

THE UNIVERSITY OF NEW SOUTH WALES



SYDNEY • AUSTRALIA

School of Computer Science and Engineering
Faculty of Engineering

Type Systems for Systems Types

A thesis submitted for the degree of
DOCTOR OF PHILOSOPHY

LIAM O'CONNOR-DAVIS

June 2019



Australia's
Global
University

Thesis/Dissertation Sheet

Surname/Family Name	: O'Connor-Davis
Given Name/s	: Liam John
Abbreviation for degree as give in the University calendar	: PhD
Faculty	: Faculty of Engineering
School	: School of Computer Science and Engineering
Thesis Title	: Type Systems for Systems Types

Abstract 350 words maximum: (PLEASE TYPE)

This thesis presents a framework aimed at significantly reducing the cost of proving functional correctness for low-level operating systems components, designed around a new programming language, Cogent. This language is total, polymorphic, higher- order, and purely functional, including features such as algebraic data types and type inference. Crucially, Cogent is equipped with a uniqueness type system, which eliminates the need for a trusted runtime or garbage collector, and allows us to assign two semantics to the language: one imperative, suitable for efficient C code generation; and one functional, suitable for equational reasoning and verification. We prove that the functional semantics is a valid abstraction of the imperative semantics for all well-typed programs. Cogent is designed to easily interoperate with existing C code, to enable Cogent software to interact with existing C systems, and also to provide an escape hatch of sorts, for when the restrictions of Cogent's type system are too onerous. This interoperability extends to Cogent's verification framework, which composes with existing C verification frameworks to enable whole systems to be verified.

Cogent's verification framework is based on certifying compilation: For a well- typed Cogent program, the compiler produces C code, a high-level representation of its semantics in Isabelle/HOL, and a proof that the C code correctly refines this embedding. Thus one can reason about the full semantics of real-world systems code productively and equationally, while retaining the interoperability and leanness of C. The compiler certificate is a series of language-level proofs and per-program translation validation phases, combined into one coherent top-level theorem in Isabelle/HOL.

To evaluate the effectiveness of this framework, two realistic file systems were implemented as a case study, and key operations for one file system were formally verified on top of Cogent specifications. These studies demonstrate that verification effort is drastically reduced for proving higher-level properties of file system implementations, by reasoning about the generated formal specification from Cogent, rather than low-level C code.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all property rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstracts International (this is applicable to doctoral theses only).

.....

Signature

Witness Signature

Date

The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

FOR OFFICE USE ONLY Date of completion of requirements for Award:

ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed

Date

INCLUSION OF PUBLICATIONS STATEMENT

UNSW is supportive of candidates publishing their research results during their candidature as detailed in the UNSW Thesis Examination Procedure.

Publications can be used in their thesis in lieu of a Chapter if:

- The student contributed greater than 50% of the content in the publication and is the “primary author”, ie. the student was responsible primarily for the planning, execution and preparation of the work for publication
- The student has approval to include the publication in their thesis in lieu of a Chapter from their supervisor and Postgraduate Coordinator.
- The publication is not subject to any obligations or contractual agreements with a third party that would constrain its inclusion in the thesis

Please indicate whether this thesis contains published material or not.

This thesis contains no publications, either published or submitted for publication

Some of the work described in this thesis has been published and it has been documented in the relevant Chapters with acknowledgement

This thesis has publications (either published or submitted for publication) incorporated into it in lieu of a chapter and the details are presented below

CANDIDATE'S DECLARATION

I declare that:

- I have complied with the Thesis Examination Procedure
- where I have used a publication in lieu of a Chapter, the listed publication(s) below meet(s) the requirements to be included in the thesis.

Name

Signature

Date (dd/mm/yy)

COPYRIGHT STATEMENT

'I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International (this is applicable to doctoral theses only).

I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.'

Signed

Date

AUTHENTICITY STATEMENT

'I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.'

Signed

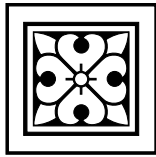
Date

ACKNOWLEDGEMENTS

I would first like to thank Gabriele Keller, my supervisor, colleague, and friend, for her invaluable support and guidance not just during this doctoral undertaking, but for the majority of my academic life throughout the last decade. I'm also immensely grateful to Kai Engelhardt, whose courses I had the pleasure both of taking and teaching, and without whom I certainly would not have developed the skills necessary to pull this off.

Thanks are also due to my co-supervisors (at various times) Gernot Heiser, Toby Murray and Manuel Chakravarty, as well as Ben Lippmeier and all the other regulars at the Programming Languages and Systems research group for their sometimes invaluable advice and mentorship.

Several others in our group worked extremely hard on this project, and without their contributions this thesis would never have happened. In particular I would like to thank Zilin Chen for his tireless efforts bringing our compiler up to scratch and on polishing the user experience for our language. I would also like to thank Christine Rizkallah, Thomas Sewell, Japheth Lim and anyone else my memory neglects that worked on the C refinement framework, as well as Sidney Amani, Alex Hixon, Peter Chubb and Partha Susarla who worked on the systems side of things. Gerwin Klein also deserves thanks for the considerable Isabelle and verification expertise he lent to the project. I leaned considerably on the support of my friends and relatives, without whom I would no doubt have failed to complete this degree. I offer my heartfelt thanks to my parents Michelle and Peter, my friends Annie Liu, Christina Jeong (정지영), Edward Lee, Joey Tuong, Takashi Matsuoka (松岡 崇), Tran Ma (馬玉珍), Vivian Dang (鄧懿婷), and many others I relied upon from time to time.

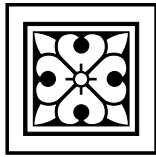


ABSTRACT

This thesis presents a framework aimed at significantly reducing the cost of proving functional correctness for low-level operating systems components, designed around a new programming language, Cogent. This language is total, polymorphic, higher-order, and purely functional, including features such as algebraic data types and type inference. Crucially, Cogent is equipped with a uniqueness type system, which eliminates the need for a trusted runtime or garbage collector, and allows us to assign two semantics to the language: one imperative, suitable for efficient C code generation; and one functional, suitable for equational reasoning and verification. We prove that the functional semantics is a valid abstraction of the imperative semantics for all well-typed programs. Cogent is designed to easily interoperate with existing C code, to enable Cogent software to interact with existing C systems, and also to provide an escape hatch of sorts, for when the restrictions of Cogent's type system are too onerous. This interoperability extends to Cogent's verification framework, which composes with existing C verification frameworks to enable whole systems to be verified.

Cogent's verification framework is based on certifying compilation: For a well-typed Cogent program, the compiler produces C code, a high-level representation of its semantics in Isabelle/HOL, and a proof that the C code correctly refines this embedding. Thus one can reason about the full semantics of real-world systems code productively and equationally, while retaining the interoperability and leanness of C. The compiler certificate is a series of language-level proofs and per-program translation validation phases, combined into one coherent top-level theorem in Isabelle/HOL.

To evaluate the effectiveness of this framework, two realistic file systems were implemented as a case study, and key operations for one file system were formally verified on top of Cogent specifications. These studies demonstrate that verification effort is drastically reduced for proving higher-level properties of file system implementations, by reasoning about the generated formal specification from Cogent, rather than low-level C code.



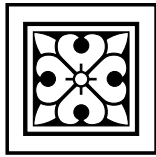
NOTES ON TYPOGRAPHY

Even before the completion of this thesis, I have been asked for information about how I typeset it. The basics are as follows: I am using \LaTeX with a custom version of the `unswthesis` class. The text font is Concrete Roman, and the mathematics font is Euler. The only boldface font used is for diagrams and mathematics (for language keywords etc.) and it is the boldface sans-serif Computer Modern. Initials are provided by the `lettrine` package and ornaments provided by `pgfornament` with the `vectorian` set. Quotes are typeset with `epigraph`.

All diagrams are typeset with `tikz`. For Cogent code snippets in Chapter 2, I have a customised fork of the Cogent parser which then “pretty” prints the Cogent code as a series of \TeX commands in the `algorithm2e` package. I have a custom environment that shells out to this parser for Cogent snippets.

For all mathematical figures in Chapters 3 and 4, I have a Agda library to generate \TeX code, which uses mix-fix syntax to allow me to typeset inference rules and grammars using Agda syntax. The Agda terms are also given phantom types to enable me to “type-check” my definitions before exporting to \TeX . The rules themselves are typeset with the `mathpartir` package.

I have occasionally opted to present theorems dually, in formal notation on the right, and in English text on the left. The text on the left corresponds line by line to the formalism on the right. I hope this makes the theorems easier to grasp, but I’m not certain that it does.



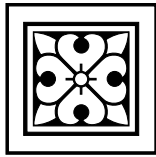
CONTENTS

1	Introduction	1
1.1	Automating Refinement-based Verification	2
1.1.1	The seL4 Microkernel	2
1.1.2	AutoCorres	4
1.1.3	CakeML	5
1.2	Pure Functions, Efficiently	5
1.2.1	Type Systems for Memory Management	7
1.2.2	Linear and Uniqueness Types	7
1.3	A New Language	9
1.3.1	Cogent	11
1.4	Summary of Contributions	12
2	Programming in Cogent	15
2.1	A Cogent Tutorial	16
2.1.1	Variant Types	17
2.1.2	Abstract Types and Functions	19
2.1.3	Suspending Uniqueness	20
2.1.4	Higher-order Functions	21
2.1.5	Polymorphism	22
2.1.6	Records	24
2.1.7	Combined Example	26
2.2	Systems Programming in Cogent	26
2.2.1	Experience with Cogent	28
2.2.2	Performance	30
3	Static Semantics	35
3.1	Constraint-Based Type Inference	36
3.2	Substructural Type Systems	37

3.3	Elements of Constraint Generation	41
3.4	Elaboration	44
3.5	Variant Types	44
3.5.1	Adding Subtyping	44
3.5.2	Type Inference	47
3.6	Abstract and Observer Types	49
3.6.1	Observation and Escape Analysis	50
3.7	Record Types	53
3.7.1	Type Inference	56
3.8	Constraint Generation Theorems	56
3.8.1	Soundness	58
3.8.2	Completeness	59
3.8.3	Totality	59
3.9	Polymorphism	60
3.10	Constraint Solver	61
3.10.1	The Simplify Phase	62
3.10.2	The Unify Phase	63
3.10.3	The Join/Meet Phase	67
3.10.4	The Equate Phase	71
3.10.5	The Solver Overall	73
3.11	Type Inference Results	74
4	Dynamic Semantics	77
4.1	A Tale of Two Semantics	78
4.1.1	Value Semantics	79
4.1.2	Update Semantics	83
4.2	Refinement and Type Preservation	84
4.2.1	Framing	88
4.2.2	Proving Refinement	90
4.2.3	Foreign Functions	91
5	Refinement Framework	93
5.1	Refinement and Forward Simulation	96
5.2	Well-typedness Proof	96
5.3	Refinement Phases	97
5.3.1	SIMPL and AutoCorres	98

5.3.2	AutoCorres and Cogent	100
5.3.3	Monomorphisation	107
5.3.4	A Normal and Deep Embeddings	108
5.3.5	Desugared and Neat Embeddings	109
5.3.6	Combined Predicate for Full Refinement	110
5.4	Connecting to Abstract Specifications	111
5.5	Evaluation	112
6	Conclusions and Future Work	115
6.1	Optimisations	116
6.2	Binary Verification	117
6.3	Data Description Language	118
6.4	Richer Type System	123
6.5	Recursion and Non-termination	124
6.6	Concurrency	125
6.7	Testing Frameworks	126





LIST OF FIGURES

1.1	The refinement hierarchy in the seL4 verification	3
1.2	A simplified view of the Cogent refinement framework.	11
2.1	A full example of a Cogent program.	27
2.2	IOZone throughput for 4KiB writes	30
2.3	IOZone throughput for random 4KiB writes to a RAM disk	30
3.1	Syntax of the basic fragment of Cogent	38
3.2	Context relations	40
3.3	Some non-algorithmic typing rules	40
3.4	Some elementary constraint generation rules	42
3.5	Algorithmic context join	43
3.6	Constraint semantics	43
3.7	Syntax for variants	45
3.8	Typing Rules for variants	46
3.9	Constraint semantics for variants	47
3.10	Constraint generation for variants	48
3.11	Syntax for abstract and observer types	49
3.12	Constraint semantics for abstract and observer types	51
3.13	Typing and constraint generation rules for let!	52
3.14	Syntax for records	53
3.15	Typing rules for records	54
3.16	Constraint semantics for records	55
3.17	Constraint generation rules for records	57
3.18	Solver data flow	61
3.19	Basic simplification rules of constraint solving.	64
3.20	Simplification rules for equality and subtyping.	65
3.21	Example of the interaction of the join/meet and equate phases.	68

3.22	Meet rules of constraint solving.	69
3.23	Join rules of constraint solving	70
4.1	Syntax for both dynamic semantics interpretations.	80
4.2	The value semantics evaluation rules.	81
4.3	The straightforward update semantics evaluation rules.	82
4.4	The update semantics evaluation rules concerning pointers.	83
4.5	The value typing and update/value refinement rules.	86
5.1	Refinement phases of Cogent	95
5.2	The monadic embedding do-notation.	100
5.3	An example program, its A-normalisation, and monadic embedding.	101
5.4	Partial type erasure to determine C representation	102
5.5	Some example corres rules	105
5.6	Neat embedding of the program from Figure 5.3.	109
6.1	The binary verification framework of Sewell, Myreen, and Klein [122]	117
6.2	The grammar of our Dargent prototype.	120
6.3	A Cogent type laid out according to a Dargent specification.	121




CHAPTER 1

INTRODUCTION

Every now and then I feel a
temptation to design a programming
language but then I just lie down
until it goes away.

L. Peter Deutsch

 COMPLETE, formal, end-to-end verification of low-level software systems, such as operating systems, drivers and file systems, has been traditionally considered prohibitively expensive and too difficult for realistic software development. This prejudice is not grounded in reality: projects such as the seL4 microkernel [75] have demonstrated that such verification is actually cheaper than existing methods for high-assurance software engineering certification, and offers a much greater degree of trustworthiness [73].

The verification of seL4 is a very significant milestone, but a microkernel alone is not a sufficient platform to support most applications. In the Linux kernel code base, device drivers and file systems form the largest fraction of the code, and file systems have the highest density of faults [104]. Each file system is approximately the same size as the seL4 microkernel, and a typical Linux distribution includes dozens of file systems. Replicating the ≈ 25 person years required to write and verify seL4 for each of the many file system drivers in use would be completely impractical, so new methods are needed. The $\approx 10,000$ lines of the seL4 microkernel are dwarfed by the 15 million lines of code in the Linux 3.10 kernel tree.

The aim of the Trustworthy Systems project, of which this project is a key component [74], is to reduce the cost of verification to be competitive with traditional

widely-used methods of software engineering, at least for the development of these operating systems components like device drivers and file systems, thus creating a platform on which large scale systems can be guaranteed to be free of faults [56, 55]. Klein et al. [73] estimate that implementation and verification of seL4 was about five times more expensive than traditional software development practices without high-assurance certification. As we have previously stated [74], a factor of five effort reduction is not necessarily an unachievable goal, but even an effort reduction of a smaller constant factor could make verification an attractive option for software development where some degree of assurance is desired.

1.1 AUTOMATING REFINEMENT-BASED VERIFICATION

In recent years, a number of large-scale software verification projects have been carried out. In addition to the aforementioned seL4 microkernel [75], these breakthroughs include verified compilers for C [81] and ML [79], verified theorem provers Milawa [30] and Candle [78], a verified conference system [70], a crash-resistant file system [18], the concurrency verification in CertiKOS [50], the verified cryptographic routines of OpenSSL HMAC [12], the verified distributed system Ironfleet [54] and many more — not to mention the mechanisation of large mathematical proofs such as that of the Four Colour Theorem [46], the Kepler Conjecture [51, 52], or the Odd Order Theorem [47].

1.1.1 THE SEL4 MICROKERNEL

Existing approaches to functional correctness verification, including that of seL4, typically rely on a hierarchy of specifications, of varying levels of abstraction, and a manual proof of *refinement* [31] between each specification in the hierarchy. At the top of the hierarchy, the specifications are abstract, small, and obviously capture functional correctness for the system. The bottom of the hierarchy consists of a concrete, deterministic representation of an implementation in terms of a low level language or machine model. The refinement proofs guarantee that any functional correctness property proved about a higher level specification also applies to all lower levels.

In the case of seL4, the refinement hierarchy is comprised of three layers: a C implementation, an executable specification, and a smaller abstract specification, with refinement proofs between each layer (see Figure 1.1). The executable specification is the output of a straightforward syntactic translation from a Haskell prototype [33]. The C code is given semantics in terms of the SIMPL language [121], and refinement is

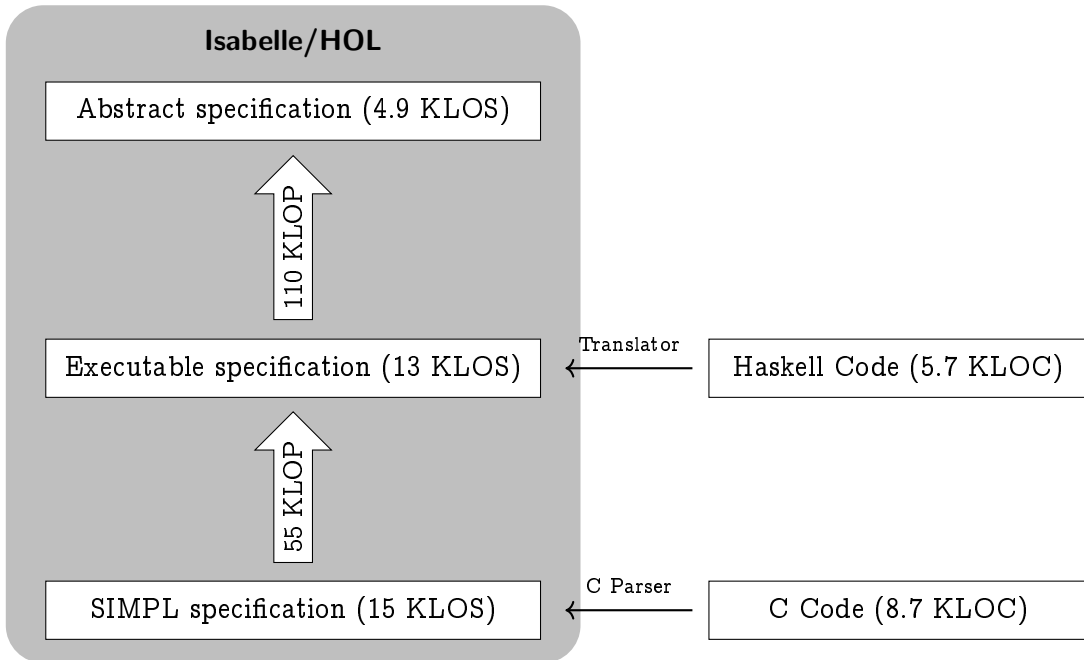


Figure 1.1: The refinement hierarchy in the seL4 verification

proven from the monadic executable specification to this SIMPL representation or *embedding* [23, 76]. All of the specifications and proofs are written in the Isabelle/HOL theorem prover [94]. Sewell, Myreen, and Klein [122] used an SMT solver to automatically relate a SIMPL embedding (interpreted as an automaton) directly to the machine-code output of the C compiler, thus verifying the kernel all the way down to the binary level.

In terms of performance, seL4 rates well, performing comparably with the best-performing L4 microkernels on inter-process communication benchmarks. If we were to evaluate the possible automation of some or all of these proof stages, we must consider the performance cost in doing so. For example, it may be possible to compile the abstract specification directly to C code and prove refinement automatically, but this would result in a highly inefficient kernel: The abstract specification does not contain enough information to produce an efficient implementation, and significant creativity was required to prove refinement from the abstract specification to the executable specification.

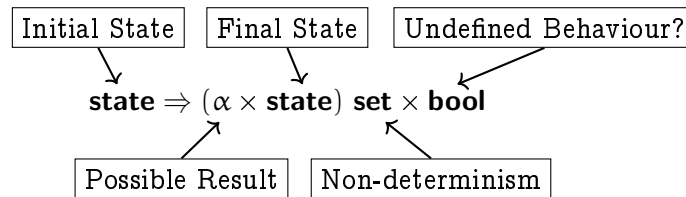
The proof engineers for seL4 report that the refinement proof *below* the executable specification, however, was very straightforward, as was writing the C implementation after the Haskell code had already been written. These areas are prime candidates for automation: saving the user from manually writing 55 thousand lines of proof

and 8.7 thousand lines of C code would be a substantial improvement. Furthermore, the Haskell specification is written in a fairly low-level way, and it may be possible increase the abstraction of the executable specification, while still having efficient implementation, thus reducing proof effort even further.

1.1.2 AUTOCORRES

AutoCorres [49] is a tool embedded within Isabelle which automates some of this lower level refinement proof and specification. It works by generating higher-order logic (HOL) abstractions of imported SIMPL code, and automatically proving that the SIMPL code is a refinement of the generated abstractions.

This abstraction is in terms of a *non-deterministic state monad*. The use of this monad for refinement specifications was first described by Cock, Klein, and Sewell [23]. In the AutoCorres-produced abstraction, a computation returning type α would be represented using the following HOL type:



Here, **state** represents all the global state of the C program, including any global variables and a memory model for the heap. Local, stack-allocated variables are represented as local bindings within the monadic computation, and are not part of the **state**. Given an input **state**, the computation will produce a set of possible results (a return value and final **state**), as well as a flag to indicate if undefined behaviour is possible.

One of the major abstractions that AutoCorres provides is in the heap memory model. After importing C code into SIMPL, the heap is represented simply as a large collection of bytes [128], and interpreting data on the heap must be done more or less by hand when writing refinement proofs. In AutoCorres, however, values on the heap are *typed*, and the memory model consists of a collection of heaps, one for each type.

AutoCorres also provides some other abstractions, such as the translation of potentially overflowing machine-word arithmetic into arithmetic on convenient integers (where possible), as well as conveniences such as heuristics to prove the absence of undefined behaviour.

These abstractions, while undoubtedly useful for manual verification, are still significantly lower-level than even the executable specification of seL4. This is because the C language is too underspecified, in the sense of undefined behaviour, and overpowered, in the sense of omnipresent effects and type coercions, to allow for substantial abstractions that dramatically reduce verification effort. For the cost reduction we seek, we shall have to abandon C in favour of a language that operates on a higher level of abstraction.

1.1.3 CAKEML

CakeML [79] is a verified compiler and language runtime for a significant subset of the language Standard ML [93]. Compared to C, Standard ML is certainly more high-level, supporting algebraic data types, first-class functions and functional programming, as well as a robust module system.

CakeML also inherits *refs* from Standard ML, which are mutable cells that are stored on the heap. While the destructive update afforded by *refs* can be useful for the efficient implementation of many algorithms, the addition of mutable state complicates both refinement proofs and executable specifications, as described in the next section.

