statements about a step in the lifted version of the sub-language:

```
lemma Vlift :
  assumes Valid : "lifting_valid_weak l S"
  assumes V: "(sem) % {{P}} x {{Q}}"
  assumes Syn : "l' x' = x"
  shows "(lift_map_s l' l sem) %
           {{lift_pred_noS_s l' l x' P}}
           x'
           {{lift_pred_noS_s l' l x' Q}}"
```

Such lifting is usually done in the service of constructing a merged language (i.e., lifting from a sub-language state type into a combined-language state type). When merging languages, we can make use of the following rule:

```
lemma Vmerge :
  assumes Pres : "sups_pres (set l) S"
  assumes Sem : "f ∈ set l"
  assumes P_S : "⋀ st . P st ⟹ st ∈ S x"
  assumes V : "(f) % {{P}} x {{Q}}"
  shows "(pcomps l) %
          {{P}}
          x
          {{(λ st . ∃ st_sub . Q st_sub ∧ st_sub <[ st)}}"
```

That is, suppose we have a list of compatible language-component semantics `l`, and merge them together with `pcomps`, and that for some particular language in `l`, `f`, we can show the Hoare triple `f % {{P}} x {{Q}}` (for command `x`). Then, as long as our precondition `P` is

strong enough to entail that the starting state is valid, if we run the combined language starting in a state satisfying `P`, then `Q` holds on some state informationally smaller than the final state. Specifically, `Q` holds on `f x` applied to the initial state; but the influence of the other languages in `l` may lead to a greater result, on which `Q` may not hold.

Another way to look at this rule is that if the predicate `Q` is monotone (in the sense that `x <[ x' ⇒Q x ⇒Q x'`), it will hold in the final (merged) state. We can capture this with the following, alternative rule for merging:

```
lemma Vmerge_mono :
  assumes Pres : "sups_pres (set l) S"
  assumes Sem : "f ∈ set l"
  assumes Mono : "Pord.is_monop1 Q"
  assumes P_S : "⋀ st . P st ⟹ st ∈ S x"
  assumes V : "(f) % {{P}} x {{Q}}"
  shows "(pcomps l) %
         {{P}}
         x
         {{Q}}"
```

Because of the restrictions on output predicates imposed by these rules, they do not lead us to a general approach to reasoning about merged semantics (we will see in our discussion of the implementation of `Imp` how we handle cases involving merges with less well-behaved predicates). Nonetheless, they can be useful for reasoning about certain cases of merged languages. When they apply, they can give a very straightforward account of the behavior of the result of the merged languages.

4.3 CPS-Flavored Hoare Logic

In order to better support complex control-flow constructs, we adopt a different and slightly counterintuitive definition of Hoare triples [4]. First, we define *immediate safety* as:

```
definition imm_safe :: "('syn, 'mstate) semc ⇒
```

[4]These definitions are heavily based on those given in [ADH+14, ch.4] for reasoning about multi-step executions.

```
    ('syn, 'mstate) control ⇒
    bool" where
"imm_safe gs m ≡
  ((cont m = Inl []) ∨
    (∃ m' . sem_step_p gs m m'))"
```

That is, an immediately-safe state is a state that is either halted or can take a step under our semantics. We then define *safety* in terms of immedate safety:

```
definition safe :: "('syn, 'mstate) semc ⇒
    ('syn, 'mstate) control ⇒
    bool" where
"safe gs m ≡
    (∀ m' . sem_exec_p gs m m' ⟶ imm_safe gs m')"
```

So a *safe* state is any state in which execution always leads to an immediately safe state, regardless of how many execution steps are taken. (Note that nonterminating programs are safe under this definition). Next, we define *guardedness*, which provides a notion of "safety under a precondition `P`":

```
definition guarded :: "('syn, ('mstate :: Okay)) semc ⇒
    ('mstate ⇒ bool) ⇒
    'syn gensyn list ⇒ bool"
("|_| {_} _" [200, 202, 204])
  where
"guarded gs P c =
    (∀ m . m ∈ ok_S ⟶ P (payload m) ⟶
      cont m = Inl c ⟶ safe gs m)"
```

Note the use of `payload` in the definition above, defined in section 2.3.2.1. `payload` extracts the non-control-flow data from a GAZELLE state. We need to use this here, since our goal is to have the predicates used in Hoare logic be independent of control flow; control flow is to be handled by the Hoare logic itself, through the application and composition of rules. We can write "when using `gs` as our single-step semantics, program `c` is guarded under predicate `P`" as `|gs| {-P-} c`.

A few details here bear some discussion. First, we add the precondition that the state `m` (over which `guarded` quantifies) be in `ok_S` (that is, have valid data inside of it; see section

2.2.2.5); this is because, in our definition of Hoare triples, we will want this as an additional assumption (so that, in turn, we can use the fact that the projections `LOut` will return meaningful data when proving Hoare rules). Next, note that the predicate `P` is a predicate over `'mstate` rather than over the entire interpreter state `('syn, 'mstate) control`. This is to enable a separation of concerns between control-flow and Hoare predicates; if our Hoare predicates were allowed to directly reference the continuation representing subsequent instructions to be executed, it would be very difficult to write useful Hoare rules. When we discuss IMP we will see a lifting `no_control_l` (in section 5.2.3), which bridges the gap between the full state with control information and the state over which Hoare predicates operate.

Finally, notice that instead of `c` being a *single* command, we instead have a *list* of commands (`syn gensyn list`) representing the next statements to be executed. When running a program `P`, we will start with a singleton list `Inl [P]` in the `cont` field, but the list will likely grow as the program executes.

We can now define Hoare triples in terms of `guarded`:

```
definition HT :: "('syn, ('mstate :: Okay)) semc ⇒
  ('mstate ⇒ bool) ⇒
  'syn gensyn list ⇒
  ('mstate ⇒ bool) ⇒ bool"
  ("|_| {-_-} _ {-_-}" [206, 208, 210])
  where
"HT gs P c Q =
  (∀ c' .  |gs| {Q} (c') ⟶ |gs| {P} (c @ c'))"
```

Essentially, `|gs| {-P-} c {-Q-}` means that (using single-step semantics `gs`), if some suffix continuation `c'` is safe under `Q`, then prepending `c` to that suffix is safe under `P`. Notice how this frees us from having to "commit" to the program actually ending up in a state satisfying `Q`. If, for instance, we were to discard `c'` somewhere in the course of executing `c`, we might never reach a state where `Q` holds. As argued in [ADH+14, ch.4], being able to reason about safety of programs is sufficient to allow us to reason about correctness of programs, provided we are working in a language that has the ability to check the correctness condition at runtime and crash

the program if the condition is not met (thus turning a correctness property into a safety property).

Using this definition we can prove a version of the rule of consequence. We can also prove the following general sequencing-like rule, which will be useful for implementing sequencing and other constructs later (see chapter 5):

```
lemma HCat :
  assumes H : "|gs| {- P1 -} c1 {- P2 -}"
  assumes H' : "|gs| {- P2 -} c2 {- P3 -}"
  shows "|gs| {- P1 -} (c1 @ c2) {- P3 -}"
```

This rule allows us to append a suffix continuation `c2` to an existing continuation `c1`; if its precondition matches `c1`'s postcondition, then the postcondition of the concatenation is the postcondition of `c2`. This is quite similar to a standard Hoare sequencing rule, but differs from it in that there is no explicit mention of a program (syntax) corresponding to sequencing: this is a sequencing rule that operates purely at the level of the program state representation. (We do this to keep the rule general by not making unnecessary assumptions about the structure of the language encoded in `gs`).

4.4 CPS-Flavored Hoare Logic with Step-Counts

When reasoning about loop constructs, we find it useful to make a further modification to the Hoare logic formalism described above. We replace the notion of *safety* defined above with a notion of *safety for* `n` *steps*:

```
definition safe_for :: "('syn, 'mstate) semc ⇒
  ('syn, 'mstate) control ⇒
  nat ⇒ bool" where
"safe_for gs full n =
  ((∃ n0 full' . n0 ≤ n ∧ sem_exec_c_p gs full n0 full' ∧
     cont full' = Inl []) ∨
   (∀ n0 . n0 ≤ n ⟶
     (∃ full' h t . sem_exec_c_p gs full n0 full' ∧
     cont full' = Inl (h#t))))"
```

We say a program `full` is safe for `n` steps if either

- It halts without error in n or fewer steps, or
- It neither halts nor reaches an error state for n steps

This enables us to define an indexed notion of guardedness:

```
definition guarded :: "('syn, 'mstate :: Okay) semc ⇒
  ('mstate ⇒ bool) ⇒
  nat ⇒
  'syn gensyn list ⇒ bool"
("|#_#| {#_, _#} _" [210, 212, 214, 216])
 where
"guarded gs P n c =
  (∀ m . m ∈ ok_S ⟶
    P (payload m) ⟶
    cont m = Inl c ⟶
    safe_for gs m n)"
```

So we write |#gs#| {#P, n#} c if c is guarded under P for n steps (we use # signs in the notation to avoid collision with the notation defined above for standard guardedness, as well as to suggest the indexed nature of the predicate). With this definition of guardedness we can define a corresponding definition of Hoare triple:

```
definition HT :: "('syn, 'mstate :: Okay) semc ⇒ ('mstate ⇒ bool)
⇒ nat ⇒ 'syn gensyn list ⇒ ('mstate ⇒ bool) ⇒ nat ⇒ bool"
  ("|#_#| {#-_, _-#} _ {#-_, _-#}"
   [220, 222, 224, 226, 228, 230])
  where
"HT gs P np c Q nq =
  (∀ c' . |#gs#| {#Q, nq#} (c') ⟶ |#gs#| {#P, np#} (c @ c'))"
```

In other words, |#gs#| {#P, np#} c {#Q, nq#} means that for any c' safe under Q for nq steps, the concatenation c @ c' is safe under P for np steps. One subtlety here is that np is not required to be greater than nq. This might occur, for instance, in a program that looks like the following pseudocode:

```
(* c *)
if(cond == true) {
```

```
    do_something(); (* 5 steps of computation *);
    crash;
}

(* c' *)
do_something(); (* 10 steps of computation *)
```

In this case, `c @ c'` is safe under the precondition `True` (i.e., any initial state) for 5 computation steps (for simplicity, assume the step counter does not increase when evaluating conditionals). However, to show this, we need to make use of the fact that `c'` is safe for 10 steps (if, for instance, `c'` crashed immediately, `c @ c'` would not be safe for 5 steps when `cond = false`).

We can prove consequence and concatenation lemmas for this definition of Hoare logic; naturally, these lemmas must take into account the indices carried around along with the Hoare triples:

```
lemma HConseq :
  assumes H : "|#gs#| {#- P', np' -#} c {#-Q', nq'-#}"
  assumes HP1 : "⋀ st . P st ⟹ P' st"
  assumes HP2 : "np ≤ np'"
  assumes HQ1 : "⋀ st . Q' st ⟹ Q st"
  assumes HQ2 : "nq' ≤ nq"
  shows "|#gs#| {#-P, np-#} c {#-Q, nq-#}"
lemma HCat :
  assumes H : "|#gs#| {#- P1, np1 -#} c1 {#- P2, np2 -#}"
  assumes H' : "|#gs#| {#- P2, np2 -#} c2 {#- P3, np3 -#}"
  shows "|#gs#| {#- P1, np1 -#} (c1 @ c2) {#- P3, np3 -#}"
```

Worth noting is that this version of the rule of consequence (`HConseq` for the step-count Hoare triples) allows us to *decrease* the step count attached to the precondition as we *strengthen* the precondition, and to *increase* the step count attached to the conclusion as we *weaken* the conclusion. This may seem counterintuitive, but recall that the step-count in the "conclusion" of the Hoare triple corresponds to an assumption when unfolding the definition of the Hoare triple

(i.e., have arbitrary `c'`, and we get to *assume* `c'` is safe for `nq` steps); whereas the step-count attached to the "premise" of the Hoare triple actually corresponds to the conclusion when the Hoare triple definition is unfolded (since our end goal is to prove safety for `c @ c'` for `np` steps).

The concatenation lemma `HCat` is a straightforward extension of the one we saw above for the non-step-counting Hoare logic.