A significant benefit of CakeML is that it ensures memory and type safety; two key properties that would have to be manually established for any C verification. Unfortunately, CakeML ensures this by including a language runtime with a garbage collector. While the garbage collector is verified, it makes performance more unpredictable in both time and space. This trade-off is acceptable for CakeML's intended use case of theorem prover implementation, but it renders CakeML unsuitable for low-level systems code running on hardware with potentially limited resources.

1.2 PURE FUNCTIONS, EFFICIENTLY

A purely functional language is a language where functions which perform side-effects such as mutating shared data or which depend on state outside of their input are disallowed. The default specification language for all widely used interactive proof assistants, including Isabelle/HOL, is some variant of the λ -calculus [21], itself a (or *the*) purely functional language. In the case of Isabelle/HOL, System \mathcal{F} is used [45, 113], with some extensions such as type classes. This is no historical accident — purely functional programming enables simple *equational reasoning* about programs and straightforward composition of larger programs from smaller components [63]. This is

also why seL4 was prototyped [33] in Haskell [85], a purely functional language.¹

Without mutation or other side-effects, a program can be represented as a pure, mathematical function, and reasoned about equationally in much the same way as in high-school algebra. With mutation, our assertions about variables become *state-dependent*, in that the value of a variable depends on some global state at that particular point in time. Thus, if our language allows mutation, we must turn to *program logics* such as that of Hoare [57] or Floyd [40] in order to prove properties about programs.

While these early program logics are not overly complex, verifying complex software with this approach proves quite cumbersome, due to the possibility of *aliasing*. Aliasing is when two or more variables refer to the same mutable cell. Thus, updating the value of one variable will affect the other variable also. This so-called “spooky action at a distance” means that, when specifying an algorithm that mutates memory, it is not sufficient to state how variables might be changed by the program. One must additionally, and tediously, specify *inertia* or *frame conditions* for every unrelated variable, stating that those variables remain unchanged after the program has executed. This problem, called the *frame problem*, was originally identified by McCarthy and Hayes [90]. While more sophisticated logics such as separation logic [112] can alleviate this problem, ultimately a purely functional language is preferable for us, as they avoid the problems posed by mutation entirely, and integrate better with interactive proof assistants.

Typically, purely functional languages such as Haskell model updates without mutation by instead allocating a new data structure, and sharing any unchanged parts with the old data structure. If tree data structures are used, the time overhead for this approach is usually $\mathcal{O}(\log n)$ where n is the size of the structure being updated. Once the old structure is no longer used, a garbage collector will eventually free up the memory it occupies.

For systems programming, this kind of purely functional update is unsuitable for two reasons. Firstly, any performance overhead, even a logarithmic one, can represent severe practical performance degradation if the update occurs, for example, in a tight loop. Secondly, as with CakeML, frequent allocation combined with a garbage collector causes unpredictable runtimes and space usage, which is not ideal in a low-level systems context.

¹Haskell is purely functional only modulo certain “benign” effects like non-termination

1.2.1 TYPE SYSTEMS FOR MEMORY MANAGEMENT

As the context of systems programming precludes automatic dynamic memory management, we must track the *lifetime* of allocated memory in the *static semantics* of the language to ensure memory safety.

While specialised static analyses for allocation are also a well-studied problem [116], a number of static approaches integrate the analysis into the type system: *Linear types*, the types analogue of Girard’s Linear Logic [44], were first developed by Lafont [80] and Holmström [60], and popularised by Wadler [132]. Subsequently, their utility for memory management was realised in systems languages such as ATS [17] and Vault [32]. *Affine types*, a close cousin of linear types, were also used in the general-purpose Alms programming language [127]. Another approach, *region typing*, initially developed by Tofte and Talpin [126] for the ML Kit, was later applied to systems programming in the languages Cyclone [66] and Discus [82].

The increasingly popular systems language Rust [118] combines both region and affine types to form the basis of its “borrow checker”², and linear types extensions have been proposed for other popular languages such as Haskell [13] and Swift [125].

1.2.2 LINEAR AND UNIQUENESS TYPES

Linear types are a type of substructural type system [133], which places restrictions on the *use* of variables. While we give a formal definition in Chapter 3, the intuition is that variables of linear type must be used exactly once. Such a syntactic restriction, if applied globally, ensures a very useful dynamic property: throughout the lifetime of an object, objects of linear type have exactly one usable reference to them at a time. Type systems that ensure this uniqueness property are appropriately called *uniqueness type systems*.

As observed by Wadler [132], and implemented in the language Clean [10], this uniqueness property means that destructive update by mutation becomes a mere implementation detail behind a purely functional semantics, as any program that can observe the destructive update would have to share a reference to the old value, which is disallowed by the type system. For example, the following program can be compiled into a single allocation with a destructive update, even though the update appears to

²Despite the separate name, this checker is not distinct from Rust’s type checker.

return a new value:

```

let x1 = alloc ()
      x2 = update x1
in x2

```

If we attempted to retain x_1 as well as x_2 , this would allow the destructive update to x_1 to be observed. The linear type discipline, however, ensures that x_1 is no longer usable after x_2 is created, as it was already used as a parameter to update. For example, the following program is rejected under a linear type system:

```

let x1 = alloc ()
      x2 = update x1
in (x1, x2) (using x1 twice)

```

By similar reasoning, we are unable to use a variable after it has been freed, rejecting a whole class of common memory safety errors:

```

let x1 = alloc ()
      - = free x1
in update x1 (using x1 twice)

```

All of the errors seen so far result from a variable being used more than once, but linear type systems also require that variables of linear type are not discarded without being used.

```

let x1 = alloc ()
      x2 = alloc ()
in x1 (x2 never used)

```

This prevents the leaking of memory, as in order to discard a variable, a programmer must explicitly invoke a free function or some other cleanup routine.

Thus, by requiring code that “owns” a reference to an object to take responsibility for disposal of that object, uniqueness type systems enforce statically the manual memory management discipline that systems programmers previously only enforced by convention, eliminating the need for a garbage collector.

Even better, the aforementioned elimination of aliasing allows us to give well-typed programs two equivalent semantic interpretations: A high-level, abstract and purely functional semantics, suitable for reasoning, and a low-level imperative semantics using destructive update, suitable for generating efficient systems code.

The syntactic requirements of linear types are very restrictive, however, and considerable research has aimed at improving the usability of linear type systems [99, 127,

132]. In particular, constructs to make values of linear type *temporarily* shareable, sometimes called *observers* or *borrows*, are widely employed in many programming languages which use linear and affine types. However, even with these features, data structures that make extensive use of shared pointers are difficult or impossible to encode in such a typing discipline. In the case of Rust, which does not present a purely functional semantics, programmers must nonetheless make use of escape hatches from the restrictions of their type system, even though the language compiler performs detailed lifetime analysis based on region types. Specifically, Rust allows blocks of code to be marked `unsafe`, within which the affine type system rules are suspended.

1.3 A NEW LANGUAGE

When we survey the landscape of pre-existing programming languages, we come to the unfortunate conclusion that none of them are suited to our particular needs:

- C — As AutoCorres demonstrates, there is a limit to the amount of abstraction that can be applied to C code, and that is still substantially lower-level than an executable specification.
- RUST — While efforts are underway to assign a formal semantics to Rust [68], with an axiomatic semantics for verification [69], this project does not aim to certify compilation of Rust programs.³ Even with certified compilation, Rust programs are still imperative and stateful, complicating formal reasoning.
- ML — As previously mentioned, ML programs are not purely functional, and fundamentally depend on garbage collection, even when region-based memory management is used [126].
- HABIT — While the High-Assurance Systems Programming project (HASP) and their language Habit appears similar in goals to our project [1], McCreight, Chevalier, and Tolmach [91] show the correctness of a garbage collector for Habit, which suggests that HASP differs from our own project in the details. While a full formalisation of Habit’s semantics for verification was one of the priorities of the project, the project was never completed, and the language was never formalised.

³At least, not at the time of writing.

- `CLEAN` — While Clean uses uniqueness types to present a purely functional semantics for destructive update, the language still makes use of garbage collection and lazy evaluation, like Haskell, rendering it unsuitable for our intended domain of systems programming.

While there are far too many languages to list here, there are no pre-existing usable languages which possess all three desiderata for our language: a complete formalisation, a purely functional semantics, and no garbage collection.

If, then, we are to make a new language, what kind of language should it be? There are significantly more constraints on our design than would be conventionally encountered when developing a programming language.

Because we must produce a formal proof that the compilation process is correct, the compiler is not free to make large, cross-cutting optimisations or other transformations in the compilation process. Each transformation made by the compiler must be verified, which drastically increases their cost in terms of development effort. Therefore, our language must be as simple as possible, in order to reduce this difficulty.

As we intend our language to be used as an executable specification for further manual verification, it must correspond closely to the object language of a proof assistant, in our case HOL functions in Isabelle. This means that language features that are not easily expressible in HOL cannot easily be included in our language, as they would have to be translated into something that the user can reason about in HOL. To ease verification, we should aim to make our language as high level as possible without sacrificing efficiency.

Lastly, as our intended domain is systems programming, the compiler must generate code that is reasonably efficient, and does not depend on any support from a run-time system. In particular, features such as lazy evaluation, dynamic memory management, dynamic types and closures are mostly precluded by this restriction. As no system exists in isolation, we would like programs written in this language to interface with existing C code, including the seL4 microkernel.

This language need not be general purpose, but must be expressive enough to model the executable specification of operating system components like file systems, although some components of the system, such as complex data structures, could be implemented as separate C libraries.

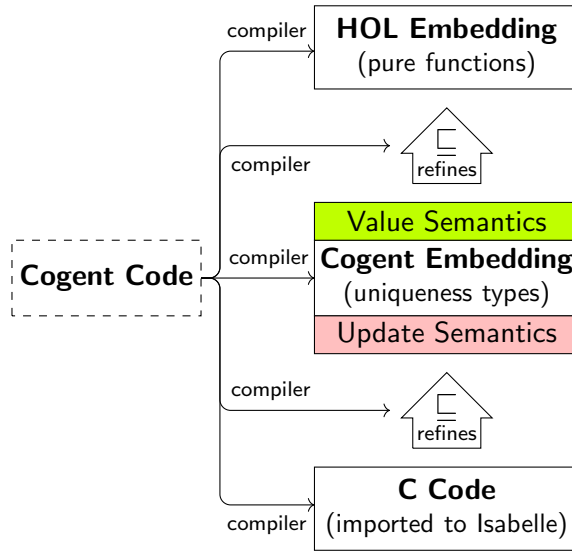


Figure 1.2: A simplified view of the Cogent refinement framework.

1.3.1 COGENT

This thesis presents the language Cogent, a purely functional language intended for systems programming. As explained above, it uses uniqueness types to eliminate the need for garbage collection, and to compile to efficient code with destructive updates.

Cogent also features a *certifying compiler*, which, given a Cogent program, will generate C code, a direct embedding of the Cogent program in Isabelle/HOL, and a proof that the C code (imported into Isabelle) is a refinement of the Cogent embedding. Central to this refinement proof is the *semantic duality* of Cogent: In addition to a purely functional *value* semantics, Cogent programs can also be assigned an equivalent imperative *update* semantics which involves mutable state. A birds-eye view of this framework is presented diagrammatically in Figure 1.2. This means that engineers can write Cogent code, and then reason about it simply and equationally in Isabelle in terms of straightforward HOL embeddings, confident in the knowledge that the refinement proof guarantees that any functional correctness property proven about their HOL embeddings applies also to the generated C code.

The compilation target is C, because C is the language in which most existing systems code is written, including seL4, and because with the advent of tools like CompCert [81]⁴ and the aforementioned gcc translation validation [122], large subsets

⁴Mind the potential logical gap between the SIMPL-based C semantics used here [128] and that of CompCert.

of C now have a formalised semantics and an existing formal verification infrastructure.

Cogent is, however, a heavily restricted language. In addition to the restrictions imposed by the uniqueness type system, we also require Cogent functions to be *total*. This means that Cogent functions must be defined for all their inputs, and cannot loop infinitely or crash. Aside from being helpful to reduce bugs, this restriction greatly simplifies the Isabelle embeddings we generate.

We enforce totality using a crude but effective mechanism: Cogent features no recursion, and all branching must be total. But, even in the restricted target domains of Cogent, some iteration is required. This is where the foreign function interface (FFI) of Cogent comes in: The programmer provides abstract data types (ADTs) in C, and higher-order iterator functions for them. These C functions and iterator functions can be used seamlessly from within Cogent. It is also possible to compose the generated Cogent refinement certificate with manually-written refinement proofs for this C code, without any room for unsoundness. This FFI can also be used to interface with existing systems, such as Linux or seL4, which are written in C.

Cogent is restricted, but it is by no means a toy language. With Amani [2], we used Cogent to successfully implement two efficient full-scale Linux file systems [4] — the standard Linux `ext2` and the BilbyFs flash file system [72] — and proved two core functional correctness properties of BilbyFs. These case studies demonstrate the suitability of Cogent for both systems programming and verification, confirming our initial hypothesis [72] that a language-based approach would dramatically reduce the cost of verifying correctness for practical file system implementations. In addition, these case studies are beneficial in their own right because, as previously mentioned, file systems constitute the second largest proportion of operating system code, and have among the highest density of faults [104].

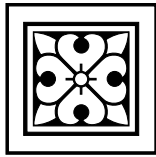
1.4 SUMMARY OF CONTRIBUTIONS

This thesis is the synthesis of a number of research contributions and technical developments. Where possible, citations to the relevant publications where these contributions were first presented will be provided. All of these contributions, except where otherwise noted, were made by the author.

1. The Cogent language itself, including its design and compiler. A tutorial on Cogent is presented in Chapter 2.
2. An evaluation, conducted by Amani [2], of Cogent-implemented systems com-

pared to systems implemented directly in C, in terms of performance and effort. This was originally presented at ASPLOS in 2016 [4] and is repeated here in Chapter 2.

3. The formalisation of the Cogent type system in Chapter 3, drawing from the version of the type system originally presented at ICFP in 2016 [97].
4. A sound type inference algorithm for Cogent, including support for polymorphism, subtyping, and uniqueness types, also in Chapter 3.
5. A formalisation of the dual dynamic semantics of Cogent, and the machine-checked formal proof that connects the two for any well-typed program. This result builds on earlier work for linear type systems [59], extended to account for a fully fledged language and heap-allocated objects. This was originally presented at ICFP in 2016 [97] and is expanded upon in Chapter 4.
6. A characterisation of the requirements placed on foreign functions to maintain this refinement result, also presented at ICFP in 2016 [97] and in Chapter 4.
7. The overall translation validation framework that generates the certificate of refinement from the HOL embedding through the Cogent formalisation and down to C. This involves a number of intermediate translation validation phases, which were summarised at ICFP in 2016 [97] and are significantly expanded upon in Chapter 5. One of the phases connecting Cogent's update semantics to C was a collaborative effort with a number of others, and is separately published at ITP in 2016 [114].
8. An evaluation, conducted along with Amani [2], of the effort involved in the verification of high-level properties on top of the Cogent framework, presented at ASPLOS in 2016 [4] and repeated in Chapter 5.
9. A proposal for a data description language extension to Cogent, intended to address some of the performance and language shortcomings observed in the existing case studies. This was first proposed at ISoLA 2018 [98] and is summarised as part of future work in Chapter 6.



CHAPTER 2

PROGRAMMING IN COGENT

Like dreams, statistics are a form of wish fulfilment.

Jean Baudrillard

DESIGNING a programming language like Cogent is a delicate balancing act between competing and often contradictory interests. From a systems programming perspective, it must offer good performance, interfacing with existing C code, and predictable execution times. From a verification perspective, it must be high-level, easy to reason about, and abstract away from as many cumbersome details as possible. These goals are often very much in tension, and it falls to the poor programming language designer to strike a balance that hopefully satisfies both sides.

In this chapter, we focus on Cogent as a *programming language*, and not as a verification framework. We discuss how Cogent is used for verification in Chapter 5. Instead, here, we will introduce the Cogent language itself [97], the Cogent-implemented file systems used as a case study for the language [4], and an evaluation of how Cogent file systems perform relative to their C counterparts.

These case studies show that Cogent is a highly usable language for systems programming, offering competitive performance with native C implementations, and also reveal a number of avenues for future improvement of the language, which we revisit in Chapter 6.

Before presenting the case studies, however, we shall give a sense of the experience of programming in Cogent by way of a short tutorial, explaining the features of the language, its design philosophy, and its concrete syntax.

2.1 A COGENT TUTORIAL

As Cogent is a functional language, it inherits the syntactic style of earlier functional languages such as Haskell and ML.

Functions and constants are given a top-level type signature and a defining equation, as in the following example:

```
1 add : (U32, U32) → U32
2 add (x, y) = x + y
```

Arithmetic operations (e.g. $+$) may be used on any numeric type (e.g. U8), however they must be used on the *same* numeric type. The type checker inserts no implicit coercions between numeric types. As we don't support closures, partial application via currying is also not common, so it is typical for multi-parameter functions to take a tuple (a.k.a. product type) of their arguments.

We also allow **if** and (non-recursive) **let** expressions with the usual semantics. Comments are preceded with a double-dash (`--`), as in Haskell.

```
1 -- detects overflow
2 add' : (U32, U32) → U32
3 add' (x, y) =
4   let out = x + y
5   in if out < x || out < y then 0 else out
```

We also support pattern matching, however the syntax is a good deal more lightweight than in Haskell or ML, because pattern matching is used in Cogent for error-handling situations that would make use of exceptions in Haskell or ML. Also unlike those languages, our patterns must also be *exhaustive*. Omitting a case is not just a warning but a compile error. To match on an expression, a series of vertically-aligned pipe characters (`|`) are placed after the expression, one for each case. This is the only alignment-sensitive part of the language syntax. In this document, we use a single vertical rule to make the matches both clearer to read and more aesthetically pleasing.

```

1 -- detects overflow
2 add'' : (U32, U32) → U32
3 add'' (x, y) =
4   let out = x + y
5   in out < x || out < y
6     | True → 0
7     | False ⇒ out

```

After each pattern, one of three symbols (\Rightarrow , \rightarrow and \rightsquigarrow) may be used to indicate the corresponding expression to the pattern. Each symbol conveys optimisation information to the underlying C compiler, to determine the likelihood of each branch. We use \Rightarrow to indicate that the branch is *likely* to be taken, and \rightsquigarrow to indicate that the branch is *unlikely* to be taken. The intermediate symbol \rightarrow passes no particular likelihood to the compiler.

Patterns may take the form of a literal value (True, 32, etc.), which only accepts values that are equal to the literal; a variable (e.g. x), which matches against any value and binds it to that variable name in the subsequent case; or a wildcard (written $_$), which accepts any value without binding it to a particular variable names. There are also special patterns for *variant* and *record* types, described in subsequent sections.

2.1.1 VARIANT TYPES

Known in other languages as a *tagged union* or a *sum type*, a variant type describes values that may be one of several types, disambiguated by a *tag* or *constructor*. For example, a value of type $\langle \text{Failure U16} \mid \text{Success U8} \rangle$ may contain *either* an eight-bit or sixteen-bit unsigned integer, depending on which constructor (Success or Failure) is used.

Using the unit type (written $()$), the type with a single trivial inhabitant (also written $()$), we can also use variant types to construct the familiar *Option* or *Maybe* types from ML or Haskell:

```
type Option a =  $\langle \text{None } () \mid \text{Some } a \rangle$ 
```

Here we have used the syntax for *type synonyms* in Cogent. While *Option U8* is easier for humans to write, the Cogent type system makes absolutely no distinction between *Option U8* and $\langle \text{None } () \mid \text{Some U8} \rangle$.

A variant type may include any number of constructors:

```
type CarState = ⟨Drive U32 | Neutral () | Reverse U32⟩
```

Variant types are deconstructed via pattern matching, and constructed by simply typing a constructor name followed by its parameter. Constructor names are required to begin with a capital letter, so that they can be disambiguated from variables and functions.

To ease implementation, compatibility with Isabelle, and to avoid costly backtracking, we require that the pattern for a constructor's argument be *irrefutable*. An irrefutable pattern will always successfully match against any well-typed value. The same restriction is placed on those patterns that occur on the left hand side of a **let** binding or top-level function definition.

```

1 accelerate : (CarState, U32) → CarState
2 accelerate (st, δ) =
3   st
4   | Drive vel → Drive (vel + δ)
5   | Neutral () → Drive δ
6   | Reverse vel → Reverse (vel + δ)

```

While the above example type-checks, the type can be frustratingly imprecise. For example, if we were to match on the result of a call to the `accelerate` function, we would be required to handle the case for the `Neutral` constructor, despite the fact that this case would never be executed. One simple way to solve this problem is to have a version of `CarState` without the `Neutral` constructor:

```
type CarState' = ⟨Drive U32 | Reverse U32⟩
```

We can then use this to give the above function a more precise type:

```
accelerate : (CarState', U32) → CarState'
```

While this would type-check, the representation of `CarState'` is completely independent of `CarState`. This means that, even though a trivial injection exists from `CarState'` to `CarState`, any function that makes use of `CarState` cannot accept a `CarState'` without a potentially expensive copy.

We address this problem by allowing the programmer to specify that certain constructors of a variant type are statically known not to be present, using the **take** keyword:

```
accelerate : (CarState, U32) → CarState take Neutral
```

Unlike *CarState'*, the type *CarState take Neutral* has the same run-time representation as *CarState*, and thus can be trivially coerced into the broader type. Indeed, the type checker will automatically perform such coercions via subtyping.

Such additional static information also becomes useful when default cases are used in pattern matching, for example:

```
1 bounce : CarState → CarState take Drive
2 bounce -- "x = x" can be omitted here.
3 | Drive vel → Reverse vel
4 | st → st
```

Note that the local variable *st* here is of type *CarState take Drive* because the *Drive* constructor has already been matched.

2.1.2 ABSTRACT TYPES AND FUNCTIONS

By omitting the implementations of functions and type definitions, we declare them to be *abstract*. Abstract types and functions are defined outside of Cogent. Typically, an implementation is provided in C. The Cogent compiler includes powerful infrastructure for compiling C implementations along with Cogent code, including the embedding of Cogent types and expressions inside C code using quasi-quotation.

```
1 type Buffer
2 poke : (Buffer, U32, U8) → Buffer
```

Outwardly, the interface of this *Buffer* type seems purely functional, however Cogent assumes by default that all abstract types are *linear*. This means that any variable of type *Buffer*, or any compound type such as a variant that could potentially contain a *Buffer*, must be used exactly once. This scheme of *uniqueness types* ensures that there is only one active reference to a given *Buffer* object at any given time. Therefore, the C implementation of *poke* is free to destructively update the provided *Buffer* without contradicting the purely functional semantics of Cogent.

```

1 hello : Buffer → Buffer
2 hello buf = -- shadowing is often used to indicate update
3   let buf = poke (buf, 0, 'H')
4   and buf = poke (buf, 1, 'e')
5   and buf = poke (buf, 2, 'l')
6   and buf = poke (buf, 3, 'l')
7   and buf = poke (buf, 4, 'o')
8   in buf

```

In the above example, while it would appear that many intermediate buffers are created, the real implementation is merely a series of destructive updates to the same buffer.