This step-counting formulation of Hoare logic is necessary because of the generality and open-endedness of GAZELLE's syntax and instruction set. When reasoning about non-local control flow (`break` and `continue`), [AB07] are able to perform an exhaustive case-analysis at a key point in the proof of correctness of the rules for `while` and `break`/`continue`. We are not able to make the same argument here, because of GAZELLE's extensibility: our Hoare logic is designed *not* to need to know exhaustively which other instructions might affect control-flow.

4.5 Soundness of Step-Counting Hoare Logic

While the step-counting formalism ends up being extremely useful when reasoning about loops in GAZELLE, it is important to justify that the formalism is reasonable; that is, that when we say `|#gs#| {#P, np#} c {#Q, nq#}` it means something along the lines of what one would expect from a Hoare triple. This is especially important as adding an explicit step-count to Hoare logic is a non-standard extension to an already (somewhat) non-standard development of Hoare logic (in terms of its "CPS-flavor"). Otherwise, we run the risk of building a system that is not useful, or even misleading.

In order to resolve this problem, we relate the step-counting Hoare logic back to the CPS-flavored Hoare logic just developed (in section 4.4). To begin with, we define yet another notion of Hoare triple - `HT'` - that wraps the step-counting version of `HT` so as to hide the step counts:

definition `HT' :: "('syn, 'mstate :: Okay) semc ⇒ ('mstate ⇒ bool) ⇒ 'syn gensyn list ⇒ ('mstate ⇒ bool)⇒ bool"`

```
  ("|_| {~_~} _ {~_~}" [250, 252, 254, 256])
  where
"HT' gs P c Q =
  ((∀ npost . ∃ npre . |#gs#| {#- P, (npre + npost) -#} c {#-
Q, npost -#}))"
```

That is, we say |*gs*| {~*P*~} *c* {~*Q*~} if, for any step-count *npost*, we can come up with a *npre* such that, |*#gs#*| {*#P, (npre + npost)#*} *c* {*#Q, npost#*}. Unpacking this further (and thinking about the quantifiers in terms of a game), that means that for any *npost* and *c'* provided by an adversary, where *c'* is safe for *npost* steps, we must be able to show that the concatenation *c @ c'* is safe for *npre + npost* steps. Another way to put this, bearing in mind that all these indices are nonnegative, is that we can come with an *npre* such that *npost* ≤*npre* and |*#gs#*| {*#P, npre#*} *c* {*#Q, npost#*}.

Our goal is to show that this definition of Hoare triples matches the non-step-counted, CPS version of *HT*. Ideally we'd like to show that *HT'* is *sound* and *complete* with respect to *HT'*. The soundness lemma does indeed hold; it says the following:

```
lemma HT'_imp_HT :
  assumes H : "|gs| {~P~} c {~Q~}"
  shows "|gs| {-P-} c {-Q-}"
```

The completeness lemma, which we believe does not hold, would say the following:

```
lemma HT_imp_HT' :
  assumes "|gs| {-P-} c {-Q-}"
  shows "|gs| {~P~} c {~Q~}"
```

Unfortunately, it appears that *HT'* is a more informative (stronger) statement than *HT*. When attempting to prove *HT'* from *HT*, we reach a point where we need to show |*gs*| {-*Q*-} *c'* (for the arbitrarily chosen *c'* fixed as part of the definition of *HT'*), so that we can use the assumption |*gs*| {-*P*-} *c* {-*Q*-}. Unfortunately, all we know is that |*#gs#*| {*#Q, npost#*} *c'* (for *npost* fixed as part of the definition of *HT'*), and so we are stuck.

Despite not being able to prove completeness, showing soundness is still a key result, since it means that if *HT'* fails to be useful, its failure will be in that it is too strong; i.e., too hard

to prove, rather than too weak; i.e., not meaningful. As we show later when discussing the Hoare logic for the IMP language, (section 5.3) we are able to prove and use a rich set of Hoare rules using *HT'* in practice, making this much less of a concern.

4.6 Reasoning about Multi-Step Composition

4.6.1 Dominance and Toggling

Putting together the Hoare logic just described with the formalization of language-component composition described in chapter 2 gives us a very general framework for reasoning about programs whose semantics are given by such compositions. However, the downside to this generality is that, without further simplifying assumptions, such reasoning can be challenging. In particular, we may have to break the Hoare-rule abstractions at certain points in order to show that the rules do not interfere with each other.

Fortunately, in many instances of composition, we can make use of additional structure of the languages and their interactions to dramatically simplify our reasoning and minimize the need to break the modularity provided by the Hoare abstraction. One example we've already seen is the *Vmerge* single-step Hoare rule in section 4.2. Another common case is when one language can be shown to produce a result informationally greater than (or equal to) the merging of all other languages it is being composed with, for particular instructions (syntax-elements). We call this *dominance* and define it formally as:

```
definition dominant ::
   "('a ⇒ 'c ⇒ ('c :: Pord_Weak)) ⇒
    ('a ⇒ 'c ⇒ 'c) set ⇒
    'a set ⇒ bool"
("_ ↓ _ _" [250, 252, 254])
where
"(f ↓ S X) =
   (∀ x b . x ∈ X ⟶
    (is_sup ((λ g . g x b) ` S) (f x b)))"
```

That is, `f` dominates the set of functions `S` for syntax-set `X` if, the result of applying `f` to any input `b` in `X` is the least upper bound of the result of mapping all functions in `S` over that same input. We write this as `f ↓ S X` In practice we will often use `ok_S` as the `X` parameter (for more information on `ok_S`, see section 2.2.2.5). From this definition we can easily derive:

```
lemma dominant_pcomps :
  assumes Hpres : "sups_pres (set fs) (λ _ . ok_S)"
  assumes Hne : "z ∈ set fs"
  assumes H : "(f ↓ (set fs) X)"
  assumes Xin : "x ∈ X"
  assumes Bin : "b ∈ ok_S"
  shows "(pcomps fs x b) = (f x b)"
```

That is, if the set `S` is compatible in the sense of `sups_pres` for inputs which are in `ok_S` (i.e., have valid data inside them), `S` is nonempty, and `f` dominates `S` for a set of syntax elements `X`, then the result of the composition of the functions in `S` when applied to a syntax element of the specified set `X` and a valid input state will be equal to the result of applying `f` to that syntax and input state. This lemma underlines the usefulness of dominance as an abstraction: when we can show it holds, we can reduce reasoning about the composition of several functions to reasoning about the behavior of the one function that is known to dominate the others (when reasoning about instructions from the specified set `X`). We use the `dominant` abstraction to build a set of modular proof-rules for the IMP language as part of our case study (see chapter 5).

When dealing with languages with extensible state types (i.e., languages having a type-parameter corresponding to extensions to the state), we run into a subtlety that gets in the way of showing dominance directly, even in cases where one would intuitively think it should hold. Consider, for instance, a state with two wrapped integers `(int md_triv option md_prio * int md_triv option md_prio)`. Suppose further we have two semantics over the state and a trivial instruction language with a single instruction (`() :: unit`). The first function, `sem1`, increments the first component of the tuple, but is generic in the second component of the tuple (this corresponds to the situation we encounter when using `no_control_lifting`; see section 5.3.1). The second function has access to the full tuple state, and increments the second

parameter. *sem1_p* and *sem2_p* supply priority increments for *sem1* and *sem2* respectively, used when merging their results (see section 3.2.4).

```
datatype syn =
  Op1
  | Op2

fun sem1 :: "syn ⇒ int ⇒ int" where
"sem1 Op1 x = (1 + x)"
| "sem1 _ x = x"

fun sem1_p :: "syn ⇒ nat" where
"sem1_p Op1 = 2"
| "sem1_p _ = 1"

fun sem2 :: "syn ⇒ (int * int) ⇒ (int * int)" where
"sem2 Op2 (x1, x2) = (x1, (2 + x2))"
| "sem2 _ x = x"

fun sem2_p :: "syn ⇒ nat" where
"sem2_p Op2 = 2"
| "sem2_p _ = 1"

type_synonym state =
  "int md_triv option md_prio *
   int md_triv option md_prio"

definition sem1_lift ::
  "(syn, int,
    int md_triv option md_prio *
    ('a :: Mergeableb)) lifting" where
"sem1_lift = schem_lift NA (SP (SPRC sem1_p (SO NA)) NX)"

definition sem2_lift :: "(syn, (int * int), state) lifting" where
"sem2_lift =
  schem_lift (SP NA NB)
    (SP (SPRI (SO NA)) (SPRC sem2_p (SO NB)))"
```

We would like to be able to show that the result of lifting *sem1* by *sem1_lift* dominates the result of lifting *sem2* by *sem2_lift*, for instruction *Op1*. Intuitively, we know that for *Op1*, *sem1_lift* will cause the first component's priority to be incremented by 2, whereas *sem2_lift* at *Op1* will only increment by 1 for the first parameter. However, the second parameter will be left

untouched by the lifted *sem1* (its structure is hidden behind the type variable *'a* in *sem1_lift*), yet it will be incremented by *sem2_lift*. Even though the data is left unchanged, we end up in a situation where neither *sem1* nor *sem2* produces a larger result after lifting.

One might ask why it is necessary to increment the priority of the first component in *sem2_lift*, since the data is not being modified. Unfortunately, in order for the lifting to obey *lifting_valid_strong*, the priority must always strictly increase (since the lifting itself is not aware that this parameter will never be changed by *sem2*). We want to work with liftings obeying *lifting_valid_strong* whenever possible, as such liftings are far easier to reason about than those merely implementing *lifting_valid* (for details on this distinction, see section 3.1.2).

We need to adopt a different solution for such cases, which we refer to as *toggling*. The idea is that when lifting a semantics that is designed to be a "no-op" on certain arguments, we supply a Boolean predicate characterizing which syntax elements to run the semantics on, and which to simply ignore, returning the original result unchanged:

```
definition
  lift_map_t_s ::
  "('b1 ⇒ 'a1) ⇒
   ('a1, 'a2, 'b2::Pord) lifting ⇒
   ('b1 ⇒ bool) ⇒
   ('a1 ⇒ 'a2 ⇒ 'a2) ⇒
   'b1 ⇒ 'b2 ⇒ 'b2" where
"lift_map_t_s l' l tg f syn st =
  (if tg syn then lift_map_s l' l f syn st
   else st)"

definition toggle ::
  "('syn ⇒ bool) ⇒
   ('syn ⇒ 'b ⇒ 'b) ⇒
   ('syn ⇒ 'b ⇒ 'b)" where
"toggle tg f syn st =
  (if (tg syn) then f syn st else st)"
```

When composing several toggled semantics, we have an easy way to prove dominance, using the following theorem:

```
lemma dominant_toggles :
  assumes Valid : "lifting_valid l1 S1"
```

```
assumes Fs_fin :
  "finite (Fs :: (_ ⇒ (_ :: Mergeable) ⇒ _) set)"
assumes Fs_f1 : "lift_map_t_s l'1 l1 t1 f1 ∈ Fs"
assumes Toggle1 : "⋀ s . s ∈ X ⟹ t1 s"
assumes Toggles: "⋀ f . f ∈ Fs ⟹
  f ≠ lift_map_t_s l'1 l1 t1 f1 ⟹
  (∃ tg g . f = toggle tg g ∧ (∀ s . s ∈ X ⟶ ¬ tg s))"
shows  "(lift_map_t_s l'1 l1 t1 f1) ↓ Fs X"
```

That is, if toggle predicate `t1` is true for all syntax elements in `X`, the lifting of `f1` using `t1` is in the set of functions `Fs`, and all other functions in `Fs` are toggled using a toggling predicate that is false for all syntax elements in `X`, then the lifting of `f1` dominates `Fs` for inputs `X`. Using this fact, we can complete our example, defining toggling functions for our languages and making use of `dominant_toggles` to show that the lifted `sem1` dominates the lifted `sem2` on `Op1`:

```
fun sem1_toggle :: "syn ⇒ bool" where
"sem1_toggle Op1 = True"
| "sem1_toggle Op2 = False"

fun sem2_toggle :: "syn ⇒ bool" where
"sem2_toggle Op1 = False"
| "sem2_toggle Op2 = True"

definition sems :: "(syn ⇒ state ⇒ state) set" where
"sems = {lift_map_t_s id sem1_lift sem1_toggle sem1
        ,lift_map_t_s id sem2_lift sem2_toggle sem2}"

lemma sem1_dominant :
  "(lift_map_t_s id sem1_lift sem1_toggle sem1) ↓ sems ({Op1})"
proof (rule dominant_toggles)
```

4.7 Discussion

This completes the picture of the reasoning framework offered by GAZELLE for constructing proofs about programs written using languages defined via composition in GAZELLE. A theme here is that we adopt extremely general notions whenever possible, in order to enable maximum expressiveness and maximally powerful reasoning. Such generality leads to challenging proofs, however, so we adopt some additional primitives (such as dominance and toggling)

in order to ease the proof burden for common cases. In chapter 5, when we discuss IMP, we will see how these common cases largely suffice to build a program logic for and prove correct programs in a nontrivial imperative language, while still allowing for interesting interactions between language-components.

Chapter 5

IMP: An Extended Example of GAZELLE in Practice

At this point, we have discussed the entire GAZELLE framework. GAZELLE enables composition of separately-defined programming-language components (as described in chapter 2). GAZELLE enables retrofitting of existing formal developments of (denotational) language-component semantics, enabling them to work with the its infrastructure via its lifting framework (as described in chapter 3). Finally, GAZELLE supports reasoning about such compositions of language components using Hoare logic (described in chapter 4).

To put everything together, we present an extended example of the usage of GAZELLE. Namely, we show how it can be used to define a simple imperative language - quite similar to the IMP language frequently presented in the literature (for instance, [PAdAG+21b, ch.12]) - from self-contained components. We also show how GAZELLE enables reasoning about that language using self-contained reasoning rules for each component. In the process, we will build several components useful for implementing languages beyond IMP - in keeping with GAZELLE's goals of enabling reuse - and demonstrate useful patterns for productively using the GAZELLE framework.

5.1 IMP's Sub-Languages

IMP programs are defined as *syn gensyn* syntax trees, for the following combined instruction-syntax *syn*:

```
datatype syn =
  Sc "calc"
  | Sm "Mem_Simple.syn"
  | Sb "cond"
  | Si "Imp_Ctl.syn'"
  | Ss "Seq.syn"
  | Ssk
```

Each sub-language of IMP injects into *syn*; each case of *syn* corresponds to one sub-language of IMP. In particular:

- *Sc* injects an instruction from the *Calc* arithmetic language
- *Sm* injects an instruction from the memory-store language (*Mem*, called *Mem_Simple* in the formal development)
- *Sb* injects an instruction from the language *Cond* of boolean conditions
- *Si* injects an instruction from the *Imp_Ctl* language implementing *Imp*'s conditional control-flow
- *Ss* injects an instruction from the *Seq* language, implementing sequencing
- *Ssk* represents a no-op or "skip" instruction

In this section, we give an overview of these sub-languages, which we compose to implement IMP using GAZELLE.

5.1.1 Arithmetic Language

We want our implementation of IMP to be able to perform arithmetic operations. This is the responsibility of *calc*, a language giving access to the operations of a four-function calculator. Its syntax is as follows:

```
datatype calc =
  Cadd
  | Csub
  | Cmul
  | Cdiv
  | Cnum int
  | Cskip
```

Cadd, *Csub*, *Cmul*, and *Cdiv* implement the basic arithmetic operations their names imply. *Cnum* corresponds to an integer-literal instruction, and *Cskip* is an operation with no effect. As we'll see when defining the remaining language-components comprising IMP, it is generally useful to have a such a "no-op" instruction in language-components used with GAZELLE (for those that lack one, it is of course quite easy to extend the syntax and semantics to add one). Such an instruction allows us to express the idea that one language-component does nothing while an instruction exclusive to another language-component is being executed.

calc operates over what is essentially a three-register machine, represented as three components of a triple:

```
type_synonym calc_state =
  "(int * int * int)"
```

The first two components are the arguments to the arithmetic operation being performed; the third is the output of the operation. *Cadd* (and the other arithmetic operations) read their inputs from the first two components and write to the third; *Cnum* writes its literal argument to the third. Notice that from within *Calc* we don't have a way of reading from the output or writing to the inputs, nor any kind of control construct for sequencing arithmetic operations. All of these will be provided by other language-components.

Here is the semantics of *Calc*:

```
fun calc_sem :: "calc ⇒ calc_state ⇒ calc_state" where
"calc_sem (Cadd) (x1, x2, x3) =
   (x1, x2, x1 + x2)"
| "calc_sem (Csub) (x1, x2, _) = (x1, x2, x1 - x2)"
| "calc_sem (Cmul) (x1, x2, _) = (x1, x2, x1 * x2)"
| "calc_sem (Cdiv) (x1, x2, _) =
   (x1, x2, divide_int_inst.divide_int x1 x2)"
| "calc_sem (Cnum i) (x1, x2, _) = (x1, x2, i)"
| "calc_sem (Cskip) st = st"
```

`Calc` injects into `syn` using the `Sc` constructor. To recover a `Calc` instruction from a combined-language instruction, we use the following translation function:

```
fun calc_trans :: "syn ⇒ calc" where
"calc_trans (Sc x ) = x"
| "calc_trans _ = Cskip"
```

We will see this pattern with the translation functions for the other sub-languages of IMP (aside from `seq`, which requires some special handling). For any combined-language instruction, if it corresponds to an injected instruction of our sub-language (in this case `Sc x` for a `calc` instruction), we project out that sub-language instruction. Otherwise, we return the skip instruction from the sub-language.

5.1.2 Boolean Language

Along similar lines to `Calc`, we define another language, `Cond`, corresponding to operations of Boolean logic. Such operations are important to expressing interesting conditions for the `Sif` and `SwhileC` statements introduced below as part of the `Imp_Ctrl` language-component.

Our approach to Boolean expressions is nearly identical to that for arithmetic. One important point is that we do not have a separate Boolean type, instead choosing to represent Boolean values as integers. This simplifies the process of defining the liftings necessary to compose `Cond` with the other language-components. It should be emphasized, however, that GAZELLE is not fundamentally restricted to unityped languages. Implementing languages with more interesting notions of datatype is possible, and is future work.

The syntax of *Cond* is as follows:

```
datatype cond =
  Seqz
  | Sltz
  | Sgtz
  | Sskip_cond
```

Cond corresponds to a simple language of conditionals comparing a natural-number input to zero. Much like *Calc*, *Cond* represents its state as a tuple:

```
type_synonym cond_state = "int * int"
```

The first *int* represents the Boolean input; the second represents the output. The conversion between *bool* and *int* is given as:

```
abbreviation encode_bool :: "bool ⇒ int" where
"encode_bool b ≡
  (if b then 1 else 0)"
```

We give the semantics for *Cond*:

```
definition cond_sem :: "cond ⇒ cond_state ⇒ cond_state" where
"cond_sem x s =
  (case s of (b, i) ⇒
    (case x of
      Seqz ⇒ (encode_bool (i = 0), i)
      | Sltz ⇒ (encode_bool (i < 0), i)
      | Sgtz ⇒ (encode_bool (i > 0), i)
      | Sskip_cond ⇒ s))"
```

Cond injects into *syn* using the *Sb* constructor. To recover *Cond* instructions from *syn*, we use the following translation function:

```
fun cond_trans :: "syn ⇒ Cond.cond" where
"cond_trans (Sb x) = x"
| "cond_trans _ = Sskip_cond"
```

5.1.3 Variable-Store Language

Next we introduce the *Mem* language, which provides access to a memory store mapping *String.literal* names to *int* values. It has the following syntax:

```
datatype reg_id =
  Reg_a
  | Reg_b
  | Reg_c
  | Reg_flag

datatype syn =
  Sread "str" "reg_id"
  | Swrite "str" "reg_id"
  | Sskip
```

`Mem` handles moving data between the registers used by `Calc` and `Cond` and the memory-store. We use `reg_id` to specify the register target of our reads and writes. `Mem` has access to all the registers used by the other language components of IMP, laid out in a tuple along with the memory-store itself. In order, the components of this state tuple are:

1. The `Reg_flag` register, from which the `if` and `while` control-flow constructs (discussed below) read their Boolean condition
2. The result register `Reg_c` (an `int`), to which `Calc` and `Cond` write the results of their operations
3. Input register `Reg_a` (an `int`), from which `Cond` reads its input and `Calc` reads its first input
4. Input register `Reg_b` (an `int`), from which `Calc` reads its second input
5. The memory-store, which is an association-list mapping `String.literal` keys to `int` values

We abbreviate this state type as `state0`. The semantics of `Mem` move values into and out of the memory store, leaving the remaining tuple components untouched. Here is an excerpt of the semantics function for `Mem`:

```
fun mem0_sem :: "syn ⇒ state0 ⇒ state0" where
"mem0_sem (Sread s r) (reg_flag, reg_c, reg_a, reg_b, mem) =
  (case get mem s of
    Some v ⇒
      (case r of
        Reg_a ⇒ (reg_flag, reg_c, v, reg_b, mem)
        | Reg_b ⇒ (reg_flag, reg_c, reg_a, v, mem)
        — ... remaining cases similar)
    | None ⇒ (reg_flag, reg_c, reg_a, reg_b, mem))"
| "mem0_sem (Swrite s r) (reg_flag, reg_c, reg_a, reg_b, mem) =
  (case r of
    Reg_a ⇒
      (reg_flag, reg_c, reg_a, reg_b, update s reg_a mem)
    | Reg_b ⇒
      (reg_flag, reg_c, reg_a, reg_b, update s reg_b mem)
    — ... remaining cases similar)"
| "mem0_sem _ st = st"
```

`get` and `update` perform the association-list accesses. We have made the choice to ignore reads from an undefined variable. We could instead have `Mem` signal an error to the interpreter when it encounters such a situation, but this would require giving `Mem` access to the interpreter's error flag, which would complicate the liftings involved slightly.

`Mem` injects into `syn` using the `Sm` constructor. To recover a `Mem` instruction from `syn`, we use the following translation function:

```
fun mem_trans :: "syn ⇒ Mem_Simple.syn" where
"mem_trans (Sm m) = m"
| "mem_trans _ = Mem_Simple.Sskip"
```

5.1.4 Sequencing Language

So far we have still not discussed the control-flow aspects of IMP. With GAZELLE, we can separate the control-flow aspects of IMP from the remainder of the language; in fact, we can even separate different parts of control-flow from each other. The `Seq` language serves as an example of this pattern - it is a component that is suitable for use in any language that desires sequencing behavior, and is not tied to the choice of other control-flow primitives used in IMP.

Syntactically, the sequencing language is quite straightforward: it consists of just two node labels:

```
datatype syn =
  Sseq
  | Sskip
```

The state of the *Seq* language is a list of *'x gensyn* syntax trees. This list will ultimately be mapped via a lifting into the GAZELLE interpreter's continuation field. We leave *'x*, the type of syntax-tree node labels for the *entire* combined language as a type parameter so that we can compose *Seq* with other languages later on. *Seq* does not need to know the details of the node labels' contents, so this does not create any problems.

The semantics for *Seq* are given by the following evaluation function:

```
type_synonym 'x state' = "'x gensyn list"

definition seq_sem :: "syn ⇒ 'x state' ⇒ 'x state'" where
"seq_sem x st =
  (case st of [] ⇒ []
   | (G s l)#t ⇒
     (case x of
      Sskip ⇒ t
      | Sseq ⇒ l@t))"
```

Seq updates the list of syntax-trees representing the continuation. When it encounters a root *Sseq* node with descendants at the head of the continuation list, it removes the node from the list and adds its descendants to the list in its place. Once we connect *Seq*'s state to the interpreter's continuation field via liftings (see sections 5.1.4 and 5.2.7), this will cause the GAZELLE control-flow interpreter to descend into subtrees rooted at *Sseq* nodes (i.e., execute a the sequence of constructions contained therein). Otherwise, for non *Sseq* nodes (*Sskip*), we simply discard the head of the continuation and replace the continuation with its tail. This corresponds to moving on to the next instruction (the head is assumed to be executed by some other language-component, *Seq* does not need to worry about the specifics of how this happens).