2.1.3 SUSPENDING UNIQUENESS

Uniqueness types can become a hindrance, however, when reading from a data structure, such as in this proposed type for a peek function:

$$\text{peek} : (\text{Buffer}, \text{U32}) \rightarrow \langle \text{Err } () \mid \text{Ok U8} \rangle$$

This function's type states that it will *consume* the *Buffer* and return the byte at the index specified. This means that the only way a C program could validly implement this signature while maintaining the invariants of the Cogent type system would be if it *deallocated* the buffer it received.

We could consider allowing peek to return the buffer as well as a result:

$$\text{peek}' : (\text{Buffer}, \text{U32}) \rightarrow \langle \text{Err Buffer} \mid \text{Ok (U8, Buffer)} \rangle$$

But, in addition to being frightfully inconvenient to use, this makes the type less expressive: Because it says that a new *Buffer* will be returned, this type allows peek' to freely modify the buffer. As mutable state complicates verification, it would be better if we could declare *in the type* that peek' does not modify the buffer it is given.

We do this by using the ! type operator. This operator converts any *linear, writable* type to a *read-only* type that can be freely shared or discarded. A function that takes a value of type *Buffer!* is free to *read* from the buffer, but is unable to *write* to it.

$$\text{peek}'' : (\text{Buffer!}, \text{U32}) \rightarrow \langle \text{Err } () \mid \text{Ok U8} \rangle$$

A value of type *Buffer* can be temporarily converted to a *Buffer!* using the expression-level `!` construct. By placing a `!` followed by a variable name after any `let` binding, `match scrutinee` or `if` condition, the variable will be made temporarily read-only for the duration of that expression.

For example, a function that writes a character to the address specified at the beginning of a buffer combines both read-only and writable uses of the same buffer:

```

1 writeChar : (U8, Buffer) → ⟨Err Buffer | Ok Buffer⟩
2 writeChar (c, buf) =
3   peek'' (buf, 0) !buf
4   | Ok i ⇒ Ok (poke (buf, upcast i, c)) -- upcast to convert U8 to U32
5   | Err () ∼⇒ Err buf

```

Here the use of the `!` post-fix on line 3 allows the *buf* variable to be used both in a read-only way as an argument to `peek''` and in a writable way as an argument to `poke` on line 4.

To ensure that read-only references are never simultaneously live with writable references, we require that any such `!`-annotated expression must not contain any use of the `!` operator in its type. For example, the following program would be rejected:

```

1 -- Rejected by type checker
2 bad : Buffer → (Buffer, Buffer!)
3 bad (c, buf) = let x = buf !buf in (buf, x)

```

While the multiple uses of *buf* are considered valid as *buf* is of shareable type during the binding for *x*, the type checker will reject this program because the type of *x* would be *Buffer!*, which contains the `!` type operator.

This restriction is necessary in order to be able to reason equationally about Cogent programs. If the above program were accepted by the type checker, a subsequent write to the buffer *buf* would also be visible through *x* in the update semantics but not in our equational value semantics. Thus, we must disallow this program in order to preserve the refinement theorem connecting the two semantics.

2.1.4 HIGHER-ORDER FUNCTIONS

To ensure that all shallow embeddings can be automatically shown to terminate, Cogent does not include any form of recursion or any built-in looping construct. This is

a severe restriction, but the vast majority of loops present in the low-level domains targeted by Cogent are traversals of data structures, which in Cogent typically take the form of abstract types.

Therefore, iteration in Cogent is accomplished through the use of abstract *higher-order functions*, providing basic functional traversal combinators such as `map` and `fold` for abstract types. For example, the *Buffer* type described above could have a `map` function like:

$$\text{map} : (\text{U8} \rightarrow \text{U8}, \text{Buffer}) \rightarrow \text{Buffer}$$

Here our `map` function is able to destructively overwrite the buffer with the results of the function applied to each byte.

Note that while Cogent does support higher-order functions (functions that accept functions as arguments or return functions), it does not support nested lambda abstractions or closures, as these can require allocation if they capture variables. Thus, to invoke this `map` function, a separate top-level function must be defined for its argument.

```

1 incrementByte : U8 → U8
2 incrementByte x = x + 1
3 incrementBuf  : Buffer → Buffer
4 incrementBuf buf = map (incrementByte, buf)

```

2.1.5 POLYMORPHISM

Cogent also supports *parametric polymorphism*, allowing functions that operate identically over any type to be defined generically in terms of a type variable. For example, we can define an abstract polymorphic function to fold over each byte in a buffer:

$$\text{foldBuf} : \forall a. (\text{Buffer!}, (\text{U8}, a) \rightarrow a, a) \rightarrow a$$

Seeing as C does not support polymorphism, our compiler will generate multiple specialised C implementations from a polymorphic C template, one for each concrete instantiation used in the Cogent code.

Polymorphic functions can be instantiated to concrete types using square brackets. This type application syntax is not always necessary — the type checker can often infer the omitted types.

```

1 sumBuf : Buffer! → U32
2 sumBuf buf = foldBuf[U32] (buf, sumHelper, 0)
3 sumHelper : (U8, U32) → U32
4 sumHelper (x, y) = (upcast x) + y

```

Note that polymorphic functions are not first class — we only allow polymorphic definitions on the top-level, as with ML. This enables us to implement polymorphism via monomorphisation, which has minimal performance impact at run-time.

Variables of polymorphic type are by default treated as *linear* — they must be used exactly once — this allows the polymorphic type variable to be instantiated to any type, shareable or not. Additional *constraints* can be placed on the type variable, to restrict the possible instantiations to those that can be shared:

```

1 dup : ∀a. (Share a) ⇒ a → (a, a)
2 dup a = (a, a)

```

In addition to Share constraints, we also allow Drop constraints, which require instantiations to be discardable without being used; and Escape constraints, which require instantiations to be safe to return from a !-annotated expression. Multiple constraints can be combined with a comma-separated list:

```

1 dupOrDrop : ∀a. (Drop a, Share a) ⇒ (Bool, a) → (Drop () | Dup (a, a))
2 dupOrDrop (b, a) = if b then Dup (a, a) else Drop

```

Abstract types may be given type parameters also, such as in the *Array* type given below. As with abstract functions, this will correspond to a family of automatically generated C types for each concrete type used in the Cogent code.

The type-level ! operator can also be applied to type variables, as shown in the abstract fold function below, for abstract arrays:

```

1 type Array a
2 fold : ((Array a)!, (a!, b) → b, b) → b

```

2.1.6 RECORDS

In addition to simple tuples, Cogent also supports *record types* with named fields. It supports both *boxed* (stored on the heap) and *unboxed* (stored on the stack) variants.

STRUCTS AND RECORD NOTATION

The unboxed records, also called *structs*, are morally similar to tuples — indeed, tuples are compiled identically to unboxed records internally. Unboxed records can be constructed using a *struct literal*, as follows:

```
1 makeRec : Buffer → # {a : U8, b : Buffer}
2 makeRec buf = # {a = 42, b = buf}
```

Here the sharp sign (#) indicates that the record is unboxed. These structs can be deconstructed with pattern matching analogously.

```
1 getBuf : # {a : U8, b : Buffer} → Buffer
2 getBuf # {a = _, b = buf} = buf
```

We also support *field punning*, and any unused non-linear field can be simply omitted from the pattern:

```
1 getBuf : # {a : U8, b : Buffer} → Buffer
2 getBuf # {b} = b -- Equivalent to above
```

For read-only or unboxed records that contain no linear values, it is also permitted to use a more conventional field-access notation:

```
1 addNamed : # {a : U32, b : U32} → U32
2 addNamed r = r.a + r.b
```

Unboxed records that contain linear fields are linear — they cannot be shared, nor can they be discarded without first matching on the record to extract the linear value. If an unboxed record contains no linear values, it can be shared or discarded freely.

BOXED RECORDS

As Cogent has no in-built notion of heap allocation, boxed records, which are stored on the heap, must be allocated by an externally defined constructor function. As the record is stored on the heap, it is treated as *linear*, and must be explicitly freed, much like an abstract type.

```

1 type Heap
2 type Rec = {buf : Buffer, num : U32}
3 allocRecord : Heap → ⟨Failure Heap | Success (Heap, Rec take (buf, num))⟩
4 freeRecord : (Heap, Rec take (..)) → Heap -- (..) means all fields

```

We define an abstract *Heap* type to represent the allocator state, and use a variant type to model possible allocation failure. The *Heap* parameter is needed as Cogent has a deterministic, equational semantics, so the behaviour of a function cannot depend on *any* implicit global state, including the state of available memory. If allocation succeeds, the desired record type is returned. Similarly with variants, the **take** keyword is used to indicate that a field is not initialised or *unavailable*. These fields can subsequently be supplied using a record assignment expression:

```

1 example : Heap → Heap
2 example h =
3   allocRecord h
4   | Failure h → h
5   | Success (h, r) → let r' = r {num = 42} in freeRecord (h, r')

```

While this assignment expression appears to return a new record r' of type *Rec take* buf; because r is linear, it cannot be used again, so Cogent can compile this expression into an efficient destructive update of the record.

Note also that while `freeRecord` expects a record of type *Rec take* (..), we supply r' , of type *Rec take* buf. Cogent resolves this by implicitly discarding the field `num` from r' via subtyping. Because this field has the non-linear, freely discardable type `U32`, we know this implicit discarding of data is safe. Similarly, a record assignment expression can only assign to a field that is either unavailable or of a discardable type.

Symmetrically, pattern matching on a linear, non-shareable field makes the field unavailable in the resulting record, to prevent aliasing of a unique pointer:

```

1 getBuffer : Rec → (Rec take buf, Buffer)
2 getBuffer (r {buf}) = (r, buf)

```

A linear, boxed record can be made temporarily non-linear using the ! operator, just as with abstract types. This also applies the ! operator transitively to the record's fields.

```

1 getBuffer : Rec → U32
2 getBuffer r =
3   let ret = bufSize r.buf + r.num !r
4   in ret

```

2.1.7 COMBINED EXAMPLE

Figure 2.1 contains an example of a complete Cogent program. Assuming an abstract *List* data structure with a reduce function (Lines 13-14) which aggregates the *List* content using a given aggregation function and identity element, the function *average* computes the average of a list of 32-bit unsigned integers. It accomplishes this by storing the running total and count in a heap-allocated data structure called a *Bag*. Line 2 defines the *Bag* as a heap-allocated record containing two 32-bit unsigned integers. Lines 3 and 4 introduce allocation and free functions for *Bags*. The *newBag* function returns a variant, indicating that either a bag and a new heap will be returned in the case of Success, or, in the case of allocation Failure, no new bag will be returned. The *addToBag* function (lines 5-8) demonstrates the use of pattern-matching to destructure the heap-allocated record to gain access to its fields, and update it with new values for each. The *averageBag* function (lines 9-12) returns, if possible, the average of the numbers added to the *Bag*. The input type *Bag!* indicates that the input is a read-only, freely shareable view of a *Bag*. This view of the *Bag* is made on line 19 with the ! notation. Lastly, lines 15-23 define the overall *average* function, which creates a *Bag* with *newBag*, pattern matches on the result, and, if allocation was successful, adds every number in the given list to it, and then returns their average.

2.2 SYSTEMS PROGRAMMING IN COGENT

Together with Amani [2], we conducted a case study into the implementation and verification of software systems written in Cogent [4]. Two file systems were implemented.

```

1 type Heap
2 type Bag = {count : U32, sum : U32}
3 newBag : Heap → ⟨Failure Heap | Success (Bag, Heap)⟩
4 freeBag : (Heap, Bag) → Heap
5 addToBag : (U32, Bag) → Bag
6 addToBag (x, b {count = c, sum = s}) =
7   b {count = c + 1, sum = s + x}
8
9 averageBag : Bag! → ⟨EmptyBag | Success U32⟩
10 averageBag (b {count, sum}) =
11   if count == 0 then EmptyBag else Success (sum / count)
12
13 type List a
14 reduce : ∀a b. (List a!, (a!, b) → b, b) → b
15 average : (Heap, List U32!) → (Heap, U32)
16 average (h, ls) =
17   newBag h
18   | Success (bag, h') → let bag' = reduce (ls, addToBag, bag)
19   | in averageBag bag' !bag'
20   | Success n → (freeBag (h', bag'), n)
21   | EmptyBag → (freeBag (h', bag'), 0)
22
23   Failure h' → (h', 0)

```

Figure 2.1: A full example of a Cogent program.

The first is an almost feature-complete implementation of the ext2 revision 1 file system, passing the POSIX File System Test Suite [96] for all implemented features. Its performance is comparable to the implementation of ext2 that is included as part of the Linux Kernel. The second is a flash file system BilbyFs, designed from the ground up to be easy to verify [72].

The Cogent implementations of ext2 and BilbyFs share a common C library of abstract data types that includes fixed-length arrays for words and structures, simple iterators for implementing loops, and Cogent stubs for accessing a range of Linux APIs such as the buffer cache and its native red-black tree implementation. The interfaces exposed by this library are carefully designed to ensure compatibility with Cogent's uniqueness type system.

The ext2 implementation demonstrates Cogent's ability to enable re-engineering of existing file systems, and thus its potential to provide an incremental upgrade path to increase the reliability of existing systems code. BilbyFs, on the other hand, provides a glimpse of how to design and engineer new file systems that are not only performant, but amenable to being verified as correct against a high-level specification of file system correctness.

2.2.1 EXPERIENCE WITH COGENT

In order to shed light on Cogent's usability as a systems programming language, we briefly describe the experience of developing the ext2 and BilbyFs implementations. In both cases, a manually-written C implementation was used as a starting point: In the case of ext2, this was Linux's ext2fs implementation; for BilbyFs, it was our own implementation of the file system that was used to prototype its design [72]. The two file systems were written by separate developers, but in the case of BilbyFs, the same developer wrote both the C and Cogent implementations. Both developers were already familiar with functional programming.

Naturally, Cogent itself evolved in the process — at the time of the initial implementations, the language had uniqueness types, but no polymorphism nor higher-order functions. The developers jointly wrote the shared C library, and the ext2 developer spent considerable time assisting with Cogent tool-chain design and development. Unfortunately, this makes it infeasible to give accurate effort estimates for how long each file system would have taken to write had the language and tool-chain been stable, as they are now. Having to adopt Cogent's functional style was not a major barrier for either developer; indeed one reported that Cogent's use of **let**-expressions for sequenc-

	Original C	Cogent	Generated C
ext2	4,077	2,789	12,066
BilbyFs	4,021	4,643	18,182

(Generated line counts include C library.)

Table 2.1: Implementation source lines of code, measured with `sloccount`.

ing and pattern matching for error handling aided his understanding of the potential control paths of his code. While both had to get used to the uniqueness type system, both reported that this happened quite quickly and that the type system generally did not impose much of a burden when writing ordinary Cogent code. Both developers noted the usefulness of Cogent’s uniqueness types for tracking memory allocation and catching memory leaks. Uniqueness types were reported to cause some friction when having to design the shared C library interfaces to respect the constraints of the type system.

Both developers reported that the strong type system provided by Cogent decreased the time they usually would have spent debugging, which is to be expected. Of course, logic bugs which cannot be captured by the static semantics could remain in Cogent code. Such bugs are harder to debug than in a comparable C implementation, because of the lack of debugging tool support for Cogent. The developers, however, found comparatively few bugs in the Cogent code; the vast majority of bugs were in the C code that accompanies it.

Table 2.1 shows the source code sizes of the two systems. For the original `ext2` system (i.e. the Linux code) we exclude code that implements features that the Cogent implementation does not support. We can see that for the `ext2` system, the Cogent implementation is about two-thirds the size of C.

BilbyFs’ Cogent implementation is larger than `ext2`’s, relative to their respective original C implementations. This is because BilbyFs makes heavier use of the various abstract data types available in the C library, some of which present fairly verbose client interfaces in their current implementation.

The blowout in size of the generated C code is mostly a result of normalisation steps applied by the Cogent compiler, which we discuss in detail in Chapter 5. Most of this is easily optimised away by the C compiler. However, we found that `gcc`’s optimiser does an unsatisfactory job of optimising operations on large structs, resulting in some unnecessary copy operations left in the code. This could be addressed by producing more optimised Cogent output for such cases. We discuss future work to improve the generated C code from Cogent in Chapter 6.

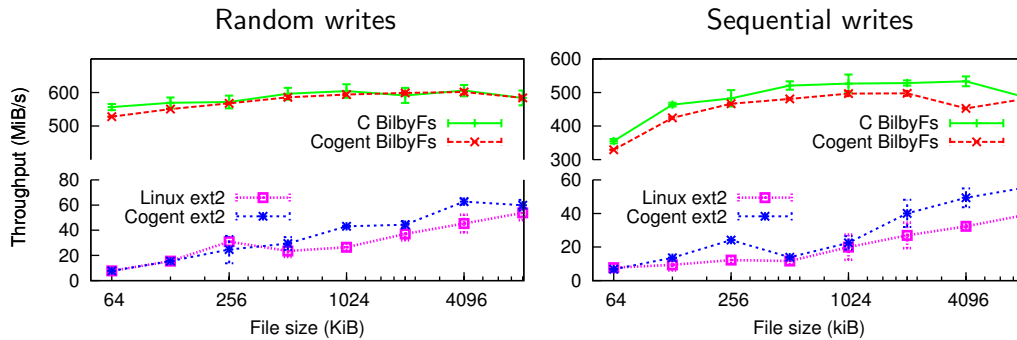


Figure 2.2: IOZone throughput for 4KiB writes

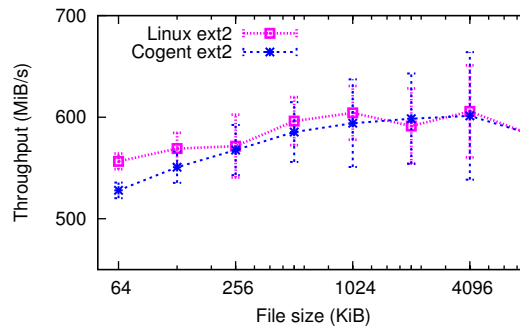


Figure 2.3: IOZone throughput for random 4KiB writes to a RAM disk

2.2.2 PERFORMANCE

Because both file systems were implemented in both C and Cogent, benchmarking both versions gives us an idea of the overheads introduced by programming in Cogent rather than directly in C. Where performance overheads are found, profiling these file systems can indicate which areas of Cogent code generation need improvement to reduce the gap between Cogent and hand-written C.

In Amani et al. [4], we conducted a series of existing file system benchmarks on both file systems. The evaluation platform for the ext2 file system was a four-core Intel Core i7-6700, running at 3.1 GHz, with a Samsung HD501JL 7200RPM 500GiB SATA hard disk, running Linux kernel 4.3.0-1 from the Debian 8.0 distribution. To reduce the run-to-run variability, we disabled all but one core. For the BilbyFs evaluations, we used a Mirabox with 1GiB NAND flash, a Marvell Armada 370 single-core 1.2GHz ARMv7 processor and 1GiB of DDR3 memory, running Linux 3.17 from the Debian 6a distribution.

IOZONE BENCHMARKS

We use the IOZone file system microbenchmarks [65] to evaluate basic performance. We use the default settings for an automated run, except that we include the cost of flush at the end of each write for ext2. We do not include flush for BilbyFs, as doing so completely hides the overhead of the Cogent implementation.

Figure 2.2 shows the throughput of the various file system implementations for random-access and sequential writes. Higher throughput indicates better performance. BilbyFs shows a $\approx 5\%$ and $\approx 10\%$ throughput degradation respectively in the worst case for the Cogent version. The CPU load is around 20% compared to 15% with the C version. This is mostly the effect of redundant memory copy operations in the generated C code when passing structs on the stack. Since these benchmarks were conducted, we have made some improvements to Cogent's code generator that reduces this overhead. We discuss further improvements in Chapter 6.

Because ext2 uses magnetic media, rather than flash memory, we expect significantly lower throughput. Surprisingly, Cogent appears to perform better than the Linux ext2 implementation on this benchmark. CPU usage for Cogent and native Linux are the same at around 10%. After some investigation, we discovered that, because the Cogent implementation is slightly slower, disk I/O operations hit the disk more often, instead of being merged in the Linux I/O queue. We speculate that the on-disk firmware does a better job of scheduling disk writes than the Linux CFQ I/O scheduler.

Figure 2.3 shows write performance for ext2 on a RAM disk, without physical media. Without physical I/O, Cogent is slightly slower than native Linux, as expected. This confirms that the performance differences observed in Figure 2.2 are indeed the result of disk artefacts. The $\pm 8\%$ error bars (showing standard deviation on ten runs) are because of CPU contention with other system activity; they are larger for Cogent because its slightly longer running time gives more opportunity for such contention.

MACROBENCHMARKS

For a macrobenchmark that exposes present limitations of Cogent, we ran the benchmark suite postmark [71], a benchmark that emulates a busy mail server by creating and deleting many small files. For ext2 tests, we configure the benchmark to start with 50,000 files of size 10,000 bytes each, and run on a RAM disk so that I/O latency will not mask Cogent overheads. For BilbyFs, we used a RAM disk that emulates the flash memory interface. Because BilbyFs is a lot faster to create files than ext2, we

	Total Time	Creation	Read Rate
	sec	files/sec	KiB/sec
C ext2	10	5,025	248
Cogent ext2	21	2,393	118
C BilbyFs	6	33,375	431
Cogent BilbyFs	10	20,025	259

(CPU Usage is 100% in all cases)

Table 2.2: Postmark run summary

increased the initial number of files in the benchmark to 200,000. For both of these benchmarks, we used the same Intel Core i7 machine that was used for the ext2 IOZone benchmarks.

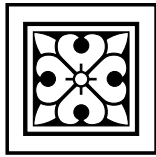
Table 2.2 shows the results, where each of the values is the mode of ten runs. We can see significant degradation of the Cogent implementations, a factor around two for ext2 on the RAM disk, and 1.5 times for BilbyFs. Profiling Cogent ext2 performance shows that most of the time is spent converting in-buffer directory entries to an algebraic data type in Cogent. In Chapter 6, we investigate a way to eliminate this marshalling code and have Cogent code work on the in-buffer entries directly. Another bottleneck observed in BilbyFs is in a function that summarises information about newly created files for the log. The same function is a bottleneck in both C and Cogent versions, but the Cogent version takes about three times as long, once again due to marshalling between various representations of data types.

In addition to these overheads, the Cogent compiler presently relies very heavily on the optimiser in the C compiler. In general, Cogent tends to create larger structs, which is not optimised as well as the idiomatic style of C used in the hand-written implementations. Further work is needed to generate code that is more in line with the C compiler's optimisation capabilities.



In many ways, Cogent appears entirely typical for a functional language in the ML family: higher-order functions, pattern matching, algebraic data types, and so on. But, unlike these languages, Cogent should be characterised not by what it *has*, but what it *has not*. By carefully *restricting* the language to only support features that can be straightforwardly compiled to efficient code with a certificate of compilation correctness, we make those programs that *can* be expressed in Cogent more verifiable. Of course, it is possible to take this too far, reducing the expressiveness of the language

to the point where no realistic programs are accepted. Our case study file systems, however, show that Cogent is able to express real-world system implementations, relying on a single shared library of common data types and abstractions in C. As this library is shared between systems, the cost of its verification could be amortised over each system which uses it. These case studies demonstrate that Cogent's restrictions, which give it a significant advantage in verifiability and generated code efficiency, do not overly impede its expressiveness for systems programming.




CHAPTER 3

STATIC SEMANTICS

It is no use trying to sum people up.
One must follow hints, not exactly
what is said, nor yet entirely what is
done.

John Greenleaf Whittier

OGENT makes use of a number of novel or uncommon *static semantics* features, being a structurally typed language with uniqueness types and a foreign function interface. This chapter outlines the typing rules, the algorithm for type inference and type checking, and the relationship between the algorithm and the typing rules.

Uniqueness types by themselves do not overly complicate the type-checking process (see Walker [133] for a simple algorithmic typing presentation of linear lambda calculi), but from the perspective of a compiler implementation, there are a number of additional confounding features particular to Cogent.

The introduction of parametric polymorphism, even in the restricted sense of prenex rank-1 polymorphism *sec.* ML [93], means that straightforward type checking algorithms require explicit type applications for all polymorphic functions, which is both verbose and cumbersome for the Cogent programmer.

If we turn to more sophisticated techniques to handle polymorphism, the presence of *subtyping* in the language renders many of the existing complete inference approaches (such as the \mathcal{W} algorithm of Damas and Milner [29]) useless, and the undecidability of type inference for System F_{\leq} [107] is a discouraging result.

Many languages with advanced type systems (i.a. dependent type systems), abandon completeness entirely, relying on so-called *bidirectional* techniques to accomplish

local type inference [108]. That is, they require a type signature to be ascribed to each top-level binding, but then infer types for all expressions, patterns, and specialisations.

While a complete algorithm may be possible for our particular flavour of subtyping (perhaps Dolan and Mycroft [35] can provide inspiration), it is not immediately apparent how a sound and complete reconstruction algorithm could be devised that supports simultaneously uniqueness types, subtyping, and the various non-injective type operators used for records and pointer types in Cogent. On the other hand, if we are to abandon completeness, it is equally unclear at the outset where completeness ought to be sacrificed and annotations required.

3.1 CONSTRAINT-BASED TYPE INFERENCE

Those from the French school of type inference, such as [110], advocate for the decomposition of the type inference problem into two parts, each with their own soundness and completeness conditions:

1. The *generation* of a type *constraint* in the form of a set of equations (or inequalities) between types. Here types may involve placeholder variables called *unknowns* or *unification variables*. This phase of type inference is given as a relation of the form $\Gamma \vdash e : \tau \rightsquigarrow C$, where given as input a context Γ , a term e , and a type τ that may involve unknowns, the constraint C is generated. The *soundness* condition of this phase can be stated directly: If an assignment \mathcal{S} to all unknowns can be devised that satisfies C , then the term e really will have the type $\mathcal{S}(\tau)$.

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow C \quad \mathcal{S}(C)}{\mathcal{S}(\Gamma) \vdash e : \mathcal{S}(\tau)}$$

The *completeness* condition ensures that we generate solvable constraints for well-typed expressions, and can be thought of as the opposite direction to soundness: If a program e can be given a type under some assignment \mathcal{S} , and the type checker produces a constraint C , then \mathcal{S} satisfies C .

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow C \quad \mathcal{S}(\Gamma) \vdash e : \mathcal{S}(\tau)}{\mathcal{S}(C)}$$

In reality, this completeness condition is slightly complicated by the fact that the constraint C may involve unification variables that do not occur in Γ or τ , but this problem can be addressed by elaborating the input expression with type signatures, as seen in Section 3.4.

2. The *solving* of the generated constraints, *i.e.* the devising of an assignment S to all unknowns in a given constraint C such that $S(C)$ holds. The soundness and completeness conditions of this phase are comparatively straightforward: The solver is sound iff any assignment it produces is a satisfying assignment, and it is complete iff it will produce an assignment for any satisfiable constraint.

The advantage of the constraint-based method for our purpose is precisely this decomposition, as it naturally allows us to formulate the constraint generation phase completely independently of completeness concerns, merely producing the necessary constraints on unknowns for sound type reconstruction. This constraint generation can be proven both sound and complete, even for an overall-incomplete type inference algorithm — the incompleteness can be compartmentalised entirely into the solver. This means we do not need to make any decision *a priori* about where signatures should be required, and can formalise a type system that merely generates the necessary constraints and attempts to solve them. Signatures must be added in the cases where the solver is unable to solve the constraints. Furthermore, this gives rise to a development methodology whereby the solver can be incrementally improved, making more and more type signatures redundant as more and more constraints can be solved.

The well-known $HM(X)$ system of Odersky, Sulzmann, and Wehr [100] is a classic example of a constraint-based system for ML-style languages, parameterised by the specific solver and constraint infrastructure (the X in the name). A clear presentation of $HM(X)$ can be found in Pottier and Rémy [110]. The $OUTSIDEIN(X)$ system [131], originally designed for GHC Haskell, extends $HM(X)$ with *local assumptions*, necessary for type inference in the presence of generalised algebraic data types.

3.2 SUBSTRUCTURAL TYPE SYSTEMS

For Cogent, existing constraint-based systems are simultaneously too rich, as they support local polymorphic bindings which Cogent disallows; and somewhat deficient, as they assume that the theory includes a *structural context*. That is, they accept the following *structural* laws implicitly:

$$\frac{\Gamma_1 \Gamma_2 \vdash e : \tau}{\Gamma_2 \Gamma_1 \vdash e : \tau} \text{EXCHANGE} \quad \frac{\Gamma_1 \vdash e : \tau}{\Gamma_1 \Gamma_2 \vdash e : \tau} \text{WEAKENING} \quad \frac{\Gamma_1 \Gamma_1 \Gamma_2 \vdash e : \tau}{\Gamma_1 \Gamma_2 \vdash e : \tau} \text{CONTRACTION}$$

These laws, which respectively state that we may swap, drop, or duplicate assumptions whenever necessary, allow the typing context to be treated as a *set*. Indeed, in many

expressions	e	$::=$	$x \mid \ell$	
			$\mid e_1 \wr e_2$	(<i>primops</i>)
			$\mid e_1 e_2 \mid f[\vec{\tau}_i]$	(<i>applications</i>)
			$\mid \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2$	
			$\mid \mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3$	
			$\mid e :: \tau$	(<i>type signatures</i>)
			$\mid \dots$	
types	τ, ρ	$::=$	α	(<i>type variables</i>)
			$\mid \alpha$	(<i>unknowns</i>)
			$\mid \tau_1 \rightarrow \tau_2$	(<i>functions</i>)
			$\mid \top \mid \dots$	
prim. types	T	$::=$	$\mathbf{U8} \mid \mathbf{U16} \mid \mathbf{U32} \mid \mathbf{U64}$	(<i>integral types</i>)
			$\mid \mathbf{Bool}$	
operators	\wr	$::=$	$+$ \mid \leq \mid \neq \mid \wedge \mid \dots	
literals	ℓ	$::=$	$\mathbf{True} \mid \mathbf{False} \mid \mathbb{N}$	
constraints	C	$::=$	$C_1 \wedge C_2$	(<i>conjunction</i>)
			$\mid \ell \in \tau$	(<i>integer bounds</i>)
			$\mid \tau_1 \approx \tau_2$	(<i>equality</i>)
			$\mid \tau_1 \sqsubseteq \tau_2$	(<i>subtyping</i>)
			$\mid \tau \mathbf{Share} \mid \tau \mathbf{Drop}$	(<i>contract/weaken</i>)
			$\mid \top \mid \perp$	
contexts	Γ	$::=$	$\overline{x : \tau}$	
alg. contexts	G	$::=$	$\overline{x : \langle n \rangle \tau}$	
axiom sets	A	$::=$	$\overline{a_i \mathbf{Drop}, b_j \mathbf{Share}}$	
polytypes	π	$::=$	$\forall \vec{d}. C \Rightarrow \tau$	
type vars	a, b, c			
unknowns	α, β, γ			
variables	x, y, z			
usage counts	m, n			

$\overline{\hspace{1cm}}$ Harpoons indicate a list of zero or more.
 $\overline{\hspace{1cm}}$ Overlines indicate a set, i.e. order is not important.

(Continued in Figures 3.7, 3.11, and 3.14)

Figure 3.1: Syntax of the basic fragment of Cogent

such calculi the rule for variables is presented as

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \text{VAR}$$

where the WEAKENING rule is implicitly used to discard unneeded assumptions, rather than the more precise version of the rule:

$$\frac{}{x : \tau \vdash x : \tau} \text{VAR}$$

As Cogent makes use of a *substructural* type system, specifically uniqueness types, we must be substantially more precise when dealing with contexts. We do not accept the rules of CONTRACTION and WEAKENING universally. Admitting CONTRACTION for any type would allow multiple references to a mutable object to be accessible at one time, thus breaking the semantic correspondence Cogent enjoys. Admitting WEAKENING for any type would allow resources to be discarded without being properly disposed.¹ Rather than a set, a context is now a *multiset*, where each assumption about a variable is viewed as a one-use *permission* to type that variable.

Of course, not *all* types benefit from such linearity restrictions. For example, it would be most inconvenient if one was forced to use a variable of type `Bool` exactly once. Thus, it becomes beneficial to allow contraction and weakening for some types, but not others.

To cleanly accomplish this, we move the manipulation of contexts out of the structural rules; instead reifying them as the explicit relations given in Figure 3.2. We define a *context-splitting* operation, used for typing the *branches* of the abstract syntax tree, which, given assumptions A about the linearity of polymorphic type variables, splits a context Γ into two sub-contexts Γ_1 and Γ_2 . Each assumption from Γ must be put into either Γ_1 or Γ_2 . An assumption may only be distributed into *both* sub-contexts if it is *shareable*, i.e. it contains no unique references. We also define a *weakening* relation, used for typing the *leaves* of the abstract syntax tree, which, under assumptions A , weakens a context Γ into a smaller context Γ' , where each discarded assumption must have a *discardable* type. The specifics of what makes a type *shareable* or *discardable* are encapsulated by the **Share** and **Drop** judgements respectively, definitions of which are provided later in Figure 3.6. The fragment of Cogent defined in Figure 3.1 contains only primitive types, however, which are all freely shareable and discardable.

¹Although it is possible to statically insert destructor code as linear variables go out of scope, giving an *affine* type system, this complicates implementation and is omitted for now.

$$\boxed{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2}$$

$$\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2}{A \vdash x : \tau, \Gamma \rightsquigarrow x : \tau, \Gamma_1 \boxplus \Gamma_2}^L \quad \frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2}{A \vdash x : \tau, \Gamma \rightsquigarrow \Gamma_1 \boxplus x : \tau, \Gamma_2}^R$$

$$\frac{}{A \vdash \varepsilon \rightsquigarrow \varepsilon \boxplus \varepsilon} \quad \frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad A \vdash \tau \text{ Share}}{A \vdash x : \tau, \Gamma \rightsquigarrow x : \tau, \Gamma_1 \boxplus x : \tau, \Gamma_2}^C$$

$$\boxed{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma'}$$

$$\frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma'}{A \vdash x : \tau, \Gamma \overset{\text{weak}}{\rightsquigarrow} x : \tau, \Gamma'}^K \quad \frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma' \quad A \vdash \tau \text{ Drop}}{A \vdash x : \tau, \Gamma \overset{\text{weak}}{\rightsquigarrow} \Gamma'}^D$$

$$\frac{}{A \vdash \varepsilon \overset{\text{weak}}{\rightsquigarrow} \varepsilon}$$

Figure 3.2: Context relations

$$\boxed{A; \Gamma \vdash e : \tau}$$

$$\frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} x : \tau}{A; \Gamma \vdash x : \tau}^{\text{VAR}} \quad \frac{A; \Gamma \vdash e : \tau}{A; \Gamma \vdash e :: \tau : \tau}^{\text{SIG}}$$

$$\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad A; \Gamma_1 \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad A; \Gamma_2 \vdash e_2 : \tau_1}{A; \Gamma \vdash e_1 e_2 : \tau_2}^{\text{APP}} \quad \frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \varepsilon \quad \text{typeOf}(f) = \forall \vec{a}_i. C \Rightarrow \tau \quad A \vdash C \left[\frac{\tau_i}{a_i} \right]}{A; \Gamma \vdash f[\vec{\tau}_i] : \tau \left[\frac{\tau_i}{a_i} \right]}^{\text{TAPP}}$$

$$\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad A; \Gamma_1 \vdash e_1 : \tau_1 \quad A; x : \tau_2, \Gamma_2 \vdash e_2 : \tau_2}{A; \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}^{\text{LBT}} \quad \frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad A; \Gamma_1 \vdash e_1 : \text{Bool} \quad A; \Gamma_2 \vdash e_2 : \tau \quad A; \Gamma_2 \vdash e_3 : \tau}{A; \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}^{\text{IF}}$$

$$\frac{T \neq \text{Bool} \quad \wr \in \{+, -, \times, \div, \dots\} \quad A; \Gamma_1 \vdash e_1 : T \quad A; \Gamma_2 \vdash e_2 : T}{A; \Gamma \vdash e_1 \wr e_2 : T}^{\text{IOP}} \quad \frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad T \neq \text{Bool} \quad \wr \in \{=, \neq, <, >, \leq, \geq\} \quad A; \Gamma_1 \vdash e_1 : T \quad A; \Gamma_2 \vdash e_2 : T}{A; \Gamma \vdash e_1 \wr e_2 : \text{Bool}}^{\text{COP}}$$

$$\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad \wr \in \{\wedge, \vee\} \quad A; \Gamma_1 \vdash e_1 : \text{Bool} \quad A; \Gamma_2 \vdash e_2 : \text{Bool}}{A; \Gamma \vdash e_1 \wr e_2 : \text{Bool}}^{\text{BOP}}$$

$$\frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \varepsilon \quad \ell \in \mathbb{N} \quad \ell < |T|}{A; \Gamma \vdash \ell : T}^{\text{ILIT}} \quad \frac{A \vdash \Gamma \overset{\text{weak}}{\rightsquigarrow} \varepsilon \quad \ell \in \{\text{True}, \text{False}\}}{A; \Gamma \vdash \ell : \text{Bool}}^{\text{BLIT}}$$

(Continued in Figures 3.8, 3.13, and 3.15)

Figure 3.3: Some non-algorithmic typing rules

Figure 3.3 contains the typing rules for this elementary fragment of Cogent: just variables (VAR), literals (ILIT and BLIT), binary operators (IOP, BOP and COP), conditionals (IF), and local monomorphic bindings (LET). For simplicity, Cogent does not currently include lambda abstractions or local polymorphism. Thus, all function definitions or polymorphic definitions must occur on the top-level. We assume the existence of a global environment $typeOf(\cdot)$ that includes the complete types of all top-level definitions so far. The rule TAPP allows these top-level polymorphic definitions to be used and instantiated.

3.3 ELEMENTS OF CONSTRAINT GENERATION

This fragment of Cogent is nothing unusual, and yet the typing rules do not correspond to a tractable type-checking algorithm, for two reasons:

1. Each time the context is split, the decision about where to put each assumption cannot be made *prima facie* without first examining the sub-expressions to determine which variables are used.
2. The exact type assigned to integer literals (ILIT) is overloaded, and therefore non-deterministic. Such literals can be assigned any integral type that contains the literal value.

Figure 3.4 outlines the rules for *constraint generation* for the same fragment of Cogent. Unlike the typing rules, these constraint generation rules have an algorithmic interpretation, where everything to the left of the \rightsquigarrow symbol can be viewed as an *input*, and everything to the right an *output*.

To make our context-splitting algorithmic, we avoid the *a priori* splitting relations used in the typing rules, instead borrowing a trick from Walker [133] and making the context an *output* as well as an input to the constraint generation process. As Cogent combines linear and non-linear types, we do not *remove* assumptions from the context when they are used as Walker does. Instead, we keep a count of the uses of each assumption, incrementing it whenever it is used. The assumption $x :_{\langle n \rangle} \tau$ signifies that the variable x is of type τ , and has already been used n times. If an assumption $x :_{\langle n \rangle} \tau$ is used and $n > 0$, an additional Share constraint is emitted (see rule CG-VAR₂). Similarly, when values go out of scope unused, a Drop constraint is emitted for their type (see rule CG-LET). Contexts which include these usage annotations are called *algorithmic contexts* and they are denoted by G rather than Γ .

$$\boxed{G \vdash e : \tau \rightsquigarrow G' \mid C}$$

$$\frac{}{x : \langle 0 \rangle \rho, G \vdash x : \tau \rightsquigarrow x : \langle 1 \rangle \rho, G \mid \rho \sqsubseteq \tau} \text{CG-VAR}_1$$

$$\frac{n > 0}{x : \langle n \rangle \rho, G \vdash x : \tau \rightsquigarrow x : \langle n+1 \rangle \rho, G \mid \rho \sqsubseteq \tau \wedge \rho \text{ Share}} \text{CG-VAR}_2$$

$$\frac{G_1 \vdash e : \tau' \rightsquigarrow G_2 \mid C}{G_1 \vdash e :: \tau' : \tau \rightsquigarrow G_2 \mid C \wedge \tau' \sqsubseteq \tau} \text{CG-SIG}$$

$$\frac{\alpha \text{ fresh} \quad G_1 \vdash e_1 : \alpha \rightarrow \tau \rightsquigarrow G_2 \mid C_1 \quad G_2 \vdash e_2 : \alpha \rightsquigarrow G_3 \mid C_2}{G_1 \vdash e_1 e_2 : \tau \rightsquigarrow G_3 \mid C_1 \wedge C_2} \text{CG-APP}$$

$$\frac{\alpha \text{ fresh} \quad G_1 \vdash e_1 : \alpha \rightsquigarrow G_2 \mid C_1 \quad x : \langle 0 \rangle \alpha, G_2 \vdash e_2 : \tau \rightsquigarrow x : \langle n \rangle \alpha, G_3 \mid C_2 \quad \text{if } n = 0 \text{ then } C_3 = \alpha \text{ Drop else } C_3 = \top}{G_1 \vdash \text{let } x = e_1 \text{ in } e_2 : \tau \rightsquigarrow G_3 \mid C_1 \wedge C_2 \wedge C_3} \text{CG-LBT}$$

$$\frac{\ell \in \{\text{True}, \text{False}\}}{G \vdash \ell : \tau \rightsquigarrow G \mid \tau \approx \text{Bool}} \text{CG-BLIT} \quad \frac{\ell \in \mathbb{N}}{G \vdash \ell : \tau \rightsquigarrow G \mid \ell \in \tau} \text{CG-ILIT}$$

$$\frac{\vec{\beta}_j \text{ fresh} \quad \text{typeOf}(f) = \forall \vec{a}_i \vec{b}_j. C \Rightarrow \rho \quad C' = C \left[\frac{\vec{\tau}_i}{\vec{a}_i} \right] \left[\frac{\vec{\beta}_j}{\vec{b}_j} \right]}{G \vdash f[\vec{\tau}_i] : \tau \rightsquigarrow G \mid \rho \left[\frac{\vec{\tau}_i}{\vec{a}_i} \right] \left[\frac{\vec{\beta}_j}{\vec{b}_j} \right] \sqsubseteq \tau \wedge C'} \text{CG-TAPP}$$

$$\frac{G_1 \vdash e_1 : \text{Bool} \rightsquigarrow G_2 \mid C_1 \quad G_2 \vdash e_2 : \tau \rightsquigarrow G_3 \mid C_2 \quad G_2 \vdash e_3 : \tau \rightsquigarrow G'_3 \mid C_3 \quad G_3 \boxtimes G'_3 \rightsquigarrow G_4 \mid C_4}{G_1 \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau \rightsquigarrow G_4 \mid C_1 \wedge C_2 \wedge C_3 \wedge C_4} \text{CG-IF}$$

$$\frac{\lambda \in \{+, -, \times, \div, \dots\}}{G_1 \vdash e_1 : \tau \rightsquigarrow G_2 \mid C_1 \quad G_2 \vdash e_2 : \tau \rightsquigarrow G_3 \mid C_2}{G_1 \vdash e_1 \lambda e_2 : \tau \rightsquigarrow G_3 \mid 0 \in \tau \wedge C_1 \wedge C_2} \text{CG-IOP}$$

$$\frac{\alpha \text{ fresh} \quad \lambda \in \{=, \neq, <, >, \leq, \geq\}}{G_1 \vdash e_1 : \alpha \rightsquigarrow G_2 \mid C_1 \quad G_2 \vdash e_2 : \alpha \rightsquigarrow G_3 \mid C_2}{G_1 \vdash e_1 \lambda e_2 : \tau \rightsquigarrow G_3 \mid 0 \in \alpha \wedge \tau \approx \text{Bool} \wedge C_1 \wedge C_2} \text{CG-COP}$$

$$\frac{\lambda \in \{\wedge, \vee\}}{G_1 \vdash e_1 : \tau \rightsquigarrow G_2 \mid C_1 \quad G_2 \vdash e_2 : \tau \rightsquigarrow G_3 \mid C_2}{G_1 \vdash e_1 \lambda e_2 : \tau \rightsquigarrow G_3 \mid \tau \approx \text{Bool} \wedge C_1 \wedge C_2} \text{CG-BOP}$$

(Continued in Figures 3.10, 3.13, and 3.7)

Figure 3.4: Some elementary constraint generation rules

$$\boxed{G_1 \bowtie G'_1 \rightsquigarrow G_2 \mid C}$$

$$\frac{
\begin{array}{l}
G = \overline{x_i : \langle n_i \rangle \tau_i} \quad G' = \overline{x_i : \langle n'_i \rangle \tau_i} \quad m_i = \mathit{max}(n_i, n'_i) \\
\text{for each } i, \text{ if } n_i = n'_i \text{ then } C_i = \top \text{ else } C_i = \tau_i \text{ **Drop**}
\end{array}
}{G \bowtie G' \rightsquigarrow \overline{x_i : \langle m_i \rangle \tau_i} \mid \bigwedge_i C_i} \text{COMBINE}$$

Figure 3.5: Algorithmic context join

$$\boxed{A \vdash C}$$

$$\frac{C \in A}{A \vdash C} \text{ASM} \quad \frac{A \vdash C_1 \quad A \vdash C_2}{A \vdash C_1 \wedge C_2} \text{CONJ} \quad \frac{\ell < |T|}{A \vdash \ell \in T} \text{INT} \quad \frac{}{A \vdash \top} \text{TOP}$$

$$\frac{}{A \vdash \tau \approx \tau} \text{REPL} \quad \frac{A \vdash \tau \approx \rho}{A \vdash \tau \sqsubseteq \rho} \text{EQUAL} \quad \frac{A \vdash \rho_1 \sqsubseteq \tau_1 \quad A \vdash \tau_2 \sqsubseteq \rho_2}{A \vdash (\tau_1 \rightarrow \tau_2) \sqsubseteq (\rho_1 \rightarrow \rho_2)} \text{FUN}$$

$$\frac{}{A \vdash \tau_1 \rightarrow \tau_2} \text{FUN-S} \text{ **Share**} \quad \frac{}{A \vdash \tau_1 \rightarrow \tau_2} \text{FUN-D} \text{ **Drop**}$$

$$\frac{}{A \vdash \top} \text{PRIM-S} \text{ **Share**} \quad \frac{}{A \vdash \top} \text{PRIM-D} \text{ **Drop**}$$

(Continued in Figures 3.9, 3.12, and 3.16)

Figure 3.6: Constraint semantics

Our constraint generation relation takes a context G , expression e and type τ , and produces an output context G' and a constraint C that must be true for e to have the type τ . The semantics of constraints (i.e. the proof rules to show that a constraint holds) are given in Figure 3.6. After constraint generation, all types, contexts and constraints may include *unknowns*, or unification variables. Devising a satisfying assignment to each unknown is the job of the solver. See Section 3.10 for discussion of the solver, and Section 3.8 for discussion of constraint generation soundness and completeness properties.