Seq injects into *syn* using the *Ss* constructor. To recover a *Seq* instruction from *syn*, we use the following translation:

```
fun seq_trans :: "syn ⇒ Seq.syn" where
"seq_trans (Ss x) = x"
| "seq_trans _ = Seq.Sskip"
```

5.1.5 IMP-Control Language

Finally, we have the *Imp_Ctl* language, which implements the conditional and looping constructs used by our GAZELLE implementation of IMP. *Imp_Ctl*'s syntax is as follows:

```
datatype syn' =
  Sif
  | Sskip
  | SwhileC
```

Note that we could easily have separated *Sif* and *Swhile* into distinct language components if we wanted to break down *Imp*'s control-flow constructs on a finer granularity. The state type for *Imp* (which we call *'x imp_state'*) is:

```
type_synonym 'x imp_state' = "'x gensyn list * int"
```

The first component corresponds to the control-flow information *Imp_Ctl* needs to manage (as with *Seq*, above; see section 5.1.4). The second field is the *flag* register used when evaluating conditionals (and meant to be accessible by *Mem*, as noted above in section 5.1.3.)

Imp_Ctl provides two branching control-flow primitives. The first is an *if*-statement. *if* nodes are meant to have two children: the first corresponds to a boolean expression to evaluate for the condition of the *if*; the second is the body.

The second primitive, *SwhileC*, is essentially a *while* loop without a built-in condition [1]: it iterates the loop body until the beginning of an iteration at which the boolean flag is zero (i.e., *false*).

To encode these compound statements, we make use of GAZELLE's *gensyn* datatype: an if-statement will look like ◊*Sif [cond, body]*, and a while-loop will look like ◊*SwhileC*

[1] Omitting the condition leads to a more straightforward Hoare rule without sacrificing expressivity; a variant with a condition could be supported easily.

`[body]`. The function giving the semantics for `Imp_Ctl` is a bit more complicated than the ones we've seen so far:

```
definition imp_ctl_sem :: "syn' ⇒ 'x imp_state' ⇒ 'x imp_state'"
where
"imp_ctl_sem x st =
   (case st of
     ([], b) ⇒ ([], b)
     | ((G z l)#t, b) ⇒
       ((case x of
         Sskip ⇒ t
         | Sif ⇒
         (case l of
           [body] ⇒ (if (b ≠ 0) then body#t else t)
           | [cond, body] ⇒ cond# ((G z [body])#t)
           | _ ⇒ [] — error)
         | SwhileC ⇒
         (case l of [body] ⇒
           (if (b ≠ 0) then body # (G z [body]) # t
            else t)
          | _ ⇒ [] — error))
       , b))"
```

The most noteworthy thing about `imp_ctl_sem` is the way that `Sif` is implemented. `Sif` is evaluated in multiple steps:

1. We begin with `(◊Sif [cond, body])` at the head of the continuation list
2. We tell the GAZELLE interpreter to execute `cond` first, then `◊Sif [body]` (followed by the original tail of the list)
3. After evaluating `cond`, we are left with `◊Sif [body]`, followed by the tail
4. We then execute `◊Sif [body]`, which involves checking the condition-flag register (which will have already been set since we just executed `cond`). If the condition is true (nonzero), we execute the `body` followed by the original tail. Otherwise, we just execute the tail.

In other words, `Sif` occurs in two variants: one, `◊Sif [cond, body]` corresponds to an if-statement that has not yet been evaluated (this is the version that will occur in the original syntax

tree of IMP programs being run by GAZELLE). The second variant, `◊Sif [body]`, corresponds to the condition having been evaluated already, and causes the interpreter to branch based on the value of the condition-flag register. This breaking-down of statements into multiple continuation variants is inspired by techniques from the defunctionalization and continuation-passing style literature (e.g. [Dan08] and [HB16]).

Another point worth observing here is the distinction between the values `z` (the syntax-node label pulled out of the continuation field via case analysis) and `x` (the syntax-node label given as input to `imp_ctl_sem`). Significantly, these two are of different types: `z` has type `'x`, a type in which `imp_ctl_sem` is parametrically polymorphic and thus cannot use for pattern-matching; `x`, on the other hand, is in the form of a translated label (from the full language to the `Imp_Ctl` sub-language) of type `syn'`. This means that `imp_ctl_sem` can understand and perform case-analysis on syntax node labels without needing access to a full definition of the syntax of the combined language. All languages implementing flow, including `Sseq`, will exhibit this behavior.

Due to the way the GAZELLE interpreter works (see section 2.3.2), `x` will always directly correspond to the translation of `z` into `Imp_Ctl` syntax. Because the manipulations performed by `imp_ctl_sem` on the continuation field care only about the shape of the syntax trees involved rather than the contents of the syntax-tree labels, we can still perform the manipulations while leaving the labels' type as a type parameter `'x` in the actual state. In this way, we are able to define useful control-flow primitives while maintaining modularity.

It should be noted that the cases where `l = []` and `x = Sif` or `x = SwhileC` correspond to errors: a childless `Sif` or `SwhileC` node is not syntactically valid. While we could have made these interpreter errors, we choose instead to simply halt the program, since correct handling of invalid syntax is not our focus here (such errors could be eliminated via a static check).

`Imp_Ctl` injects into `syn` using the `Si` constructor. To recover an `Imp_Ctl` instruction

from *syn*, we use the following translation function:

```
fun imp_trans :: "syn ⇒ Imp_Ctl.syn'" where
"imp_trans (Si x) = x"
| "imp_trans _ = Imp_Ctl.Sskip"
```

5.1.6 Discussion

It is commonplace to separate, for instance, arithmetic-language semantics (which can easily be given denotationally) from control flow (this is the approach taken in *Software Foundations*' formalization of *Imp*; see [PAdAG+21b, ch.14]). However, with GAZELLE we are able to break down control-flow itself into isolated components, as evidenced here with the separation between *Seq* and *Imp_Ctl*. These semantics definitions, in and of themselves, do not tell the full story of how these language-components fit together to give an implementation of IMP. For this, we need to specify the *liftings* that show how the state types for each of these sub-languages map into a common state-type for the combined language. We turn our attention there next.

5.2 Liftings for Constructing IMP

In this section, we give definitions for each language-component's lifting, describing how precisely each component contributes to the overall state-transitions of IMP (for more details on liftings, see chapter 3). It should be noted that, while we focus here on liftings, in the development we also construct corresponding valid-sets for the liftings, in order to be able to prove their validity. These match the structure of their corresponding liftings, and are omitted for brevity. The interested reader can refer to the formal development for more information about the valid-sets used in IMP. (Language-component definitions can be found in `Gazelle/Language_Components`; the composed IMP language can be found in `Gazelle/Languages/Imp`).

5.2.1 An Overview of IMP State

First, we define a combined state type for IMP. This state packages together all the components of the state-types used by the language-component semantics defined above (section 5.1). This is not simply a tuple of all the language-component states side by side, however, as some of the states overlap. For instance, the flag register used by *Mem* and the flag register from which *Imp_Ctl* reads its conditions need to be the same, so that IMP can branch based on the results of memory reads.

To ease the expression of IMP's combined state, we define a helpful type synonym *'x swr*, which captures the "standard wrapping" of putting the *'x* data inside a nesting of wrapper types:

- At the innermost layer, we have *'x*, the data type we are embedding
- We wrap this in *md_triv* to impose a trivial ordering on it (see section 2.2.3.1)
- We further wrap with *option* to add a least element to the ordering (see section 2.2.3.2)
- We finally wrap with *md_prio*, in order to enable GAZELLE's priority-ordering system to handle conflicts when merging data (see section 2.2.3.4)

  In code, we define this as:

```
type_synonym 'a swr =
  "'a md_triv option md_prio"
```

  The combined state of IMP contains the following elements:

- The first component of the GAZELLE interpreter state: a list of syntax trees representing the current continuation
- The second component of the GAZELLE interpreter state: a field for signaling errors with an error message

- The *flag* register (an *int*)
- The result register, *Reg_c* (an *int*)
- The first input register, *Reg_a* (an *int*)
- The second input register, *Reg_b* (an *int*)
- The memory-store
- A field of a parameterized type *'x*, allowing the state to be extended further

Each of these fields (except for the last one, since we want to allow extensions flexibility with respect to the ordering they choose on the remaining elements they add), is wrapped in the standard wrapping given as the *swr* type. Putting this together, the state of IMP is

```
type_synonym ('s, 'x) entire_state =
  "('s gensyn list md_triv option md_prio *
       String.literal option md_triv option md_prio *
       int md_triv option md_prio *
       int md_triv option md_prio *
       int md_triv option md_prio *
       int md_triv option md_prio *
       (String.literal,
        int) oalist md_triv option md_prio *
       'x)"
```

We can restate this a bit more simply. The first two fields come from the GAZELLE control-flow system. Recall (see section 2.3.2) that the *('s, 'x) control* type alias packages a datum of type *'x* with the interpreter-state fields. Therefore, the above is equivalent [2] to:

```
type_synonym ('s, 'x) entire_state_alt =
  "('s, 'x imp_state') control"
```

For a visual representation of the combined state structure of IMP, and its relationship to the structures of the individual sub-languages' states, see figure 5.1.

[2] In the codebase, this definition is inside of *Mem_Simple.thy* and has qualified name *Mem_Simple.state*

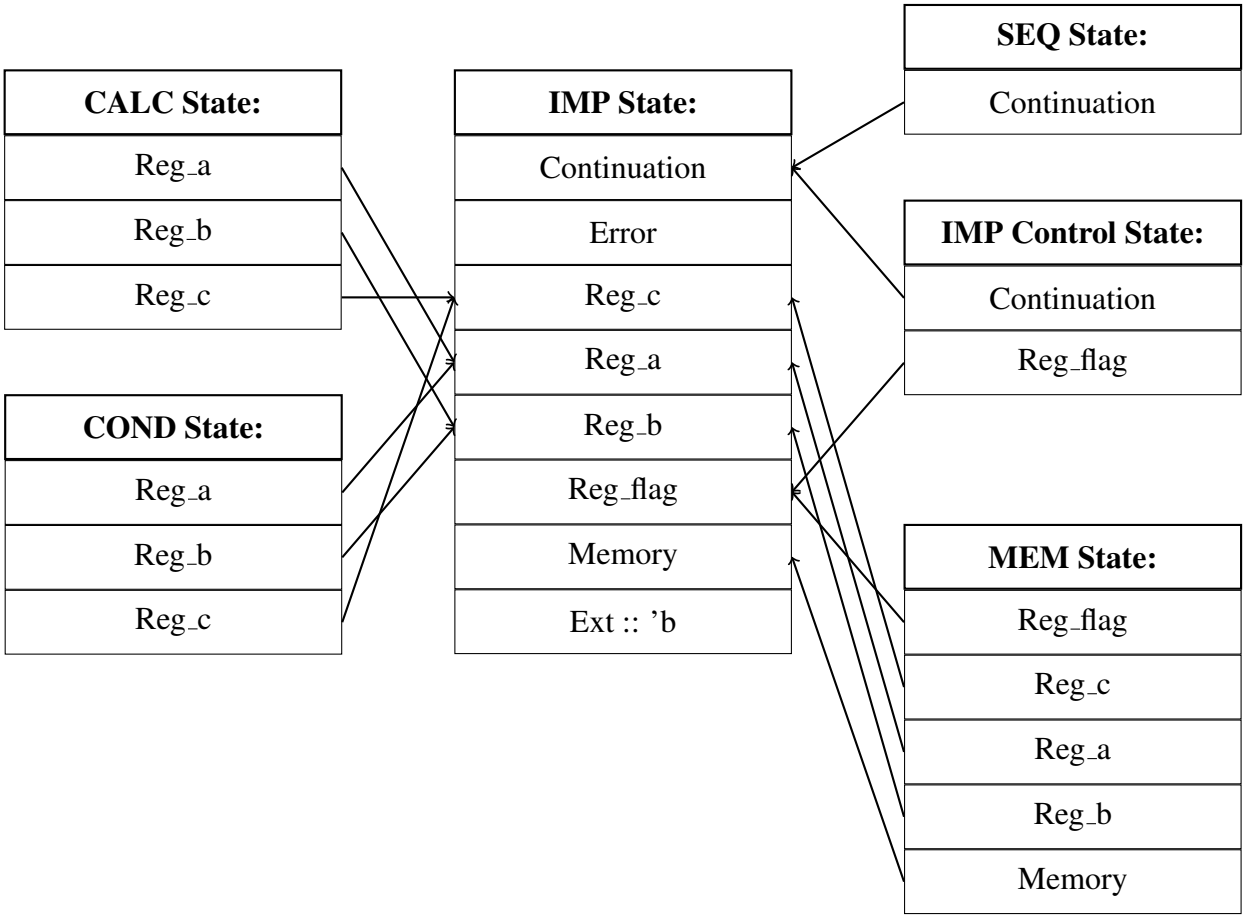

Figure 5.1: Visual depiction of lifting into IMP's combined state; arrows denote injection of data

5.2.2 Priority Protocol for IMP

We use priorities (see section 2.2.3.4) - calculated by *priority functions* based on syntax - in order to control the interactions between language components on overlapping parts of IMP's state. In order to ensure that priorities are used consistently, we establish a protocol on the meaning of priorities - more specifically, on the meaning of the *priority increments* - used by priority functions defined later in this section.