The constraints include a *subtyping* constraint $\tau \sqsubseteq \tau'$, however as we have not yet introduced any types (such as records and variants) where subtyping applies, these constraints can be interpreted for basic types as *equality* constraints, denoted $\tau \approx \tau'$.

For conditionals (i.a. CG-IF), we require the two output contexts for each branch to be *compatible*, in the sense that all linear values used in one branch are also used in the other. To express this, we define a *context join* relation in Figure 3.5 which merges two contexts G_1 and G'_1 , producing a context G_2 and a constraint C . The generated

constraint enforces that all assumptions used in one branch but not the other have a discardable type. The only assumptions left unused in G_2 are those that are unused in both G_1 and G'_1 .

To nail down our integer types, and to eliminate the non-determinism we saw in the original typing rules, we also introduce a constraint $n \in \tau$ that states that the number n must be contained within the type τ . The solver is free to choose the smallest integral type that simultaneously satisfies all such constraints.

3.4 ELABORATION

When a polymorphic function is supplied with type parameters (CG-TAPP), our constraint generation allows some or all of the type parameters to be omitted, instead relying on inference to elaborate the expression. Strictly speaking, to make this constraint generation rule compatible with the typing rule TAPP, the expressions themselves would have to be elaborated with explicit types, but, in keeping with existing type inference literature [29], we omit the elaboration from the constraint generation rules.

The completed constraint generation relation, called the *elaboration* relation, includes an additional output expression. Written $G \vdash e : \tau \rightsquigarrow G' \mid C \mid e'$, the rules are analogous to the constraint generation rules, where the additional expression e' is the same as e except with type annotations added:

- Each type application expression is expanded to be fully saturated with type parameters.
- Every sub-expression is annotated with an explicit type signature. This type signature may include unification variables, later substituted by the assignment determined by the solver.

Because every expression is annotated with a signature, every unification variable that occurs in the generated constraint will also occur in the elaborated expression.

3.5 VARIANT TYPES

3.5.1 ADDING SUBTYPING

Variants in Cogent are an anonymous n-ary sum type consisting of a set of *constructor* names paired with types. The syntax for variants is given in Figure 3.7. Users may

expressions	e	$::= \dots \mid K e$	(<i>variant constructor</i>)
		$\mid \mathbf{case} e_1 \mathbf{of} K x. e_2 \mathbf{else} y. e_3$	(<i>pattern matching</i>)
		$\mid \mathbf{case} e_1 \mathbf{of} K x. e_2$	(<i>irrefutable match</i>)
types	τ, ρ	$::= \dots \mid \langle \overline{K^u \tau} \rangle$	(<i>variant types</i>)
		$\mid \langle \overline{K^u \tau} \mid \alpha \rangle$	(<i>incomplete variant</i>)
constraints	C	$::= \dots \mid \tau \mathbf{Exhausted}$	(<i>exhaustiveness check</i>)
usage tags	u	$::= \circ$	(<i>unused</i>)
		$\mid \bullet$	(<i>used</i>)
constructors	K		

Figure 3.7: Syntax for variants

construct a value of variant type by invoking a constructor, as in

$$K\ 42 : \langle K^\circ U8, J^\bullet \text{Bool} \rangle$$

We also tag each constructor with a usage tag, either \bullet or \circ , for use in exhaustivity checking for pattern matching. These usage tags are how we represent the type-level **take** annotations on variant types seen in Chapter 2 in our core language. A constructor is marked with \bullet if it is statically known that this constructor is *not* the one actually used to construct the value. In this way, we can ensure exhaustivity by only permitting irrefutable patterns when *every* other constructor is marked with \bullet .

Unfortunately, this necessitates the addition of subtyping to the type system. Take this simple example:

```

if (condition) then
  K 42 :  $\langle K^\circ U8, J^\bullet \text{Bool} \rangle$ 
else
  J True :  $\langle K^\bullet U8, J^\circ \text{Bool} \rangle$ 
:  $\langle K^\circ U8, J^\circ \text{Bool} \rangle$ 

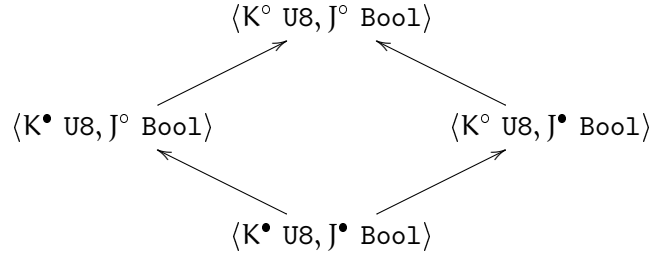
```

Note that the types for the two branches of the conditional differ only in the static knowledge we have of the constructor. The two types have the same run-time representation, so it is safe to *discard* information in order to type the expression. Fortunately,

$$\begin{array}{c}
\boxed{A; \Gamma \vdash e : \tau} \\
\dots \\
\frac{A; \Gamma \vdash e : \tau' \quad A \vdash \tau' \sqsubseteq \tau}{A; \Gamma \vdash e : \tau} \text{SUB} \quad \frac{A; \Gamma \vdash e : \tau}{A; \Gamma \vdash K e : \langle K^\circ \tau, \overline{K_i^\bullet \tau_i} \rangle} \text{VCON} \\
\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad A; \Gamma_1 \vdash e_1 : \langle K^\circ \rho, \overline{K_i^u \tau_i} \rangle \quad A; x : \rho, \Gamma_2 \vdash e_2 : \tau \quad A; y : \langle K^\bullet \rho, \overline{K_i^u \tau_i} \rangle, \Gamma_2 \vdash e_3 : \tau}{A; \Gamma \vdash \mathbf{case} e_1 \mathbf{of} K x. e_2 \mathbf{else} y. e_3 : \tau} \text{CASE} \\
\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad A; \Gamma_1 \vdash e_1 : \langle K^\circ \rho, \overline{K_i^\bullet \tau_i} \rangle \quad A; x : \rho, \Gamma_2 \vdash e_2 : \tau}{A; \Gamma \vdash \mathbf{case} e_1 \mathbf{of} K x. e_2 : \tau} \text{IRREF}
\end{array}$$

Figure 3.8: Typing Rules for variants

this subtyping is fairly well behaved, forming a complete lattice for each variant type:



Non-algorithmic typing rules for all expressions dealing with variants are given in Figure 3.8.

Pattern matching on variants is accomplished in our core language with two primitive forms (**case** expressions). The first is for a *refutable* match (i.e. when the pattern in question is not statically known to match the value), and it includes a default alternative in case the match fails. The second is for *irrefutable* matches, and is only well-typed when the pattern match can be shown statically to succeed.

Typically, a long chain of patterns is desugared into a nested chain of refutable **case** expressions, with a final irrefutable match when the chain of patterns is exhaustive:

$$\begin{array}{c}
\boxed{\tau \hookrightarrow \tau} \\
\langle K^\circ \tau, \overline{K_i^u \rho_i} \rangle \mathbf{take} K \hookrightarrow \langle K^\bullet \tau, \overline{K_i^u \rho_i} \rangle \\
\\
\boxed{A \vdash C} \\
\dots \\
\frac{}{A \vdash \langle \overline{K_i^\bullet \tau_i} \rangle \mathbf{Exhausted}}^{\mathbf{EXHAUST}} \\
\frac{A \vdash \bigwedge_i \tau_i \sqsubseteq \rho_i \quad A \vdash \bigwedge_j \tau_j \sqsubseteq \rho_j}{A \vdash \langle \overline{K_i^\bullet \tau_i}, \overline{K_j^u \tau_j} \rangle \sqsubseteq \langle \overline{K_i^\circ \rho_i}, \overline{K_j^u \rho_j} \rangle}^{\mathbf{VARSUB}} \\
\frac{A \vdash \bigwedge_i \tau_i \mathbf{Share}}{A \vdash \langle \overline{K_i^\circ \tau_i}, \overline{K_j^\bullet \rho_j} \rangle \mathbf{Share}}^{\mathbf{VARSHARE}} \quad \frac{A \vdash \bigwedge_i \tau_i \mathbf{Drop}}{A \vdash \langle \overline{K_i^\circ \tau_i}, \overline{K_j^\bullet \rho_j} \rangle \mathbf{Drop}}^{\mathbf{VARDROP}}
\end{array}$$

Figure 3.9: Constraint semantics for variants

$$\begin{array}{ccc}
& & \mathbf{case} \ x \ \mathbf{of} \\
\mathbf{case} \ x \ \mathbf{of} & & K_1 \ a \rightarrow \dots \\
K_1 \ a \rightarrow \dots & \text{becomes} & x' \rightarrow \mathbf{case} \ x' \ \mathbf{of} \\
K_2 \ b \rightarrow \dots & & K_2 \ b \rightarrow \dots \\
K_3 \ c \rightarrow \dots & & x'' \rightarrow \mathbf{case} \ x'' \ \mathbf{of} \\
& & K_3 \ c \rightarrow \dots
\end{array}$$

3.5.2 TYPE INFERENCE

Algorithmic type inference for variants is significantly more involved than the non-algorithmic version, for two reasons:

1. If a constructor is directly invoked or used in a pattern, this indicates only that the type in question *contains* said constructor. Indeed, any expression involving variant types could, with no modifications, involve wider variant types with any number of additional \bullet -marked constructors.
2. Without knowing all of the concrete types involved, it is impossible to check that an irrefutable pattern match is valid; i.e. we cannot tell without knowing the full type of the scrutinee whether or not patterns are exhaustive.

To solve the first problem — incomplete knowledge of variant types — we extend types with a new type of unknown, written $\langle \overline{K_i^u \tau_i} \mid \alpha \rangle$, which indicates a variant type

$$\boxed{G \vdash e : \tau \rightsquigarrow G' \mid C}$$

...

$$\frac{\alpha, \beta \text{ fresh} \quad G_1 \vdash e_1 : \alpha \rightsquigarrow G_2 \mid C}{G_1 \vdash K e : \tau \rightsquigarrow G_2 \mid C \wedge \langle K^\circ \alpha \mid \beta \rangle \sqsubseteq \tau} \text{CG-VCON}$$

$$\frac{\begin{array}{l} \alpha, \beta \text{ fresh} \quad G_1 \vdash e_1 : \langle K^\circ \beta \mid \alpha \rangle \rightsquigarrow G_2 \mid C_1 \\ x :_{\langle 0 \rangle} \beta, G_2 \vdash e_2 : \tau \rightsquigarrow x :_{\langle n \rangle} \beta, G_3 \mid C_2 \\ y :_{\langle 0 \rangle} \langle K^\bullet \beta \mid \alpha \rangle, G_2 \vdash e_3 : \tau \rightsquigarrow y :_{\langle m \rangle} \langle K^\bullet \beta \mid \alpha \rangle, G'_3 \mid C_3 \\ G_3 \bowtie G'_3 \rightsquigarrow G_4 \mid C_4 \\ \text{if } n = 0 \text{ then } C_5 = \beta \text{ Drop else } C_5 = \top \\ \text{if } m = 0 \text{ then } C_6 = \langle K^\bullet \beta \mid \alpha \rangle \text{ Drop else } C_6 = \top \end{array}}{G_1 \vdash \text{case } e_1 \text{ of } K x. e_2 \text{ else } y. e_3 : \tau \rightsquigarrow G_4 \mid \bigwedge_{i \in 1 \dots 6} C_i} \text{CG-CASE}$$

$$\frac{\begin{array}{l} \alpha, \beta \text{ fresh} \quad G_1 \vdash e_1 : \langle K^\circ \beta \mid \alpha \rangle \rightsquigarrow G_2 \mid C_1 \\ x :_{\langle 0 \rangle} \beta, G_2 \vdash e_2 : \tau \rightsquigarrow x :_{\langle n \rangle} \beta, G_3 \mid C_2 \quad C_3 = \langle K^\bullet \beta \mid \alpha \rangle \text{ Exhausted} \\ \text{if } n = 0 \text{ then } C_4 = \beta \text{ Drop else } C_4 = \top \end{array}}{G_1 \vdash \text{case } e_1 \text{ of } K x. e_2 : \tau \rightsquigarrow G_3 \mid \bigwedge_{i \in 1 \dots 4} C_i} \text{CG-IRREF}$$

Figure 3.10: Constraint generation for variants

types	$\tau, \rho ::= \dots \mid A \bar{\tau} s$	(<i>abstract types</i>)
	$a!$	(<i>observer types</i>)
	$\mathbf{bang}(\tau)$	(<i>observation operator</i>)
constraints	$C ::= \dots \mid \tau \mathbf{Escape}$	(<i>escape analysis</i>)
expressions	$e ::= \dots \mid \mathbf{let!} (\bar{y}_i) x = e_1 \mathbf{in} e_2$	(<i>observation</i>)
sigils	$s ::= \textcircled{w}$	(<i>writable</i>)
	\textcircled{r}	(<i>read-only</i>)
	\textcircled{u}	(<i>unboxed</i>)
	α	(<i>unknown sigil</i>)

Figure 3.11: Syntax for abstract and observer types

containing at least the constructors \bar{K}_i and zero or more unknown alternatives, denoted by the unification variable α . This is similar to languages with row polymorphism [43] where type variables can be used to specify incomplete variants or records, except that in Cogent, all such “row” unification variables are eliminated by constraint solving. The rules that generate these constraints are provided in Figure 3.10.

For the second problem, we introduce another constraint form for pattern matching, written $\tau \mathbf{Exhausted}$. This is true whenever τ is a variant type where all possible constructors have been matched. Using an irrefutable **case** statement emits a constraint on the type of the scrutinee, “ $\langle K^\bullet \beta \mid \alpha \rangle \mathbf{Exhausted}$ ” (see Figure 3.10), indicating that after the final constructor has been matched, there should be no further valid alternatives.

3.6 ABSTRACT AND OBSERVER TYPES

An *abstract type* is a type whose full definition must be provided in imported C code. Values of abstract type must be constructed (and, if necessary, destroyed) by imported C functions, and all operations on them must also be defined in C. Nevertheless, they must be explicitly declared when used in Cogent code. An abstract type declaration consists of a type name and a series of parameters, without any definition provided.

In our core type system, an abstract type is represented as $A \bar{\tau} s$, where A is the type name, $\bar{\tau}$ is the list of type parameters, and s is a *sigil*, which determines which constraints are satisfied by the abstract type. There are three forms of sigil:

- *Read-only* sigils (\textcircled{r}), indicating that the value is represented as a pointer that can be freely shared or dropped, as the value cannot be written to during the

lifetime of this pointer.

- *Writable* sigils (\textcircled{w}), indicating that the value is represented as a pointer, and must be linear, as the value may be destructively updated.
- *Unboxed* sigils (\textcircled{u}), indicating that the value is not represented as a pointer at all,² and may be freely shared or dropped.

Note that, according to the constraint semantics given in Figure 3.12, an abstract type can only satisfy the **Share** and **Drop** constraints *if the sigil is not* \textcircled{w} *writable*. Thus these writable abstract types are the first of the types we have introduced to be *linear*.

3.6.1 OBSERVATION AND ESCAPE ANALYSIS

In our core language, the expression-level **!** construct that allows linear values to be temporarily shared within a limited scope is desugared into the syntactic form **let!** $(\overline{y_i}) \ x = e_1 \ \mathbf{in} \ e_2$. This form is similar to a **let** expression, except that the variables $\overline{y_i} : \overline{\rho_i}$ are temporarily retyped during the typing of e_1 as $\overline{y_i} : \mathbf{bang}(\overline{\rho_i})$, where **bang**(\cdot) is a type operator that changes all linear \textcircled{w} ritable sigils in a type to shareable \textcircled{r} ead-only ones. The typing rules and algorithmic constraint generation rules for **let!** expressions are given in Figure 3.13.

We provide *normalisation rules* for this type operator, starting in Figure 3.12: Written $\tau \leftrightarrow \tau'$, these rules describe how types may be rewritten by the constraint solver to eliminate the type operators operators, after any unification variables have already been eliminated by substitution.³ The **NORM** rule in Figure 3.12 tells us that any constraint C involving a type τ (written $C[\tau]$) can be rewritten according to the type normalisation rules. Semantically, it is unimportant whether types are normalised completely, including inside sub-terms (*deep normal form*), or whether types are normalised only on top-level terms occurring in constraints (*head normal form*). In our implementation, however, we opt for *head normal form* to make error messages (much!) friendlier and to improve type checker efficiency.

To handle polymorphic type variables, which may be instantiated to types containing \textcircled{w} ritable sigils, we introduce another kind of polymorphic type variable, written $a!$, which becomes **bang**(τ) under the substitution $[\tau/a]$. Furthermore, we define

²Or, as a pointer to which no \textcircled{w} ritable pointer will ever exist.

³In our Cogent implementation, we additionally use the normalisation mechanism to support type synonyms.

$$\begin{array}{c}
\boxed{\tau \hookrightarrow \tau} \\
\mathbf{bang}(\langle \overline{K_i^u} \rho_i \rangle) \hookrightarrow \langle \overline{K_i^u} \mathbf{bang}(\rho_i) \rangle \\
\mathbf{bang}(\tau \rightarrow \rho) \hookrightarrow \tau \rightarrow \rho \\
\mathbf{bang}(A \overline{\tau_i} \textcircled{W}) \hookrightarrow A \overline{\mathbf{bang}(\tau_i)} \textcircled{I} \\
\mathbf{bang}(A \overline{\tau_i} \textcircled{I}) \hookrightarrow A \overline{\mathbf{bang}(\tau_i)} \textcircled{I} \\
\mathbf{bang}(A \overline{\tau_i} \textcircled{U}) \hookrightarrow A \overline{\mathbf{bang}(\tau_i)} \textcircled{U} \\
\mathbf{bang}(a) \hookrightarrow a! \\
\mathbf{bang}(a!) \hookrightarrow a! \\
\mathbf{bang}(T) \hookrightarrow T \\
\\
\boxed{A \vdash C} \\
\dots \\
\frac{A \vdash C[\tau] \quad \tau \hookrightarrow \rho}{A \vdash C[\rho]} \text{NORM} \\
\\
\frac{s \neq \textcircled{W} \quad A \vdash \bigwedge_i \tau_i \mathbf{Drop}}{A \vdash A \overline{\tau_i} s \mathbf{Drop}} \text{AbsDROP} \quad \frac{s \neq \textcircled{W} \quad A \vdash \bigwedge_i \tau_i \mathbf{Share}}{A \vdash A \overline{\tau_i} s \mathbf{Share}} \text{AbsSHARE} \\
\\
\frac{s \neq \textcircled{I} \quad A \vdash \bigwedge_i \tau_i \mathbf{Escape}}{A \vdash A \overline{\tau_i} s \mathbf{Escape}} \text{AbsEsc} \quad \frac{}{A \vdash \tau \rightarrow \rho \mathbf{Escape}} \text{FUNEsc} \quad \frac{}{A \vdash T \mathbf{Escape}} \text{PRIMEsc} \\
\\
\frac{A \vdash \bigwedge_i \rho_i \mathbf{Escape}}{A \vdash \langle \overline{K_i^u} \rho_i \rangle \mathbf{Escape}} \text{SUMEsc} \quad \frac{}{A \vdash a! \mathbf{Drop}} \text{OBSDROP} \quad \frac{}{A \vdash a! \mathbf{Share}} \text{OBSSHARE}
\end{array}$$

Figure 3.12: Constraint semantics for abstract and observer types

$$\begin{array}{c}
\boxed{A; \Gamma \vdash e : \tau} \\
\dots \\
\frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad A \vdash \tau' \mathbf{Escape} \quad A; x : \tau', \overline{y_i : \rho_i}, \Gamma_2 \vdash e_2 : \tau}{A; \overline{y_i : \rho_i}, \Gamma \vdash \mathbf{let!} (\overline{y_i}) x = e_1 \mathbf{in} e_2 : \tau} \text{LET!} \\
\\
\boxed{G \vdash e : \tau \rightsquigarrow G' \mid C} \\
\dots \\
\frac{\alpha \text{ fresh} \quad \overline{y_i : \langle 0 \rangle \mathbf{bang}(\rho_i)}, G_1 \vdash e_1 : \alpha \rightsquigarrow \overline{y_i : \langle n_i \rangle \mathbf{bang}(\rho_i)}, G_2 \mid C_1 \quad x : \langle 0 \rangle \alpha, \overline{y_i : \langle 0 \rangle \rho_i}, G_2 \vdash e_2 : \tau \rightsquigarrow x : \langle n \rangle \alpha, \overline{y_i : \langle m_i \rangle \rho_i}, G_3 \mid C_2}{\begin{array}{l} C_3 = \bigwedge_i \text{if } n_i = 0 \text{ then } \mathbf{bang}(\rho_i) \mathbf{Drop} \text{ else } \top \\ C_4 = \bigwedge_i \text{if } m_i = 0 \text{ then } \rho_i \mathbf{Drop} \text{ else } \top \\ C_5 = \text{if } n = 0 \text{ then } \alpha \mathbf{Drop} \text{ else } \top \quad C_6 = \alpha \mathbf{Escape} \end{array}}{\overline{y_i : \langle 0 \rangle \rho_i}, G_1 \vdash \mathbf{let!} (\overline{y_i}) x = e_1 \mathbf{in} e_2 : \tau \rightsquigarrow \overline{y_i : \langle m_i \rangle \rho_i}, G_3 \mid \bigwedge_{k \in 1..6} C_k} \text{CG-LET!}
\end{array}$$

Figure 3.13: Typing and constraint generation rules for **let!**

$\mathbf{bang}(a) \mapsto a!$. This way, we can guarantee that $\mathbf{bang}(\tau)$ is always non-linear *regardless* of τ , as no \textcircled{w} ritable sigils will remain in the type. This technique is originally due to Odersky [99].