The protocol for IMP is simple:

- A priority increment of 1 is used for data to which the language component does not intend to write. (0 cannot be used, since we desire that our liftings satisfy `lifting_valid`, which requires that priorities always increase). This allows any language component writing at any higher priority to override the data output in this way.

- A priority increment of 2 is used for data being written by the language component at a "normal priority". This priority is used in most cases when a write is intended.

- A priority increment of 3 is used for data being written at a "high priority". Since 3 is the highest priority increment used, a write at priority increment 3 will always override (i.e., be informationally greater) than outputs at any other priority increment. This property is used when merging the semantics of the sequencing language `Seq` and the `Imp_Ctl` language of other IMP control-flow constructs.

Of course, for more complex language compositions, a more complex priority protocol is likely to be appropriate. For extensibility reasons, it may be valuable to leave some extra space in between priority-increment levels, so that new language-components to be added later on can be fit in to the existing priority structure without requiring a renumbering of the others.

5.2.3 Lifting Languages Without Control-Flow

Dividing the state of IMP (using *control* and *imp_state'*) makes it easier to organize the lifting constructs needed to adapt the semantics of *Calc* and *Cond*. These languages do not interact at all (directly) with GAZELLE's control-flow constructs. This means that we can lift the semantics of, e.g., *Calc* by first building a lifting from *calc_state* to *'x imp_state'* (for *'x* arbitrary), and then *injecting* this lifting into a larger lifting that lifts from *calc_state* into *('x, 's) imp_state*.

The lifting *no_control_lifting* performs this composition:

```
definition no_control_lifting ::
"('a, 'b1, 'b2 :: {Bogus, Pord}) lifting ⇒
 ('a, 'b1, ('x, 'b2) control) lifting" where
"no_control_lifting l =
   schem_lift NC (SP NX (SP NX (SINJ l NC)))"
```

Here we see the use of the *SINJ* schema descriptor to inject the existing lifting *l* to create a lifting that lifts into states that are the same as those *l* lifts into but also take GAZELLE's control-flow data into account. (Recall that *control* is a triple of a continuation, an optional error message, and a parameterized type representing the remainder of the state; *no_control_lifting* accesses that remainder component via the input lifting *l*.) Our lifting-validity proofs (see section 3.3.7) guarantee that *no_control_lifting l* is valid if *l* itself is valid.

We can prove the following rule for *no_control_lifting*:

```
lemma HTS_imp_HT'' :
  fixes fs ::
    "('b ⇒ ('b, 'c) control ⇒
     ('b, 'c :: {Bogus, Mergeableb, Okay}) control) list"
  assumes H: "f % {{P'}} (l' x) {{Q'}}"
  assumes Valid : "lifting_valid_ok l S"
  assumes Hf' :
    "f' = lift_map_t_s l' (no_control_lifting  l) tg f"
  assumes H0 : "gs = pcomps fs"
  assumes Hpres : "sups_pres (set fs) (λ _ . ok_S)"
  assumes Hseq : "seq_sem_l_gen lfts ∈ set fs"
  assumes Skip : "lfts x = Sskip"
  assumes Active : "tg x = True"
```

assumes *Xin : "x ∈ X"*
assumes *Hnemp : "g ∈ (set fs - {seq_sem_l_gen lfts})"*
assumes *Hdom : "(f' ↓ (set fs - {seq_sem_l_gen lfts}) X)"*
assumes *HP : "⋀ st . P st ⟹ P' (LOut l (l' x) st)"*

shows
"|gs| {~ (λ st . P st) ~} [G x z]
{~ (λ st . ∃ old_big small_new .
P old_big ∧
Q' small_new ∧
st = LUpd l (l' x) small_new old_big) ~}"

The assumptions say the following:

- For any particular (fixed but arbitrary) syntax-element *x*:
- The Hoare triple *f % {{P'}} syn {{Q'}}* holds for *syn = l' x*, where *l'* is the syntax-translation function used when lifting *f*, and *x* is the combined-language syntax corresponding to an operation in the language *f*. Recall that this is a "single-step" Hoare triple talking about execution of a language component for one step.
- The lifting *l* used to lift *f* is valid according to the specifications defined in section 3.1.2
- *fs*, the list of functions being composed, obeys *sups_pres* (see section 2.2.4), and contains the lifting of *f* as well as our sequencing language
- The lifted version of *f* is not equal to the sequencing semantics
- The lifting of *f* is dominant over all the other languages in *fs* - *except* sequencing - when considering a syntax set *X* containing *x*

Then we can conclude that, when starting in a state satisfying *P* (a combined-language state predicate assumed to imply *P'* on the sub-state covered by *P'*), we end in a state satisfying three clauses. First, there must be a previous state *old_big* for which *P* held; there must be a new sub-state *small_new* for which *Q'* (the conclusion of the Hoare rule for the sub-language)

holds; and finally the ending state of the combined language is given by updating the *old_big* state with the data from *small_new*. (This characterization of the final state is reminiscent of the traditional Hoare forward rule for assignment, given for instance in [PAdAG+21a, ch.2].)

It is important to emphasize what this rule achieves. For languages with no control-flow behavior that dominate all other non-sequencing languages on any particular instruction, this rule lets us get a Hoare rule for the combined language at that instruction that talks *only* about the behavior of that one language, freeing us from having to worry about all others. This would be unremarkable (a trivial consequence of the lemmas given in section 4.6.1) except that dominance *does not hold* for arbitrary non-control-flow languages over the *Seq* language (that is, for instance, *Calc* does not dominate *Seq*). This is because *Seq* needs to update the control-flow state after executing each instruction, so it affects the part of the state the non-control-flow language is not aware of, making it impossible for the non-control-flow language to return a result informationally greater than *Seq* without also overriding its *Seq*'s control-flow behavior. The proof requires some reasoning that is specific to *Seq*; it can be found in
`Gazelle/Hoare/Hoare_Step.thy`.

We give the constructions of the control-flow-free languages using *no_control_lifting*, beginning with *Calc*.

5.2.4 Arithmetic Language

We begin with the lifting for for the *Calc* language. The priority calculation for *calc* is as follows:

```
fun calc_prio :: "(Calc.calc ⇒ nat)" where
"calc_prio (Cskip) = 1"
| "calc_prio _ = 2"
```

We return a priority-increment of one for *Cskip* instructions (no-ops, usually corresponding to instructions from another sub-language), and two for *Calc* instructions. We will see a similar pattern for most other sub-languages of IMP: we always increment the priority by at least

one (so that the resulting lifting can satisfy the laws of *lifting_valid*; see section 3.1.2), and some greater increment, usually two, otherwise.

The lifting for *Calc* is as follows:

```
definition calc_schemi where
"calc_schemi = (SP NA (SP NB NC))"
declare calc_schemi_def [simp]
definition calc_schemo where
"calc_schemo =
   (SP NX
   (SP (SPRC calc_prio (SO NC))
   (SP (SPRI (SO NA))
   (SP (SPRI (SO NB)) NX))))"
declare calc_schemo_def [simp]
definition calc_lift ::
   "(Calc.calc, Calc.calc_state,
    ('s, 'x :: {Bogus, Pord, Mergeableb, Okay, Pordps})
     Mem_Simple.state) lifting" where
"calc_lift =
   no_control_lifting (schem_lift calc_schemi calc_schemo)"
```

calc_schemi (the "input schema") simply assigns names *A*, *B*, and *C* to the three components of *Calc*'s state-type for the purposes of the lifting generator (see section 3.3). The "output schema" *calc_schemo* is more interesting. It places *NC* (the third component of the input) into the first component of the output, wrapping it using *option* and *md_prio*, using *calc_prio* for its priority calculation. *NA* and *NB* are placed following *NC* in the output tuple, wrapped in option and prio, using *SPRI* to specify that the priority should always be incremented. We then use *no_control_lifting* to inject this entire lifting into GAZELLE's state-type with control.

We use the following toggle function (for more on toggling, see section 4.6.1) when merging *Calc* with the other components of *Imp*:

```
fun calc_toggle :: "syn ⇒ bool" where
"calc_toggle (Sc _) = True"
| "calc_toggle _ = False"
```

As one would expect, this function toggles *Calc* on for combined-language instructions corresponding to injected *Calc* instructions, and off otherwise. Putting everything together, the lifted *Calc* semantics is given as:

```
definition calc_sem_l ::
  "syn ⇒ ('s, _) state ⇒ ('s, _) state" where
"calc_sem_l =
 lift_map_t_s calc_trans calc_lift calc_toggle
calc_sem"
```

5.2.5 Boolean Language

The Boolean language `Cond` uses the following priority calculation:

```
fun cond_prio :: "Cond.cond ⇒ nat" where
"cond_prio (Sskip_cond) = 1"
| "cond_prio _ = 2"
```

The lifting for `Cond` is as follows:

```
definition cond_lift ::
  "(Cond.cond, Cond.cond_state,
    ('s, 'x :: {Bogus, Pord, Mergeableb, Okay, Pordps})
      Mem_Simple.state) lifting" where
"cond_lift =
  no_control_lifting
    (schem_lift
      (SP NA NB)
      (SP (SPRC cond_prio (SO NA))
                    (SP (SPRI (SO NB)) NX))
  :: (Cond.cond, Cond.cond_state,
      ('x :: {Okay, Bogus, Mergeableb, Pordps})
        state1) lifting)
"
```

`Cond` uses the following toggle-function:

```
fun cond_toggle :: "syn ⇒ bool" where
"cond_toggle (Sb _) = True"
| "cond_toggle _ = False"
```

The lifted `Cond` semantics is given as follows:

5.2.6 Variable-Store Language

For `Mem`, the variable-store language, we have a slightly more complicated priority calculation. This is to ensure we update (at priority 2) only the fields required to perform the

operation, leaving open the possibility of further interactions with other languages. *Mem* has two priority-calculation functions. The first, *mem_prio_mem*, computes the priority-increment for updates to the memory store. The increment is 2 when the instruction being executed is *Swrite*, and is 1 otherwise.

```
fun mem_prio_mem ::
  "syn ⇒ nat" where
"mem_prio_mem (Swrite _ _) = 2"
| "mem_prio_mem _ = 1"
```

The second, *mem_prio_reg*, computes the priority for updates to each of the registers used by *Mem*. For each register, *Reg_x*, (where *x* = *a*, *b*, *c*, or *flag*), the increment is 2 when the instruction being executed is *Swrite Reg_x*, and is 1 otherwise.

```
fun mem_prio_reg ::
  "reg_id ⇒ syn ⇒ nat" where
"mem_prio_reg r (Sread _ r') =
  (if r = r' then 2 else 1)"
| "mem_prio_reg _ _ = 1"
```

Using these priority functions, we define the lifting:

```
definition mem_lift ::
  "(Mem_Simple.syn, Mem_Simple.state0,
   ('s, _ ::{Okay, Bogus, Mergeableb, Pordps}) state)
    lifting" where
"mem_lift = no_control_lifting mem_lift1"
```

That is, we define a lifting *mem_lift1* that updates each component of its state according to the priorities specified by *mem_prio_mem* and *mem_prio_reg*, corresponding to the operation being performed. We then wrap this in *no_control_lifting* in order to lift into a state with control-flow data.