THEOREM 3.1 (*bang_non_linear*). *For all types τ and assumptions A , if no unification variables occur in τ we have $A \vdash \mathbf{bang}(\tau) \mathbf{Share}$ and $A \vdash \mathbf{bang}(\tau) \mathbf{Drop}$.*

Proof. By structural induction on τ . □

The dynamic *uniqueness property*, introduced informally in Chapter 2 and formally in Chapter 4, can be stated as:

No \textcircled{w} ritable pointer can be aliased by any other pointer. A \textcircled{i} ead-only pointer may be aliased by any number of other \textcircled{i} ead-only pointers.

We prove in Chapter 4 that this property is maintained as a dynamic invariant as a consequence of the static semantics (making \textcircled{w} ritable pointers linear). A naïve implementation of the **let!** feature, however, can easily lead to this invariant being violated:

let! (x) $y = x$ **in** (x, y)

types	τ, ρ	$::= \dots \mid \overline{\{f_i^u : \tau_i\}} s$	<i>(record types)</i>
		$\mid \overline{\{f_i^u : \tau_i \mid \alpha\}} s$	<i>(record unknowns)</i>
constraints	C	$::= \dots \mid s \neq \textcircled{R}$	<i>(sigil constraint)</i>
expressions	e	$::= \dots \mid \#\{f_i = e_i\}$	<i>(unboxed allocation)</i>
		$\mid \mathbf{take} \ x \ \{f = y\} = e_1 \ \mathbf{in} \ e_2$	<i>(record patterns)</i>
		$\mid \mathbf{put} \ e_1.f = e_2$	<i>(record updates)</i>
		$\mid e_1.f$	<i>(record field read)</i>
field names	f		

Figure 3.14: Syntax for records

In this example, the freely shareable \textcircled{R} ead-only pointer x is bound to y , and thus aliases the \textcircled{W} ritable pointer x in the returned tuple. Therefore, to maintain the invariant, we must prevent the \textcircled{R} ead-only pointers available in a **let!** from escaping their scope. The first formulation to include a **let!** feature is that of Wadler [132], which imposes a type-based safety check on the type of the binding in a **let!**, essentially requiring that the type of the binding and the type of the temporarily non-linear variables have no components in common. We adopt a slightly different approach which originated from Odersky [99], although it differs in presentation.

We introduce a new type constraint,⁴ written τ **Escape**, that states that τ can be safely bound by a **let!** expression. Crucially, it does *not* hold if any \textcircled{R} sigils appear in the type. This means that read-only pointers cannot be bound in a **let!** expression, but writable, linear pointers and unboxed values can be bound without a type error. Figure 3.12 contains full definitions for this **Escape** judgement.

Both methods, our own and that of Wadler [132], are sound, type-based over-approximations of *escape analysis*. Fruitful avenues for further research may be to incorporate more sophisticated analysis techniques to improve the flexibility and predictability of this feature. One possible method may be the use of region types [126] to track the provenance of pointer variables more precisely, which Rust uses to great effect in its similar type system.

3.7 RECORD TYPES

Lastly, we must formalise the type checking process for *record types* or products. The syntax for record types is given in Figure 3.14. A record, written $\overline{\{f_i^u : \tau_i\}} s$, consists

⁴Or judgement, depending on if we are examining the algorithmic or non-algorithmic typing rules.

$$\begin{array}{c}
\boxed{A; \Gamma \vdash e : \tau} \\
\dots \\
\frac{A; \Gamma \vdash e : \{\overline{f_i^\bullet : \tau_i}, f^\circ : \tau\} s}{A; \Gamma \vdash e.f : \tau} \text{MEMBER} \quad \frac{A; \Gamma \vdash \overline{e_i : \tau_i}}{A; \Gamma \vdash \#\{f_i = e_i\} : \{\overline{f_i^\circ : \tau_i}\} \textcircled{U}} \text{STRUCT} \\
\frac{\begin{array}{c} A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ A; \Gamma_1 \vdash e_1 : \{\overline{f_i^u : \tau_i}, f^\bullet : \tau\} s \\ s \neq \textcircled{I} \quad A; \Gamma_2 \vdash e_2 : \tau \end{array}}{A; \Gamma \vdash \mathbf{put} \ e_1.f = e_2 : \{\overline{f_i^u : \tau_i}, f^\circ : \tau\} s} \text{PUT} \quad \frac{\begin{array}{c} A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ A; \Gamma_1 \vdash e_1 : \{\overline{f_i^u : \tau_i}, f^\circ : \rho\} s \quad s \neq \textcircled{I} \\ A; \chi : \{\overline{f_i^u : \tau_i}, f^\bullet : \rho\} s, y : \rho, \Gamma_2 \vdash e_2 : \tau \end{array}}{A; \Gamma \vdash \mathbf{take} \ x \{f = y\} = e_1 \ \mathbf{in} \ e_2 : \tau} \text{TAKE} \\
\boxed{A; \Gamma \vdash \overline{e_i : \tau_i}} \\
\frac{A \vdash \Gamma \xrightarrow{\text{weak}} \varepsilon}{A; \Gamma \vdash \varepsilon} \text{EMPTY} \quad \frac{A \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad A; \Gamma \vdash e : \tau \quad A; \Gamma \vdash \overline{e_i : \tau_i}}{A; \Gamma \vdash e : \tau, \overline{e_i : \tau_i}} \text{CONS}
\end{array}$$

Figure 3.15: Typing rules for records

of one or more *fields* (f_i). Due to the additional properties maintained by our type system, record types in Cogent are structured slightly differently to more traditional programming languages. Suppose we wish to access a particular field f of a record r . An expression like $r.f$ would be problematic, as this *uses* the variable r , so any non- f fields in r would need to satisfy **Drop**. If this were the only way to access the fields of a record, any record with two linear fields would be unusable.

If instead we imagine a pattern-matching expression that reintroduces the record as a new variable name, like so:

$$\mathbf{let} \ r' \{f = x\} = r \ \mathbf{in} \ \dots$$

Then this violates the uniqueness property that our type system purports to maintain, as the field f could be accessed from the resultant record r' as well as by the new variable x . To solve this problem, any field that is extracted via pattern matching is marked as *unavailable* in the type of the resultant record, by changing the usage tag associated with each of the extracted fields to be \bullet . This pattern matching is desugared into one or more **take** expressions, written $\mathbf{take} \ x \{f = y\} = e_1 \ \mathbf{in} \ e_2$. Note that the typing rules in Figure 3.15 requires that the field being taken is available (tagged with \circ), and ensures that the field is no longer available in e_2 (tagged with \bullet). Conversely, record assignment expressions are desugared into **put** expressions, of the form $\mathbf{put} \ e_1.f = e_2$. The typing rule for this expression ensures that the field being

$$\begin{array}{c}
\boxed{\tau \leftrightarrow \tau} \\
\dots \\
\mathbf{bang}(\{\overline{\mathbf{f}}_i^u : \tau_i\} \textcircled{\mathbb{W}}) \leftrightarrow \{\overline{\mathbf{f}}_i^u : \mathbf{bang}(\tau_i)\} \textcircled{\mathbb{F}} \\
\mathbf{bang}(\{\overline{\mathbf{f}}_i^u : \tau_i\} \textcircled{\mathbb{F}}) \leftrightarrow \{\overline{\mathbf{f}}_i^u : \mathbf{bang}(\tau_i)\} \textcircled{\mathbb{F}} \\
\mathbf{bang}(\{\overline{\mathbf{f}}_i^u : \tau_i\} \textcircled{\mathbb{U}}) \leftrightarrow \{\overline{\mathbf{f}}_i^u : \mathbf{bang}(\tau_i)\} \textcircled{\mathbb{U}} \\
\\
\boxed{A \vdash C} \\
\dots \\
\frac{s = \textcircled{\mathbb{W}} \vee s = \textcircled{\mathbb{U}}}{A \vdash s \neq \textcircled{\mathbb{F}}} \text{SIGIL} \\
\\
\frac{A \vdash \bigwedge_i \tau_i \mathbf{Drop} \quad A \vdash \bigwedge_i \tau_i \sqsubseteq \rho_i \quad A \vdash \bigwedge_j \tau_j \sqsubseteq \rho_j}{A \vdash \{\overline{\mathbf{f}}_i^o : \tau_i, \overline{\mathbf{f}}_j^u : \tau_j\} s \sqsubseteq \{\overline{\mathbf{f}}_i^o : \rho_i, \overline{\mathbf{f}}_j^u : \rho_j\} s} \text{RECSUB} \\
\\
\frac{s \neq \textcircled{\mathbb{W}} \quad A \vdash \bigwedge_i \tau_i \mathbf{Share}}{A \vdash \{\overline{\mathbf{f}}_i^o : \tau_i, \overline{\mathbf{f}}_j^o : \tau_j\} s \mathbf{Share}} \text{RECSHARE} \quad \frac{s \neq \textcircled{\mathbb{W}} \quad A \vdash \bigwedge_i \tau_i \mathbf{Drop}}{A \vdash \{\overline{\mathbf{f}}_i^o : \tau_i, \overline{\mathbf{f}}_j^o : \tau_j\} s \mathbf{Drop}} \text{RECDROP} \\
\\
\frac{s \neq \textcircled{\mathbb{F}} \quad A \vdash \bigwedge_i \tau_i \mathbf{Escape}}{A \vdash \{\overline{\mathbf{f}}_i^o : \tau_i, \overline{\mathbf{f}}_j^o : \tau_j\} s \mathbf{Escape}} \text{RECESCAPE}
\end{array}$$

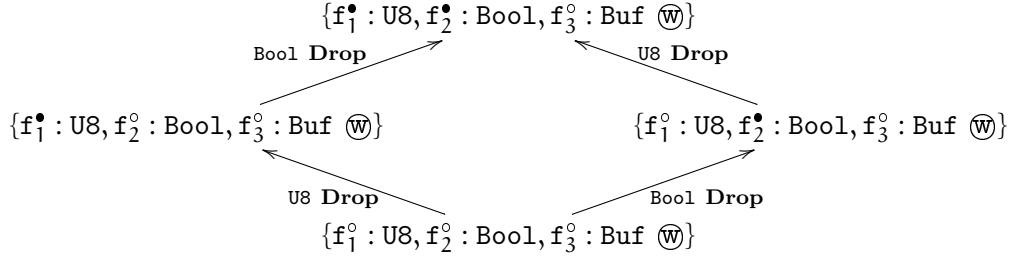
Figure 3.16: Constraint semantics for records

overwritten has already been extracted (\bullet), and makes the field available again in the resultant record (\circ).

Like abstract types, records may be stored on the heap and passed around by reference, in which case we must track uniqueness of each pointer to the record. For this reason, record types are tagged with a *sigil* s , which, much as with abstract types, allows records to be declared *unboxed* ($\textcircled{\mathbb{U}}$), where they are typically represented on the stack or as a flat structure; *read-only* ($\textcircled{\mathbb{F}}$), where they are stored on the heap and passed by a read-only, shareable pointer; or *writable* ($\textcircled{\mathbb{W}}$), where they are stored on the heap and passed by a writable, linear pointer. As can be seen in Figure 3.16, the **bang** operator interacts with these sigils in much the same way as with abstract types. The **Share**, **Drop**, and **Escape** constraints place the same constraints on the sigils as with abstract types, with the added requirement that the type of each available field also satisfies the constraint in question.

If we wish to **put** a new value into a field that is marked as available (\circ) in the original record, the typing rules seem to indicate that we would have to **take** the

field out, discard it, and **put** in a new value. To avoid having to explicitly **take** out every field we wish to discard, we allow fields that satisfy **Drop** to be automatically discarded from a record via subtyping — almost a dual of the subtyping relation used for variants:



3.7.1 TYPE INFERENCE

The constraint generation rules for records are presented in Figure 3.7. As with variant types, we introduce *incomplete* record types, where unification variables temporarily stand for as-yet undetermined fields in the record.

We also allow unification variables to be used in place of sigils, and add constraints on these sigils for **take** and **put** expressions, as they are valid for \textcircled{U} nboxed and \textcircled{W} ritable records but not for \textcircled{R} ead-only ones. This way, we can express that we expect the field f to be present in the given record type as a simple subtyping constraint, leaving the rest of the record and the sigil as unknowns in the constraint generation rules for **take** and **put**.

3.8 CONSTRAINT GENERATION THEOREMS

The above sections have cumulatively introduced the complete constraint generation and typing relations. We now wish to state the theorems that connect these two relations.

As mentioned in Section 3.1, there are two main desirable theorems for constraint generation: *soundness*, which states that our generated constraints, if satisfiable, are sufficient to ensure a typing judgement; and *completeness*, which states that a well typed expression should produce satisfiable constraints. In addition to these two theorems, we also must prove a *totality* condition, that states the constraint generator will always generate a constraint for any closed expression, satisfiable or not. Without this result, our constraint generator could trivially satisfy the soundness and completeness properties by never generating constraints at all.

$$\begin{array}{c}
\boxed{G \vdash e : \tau \rightsquigarrow G' \mid C} \\
\dots \\
\frac{\alpha, \beta \text{ fresh} \quad G \vdash e : \{f^\circ : \tau \mid \alpha\} \beta \rightsquigarrow G' \mid C_1 \quad C_2 = \{f^\bullet : \tau \mid \alpha\} \beta \text{ Drop}}{G \vdash e.f : \tau \rightsquigarrow G' \mid C_1 \wedge C_2} \text{CG-MEMBER} \\
\\
\frac{\alpha, \beta, \gamma \text{ fresh} \quad G_1 \vdash e_1 : \{f^\circ : \beta \mid \alpha\} \gamma \rightsquigarrow G_2 \mid C_1 \quad x : \langle 0 \rangle \{f^\bullet : \beta \mid \alpha\} \gamma, y : \langle 0 \rangle \beta, G_2 \vdash e_2 : \tau \rightsquigarrow x : \langle n \rangle \{f^\bullet : \beta \mid \alpha\} \gamma, y : \langle m \rangle \beta, G_3 \mid C_2 \quad C_3 = \text{if } n = 0 \text{ then } \{f^\bullet : \beta \mid \alpha\} \gamma \text{ Drop else } \top \quad C_4 = \text{if } m = 0 \text{ then } \beta \text{ Drop else } \top \quad C_5 = \gamma \neq \mathbb{I}}{G_1 \vdash \text{take } x \{f = y\} = e_1 \text{ in } e_2 : \tau \rightsquigarrow G_3 \mid \bigwedge_{k \in 1 \dots 5} C_k} \text{CG-TAKE} \\
\\
\frac{\alpha, \beta, \gamma \text{ fresh} \quad G_1 \vdash e_1 : \{f^\bullet : \beta \mid \alpha\} \gamma \rightsquigarrow G_2 \mid C_1 \quad G_2 \vdash e_2 : \beta \rightsquigarrow G_3 \mid C_2 \quad C_3 = \{f^\circ : \beta \mid \alpha\} \gamma \sqsubseteq \tau \quad C_4 = \gamma \neq \mathbb{I}}{G_1 \vdash \text{put } e_1.f = e_2 : \tau \rightsquigarrow G_3 \mid C_1 \wedge C_2 \wedge C_3 \wedge C_4} \text{CG-PUT} \\
\\
\frac{\overline{\alpha}_i \text{ fresh} \quad G \vdash \overline{e}_i : \overline{\alpha}_i \rightsquigarrow G' \mid C}{G \vdash \#\{f_i = e_i\} : \tau \rightsquigarrow G' \mid C \wedge \{f_i^\circ : \alpha_i\} \textcircled{\text{U}} \sqsubseteq \tau} \text{CG-STRUCT} \\
\\
\boxed{G \vdash \overline{e}_i : \overline{\tau}_i \rightsquigarrow G' \mid C} \\
\frac{}{G \vdash e \rightsquigarrow G \mid \top} \text{CG-EMPTY} \quad \frac{G_1 \vdash e : \tau \rightsquigarrow G_2 \mid C_1 \quad G_2 \vdash \overline{e}_i : \overline{\tau}_i \rightsquigarrow G_3 \mid C_2}{G_1 \vdash e : \tau, \overline{e}_i : \overline{\tau}_i \rightsquigarrow G_3 \mid C_1 \wedge C_2} \text{CG-CONS}
\end{array}$$

Figure 3.17: Constraint generation rules for records

Unlike the other theorems presented in this thesis, most of these results have not yet been machine formalised. This is because, as we will see in Chapter 5, we separately synthesise a proof of well-typedness for use in our refinement framework, so these results are not a vital part of our machine-checked refinement certificate, but rather simply desirable properties of our type checking algorithm. Work is currently underway to develop a machine-checked proof of these properties, however, and connect these proofs to the same Cogent formalisation in Isabelle/HOL used for our refinement certificate.

3.8.1 SOUNDNESS

THEOREM 3.2 (Soundness of generation).

If we generate a constraint for a closed term, and $\varepsilon \vdash e : \tau \rightsquigarrow G' \mid C \mid e'$
there is an assignment where the constraint holds, $\wedge A \vdash S(C)$
then the term is well-typed under that assignment. $\Rightarrow A; \varepsilon \vdash S(e') : S(\tau)$

Proof. The proof proceeds by induction, but first the goal must be generalised to open expressions using an arbitrary input context G . Define $G|_{\bar{x}}$ to be the non-algorithmic context which contains just the typing assumptions from G for the variables \bar{x} . If we algorithmically generate a constraint for an expression e from the algorithmic context G , then the non-algorithmic context for typing e is just $G|_{FV(e)}$, where $FV(e)$ is all variables occurring free in e . Then, for a compound expression containing two sub-expressions, we may prove a lemma like the following, to establish the context splitting judgement used in the non-algorithmic typing rules:

LEMMA 3.1 (Splitting for subcontexts).

If a term is made of two sub-terms $FV(e) = FV(e_1) \cup FV(e_2)$
and we run constraint generation $\wedge G_1 \vdash e_1 : \tau \rightsquigarrow G_2 \mid C_1$
on both sub-terms in order, and the $\wedge G_2 \vdash e_2 : \rho \rightsquigarrow G_3 \mid C_2$
second constraint is satisfiable, then $\wedge A \vdash S(C_2)$
the typing context for the term splits into the contexts for the two sub-terms.
 $\Rightarrow A \vdash S(G|_{FV(e)}) \rightsquigarrow S(G_1|_{FV(e_1)}) \boxplus S(G_2|_{FV(e_2)})$

Note that we only need the second constraint C_2 to be satisfiable, as the constraint generator will only emit the **Share** constraints necessary to establish the correct context splitting judgement when running over the second expression e_2 , as that is where additional uses of the variables would be observed. With such lemmas (and

similar for **let** etc.), we can inductively prove the more general version of our goal, stated as follows.

LEMMA 3.2 (Soundness of generation for open terms).

*If under an algorithmic context G ,
we generate a constraint for an open term, $G \vdash e : \tau \rightsquigarrow G' \mid C \mid e'$
and the constraint is satisfiable, then the $\wedge A \vdash \mathcal{S}(C)$
term is well-typed under the typing context derived for that term from G .*

$$\Rightarrow A; \mathcal{S}(G|_{\text{FV}(e')}) \vdash \mathcal{S}(e') : \mathcal{S}(\tau)$$

Our overall theorem is an obvious corollary of Lemma 3.2, where the context $G = \varepsilon$ and $\text{FV}(e') = \varepsilon$. \square

3.8.2 COMPLETENESS

THEOREM 3.3 (Completeness of generation).

*If we generate a constraint for a closed term, and $\varepsilon \vdash e : \tau \rightsquigarrow G' \mid C \mid e'$
that term is well typed under an assignment, then $\wedge A; \varepsilon \vdash \mathcal{S}(e') : \mathcal{S}(\tau)$
the constraint is satisfiable under that assignment. $\Rightarrow A \vdash \mathcal{S}(C)$*

Proof. First, we must generalise our goal to account for open terms, to enable rule induction on the constraint generation:

LEMMA 3.3 (Completeness of generation for open terms).

*If under an algorithmic context G ,
we generate a constraint for a term $G \vdash e : \tau \rightsquigarrow G' \mid C \mid e'$
which is well typed under an assignment S , $\wedge A; \mathcal{S}(G|_{\text{FV}(e')}) \vdash \mathcal{S}(e') : \mathcal{S}(\tau)$
then the constraint is satisfiable under S . $\Rightarrow A \vdash \mathcal{S}(C)$*

Observe that because our elaborated expression e' contains a type signature for each sub-expression in e , in order for the expression $\mathcal{S}(e')$ to be well typed, the assignment S must contain a substitution for each unification variable in C . In each rule induction case, the validity of the constraints can be established by inversion on the well-typedness assumption. \square

3.8.3 TOTALITY

THEOREM 3.4 (Totality of constraint generation).

The constraint generator will produce a constraint for any closed term.

$$\text{FV}(e) = \emptyset \Rightarrow \exists G' C. \varepsilon \vdash e : \tau \rightsquigarrow G' \mid C$$

Proof. Seeing as the constraint generator is defined for all expressions, only placing preconditions on the input context, this proof proceeds entirely straightforwardly by induction on the expression e , after a suitable generalisation to open terms, where the precondition on the input context is made explicit.

LEMMA 3.4 (Totality of constraint generation on open terms).

If all variables in the input term occur in the context G , then the constraint generator will produce a constraint for that term under G .

$$\text{FV}(e) \subseteq \{x \mid \exists \tau \text{ n. } x :_{\langle n \rangle} \tau \in G\} \Rightarrow \exists G' \text{ C. } G \vdash e : \tau \rightsquigarrow G' \mid C$$

The proof of the top-level theorem is a simple instantiation of this lemma where $G = \varepsilon$ and e is closed. \square

3.9 POLYMORPHISM

Because of the restricted polymorphism in Cogent, we may consider each polymorphic type variable in a particular top-level definition as a distinct skolem type with no dependencies on other variables. This means they can be treated just as concrete types in unification, and are global throughout constraints. Thus, constraints need not contain binders for polymorphic type variables, and any requirements on type variables to satisfy constraints can be expressed on the top-level: the assumptions A for the constraint semantics and typing rules.

We prove that instantiating polymorphic type variables does not make well-typed terms ill-typed, nor does it make satisfiable type constraints unsatisfiable. These theorems are useful for showing type preservation in Chapter 4, and for the correctness of the monomorphisation phase of our refinement framework described in Chapter 5.

To clarify notation, an *assignment* or *unifier*, written as S , is a substitution applied to *unification variables* such as $\{\alpha := U8\}$, and is generated by the constraint solver. The *substitutions* described in this section, written as σ , are applied to *polymorphic type variables*, such as $[^{U16}/_a]$, and are introduced when type applications are used (TAPP) and when the program is monomorphised in the Cogent compilation validation framework (see Chapter 5).

THEOREM 3.5 (Instantiating Type Variables in Constraints).

$$\begin{array}{ll} \text{Given a constraint that holds under assumptions } A, & A \vdash C \\ \text{and a substitution to type variables that satisfies } A, & \wedge \varepsilon \vdash \bigwedge_{C_i \in A} \sigma(C_i) \\ \text{then the substituted constraint also holds.} & \Rightarrow \varepsilon \vdash \sigma(C) \end{array}$$

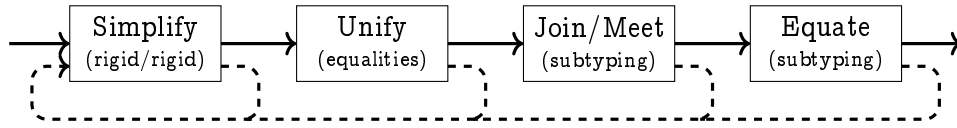


Figure 3.18: Solver data flow

Proof. Straightforward rule induction on the assumption $A \vdash C$. Wherever the rule ASM is used, we refer to the second assumption to justify the validity of the substituted constraint. Whenever observer type variables (a!) are substituted, we make use of Theorem 3.1. \square

THEOREM 3.6 (Instantiating Type Variables — instantiation).

<i>Given a term typed under requirements A,</i>	$A; \Gamma \vdash e : \tau$
<i>and a substitution to type variables that satisfies A,</i>	$\wedge \varepsilon \vdash \bigwedge_{C_i \in A} \sigma(C_i)$
<i>then the substituted term is also well-typed.</i>	$\Rightarrow \varepsilon; \sigma(\Gamma) \vdash \sigma(e) : \sigma(\tau)$

Proof. Rule induction, using Theorem 3.5. \square

3.10 CONSTRAINT SOLVER

Having generated a complete set of constraints to model a typing problem, we now turn our attention to the solving of these constraints. The solver's task is, given constraint C and a set of axioms about type variables A , to determine an assignment \mathcal{S} to all unification variables in C such that C is satisfied. We need the solver to be *sound*, in that it should not ever successfully solve an unsatisfiable set of constraints, but we do not require *completeness* — that the solver will always find a solution if one exists.

Figure 3.18 outlines the basic flow of data through the various phases of the solver. The solver consists of several phases, each of which transform the set of constraints to be solved in some way. Broadly, there are four phases in the solver, each operating on a set of constraints:

- The *simplify* phase breaks down *rigid/rigid* subtyping and equality constraints into smaller equisatisfiable constraints. A subtyping constraint $\tau_1 \sqsubseteq \tau_2$ is considered *rigid/rigid* if neither τ_1 nor τ_2 are unification variables, or type operators applied to unification variables (*mutatis mutandis* for equality constraints). The simplifier also eliminates all satisfiable constraints that do not contain any unification variables, such as **Share** or **Drop** constraints on concrete types.

- The *unify* phase solves any *flex/rigid* equality constraints — i.e. those constraints of the form $\alpha \approx \tau$ (or $\tau \approx \alpha$) where τ is not a unification variable — by substituting away the unification variable α . The combination of the first two phases is approximately equivalent to the standard first order unification of Robinson [115], extended for row variables [43].
- The *join/meet* phase examines systems of *flex/rigid* subtyping constraints and, by using the lattice structure of our subtyping system, generates a new set of constraints such that the same unification variable does not occur on the left (resp. right) hand side of multiple *flex/rigid* subtyping constraints.
- The *equate* phase locates *flex/rigid* subtyping constraints of the form $\alpha \sqsubseteq \tau$ (or $\tau \sqsubseteq \alpha$) where α does not occur on the left (resp. right) side of any other subtyping constraint. Such constraints can be safely replaced with an equality $\alpha \approx \tau$ and therefore soluble by the *unify* phase.

We run the solver to a fixed point — that is, we only move to the next phase if the current phase does not change the set of constraints in any way. If the constraint set is altered at any point, then we return to the first phase. If the constraint set is satisfiable, we will ideally be left with a set of constraints that is a subset of the given axioms A when the algorithm finishes.

Crucially for soundness of this algorithm, all phases but the *unify* phase produce output constraints that imply the input constraint under any assignment. That is, if the output is satisfiable under some assignment, then the input is also satisfiable under the same assignment. And, as the *unify* phase only substitutes for equality constraints, the output still directly implies the input under such a substitution.

In this presentation of the algorithm, unsatisfiable constraints are simply left over after solving, however in the real implementation, constraints that are clearly unsatisfiable lead to informative error messages.

3.10.1 THE SIMPLIFY PHASE

Constraints on totally rigid terms (that is, terms that do not contain any unification variables) do not provide any useful information for the solver to reduce any other constraint. Therefore, if they are satisfiable, they can be discarded. Moreover, constraints on rigid terms that contain unification variables inside, for example a subtyping constraint like $(\alpha \rightarrow U8) \sqsubseteq (U16 \rightarrow \beta)$, are not immediately useful to reduce

any constraints, but the equisatisfiable constraints that directly concern the unification variables $U16 \sqsubseteq \alpha$ and $U8 \sqsubseteq \beta$ could lead to solutions for α and β .

Therefore, the simplifier's task is to break down constraints on rigid terms into constraints directly on the unification variables (if any) that occur inside. Figure 3.19 outlines the rules for basic constraints, whereas Figure 3.20 outlines the rules for equality and subtyping constraints. The simplification relation $\xrightarrow{\text{simp}}$ maps a single constraint to a set of constraints. It may be applied to any applicable constraint in the input set. If any constraint is simplified, then the phase runs again (see Figure 3.18), so this simplification is applied as much as possible to all constraints.

As can be seen in Figure 3.20, subtyping constraints are converted to equality constraints if they involve *unordered* types. An *unordered* type is simply a type for which subtyping does not apply, such as a primitive type, an abstract type, or a type variable. For such types, subtyping is equivalent to equality.

For equality and subtyping constraints which involve unknown rows of variant alternatives or record fields, we separate those constructors or fields that the two types have in common into a separate constraint. This simplifies the unification mechanism used in the next phase for row unification variables.

LEMMA 3.5 (Soundness of simplifier).

If a constraint simplifies to a set of constraints, $C \xrightarrow{\text{simp}} \overline{C'}$
then the conjunction of the set is equivalent $\Rightarrow A \vdash S(C) \Leftrightarrow A \vdash S(\bigwedge \overline{C'})$
under any assignment to unification variables.

Proof. Proceeds by cases on the simplification relation. All cases are discharged relatively straightforwardly from the definition of the constraint semantics in Figures 3.6, 3.9, 3.12 and 3.16. \square

3.10.2 THE UNIFY PHASE

After the simplification phase has completed, all rigid/rigid equality constraints ought to be eliminated, and constraints involving row variables will have common alternatives or fields removed. Thus, we only need to concern ourselves with constraints are of the following forms:

- A flex/rigid equality constraint such as $\alpha \approx \tau$. Here, the type checker replaces α with τ if α does not occur in τ , just as in standard first-order unification [115], and adds the assignment $\alpha := \tau$ to the output assignment S of the solver.

$C \xrightarrow{\text{simp}} \bar{C}$	
$\tau \hookrightarrow \tau'$	$\ell < \mathbb{T} $
$C[\tau] \xrightarrow{\text{simp}} C[\tau']$	$\ell \in \mathbb{T} \xrightarrow{\text{simp}} \varepsilon$
$C_1 \wedge C_2$	$\xrightarrow{\text{simp}} C_1, C_2$
\top	$\xrightarrow{\text{simp}} \varepsilon$
$\textcircled{w} \neq \textcircled{r}$	$\xrightarrow{\text{simp}} \varepsilon$
$\textcircled{u} \neq \textcircled{r}$	$\xrightarrow{\text{simp}} \varepsilon$
$\tau \approx \tau$	$\xrightarrow{\text{simp}} \varepsilon$
\top Drop	$\xrightarrow{\text{simp}} \varepsilon$
\top Share	$\xrightarrow{\text{simp}} \varepsilon$
\top Escape	$\xrightarrow{\text{simp}} \varepsilon$
$\tau \rightarrow \rho$ Drop	$\xrightarrow{\text{simp}} \varepsilon$
$\tau \rightarrow \rho$ Share	$\xrightarrow{\text{simp}} \varepsilon$
$\tau \rightarrow \rho$ Escape	$\xrightarrow{\text{simp}} \varepsilon$
$\alpha!$ Drop	$\xrightarrow{\text{simp}} \varepsilon$
$\alpha!$ Share	$\xrightarrow{\text{simp}} \varepsilon$
$\langle \bar{K}_i^\circ \tau_i, \bar{K}_j^\bullet \tau_j \rangle$ Share	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Share}}$
$\langle \bar{K}_i^\circ \tau_i, \bar{K}_j^\bullet \tau_j \rangle$ Drop	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Drop}}$
$\langle \bar{K}_i^\circ \tau_i, \bar{K}_j^\bullet \tau_j \rangle$ Escape	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Escape}}$
$A \bar{\tau}_i \textcircled{r}$ Drop	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Drop}}$
$A \bar{\tau}_i \textcircled{r}$ Share	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Share}}$
$A \bar{\tau}_i \textcircled{w}$ Escape	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Escape}}$
$A \bar{\tau}_i \textcircled{u}$ Drop	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Drop}}$
$A \bar{\tau}_i \textcircled{u}$ Share	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Share}}$
$A \bar{\tau}_i \textcircled{u}$ Escape	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Escape}}$
$\{\bar{f}_i^\circ : \tau_i, \bar{f}_j^\bullet : \tau_j\}$ \textcircled{r} Share	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Share}}$
$\{\bar{f}_i^\circ : \tau_i, \bar{f}_j^\bullet : \tau_j\}$ \textcircled{r} Drop	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Drop}}$
$\{\bar{f}_i^\circ : \tau_i, \bar{f}_j^\bullet : \tau_j\}$ \textcircled{w} Escape	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Escape}}$
$\{\bar{f}_i^\circ : \tau_i, \bar{f}_j^\bullet : \tau_j\}$ \textcircled{u} Share	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Share}}$
$\{\bar{f}_i^\circ : \tau_i, \bar{f}_j^\bullet : \tau_j\}$ \textcircled{u} Drop	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Drop}}$
$\{\bar{f}_i^\circ : \tau_i, \bar{f}_j^\bullet : \tau_j\}$ \textcircled{u} Escape	$\xrightarrow{\text{simp}} \overline{\tau_i \text{ Escape}}$
$\langle \bar{K}_j^\bullet \tau_j \rangle$ Exhausted	$\xrightarrow{\text{simp}} \varepsilon$

Figure 3.19: Basic simplification rules of constraint solving.

$C \xrightarrow{\text{simp}} \overline{C}$	
...	
τ unordered	τ unordered
$\tau \sqsubseteq \rho \xrightarrow{\text{simp}} \tau \approx \rho$	$\rho \sqsubseteq \tau \xrightarrow{\text{simp}} \rho \approx \tau$
$\tau_1 \rightarrow \tau_2 \sqsubseteq \rho_1 \rightarrow \rho_2$	$\xrightarrow{\text{simp}} \rho_1 \sqsubseteq \tau_1, \tau_2 \sqsubseteq \rho_2$
$\tau_1 \rightarrow \tau_2 \approx \rho_1 \rightarrow \rho_2$	$\xrightarrow{\text{simp}} \rho_1 \approx \tau_1, \tau_2 \approx \rho_2$
$A \overline{\tau_i} s \approx A \overline{\rho_i} s$	$\xrightarrow{\text{simp}} \overline{\tau_i} \approx \overline{\rho_i}$
$\langle \overline{K_i^\bullet \tau_i}, \overline{K_j^u \tau_j} \rangle \sqsubseteq \langle \overline{K_i^\circ \rho_i}, \overline{K_j^u \rho_j} \rangle$	$\xrightarrow{\text{simp}} \overline{\tau_i} \sqsubseteq \overline{\rho_i}, \overline{\tau_j} \sqsubseteq \overline{\rho_j}$
$\{\overline{f_i^\circ : \tau_i}, \overline{f_j^u : \tau_j}\} s \sqsubseteq \{\overline{f_i^\bullet : \rho_i}, \overline{f_j^u : \rho_j}\} s$	$\xrightarrow{\text{simp}} \overline{\tau_i} \sqsubseteq \overline{\rho_i}, \overline{\tau_j} \sqsubseteq \overline{\rho_j}, \tau_i \text{ Drop}$
$\langle \overline{K_i^u \tau_i}, \overline{K_j^u \tau_j} \mid \alpha \rangle \sqsubseteq \langle \overline{K_i^u \rho_i}, \overline{K_k^u \rho_j} \mid \beta \rangle$	$\xrightarrow{\text{simp}} \langle \overline{K_i^u \tau_i} \rangle \sqsubseteq \langle \overline{K_i^u \rho_i} \rangle,$ $\langle \overline{K_j^u \tau_j} \mid \alpha \rangle \sqsubseteq \langle \overline{K_k^u \rho_j} \mid \beta \rangle$
$\{\overline{f_i^u : \tau_i}, \overline{f_j^u : \tau_j} \mid \alpha \rangle s \sqsubseteq \{\overline{f_i^u : \rho_i}, \overline{f_k^u : \rho_k} \mid \beta \rangle s'$	$\xrightarrow{\text{simp}} \{\overline{f_i^u : \tau_i}\} s \sqsubseteq \{\overline{f_i^u : \rho_i}\} s',$ $\{\overline{f_j^u : \tau_j} \mid \alpha \rangle s \sqsubseteq \{\overline{f_k^u : \rho_k} \mid \beta \rangle s'$
$\langle \overline{K_i^u \tau_i} \mid \alpha \rangle \sqsubseteq \langle \overline{K_i^u \rho_i}, \overline{K_j^u \rho_j} \rangle$	$\xrightarrow{\text{simp}} \langle \overline{K_i^u \tau_i} \rangle \sqsubseteq \langle \overline{K_i^u \rho_i} \rangle,$ $\langle \varepsilon \mid \alpha \rangle \sqsubseteq \langle \overline{K_j^u \rho_j} \rangle$
$\{\overline{f_i^u : \tau_i} \mid \alpha \rangle s \sqsubseteq \{\overline{f_i^u : \rho_i}, \overline{f_j^u : \rho_j}\} s'$	$\xrightarrow{\text{simp}} \{\overline{f_i^u : \tau_i}\} s \sqsubseteq \{\overline{f_i^u : \rho_i}\} s',$ $\langle \varepsilon \mid \alpha \rangle s \sqsubseteq \{\overline{f_j^u : \rho_j} \mid \beta \rangle s'$
$\langle \overline{K_i^u \tau_i}, \overline{K_j^u \tau_j} \rangle \sqsubseteq \langle \overline{K_i^u \rho_i} \mid \beta \rangle$	$\xrightarrow{\text{simp}} \langle \overline{K_i^u \tau_i} \rangle \sqsubseteq \langle \overline{K_i^u \rho_i} \rangle,$ $\langle \overline{K_j^u \tau_j} \mid \alpha \rangle \sqsubseteq \langle \varepsilon \mid \beta \rangle$
$\{\overline{f_i^u : \tau_i}, \overline{f_j^u : \tau_j} \mid \alpha \rangle s \sqsubseteq \{\overline{f_i^u : \rho_i} \mid \beta \rangle s'$	$\xrightarrow{\text{simp}} \{\overline{f_i^u : \tau_i}\} s \sqsubseteq \{\overline{f_i^u : \rho_i}\} s',$ $\{\overline{f_j^u : \tau_j} \mid \alpha \rangle s \sqsubseteq \langle \varepsilon \mid \beta \rangle s'$
$\langle \overline{K_j^u \tau_i} \rangle \approx \langle \overline{K_j^u \rho_i} \rangle$	$\xrightarrow{\text{simp}} \overline{\tau_i} \approx \overline{\rho_i}$
$\{\overline{f_i^u : \tau_i}\} s \approx \{\overline{f_i^u : \rho_i}\} s$	$\xrightarrow{\text{simp}} \overline{\tau_i} \approx \overline{\rho_i}$
$\langle \overline{K_i^u \tau_i}, \overline{K_j^u \tau_j} \mid \alpha \rangle \approx \langle \overline{K_i^u \rho_i}, \overline{K_k^u \rho_j} \mid \beta \rangle$	$\xrightarrow{\text{simp}} \langle \overline{K_i^u \tau_i} \rangle \approx \langle \overline{K_i^u \rho_i} \rangle,$ $\langle \overline{K_j^u \tau_j} \mid \alpha \rangle \approx \langle \overline{K_k^u \rho_j} \mid \beta \rangle$
$\{\overline{f_i^u : \tau_i}, \overline{f_j^u : \tau_j} \mid \alpha \rangle s \approx \{\overline{f_i^u : \rho_i}, \overline{f_k^u : \rho_k} \mid \beta \rangle s'$	$\xrightarrow{\text{simp}} \{\overline{f_i^u : \tau_i}\} s \approx \{\overline{f_i^u : \rho_i}\} s',$ $\{\overline{f_j^u : \tau_j} \mid \alpha \rangle s \approx \{\overline{f_k^u : \rho_k} \mid \beta \rangle s'$
$\langle \overline{K_i^u \tau_i} \mid \alpha \rangle \approx \langle \overline{K_i^u \rho_i}, \overline{K_j^u \rho_j} \rangle$	$\xrightarrow{\text{simp}} \langle \overline{K_i^u \tau_i} \rangle \approx \langle \overline{K_i^u \rho_i} \rangle,$ $\langle \varepsilon \mid \alpha \rangle \approx \langle \overline{K_j^u \rho_j} \rangle$
$\{\overline{f_i^u : \tau_i} \mid \alpha \rangle s \approx \{\overline{f_i^u : \rho_i}, \overline{f_j^u : \rho_j}\} s'$	$\xrightarrow{\text{simp}} \{\overline{f_i^u : \tau_i}\} s \approx \{\overline{f_i^u : \rho_i}\} s',$ $\langle \varepsilon \mid \alpha \rangle s \approx \{\overline{f_j^u : \rho_j} \mid \beta \rangle s'$
$\langle \overline{K_i^u \tau_i}, \overline{K_j^u \tau_j} \rangle \approx \langle \overline{K_i^u \rho_i} \mid \beta \rangle$	$\xrightarrow{\text{simp}} \langle \overline{K_i^u \tau_i} \rangle \approx \langle \overline{K_i^u \rho_i} \rangle,$ $\langle \overline{K_j^u \tau_j} \mid \alpha \rangle \approx \langle \varepsilon \mid \beta \rangle$
$\{\overline{f_i^u : \tau_i}, \overline{f_j^u : \tau_j} \mid \alpha \rangle s \approx \{\overline{f_i^u : \rho_i} \mid \beta \rangle s'$	$\xrightarrow{\text{simp}} \{\overline{f_i^u : \tau_i}\} s \approx \{\overline{f_i^u : \rho_i}\} s',$ $\{\overline{f_j^u : \tau_j} \mid \alpha \rangle s \approx \langle \varepsilon \mid \beta \rangle s'$

Figure 3.20: Simplification rules for equality and subtyping.

- An equality constraint involving an incomplete variant or record on one side only, such as $\langle \varepsilon \mid \alpha \rangle \approx \langle \overline{K}_i^u \tau_i \rangle$ or $\{\varepsilon \mid \alpha\} \approx \{f_i :^u \tau_i\}$. Note that we know that the side involving the row variable must have no concrete fields or alternatives specified as the previous phase eliminates any common components of the two types. If concrete fields or alternatives occur on the incomplete side, that indicates that the constraint is unsatisfiable. Thus, it is safe to simply apply the row substitution $\alpha := \overline{K}_i^u \tau_i$ or $\alpha := \overline{f}_i :^u \tau_i$ to the constraint set, adding it to the output assignment \mathcal{S} .
- An equality constraint involving incomplete variants or records on both sides, such as $\langle \overline{K}_i^u \tau_i \mid \alpha \rangle \approx \langle \overline{K}_j^u \tau_j \mid \beta \rangle$ where $\alpha \neq \beta$, and similarly for record types. We know that \overline{K}_i and \overline{K}_j have no constructors in common from the previous phase so, as the two variables differ, it suffices to introduce a fresh row variable γ and apply the substitutions $\alpha := \overline{K}_j^u \tau_j \mid \gamma$ and $\beta := \overline{K}_i^u \tau_i \mid \gamma$, adding them to \mathcal{S} . Here we use γ to denote alternatives not yet determined, common to both α and β . The same technique applies for record types and their fields.
- Any equality or subtyping constraint on records where one of the sigils is unknown, e.g. $\{\dots\} \alpha \sqsubseteq \{\dots\} \textcircled{w}$. Here, the unknown sigil is simply substituted for the sigil on the opposite type i.e. $\alpha := \textcircled{w}$ in this example.
- By symmetry, the above cases with the left and right hand sides swapped.

To guarantee soundness for this phase, we merely need to ensure that an unsatisfiable constraint set is never transformed into a satisfiable one. Seeing as the only transformation made by the assign phase is to instantiate a unification variable to a concrete type via substitution, this is shown straightforwardly:

LEMMA 3.6 (Soundness of unify phase).

If a substitution is applied to a constraint set, $\overline{C}' = \overline{C}[x := \tau]$
and the result is satisfiable under some assignment, $\wedge A \vdash \mathcal{S}(\wedge \overline{C}')$
then adding the substitution to that assignment gives $\wedge \mathcal{S}' = \mathcal{S} \cup \{x := \tau\}$
a satisfying assignment for the original constraints. $\Rightarrow A \vdash \mathcal{S}'(\wedge \overline{C})$

Proof. Seeing as applying an assignment $\mathcal{S}(C)$ is definitionally identical to applying each substitution in \mathcal{S} to C , adding the substitution $x := \tau$ to the satisfying assignment \mathcal{S} of the result constraint set \overline{C}' and applying it to the original constraint set \overline{C} is definitionally identical to substituting $x := \tau$ in \overline{C} and applying \mathcal{S} . \square

3.10.3 THE JOIN/MEET PHASE

After the first two phases have been executed, all rigid/rigid subtyping constraints have been eliminated, as have equality constraints. What remains is predominantly subtyping constraints which can be viewed as a graph, where the nodes in the graph are types, and the edges are the subtyping constraints between them.

This graph will consist of clusters of unification variables, with rigid terms at the boundaries of those clusters. Figure 3.21 shows an example of such a cluster.