We use the following toggle function for *Mem*:

```
fun mem_toggle :: "syn ⇒ bool" where
"mem_toggle (Sm _) = True"
| "mem_toggle _ = False"
```

Putting all these together, we define the lifted semantics for *Mem*:

```
definition mem_sem_l :: "syn ⇒ ('s, _) state ⇒ ('s, _) state" where
"mem_sem_l = lift_map_t_s mem_trans mem_lift mem_toggle mem0_sem"
```

5.2.7 Sequencing Language

To calculate priorities for *Seq*, we use the following priority function:

```
fun seq_prio :: "syn ⇒ nat" where
"seq_prio _ = 2"
```

Note that this means that regardless of instruction type, *Seq* will attempt to update the control-flow data in the output state at priority 2. This matches the intuition that sequencing behavior needs to be attached to all nodes - after executing any node, we need to proceed on to the next instruction. The exception to this is, of course, the other control-flow constructs from *Imp_Ctl*, which can lead to different control-flow behavior (i.e., something other than proceeding on to the next statement). As we will see shortly (section 5.2.8), *Imp_Ctl* makes use of a higher priority when writing to the control-flow state, in order to ensure that for such instructions it overrides the result of the *Seq* language.

Lifting from our sequencing-language state (which only contains control-flow data) into the full IMP state is given as follows:

```
definition seq_sem_lifting_gen ::
  "(syn, 'x state', ('x, 'a :: Pordb) control) lifting"
  where
"seq_sem_lifting_gen = schem_lift
    NC (SP (SPRC seq_prio (SO NC)) NX)"
```

That is, *seq_sem_l_gen* simply injects into the continuation field of the combined state using the priority increment given by *seq_prio* (i.e., 2), leaving the rest of the state untouched. Because we need *Seq* to be active for all syntax nodes, we do not define a toggle function for *Seq*. This, in turn, means that we cannot reason about the relationship between *Seq* and the other sub-languages using dominance; instead, we make use of *HTS_imp_HT''* (see section 5.2.3), which provides us with an alternative approach for reasoning in the presence of sequencing.

The lifted *Seq* semantics is given as *seq_sem_l* below:

```
definition seq_sem_l_gen ::
  "('s ⇒ syn) ⇒
```

```
    's ⇒ (('x, 'y :: Pordb) control) ⇒
    (('x, 'y :: Pordb) control)" where
"seq_sem_l_gen lfts =
  lift_map_s lfts
  seq_sem_lifting_gen
  seq_sem"

definition seq_sem_l ::
  "syn ⇒
   ('s, _ ::{Okay, Bogus, Mergeableb, Pordps}) state ⇒
   ('s, _) state" where
"seq_sem_l = seq_sem_l_gen seq_trans"
```

5.2.8 IMP-Control Language

For the *Imp_Ctl* language, we calculate priorities as follows:

```
definition imp_prio :: "(syn' ⇒ nat)" where
"imp_prio x =
(case x of
    Sskip ⇒ 1
    | _ ⇒ 3)"
```

As noted in our description of IMP's priority protocol (see section 5.2.2), a priority of 3 will suffice to override the *Seq* language's behavior at nodes corresponding to *Imp_Ctl* control-flow instructions. In this way we handle the splitting of control-flow behavior between these two language components. The lifting for *Imp_Ctl* follows:

```
definition imp_sem_lifting_gen ::
  "(syn', 'x imp_state',
   ('x, _ ) state) lifting" where
"imp_sem_lifting_gen =
 (schem_lift
    (SP NA NB)
    (SP (SPRC imp_prio (SO NA))
        (SP NX (SP (SPRI (SO NB)) NX))))"
```

Specializing the type to the one we are using for *Imp*, we obtain:

```
definition imp_sem_lifting_spec where
"imp_sem_lifting_spec =
  (imp_sem_lifting_gen ::
```

```
    (_, _, (_, (_ ::
      {Okay, Bogus, Mergeableb,
       Pordps, Pordc_all}))
      state) lifting)"
```

We use the toggle function *imp_toggle* to ensure *imp_ctl* is activated only on *Imp_Ctl* instructions:

```
fun imp_toggle :: "syn ⇒ bool" where
"imp_toggle (Si x) = (x ≠ Imp_Ctl.Sskip)"
| "imp_toggle _ = False"
```

Putting everything together, we obtain our lifted semantics for *Imp_Ctl*:

```
definition imp_sem_l ::
  "syn ⇒
   ('s, (_ :: {Okay, Bogus, Mergeableb, Pordps, Pordc_all}))
      state ⇒
   ('s, (_ :: {Okay, Bogus, Mergeableb, Pordps}))
      state" where
"imp_sem_l =
  lift_map_t_s imp_trans imp_sem_lifting_spec
    imp_toggle imp_ctl_sem"
```

5.2.9 IMP Semantics Definition

Having defined each sub-language's lifted semantics, we can compose them using *pcomps* in a straightforward manner:

```
definition sem_final ::
  "syn ⇒
   ('s, (_ :: {Okay, Bogus, Mergeableb, Pordps, Pordc_all}))
    state ⇒
   ('s, (_ :: {Okay, Bogus, Mergeableb, Pordps}))
    state" where
"sem_final =
  pcomps [calc_sem_l, mem_sem_l, cond_sem_l,
          imp_sem_l, seq_sem_l]"
```

Notice that, while there is significant complexity in the definition of IMP from its constituent sub-languages, this complexity is all contained in the definitions of the lifted semantics.

Once we have set these up, composing them becomes a straightforward matter of applying `pcomps`. This is important to the modularity offered by GAZELLE: correctly wrapping the semantics of sub-languages allows composition to be a simple operation that does not require detailed knowledge of the implementation details of the components.

Figure 5.1 (referenced earlier in section 5.2.1) helps to visualize how the components' states fit together. The "fitting together" is precisely what is accomplished by this application of `pcomps` to the lifted semantics functions.

5.3 Hoare Rules for IMP

Thus far (in chapter 4), we have seen several general-purpose Hoare logic rules for reasoning about GAZELLE programs. In order to reason about code in the IMP language we've just defined, it will be necessary to prove several more rules relating to specific IMP instructions - particularly where control flow is concerned.

5.3.1 Control-Flow-Free Instructions

For IMP instructions without control-flow, defining Hoare rules becomes a straightforward matter of stating and proving a single-step rule (i.e., a rule relating only to the language-component providing the instruction in question) and then using `HTS_imp_HT''` (see section 5.2.3) to convert it into a rule about IMP as a whole. If our initial single-step rule is `f % {{P'}} c {{Q'}}`, we need to choose `P` such that `P` on IMP's full state implies `P'` on the language-component's state.

While `HTS_imp_HT''` has several premises, in the case of IMP most of these are easily dispatched. The most interesting one involves showing that the language-component in question is dominant over all language-components in IMP other than `Seq`. This is made simple, however, due to the use of toggling in the definitions of the language components: we just need to show that the language-component defining the semantics of the instruction the rule talks about it "toggled

on" for that instruction, and that the other language components (besides *Seq*) are toggled off. The *sups_pres* condition is also easily handled by virtue of IMP's state being a member of the *Pordc_all* typeclass, allowing us to use the *sups_pres_finite_all* lemma (see section 2.2.4.1).

We define Hoare rules in this way for each *Calc*, *Cond*, and *Mem* instruction. As a representative example, we give the rule for addition below. First, we show the single-step Hoare triple:

```
lemma HCalc_Cadd :
  shows "Calc.calc_sem % {{P1}} (Cadd)
         {{(λ st .
            case st of (c1, c2, x) ⇒ x = c1 + c2 ∧
            (∃ old . P1 (c1, c2, old)))}}"
```

The proof is straightforward: once we evaluate the definitions corresponding to *calc_sem*, the result follows immediately.

```
lemma Add_Final :
  assumes P1_ok : "⋀ st . P st ⟹ st ∈ ok_S"
  assumes HP : "⋀ st . P st ⟹ P' (LOut calc_lift' Cadd st)"
  shows
"|(sem_final ::
    syn ⇒
    (syn, (_ ::{Okay,Mergeableb,Bogus, Pordps, Pordc_all}))
      state ⇒
    (syn, (_ ::{Okay,Bogus,Mergeableb, Pordps, Pordc_all}))
      state)|
{~ (λ st . P st) ~} [G (Sc (Cadd)) z]
{~ (λ st . ∃ old_big small_new . P old_big ∧
  (case small_new of
    (c1, c2, x) ⇒
      x = c1 + c2 ∧ (∃ old . P' (c1, c2, old))) ∧
  st = LUpd calc_lift' (Cadd) small_new old_big)  ~}"
```

To prove this triple, we first unfold *Cadd* into *calc_trans (Sc Cadd)*, and then apply *HTS_imp_HT''* to *HCalc_Cadd*.

For full details on these rules, the reader can consult the formal development (`Gazelle/Languages/Imp/Calc_Mem_Imp_Hoare.thy`).

5.3.2 Sequencing Rule

We give a Hoare rule for the sequencing language *Seq* next. Because the only instruction in *Seq* is the sequencing instruction *Sseq*, a single rule suffices:

```
lemma HxSeq :
  assumes H0 : "gs = pcomps fs "
  assumes HF : "f = seq_sem_l_gen lfts"
  assumes Hpres : "sups_pres (set fs) (λ _ . ok_S)"
  assumes Hnemp : "g ∈ set fs"
  assumes Hdom : "(f ↓ (set fs) {Sseq'})"
  assumes H2 : "lfts Sseq' = Sseq"
  assumes H : "|gs| {~ P1 ~} cs {~ P2 ~}"
  shows "|gs| {~ P1 ~} [G Sseq' cs] {~ P2 ~}"
```

The proof is not terribly complicated and is mostly a matter of unfolding the relevant abstractions and applying general-purpose theorems for manipulating facts about dominance, *safe_for*, and indexed Hoare triples. Notice, however, that the premises of this rule are quite general: unlike the *Add_Final* rule just given, we can use this sequencing rule on any semantics defined as a composition of language-components, provided *Seq* is dominant over the other components on the *Sseq* instruction.

5.3.3 If Rule

For *Imp_Ctl*, we have two Hoare rules. The first is a rule for *If*:

```
lemma HxIf :
  assumes H0 : "gs = pcomps fs"
  assumes HF :
    "f = lift_map_t_s lfts
    (imp_sem_lifting_gen ::
      (_, _,
        (_, (_ :: {Okay, Bogus, Mergeableb, Pordps, Pordc_all}))
          state)
        lifting)
    tg imp_ctl_sem"
  assumes Tg : "tg (Sif') = True"
  assumes Hpres : "sups_pres (set fs) (λ _ . ok_S)"
  assumes Hnemp : "g ∈ set fs"
```

```
assumes Hdom : "(f ↓ (set fs) {Sif'})"
assumes Hsyn : "lfts Sif' = Sif"
assumes P1_valid : "⋀ st.  P1 st ⟹ get_cond st ≠ None"
assumes P2_valid : "⋀ st . P2 st ⟹ get_cond st ≠ None"
assumes P1_oblivious :
  "⋀ p p' x rest .
   P1 (mdp p x, rest) ⟹
   P1 (mdp p' x, rest)"
assumes P2_oblivious :
  "⋀ p p' x rest .
  P2 (mdp p x, rest) ⟹
  P2 (mdp p' x, rest)"


assumes Hcond : "|gs| {~ P1 ~} [cond] {~ P2 ~}"
assumes Htrue :
  "|gs|
    {~ (λ st . P2 st ∧ get_cond st = Some True) ~}
    [body]
    {~ P3 ~}"
assumes Hfalse : "|gs|
  {~ (λ st . P2 st ∧ get_cond st = Some False) ~}
  []
  {~P3~}"
shows "|gs| {~ P1 ~} [G Sif' [cond, body]] {~ P3 ~}"
```

The proof of this rule is in principle quite similar to the proof for the sequencing rule. The main differences are that more definitions need to be added as hints to ISABELLE's simplifier than in the rule for `Seq`, because `Imp_Ctl`'s state is more complex than that of `Seq` (and therefore, so is the lifting it uses).

The interesting part of the proof is as follows: we take a step (evaluating the condition of the `If` statement). At this point, if the condition-flag register is `False`, we are done (the `Htrue` assumption, combined with the fact that `H2` holds after executing `cond`). Otherwise, we step through the remainder of the body of the `If` statement, and eventually apply `Hfalse`.

The structure of this proof is fairly typical for proofs of "If" Hoare rules; the primary interesting difference is that we need to have pre- and post-conditions for the conditional expression `cond`, since in principle `cond` could have side-effects. (In typical formalizations of IMP - for

instance, in [PAdAG$^+$21b, ch.12] - this problem is avoided by specifying the behavior of boolean operators by means of a side-effect-free expression language.) We also require that the predicates *P1* and *P2* do not look at the priority field of the condition (*P1_oblivious* and *P2_oblivious*) and only hold when the condition-flag register actually contains a value (*P1_valid* and *P2_valid*). For reasonable choices of predicates this is quite easy to show, although it is an annoyance.

5.3.4 While Rule

The other rule for *Imp_Ctl* corresponds to the loop construct *SwhileC*:

```
lemma HxWhileC :
  assumes H0 : "gs = pcomps fs"
  assumes HF :
    "f = lift_map_t_s lfts
    (imp_sem_lifting_gen :: (_, _,
      (_, (_ :: {Okay, Bogus, Mergeableb, Pordps, Pordc_all}))
        state)
      lifting)
    tg imp_ctl_sem"
  assumes Tg : "tg (SwhileC') = True"
  assumes Hpres : "sups_pres (set fs) (λ _ . ok_S)"
  assumes Hnemp : "g ∈ set fs"
  assumes Hdom : "(f ↓ (set fs) {SwhileC'})"
  assumes Hsyn : "lfts SwhileC' = SwhileC"
  assumes PX_valid : "⋀ st.  PX st ⟹ get_cond st ≠ None"
  assumes PX_oblivious :
    "⋀ p p' x rest .
      PX (mdp p x, rest) ⟹
      PX (mdp p' x, rest)"
  assumes Htrue :
    "|gs|
     {~ (λ st . (PX st ∧ get_cond st = Some True))~}
     [body]
      {~PX~}"
  shows "|gs|
         {~PX~}
         [G SwhileC' [body]]
         {~ (λ st . PX st ∧ get_cond st = Some False)~}"
```

Recall that with *SwhileC*, we do not evaluate a conditional expression explicitly, instead assuming that this has been done in the course of evaluating the body of the loop. In order to prove this rule, we first need to prove a lower-level rule that makes explicit use of step-counts (recall that our reason for introducing step-counts in section 4.4 was to be able to prove the soundness of this rule):

```
lemma HxWhileC' :
  assumes H0 : "gs = pcomps fs"
  assumes HF :
    "f = lift_map_t_s lfts
      (imp_sem_lifting_gen :: (_, _,
        (_, (_ :: {Okay, Bogus, Mergeableb, Pordps, Pordc_all}))
          state)
        lifting)
 tg imp_ctl_sem"
  assumes Tg : "tg (SwhileC') = True"
  assumes Hpres : "sups_pres (set fs) (λ _ . ok_S)"
  assumes Hnemp : "g ∈ set fs"
  assumes Hdom : "(f ↓ (set fs) {SwhileC'})"
  assumes Hsyn : "lfts SwhileC' = SwhileC"
  assumes PX_valid : "⋀ st.  PX st ⟹ get_cond st ≠ None"
  assumes PX_oblivious :
    "⋀ p p' x rest .
      PX (mdp p x, rest) ⟹
      PX (mdp p' x, rest)"
  assumes Htrue :
    "⋀ nb2 . ∃ nb1' .
      |#gs#|
      {#- (λ st. PX st ∧ get_cond st = Some True),
          (nb1' + nb2) -#}
      [body]
      {#- PX, nb2 -#}"
  assumes NLs : "nl1 ≤ nl2"
  shows "|#gs#|
         {#- PX, nl1  -#}
         [G SwhileC' [body]]
         {#- (λ st . PX st ∧ get_cond st = Some False), nl2 -#}"
```

Once we have *HxWhileC'*, proving *HxWhileC* is simple: we unfold the definition of the Hoare triple |*gs*| {~*P*~} *c* {~*Q*~} to get an explicit (arbitrary) step-count, then apply *HxWhileC'* using this step-count to show the rule holds at that count.

The proof of *HxWhileC'* goes roughly as follows: we proceed by induction on the number of steps for which we want to show the entire program (◊*SwhileC' [body]*) is safe. If the step-count is 0, our result holds trivially. Otherwise, we take a step (which will place the body of the loop onto the top of the continuation list), then run the body. We then check the value of the flag register. If it is false, we have exited the loop and are done. Otherwise, we can apply our inductive hypothesis (since we took at least one step, having taken one when pushing the loop body onto the continuation stack). With a bit of massaging, we can use this to prove our goal.

From a user's point of view *HxWhileC* works similarly to the rule *HxIf*, subject to the same caveats about *PX_valid* and *PX_oblivious*. Of course, a loop invariant (*PX* in the theorem statements above) must be provided.

5.4 IMP Example: Multiplication as Repeated Addition

To demonstrate the usefulness of this chapter's formalization of IMP, we conclude with the definition, specification, and verification of an example program. While small, this program showcases the main challenge of verification of imperative programs: namely, describing *loop invariants* for reasoning about iterative code, and using these invariants, along with the Hoare rules for the language, to complete proofs for loop bodies.

5.4.1 Multiplication Program

Our choice for a program that demonstrates these features of GAZELLE is a program that implements multiplication by means of repeated addition. We begin by placing one number to be multiplied in memory at *''arg1''*, and the other at *''arg2''*. By the end of the execution, memory location *''acc''* will store the result of the multiplication. Here is the code:

```
definition prog1 :: "int ⇒ int ⇒ syn gensyn" where
"prog1 i1 i2 =
  ◊ (Ss Sseq)
```

```
[ † Sc (Cnum i1)
, † Sm (Swrite (STR ''arg1'') (Reg_c))
, † Sc (Cnum i2)
, † Sm (Swrite (STR ''arg2'') (Reg_c))
, † Sc (Cnum 1)
, † Sm (Swrite (STR ''one'') (Reg_c))
, † Sc (Cnum 0)
, † Sm (Swrite (STR ''acc'') (Reg_c))

, † Sm (Sread (STR ''arg2'') (Reg_c))
, † Sb Sgtz

, ◊ (Si SwhileC)
  [ ◊ (Ss Sseq)
    [ † Sm (Sread (STR ''arg1'') (Reg_a))
    , † Sm (Sread (STR ''acc'') (Reg_b))
    , † Sc Cadd
    , † Sm (Swrite (STR ''acc'') (Reg_c))
    , † Sm (Sread (STR ''arg2'') (Reg_a))
    , † Sm (Sread (STR ''one'') (Reg_b))
    , † Sc Csub
    , † Sm (Swrite (STR ''arg2'') (Reg_c))
    , † Sm (Sread (STR ''arg2'') (Reg_c))
    , † Sb Sgtz
    ]
  ]
]"
```

Note that while in this example the program itself is parameterized over the inputs to be multiplied (i.e., `prog1` really corresponds to a procedure for generating code that will calculate the result for some specific pair of inputs), we could just as easily have assumed the inputs would already be present in, for instance, memory locations `''arg1''` and `''arg2''` in the initial state.

5.4.2 Multiplication Specification

The specification for `prog1` is what one would expect: if our inputs are positive integers (we restrict ourselves to this case for simplicity), we will find the result of multiplying these values in memory at `''acc''` after the program runs. (There is a bit of unfortunate syntactic noise

induced by having to manually unwrap some of GAZELLE's wrapper-types in the conclusion).

```
lemma prog1_spec :
  assumes Hi1 : "0 < i1"
  assumes Hi2 : "0 ≤ i2"
  shows
    "|(sem_final ::
        (syn ⇒
         (syn, ('x :: {Okay, Bogus, Mergeableb,
                       Pordps, Pordc_all}))
            state ⇒
         (syn, (_ :: {Okay, Bogus, Mergeableb,
                      Pordps})) state))|
     {~ (λ st . st ∈ ok_S) ~}
     [prog1 i1 i2]
     {~ (λ st . st ∈ ok_S ∧
        (case st of
          (reg_flag, reg_c, reg_a, reg_b, mem, xz) ⇒
            (case mem of
              (mdp p (Some (mdt mem'))) ⇒
                get mem'(STR ''acc'') = Some (i1 * i2)
              | _ ⇒ False)))
    ~}"
```

5.4.3 Multiplication Proof

The proof that the multiplication program meets its specification is unfortunately cluttered by tactic-style proofs, due to the fact that GAZELLE's automation is not as optimized as it could be, and so some "hand-holding" is required to get ISABELLE to understand the correctness of some of the proof-steps involved (particularly steps involving application of the rule of consequence). However, its structure quite closely matches the idealized version we give here.

As is typical in Hoare-logic reasoning about programs (e.g. in [PAdAG+21a, ch.2]), we can give this idealized proof by *annotating* our program, showing what predicates hold before and after each statement, and what Hoare rules are used to connect one predicate to the next. We place these in comments inside the code, which looks like:

— this is a comment

We elide the fact that the state is Okay (in the sense of being in `ok_S`) at all times, in order

to reduce noise and make the structure of the proof clearer. This property (membership in *ok_S*) is assumed to hold true on the program's initial state, and is shown to hold true at each point thereafter.

```
definition prog1_annotated :: "int ⇒ int ⇒ syn gensyn" where
"prog1_annotated i1 i2 =
  — True
  ◊ (Ss Sseq)
  [ — True
    † Sc (Cnum i1)
    — reg c = i1
  , † Sm (Swrite (STR ''arg1'') (Reg_c))
    — reg c = i1; mem["arg1"] = i1
  , † Sc (Cnum i2)
    — reg c = i2; mem["arg1"] = i1
  , † Sm (Swrite (STR ''arg2'') (Reg_c))
    — reg c = i2; mem["arg1"] = i1; mem["arg2"] = i2
  , † Sc (Cnum 1)
    — reg c = 1; mem["arg1"] = i1; mem["arg2"] = i2
  , † Sm (Swrite (STR ''one'') (Reg_c))
    — reg c = 1; mem["arg1"] = i1; mem["arg2"] = i2; mem["one"] = 1
  , † Sc (Cnum 0)
    — reg c = 0; mem["arg1"] = i1; mem["arg2"] = i2; mem["one"] = 1
  , † Sm (Swrite (STR ''acc'') (Reg_c))
    — reg c = 0; mem["arg1"] = i1; mem["arg2"] = i2; mem["one"] = 1; mem["acc"] = 0
  , † Sm (Sread (STR ''arg2'') (Reg_c))
    — reg c = i2; mem["arg1"] = i1; mem["arg2"] = i2; mem["one"] = 1; mem["acc"] = 0
  , † Sb Sgtz
    — reg c = i2; reg flag = (i2 > 0) mem["arg1"] = i1;
    — mem["arg2"] = i2; mem["one"] = 1; mem["acc"] = 0

    — LOOP INVARIANT I: exists idx such that:
     — reg flag = 1 iff idx > 0;
     — mem["arg1"] = i1; mem["arg2"] = idx; mem["one"] = 1;
     — mem["acc"] = i1 * (i2 - idx);
     — i2 >= idx;
     —
     — holds initially for idx = i2
  , ◊ (Si SwhileC)
    [ — I; reg flag = 1
      ◊ (Ss Sseq)
      [ — I; reg flag = 1.
        — We can restate this as:
        — mem["arg1"] = i1; mem["arg2"] = idx; mem["one"] = 1;
        — mem["acc"] = i1 * (i2 - idx);
```

— i2 >= idx; idx > 0
† *Sm (Sread (STR ''arg1'') (Reg_a))*
— reg a = i1;
— mem["arg1"] = i1; mem["arg2"] = idx; mem["one"] = 1;
— mem["acc"] = i1 * (i2 - idx);
— i2 >= idx; idx > 0
, † *Sm (Sread (STR ''acc'') (Reg_b))*
— reg a = i1; reg b = i1 * (i2 - idx);
— mem["arg1"] = i1; mem["arg2"] = idx; mem["one"] = 1;
— mem["acc"] = i1 * (i2 - idx);
— i2 >= idx; idx > 0
, † *Sc Cadd*
— reg a = i1; reg b = i1 * (i2 - idx); reg c = i1 * (i2 - idx) + i1;
— mem["arg1"] = i1; mem["arg2"] = idx; mem["one"] = 1;
— mem["acc"] = i1 * (i2 - idx);
— i2 >= idx; idx > 0
, † *Sm (Swrite (STR ''acc'') (Reg_c))*
— reg a = i1; reg b = i1 * (i2 - idx); reg c = i1 * (i2 - idx) + i1;
— mem["arg1"] = i1; mem["arg2"] = idx; mem["one"] = 1;
— mem["acc"] = i1 * (i2 - idx) + i1;
— i2 >= idx; idx > 0
, † *Sm (Sread (STR ''arg2'') (Reg_a))*
— reg a = idx; reg b = i1 * (i2 - idx); reg c = i1 * (i2 - idx) + i1;
— mem["arg1"] = i1; mem["arg2"] = idx; mem["one"] = 1;
— mem["acc"] = i1 * (i2 - idx) + i1;
— i2 >= idx; idx > 0
, † *Sm (Sread (STR ''one'') (Reg_b))*
— reg a = idx; reg b = 1; reg c = reg c = i1 * (i2 - idx) + i1;
— mem["arg1"] = i1; mem["arg2"] = idx; mem["one"] = 1;
— mem["acc"] = i1 * (i2 - idx) + i1;
— i2 >= idx; idx > 0
, † *Sc Csub*
— reg a = idx; reg b = 1; reg c = idx - 1;
— mem["arg1"] = i1; mem["arg2"] = idx; mem["one"] = 1;
— mem["acc"] = i1 * (i2 - idx) + i1;
— i2 >= idx; idx > 0
, † *Sm (Swrite (STR ''arg2'') (Reg_c))*
— reg a = idx; reg b = 1; reg c = idx - 1;
— mem["arg1"] = i1; mem["arg2"] = idx - 1; mem["one"] = 1;
— mem["acc"] = i1 * (i2 - idx) + i1;
— i2 >= idx; idx > 0
, † *Sm (Sread (STR ''arg2'') (Reg_c))*
— reg a = idx; reg b = 1; reg c = idx - 1;
— mem["arg1"] = i1; mem["arg2"] = idx - 1; mem["one"] = 1;
— mem["acc"] = i1 * (i2 - idx) + i1;
— i2 >= idx; idx > 0

```
      , † Sb Sgtz
        — reg a = idx; reg b = 1; reg c = idx - 1; reg flag = 1 iff idx' > 0
        — mem["arg1"] = i1; mem["arg2"] = idx - 1; mem["one"] = 1;
        — mem["acc"] = i1 * (i2 - idx) + i1;
        — i2 >= idx
        — let idx' = idx - 1. Then:
        — reg flag = 1 iff idx' > 0;
        — mem["arg1"] = i1; mem["arg2"] = idx'; mem["one"] = 1;
        — mem["acc"] = i1 * (i2 - (idx' + 1)) + i1 = i1 * (i2 - idx');
        — i2 >= idx'
        — Therefore invariant I is reestablished for idx'
      ]
    ]
    — I; reg flag = 0
  ]
  — I; reg flag = 0 ″
```

For the full details of the proof, the reader can refer to the formal development. While the actual proof is messier than this idealized version - owing mostly to the fact that GAZELLE's automation for handling the implications generated by applications of the Hoare rule of consequence could be improved - the basic structure is identical to that presented here.

Chapter 6

Conclusion

6.1 Summary

In this dissertation, we have described in detail the implementation and use of GAZELLE, a system for constructing languages by composition from smaller parts and for formally reasoning about such composed languages.

In chapter 1, we discussed the problem GAZELLE solves: making it easier to build formal semantics for programming languages out of reusable components. We also discussed why that problem matters: that lack of reuse contributes to the large amount of human effort that needs to go into formalizing and reasoning about programming languages. Through the lens of the *expression problem*, we talked about other approaches to this problem and related problems, and how GAZELLE occupies a distinct niche in a rich design space.

In chapter 2, we introduced the GAZELLE approach in practice (section 2.1), demonstrating how GAZELLE can be used to construct a simple language out of several reusable components, and contrasted the GAZELLE solution with a more ad-hoc, manual approach. We then discussed the implementation of two key primitives: `pcomps`, for merging results of a single execution step (figure 2.2) and `sem_run`, for expressing multi-step executions (figure 2.7). These allow

GAZELLE to assign a precise meaning to language composition. In section 2.2 we discussed the partial-order-based formalism that powers the definition of `pcomps`, and in section 2.3 we discussed the generic, control-flow based interpreter that defines `sem_run`.

Next, in chapter 3, we discussed GAZELLE's lifting subsystem, for adapting existing code to work with the partial orderings defined in section 2.2. In chapter 4, we then discussed how GAZELLE enables building program logics for languages defined by composition, enabling reuse of reasoning principles defined on the constituent sub-languages when reasoning about composite languages. Finally, in chapter 5, we brought everything together with an extended example of defining and reasoning about a simple but realistic programming language by composition, using the GAZELLE framework.

At this point, we believe we've made the case that GAZELLE represents an interesting and distinct approach to programming language semantics. It is our hope that this document helps readers not only understand GAZELLE, but also inspires them to consider the use of GAZELLE or a similar approach when building formalizations. We'll end this dissertation with some ideas about future directions for further work on GAZELLE.

6.2 Ideas for Future Work

The GAZELLE project achieved its initial aim: being able to express and enable reasoning about imperative programs in IMP using a compositional approach. However, GAZELLE is far from a finished project. Over the course of building GAZELLE many potential avenues for improvement (both from a research and an engineering standpoint) have come to light. Here we give an assortment of ideas about future directions for improving and building on GAZELLE. The list is by no means complete - there are a number of interesting research directions still to be pursued related to the GAZELLE system.

6.2.1 Evaluating GAZELLE on More Case Studies

While we believe that the IMP case study carried out as described in chapter 5 shows the practicality and flexibility of GAZELLE, much could be gained from applying GAZELLE to other language formalizations - particularly in terms of the reusability of components such as `Seq` between languages with very different semantics. Previous versions of GAZELLE were able to support implementation of a lambda-calculus (that is, a functional language rather than an imperative one) evaluator that was able to reuse control-flow primitives from IMP. The current version of GAZELLE should also be able to support this, and carrying out this evaluation would no doubt lead to interesting insights. Modeling other programming paradigms using GAZELLE (such as object-oriented programming) would similarly be likely to yield useful results that could help improve GAZELLE.

6.2.2 Library of Language Components

After experimenting with applying GAZELLE to other languages, it should be possible to distill the insights gained in this way to build a "standard library" of reusable components (language semantics and proof rules) designed for use with GAZELLE, with an eye toward general reusability across different language-modeling tasks. This could lower the barrier to entry for using GAZELLE to formalize a new language, and further increase user productivity.

6.2.3 Porting GAZELLE to Other Proof Assistants

While GAZELLE as a framework is implemented in ISABELLE, the approach would likely be useful in other proof assistants as well. Porting GAZELLE to COQ could be an interesting project; COQ has a richer typesystem and also more powerful typeclass inference. However, the fact that COQ requires constructive proofs might make the lifting of proof rules more challenging. Adapting GAZELLE to work with COQ would not only increase the potential audience for

GAZELLE, it might also shed light on ISABELLE-specific assumptions that might be built into the way GAZELLE is designed.

6.2.4 Fully Leveraging `Bsup`

We gave a definition for the `bsup` operator in section 2.2.2.3, but the careful reader may have noticed that in the development described here, we only use the fact that `bsup x y` computes the supremum of `x` and `y` when that supremum exists. Initially, we had hoped to support use of `bsup` (and, hence, `pcomps`) to reason about situations where no supremum exists between state elements being merged, but we had limited success doing so. Nonetheless, even when no supremum is present, `bsup` still has some useful properties, including limited forms of associativity laws. Understanding of whether `bsup` is structured enough to be useful without the presence of a supremum could be an interesting direction. If, in fact, it is possible (at least in some cases) to reason conveniently about `bsup` without assuming a supremum, this could have significant implications for the choices of partial orders used in formalizing combined-language states. For instance, the `md_prio` wrapper type is very frequently used to force the existence of a supremum, but has some undesirable qualities (see e.g. the discussion of dominance and toggling in section 4.6.1). Being able to remove some of the need for `md_prio` could be of great use.

6.2.5 Reconciling Categorical Approach to Lenses with Gazelle Liftings

While GAZELLE's lifting system (described in chapter 3) draws inspiration heavily from category-theoretic notions of lenses and other optics (as developed in e.g. [PGW17]), the precise relationship between GAZELLE's liftings and optics (as well as other related categorical constructs) is not clear. GAZELLE's definition of lifting as an abstraction (including the laws for reasoning about them) is driven primarily by practicality: we sought out an abstraction that was suitable for our purposes; namely, adapting existing semantics functions to work over ordered datatypes.

Having an understanding of the lifting abstraction from a more principled angle could help reveal other, related abstractions that might also be useful in GAZELLE, and could even lead to a simpler or cleaner definition for lifting itself.

Bibliography

[AB07] Andrew W. Appel and Sandrine Blazy. Separation Logic for Small-Step cminor. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics*, volume 4732, pages 5–21. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISSN: 0302-9743, 1611-3349 Series Title: Lecture Notes in Computer Science.

[ADH+14] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, Cambridge, 2014.

[Ahm15] Amal Ahmed. Verified Compilers for a Multi-Language World. page 17 pages, 2015. Artwork Size: 17 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany.

[Ahm21] Amal Ahmed. Semantic Soundness for Language Interoperability: Invited Presentation at the Combined 28th International Workshop on Expressiveness in Concurrency and 18th Workshop on Structural Operational Semantics. *Electronic Proceedings in Theoretical Computer Science*, 339:1–1, August 2021.

[Bal] Clemens Ballarin. Tutorial to Locales and Locale Interpretation. page 20.

[Bre17] Joachim Breitner. Isabelle functions: Always total, sometimes undefined – Blog – Joachim Breitner's Homepage, October 2017.

[Dan08] Olivier Danvy. Defunctionalized Interpreters for Programming Languages. *International Conference on Functional Programming*, 43(9):12, September 2008.

[DdSOS13] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '13*, page 207, Rome, Italy, 2013. ACM Press.

[Dev06] The Apache Velocity Developers. Velocity Users' Guide, 2006.

[FGM+05] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. Combinators for Bi-Directional Tree Transformations. page 14. ACM, January 2005.

[GKR+15] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 595–608, Mumbai India, January 2015. ACM.

[Haf21] Florian Haftmann. Haskell-style type classes with Isabelle/Isar. page 24, 2021.

[HB16] Graham Hutton and Patrick Bahr. Cutting Out Continuations. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World*, volume 9600, pages 187–200. Springer International Publishing, Cham, 2016. Series Title: Lecture Notes in Computer Science.

[KMNO14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191, San Diego California USA, January 2014. ACM.

[LBK+16] Xavier Leroy, Sandrine Blazy, Daniel Kastner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert – A Formally Verified Optimizing Compiler. page 8, 2016.

[NJ17] Shayan Najd and Simon Peyton Jones. Trees that Grow. *Journal of Universal Computer Science,*, 23(1):42–62, January 2017. Publisher: Verlag der Technischen Universität Graz.

[NPW21] Tobias Nipkow, Lawrence C. Paulson, and Makarius Wenzel. A Proof Assistant for Higher-Order Logic, December 2021.

[PAdAG+21a] Benjamin C. Pierce, Arthur Azevedo de Amorim, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. Software Foundations Volume 2: Programming Language Foundations, August 2021.

[PAdAG+21b] Benjamin C. Pierce, Arthur Azevedo de Amorim, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software Foundations Volume 1: Logical Foundations, August 2021.

[PGW17] Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. Profunctor Optics: Modular Data Accessors. *The Art, Science, and Engineering of Programming*, 1(2):7, April 2017.

[RYLG18] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 115–129, Los Angeles CA USA, January 2018. ACM.

[RYLG19] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for Proof Reuse in Coq. page 19 pages, 2019. Artwork Size: 19 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany Version Number: 1.0.

[Sch88] David Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown, December 1988.

[Swi08] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(04), July 2008.

[Tea22] The Coq Development Team. The Coq Reference Manual, March 2022.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89*, pages 347–359, Imperial College, London, United Kingdom, 1989. ACM Press.

[Wad98] Philip Wadler. The Expression Problem, November 1998.

[Wen] Makarius Wenzel. The Isabelle/Isar Reference Manual. page 356.

[ZO05] Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *In Proc. FOOL 12*, 2005.