The join/meet phase works on those cases where the same unification variable is constrained above or below by two or more rigid types. For example, the first join/meet phase in Figure 3.21 operates on the two constraints $\tau_2 \sqsubseteq \alpha_2$ and $\tau_3 \sqsubseteq \alpha_2$. Because of the lattice structure of our subtyping system, any two types have a *least upper bound* $\tau_2 \sqcup \tau_3$. Seeing as α_2 is constrained to be a supertype of both τ_2 and τ_3 if and only if it is a supertype of $\tau_2 \sqcup \tau_3$, these two subtyping constraints can be replaced with one, thereby overall reducing the complexity of the overall constraint set, and reducing the number of flex/rigid constraints concerning each unification variable. Once the number of constraints bounding each variable below is reduced to one, the subtyping constraint is safely replaced with an equality constraint by the subsequent *equate* phase, which can then be solved by substitution in the *unify* phase. Figure 3.21 demonstrates the interaction between these phases, illustrating the constraint set after each run of the *equate* and *join/meet* phases.

Just as two constraints that bound a variable below can be replaced with the least upper bound of the two rigid subtypes, two constraints that bound a variable above are replaced with the greatest lower bound of the two rigid supertypes. Figure 3.22 shows all rules for computing the greatest lower bound (read left to right), whereas Figure 3.23 shows all rules for computing the least upper bound.

We define the join/meet phase as a relation $C_1, C_2 \xrightarrow{j/m} \bar{C}$. Given two constraints that bound the same unification variable above, i.e. $\alpha \sqsubseteq \tau_1$ and $\alpha \sqsubseteq \tau_2$, we compute the greatest lower bound $\tau_1 \sqcap \tau_2$, and emit the three constraints: $\tau_1 \sqcap \tau_2 \sqsubseteq \tau_1$, $\tau_1 \sqcap \tau_2 \sqsubseteq \tau_2$, and $\alpha \sqsubseteq \tau_1 \sqcap \tau_2$. Similarly, if the variable is bound below, we compute the least upper bound. Note that now α is constrained by fewer subtyping constraints, making it more amenable to the subsequent *equate* phase.

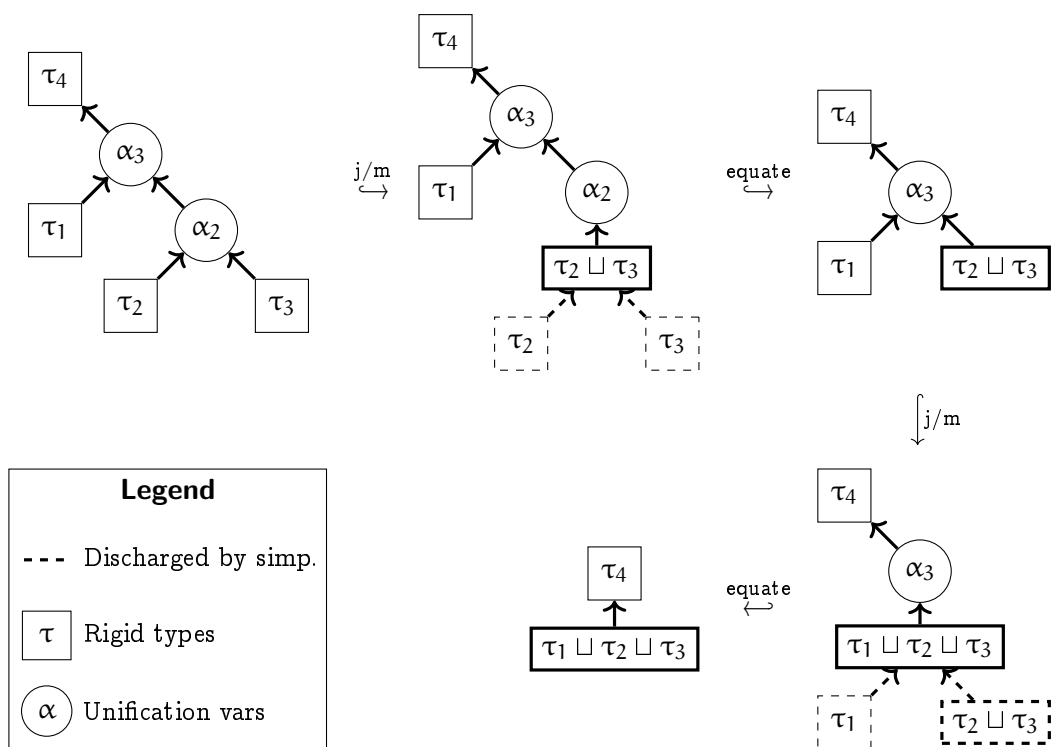


Figure 3.21: Example of the interaction of the join/meet and equate phases.

$$\begin{array}{c}
\text{MEET-FUN:} \\
\tau_1 \rightarrow \tau_2 \Downarrow \\
\rho_1 \rightarrow \rho_2 \Downarrow \beta_1 \rightarrow \beta_2 \\
(\beta_1, \beta_2 \text{ fresh})
\end{array}
\qquad
\begin{array}{c}
\text{MEET-VAR:} \\
\langle \overline{K_i^u \tau_i}, \overline{K_j^o \tau_j}, \overline{K_k^\bullet \tau_k} \rangle \Downarrow \\
\langle \overline{K_i^u \rho_i}, \overline{K_j^\bullet \rho_j}, \overline{K_k^o \rho_k} \rangle \Downarrow \langle \overline{K_i^u \beta_i}, \overline{K_j^\bullet \beta_j}, \overline{K_k^o \beta_k} \rangle \\
(\overline{\beta_i}, \overline{\beta_j}, \overline{\beta_k} \text{ fresh})
\end{array}$$

$$\begin{array}{c}
\text{MEET-REC:} \\
\{ \overline{f_i^u : \tau_i}, \overline{f_j^o : \tau_j}, \overline{f_k^\bullet : \tau_k} \} s \Downarrow \\
\{ \overline{f_i^u : \rho_i}, \overline{f_j^\bullet : \rho_j}, \overline{f_k^o : \rho_k} \} s' \Downarrow \{ \overline{f_i^u : \beta_i}, \overline{f_j^o : \beta_j}, \overline{f_k^\bullet : \beta_k} \} \gamma \\
(\overline{\beta_i}, \overline{\beta_j}, \overline{\beta_k}, \gamma \text{ fresh})
\end{array}$$

$$\begin{array}{c}
\text{MEET-REC-U:} \\
\{ \overline{f_i^u : \tau_i}, \overline{f_j^o : \tau_j}, \overline{f_k^\bullet : \tau_k} \mid \alpha \} s \Downarrow \\
\{ \overline{f_i^u : \rho_i}, \overline{f_j^\bullet : \rho_j}, \overline{f_k^o : \rho_k}, \overline{f_m^u : \rho_m} \} s' \Downarrow \{ \overline{f_i^u : \beta_i}, \overline{f_j^o : \beta_j}, \overline{f_k^\bullet : \beta_k}, \overline{f_m^u : \beta_m} \} \gamma \\
(\overline{\beta_i}, \overline{\beta_j}, \overline{\beta_k}, \overline{\beta_m}, \gamma \text{ fresh})
\end{array}$$

$$\begin{array}{c}
\text{MEET-REC-UU:} \\
\{ \overline{f_i^u : \tau_i}, \overline{f_j^o : \tau_j}, \overline{f_k^\bullet : \tau_k}, \overline{f_n^u : \tau_n} \mid \alpha_1 \} s \Downarrow \\
\{ \overline{f_i^u : \rho_i}, \overline{f_j^\bullet : \rho_j}, \overline{f_k^o : \rho_k}, \overline{f_m^u : \rho_m} \mid \alpha_2 \} s' \Downarrow \{ \overline{f_i^u : \beta_i}, \overline{f_j^o : \beta_j}, \overline{f_k^\bullet : \beta_k}, \overline{f_m^u : \beta_m}, \overline{f_n^u : \beta_n} \mid \alpha \} \gamma \\
(\overline{\beta_i}, \overline{\beta_j}, \overline{\beta_k}, \overline{\beta_m}, \overline{\beta_n}, \alpha, \gamma \text{ fresh})
\end{array}$$

$$\begin{array}{c}
\text{MEET-VAR-U:} \\
\langle \overline{K_i^u \tau_i}, \overline{K_j^o \tau_j}, \overline{K_k^\bullet \tau_k} \mid \alpha \rangle \Downarrow \\
\langle \overline{K_i^u \rho_i}, \overline{K_j^\bullet \rho_j}, \overline{K_k^o \rho_k}, \overline{K_m^u \rho_m} \rangle \Downarrow \langle \overline{K_i^u \beta_i}, \overline{K_j^\bullet \beta_j}, \overline{K_k^o \beta_k}, \overline{K_m^u \beta_m} \rangle \\
(\overline{\beta_i}, \overline{\beta_j}, \overline{\beta_k}, \overline{\beta_m} \text{ fresh})
\end{array}$$

$$\begin{array}{c}
\text{MEET-VAR-UU:} \\
\langle \overline{K_i^u \tau_i}, \overline{K_j^o \tau_j}, \overline{K_k^\bullet \tau_k}, \overline{K_n^u \tau_n} \mid \alpha_1 \rangle \Downarrow \\
\langle \overline{K_i^u \rho_i}, \overline{K_j^\bullet \rho_j}, \overline{K_k^o \rho_k}, \overline{K_m^u \rho_m} \mid \alpha_2 \rangle \Downarrow \langle \overline{K_i^u \beta_i}, \overline{K_j^\bullet \beta_j}, \overline{K_k^o \beta_k}, \overline{K_m^u \beta_m}, \overline{K_n^u \beta_n} \mid \alpha \rangle \\
(\overline{\beta_i}, \overline{\beta_j}, \overline{\beta_k}, \overline{\beta_m}, \alpha \text{ fresh})
\end{array}$$

Figure 3.22: Meet rules of constraint solving.

$$\begin{array}{c}
\text{JOIN-FUN:} \\
\tau_1 \rightarrow \tau_2 \sqsubseteq \beta_1 \rightarrow \beta_2 \\
\rho_1 \rightarrow \rho_2 \sqsubseteq \beta_1 \rightarrow \beta_2 \\
(\beta_1, \beta_2 \text{ fresh})
\end{array}
\qquad
\begin{array}{c}
\text{JOIN-VAR:} \\
\langle \overline{K_i^u \tau_i}, \overline{K_j^o \tau_j}, \overline{K_k^\bullet \tau_k} \rangle \sqsubseteq \langle \overline{K_i^u \beta_i}, \overline{K_j^o \beta_j}, \overline{K_k^o \beta_k} \rangle \\
\langle \overline{K_i^u \rho_i}, \overline{K_j^\bullet \rho_j}, \overline{K_k^o \rho_k} \rangle \sqsubseteq \langle \overline{K_i^u \beta_i}, \overline{K_j^\bullet \beta_j}, \overline{K_k^o \beta_k} \rangle \\
(\overline{\beta_i}, \overline{\beta_j}, \overline{\beta_k} \text{ fresh})
\end{array}$$

$$\begin{array}{c}
\text{JOIN-REC:} \\
\{ \overline{f_i^u : \tau_i}, \overline{f_j^o : \tau_j}, \overline{f_k^\bullet : \tau_k} \} s \sqsubseteq \{ \overline{f_i^u : \beta_i}, \overline{f_j^\bullet : \beta_j}, \overline{f_k^\bullet : \beta_k} \} \gamma \\
\{ \overline{f_i^u : \rho_i}, \overline{f_j^\bullet : \rho_j}, \overline{f_k^o : \rho_k} \} s' \sqsubseteq \{ \overline{f_i^u : \beta_i}, \overline{f_j^\bullet : \beta_j}, \overline{f_k^o : \beta_k} \} \gamma \\
(\overline{\beta_i}, \overline{\beta_j}, \overline{\beta_k}, \gamma \text{ fresh})
\end{array}$$

$$\begin{array}{c}
\text{JOIN-REC-U:} \\
\{ \overline{f_i^u : \tau_i}, \overline{f_j^o : \tau_j}, \overline{f_k^\bullet : \tau_k} \mid \alpha \} s \sqsubseteq \{ \overline{f_i^u : \beta_i}, \overline{f_j^\bullet : \beta_j}, \overline{f_k^\bullet : \beta_k}, \overline{f_m^u : \beta_m} \} \gamma \\
\{ \overline{f_i^u : \rho_i}, \overline{f_j^\bullet : \rho_j}, \overline{f_k^o : \rho_k}, \overline{f_m^u : \rho_m} \} s' \sqsubseteq \{ \overline{f_i^u : \beta_i}, \overline{f_j^\bullet : \beta_j}, \overline{f_k^o : \beta_k}, \overline{f_m^u : \beta_m} \} \gamma \\
(\overline{\beta_i}, \overline{\beta_j}, \overline{\beta_k}, \overline{\beta_m}, \gamma \text{ fresh})
\end{array}$$

$$\begin{array}{c}
\text{JOIN-REC-UU:} \\
\{ \overline{f_i^u : \tau_i}, \overline{f_j^o : \tau_j}, \overline{f_k^\bullet : \tau_k}, \overline{f_n^u : \tau_n} \mid \alpha_1 \} s \sqsubseteq \{ \overline{f_i^u : \beta_i}, \overline{f_j^\bullet : \beta_j}, \overline{f_k^\bullet : \beta_k}, \overline{f_m^u : \beta_m}, \overline{f_n^u : \beta_n} \} \alpha \\
\{ \overline{f_i^u : \rho_i}, \overline{f_j^\bullet : \rho_j}, \overline{f_k^o : \rho_k}, \overline{f_m^u : \rho_m} \mid \alpha_2 \} s' \sqsubseteq \{ \overline{f_i^u : \beta_i}, \overline{f_j^\bullet : \beta_j}, \overline{f_k^o : \beta_k}, \overline{f_m^u : \beta_m}, \overline{f_n^u : \beta_n} \} \alpha \\
(\overline{\beta_i}, \overline{\beta_j}, \overline{\beta_k}, \overline{\beta_m}, \overline{\beta_n}, \alpha, \gamma \text{ fresh})
\end{array}$$

$$\begin{array}{c}
\text{JOIN-VAR-U:} \\
\langle \overline{K_i^u \tau_i}, \overline{K_j^o \tau_j}, \overline{K_k^\bullet \tau_k} \mid \alpha \rangle \sqsubseteq \langle \overline{K_i^u \beta_i}, \overline{K_j^o \beta_j}, \overline{K_k^o \beta_k}, \overline{K_m^u \beta_m} \rangle \\
\langle \overline{K_i^u \rho_i}, \overline{K_j^\bullet \rho_j}, \overline{K_k^o \rho_k}, \overline{K_m^u \rho_m} \rangle \sqsubseteq \langle \overline{K_i^u \beta_i}, \overline{K_j^\bullet \beta_j}, \overline{K_k^o \beta_k}, \overline{K_m^u \beta_m} \rangle \\
(\overline{\beta_i}, \overline{\beta_j}, \overline{\beta_k}, \overline{\beta_m} \text{ fresh})
\end{array}$$

$$\begin{array}{c}
\text{JOIN-VAR-UU:} \\
\langle \overline{K_i^u \tau_i}, \overline{K_j^o \tau_j}, \overline{K_k^\bullet \tau_k}, \overline{K_n^u \tau_n} \mid \alpha_1 \rangle \sqsubseteq \langle \overline{K_i^u \beta_i}, \overline{K_j^o \beta_j}, \overline{K_k^o \beta_k}, \overline{K_m^u \beta_m}, \overline{K_n^u \beta_n} \mid \alpha \rangle \\
\langle \overline{K_i^u \rho_i}, \overline{K_j^\bullet \rho_j}, \overline{K_k^o \rho_k}, \overline{K_m^u \rho_m} \mid \alpha_2 \rangle \sqsubseteq \langle \overline{K_i^u \beta_i}, \overline{K_j^\bullet \beta_j}, \overline{K_k^o \beta_k}, \overline{K_m^u \beta_m}, \overline{K_n^u \beta_n} \mid \alpha \rangle \\
(\overline{\beta_i}, \overline{\beta_j}, \overline{\beta_k}, \overline{\beta_m}, \alpha \text{ fresh})
\end{array}$$

Figure 3.23: Join rules of constraint solving

LEMMA 3.7 (Soundness of join/meet phase).

Given two input constraints, if our join/meet phase produces a new set of constraints from those inputs, then the conjunction of the new constraints implies the conjunction of the two input constraints under any assignment to unification variables.

$$\begin{array}{ll} C_1, C_2 \xrightarrow{j/m} \overline{C} & \\ \Rightarrow A \vdash \mathcal{S}(\wedge \overline{C}) & \\ \Rightarrow A \vdash \mathcal{S}(C_1 \wedge C_2) & \end{array}$$

Proof. Proceeds by cases on the constraints. All cases are discharged from the definition of the constraint semantics for records and variants (see Figures 3.9 and 3.16). \square

While the join/meet phase will break down all pairs of flex/rigid subtyping constraints concerning unadorned unification variables, it is occasionally possible that a flex/rigid constraint will be encountered where the unification variable has a type operator applied, for example $\mathbf{bang}(\alpha) \sqsubseteq \tau$. Currently, the join/meet phase does not deal with these constraints, and the subsequent *equate* phase takes care to avoid converting subtyping to equality in the presence of such *flex-modulo-operator* constraints. These constraints are therefore the main source of *incompleteness* in our solving algorithm. Improving the completeness of the solver, and therefore requiring fewer typing annotations, is the subject of future investigation and research.

3.10.4 THE EQUATE PHASE

After the join/meet phase has been completed, flex/rigid subtyping constraints generally should concern unique unification variables that are not mentioned in the same position in any other flex/rigid constraint. The *equate* phase is therefore responsible for locating flex/rigid constraints of the form $\alpha \sqsubseteq \tau$ or $\tau \sqsubseteq \alpha$, where it is safe to replace the constraint with an equality $\alpha \simeq \tau$ without making the set of constraints unsatisfiable. After such a replacement is made, the unify phase will ultimately perform a substitution across the constraint set and add the substitution to the result assignment \mathcal{S} .

At the time of writing, the solver uses a very simple criteria to determine if a constraint can be safely replaced. Essentially, the solver will check all other constraints in the set, to determine if there are any constraints that could potentially become impossible to solve after replacement. A good over-approximation of this is if a constraint mentions the unification variable *in the same position* relative to the subtyping relation. For example, the presence of another constraint $\alpha \sqsubseteq \rho$ would mean that the constraint $\alpha \sqsubseteq \tau$ could not be converted into an equality, as α must be a subtype of

both τ and ρ , and it is not necessarily true that $\tau \sqsubseteq \rho$, so we cannot necessarily say that α is equal to τ .

If a unification variable is mentioned on *both* sides of the subtyping operator in two different constraints, it is only safe to replace one of these subtyping constraints with equality. For example, suppose that $\tau \sqsubseteq \rho$ but $\tau \neq \rho$, and we are given two constraints about the same unification variable: $\tau \sqsubseteq \alpha$ and $\alpha \sqsubseteq \rho$. It is only safe to replace *one* of these constraints with equality, as replacing both would imply that $\tau \approx \rho$ which is not satisfiable. Therefore, our solver arbitrarily chooses to replace constraints where the variable appears on the left-hand side first, and only replace constraints with the variable on the right-hand side if no other candidates are available.

Seeing as the previous phase will have eliminated most potentially-unsafe flex/rigid constraints, this means that flex/flex constraints (i.e. constraints between two unification variables) are the main thing that can potentially block a constraint from being replaced. Another instance is the cases where the unification variable exists in the same position modulo some type operators, because these constraints are not eliminated by the join/meet phase, as mentioned in the previous section.

Thus, each instance of the equate phase will typically replace subtyping constraints on the outer edges of the constraint graph, where concrete types constrain unification variables. After the equalities are converted into substitutions by the unify phase, flex/flex constraints which mention the substituted variables are converted into flex/rigid constraints, themselves amenable to the previous phases (as seen in Figure 3.21); and flex-modulo-operator constraints into rigid/rigid constraints that can be eliminated by the simplifier.

This phase also converts inequalities on *row* unification variables to equalities, for example in variants or records. A nearly identical strategy is used: Any constraint which mentions an incomplete variant type on one side, for example $\langle \varepsilon \mid \alpha \rangle \sqsubseteq \tau$ will be converted into an equality $\langle \varepsilon \mid \alpha \rangle \approx \tau$ so long as no other inequality exists which constrains the row variable α on the same side.

If all constraints are not fully solved after picking all of the low hanging fruit — that is, all substitutions that are obviously safe have been made — the solver will currently report any left over constraints as errors. It may be possible to use certain search-based solving techniques to eliminate more difficult constraints, however this is currently left for future work. For now, the presence of such left-over constraints indicates to the programmer that type signatures must be added to help the inference algorithm along.

All of these safety considerations and restrictions are a *completeness* concern. While our algorithm is incomplete overall, we still would like the algorithm to successfully infer types in a large number of practical use cases, and therefore we wish to avoid the solver transforming a satisfiable constraint set into an unsatisfiable one whenever possible. Soundness, on the other hand, is shown straightforwardly:

LEMMA 3.8 (Soundness of equate phase).

<i>After running the equate phase on some constraints</i>	$\overline{C} \xrightarrow{\text{equate}} \overline{C}'$
<i>the conjunction of the new constraints implies</i>	$\Rightarrow A \vdash \mathcal{S}(\bigwedge \overline{C}')$
<i>the conjunction of the input constraints</i>	$\Rightarrow A \vdash \mathcal{S}(\bigwedge \overline{C})$
<i>under any assignment to unification variables.</i>	

Proof. If a set of constraints are satisfiable after a subtyping constraint has been replaced with an equality constraint, then the original set of constraints are satisfiable due to the reflexivity of subtyping. □

3.10.5 THE SOLVER OVERALL

With each phase of the solver defined and their respective soundness conditions proven, we can now turn our attention to the composition of all these phases. As demonstrated in Figure 3.18, the solver is run until a *fixed point*. In other words, the phases continuously transform the constraint set until it is not transformed anymore by any phase.

This process will take as input a *constraint set* \overline{C} ; and produce as outputs a set of *leftover constraints* \overline{C}' and a result assignment \mathcal{S} . We shall write the solver relation as $\overline{C} \xrightarrow{\text{solve}} \overline{C}'; \mathcal{S}$.

THEOREM 3.7 (Soundness of Solver).

<i>If the solver is run on a constraint set, and the</i>	$\overline{C} \xrightarrow{\text{solve}} \overline{C}'; \mathcal{S}_0$
<i>leftover constraints are satisfiable under an assignment,</i>	$\wedge A \vdash \mathcal{S}'(\bigwedge \overline{C}')$
<i>then combining it with the output assignment produces</i>	$\wedge \mathcal{S} = \mathcal{S}_0 \cup \mathcal{S}'$
<i>a satisfying assignment for the original constraints.</i>	$\Rightarrow A \vdash \mathcal{S}(\bigwedge \overline{C})$

Proof. A mere composition of Lemmas 3.5, 3.6, 3.7 and 3.8. □

It is not proven here that the solver terminates, and termination is a necessary condition to show completeness. As completeness is not our concern, however, we have not investigated a proof of this property in great detail, but we believe that a similar

justification may be used as for the termination argument used for the first-order unification algorithm of Robinson [115], where the number of available unification variables is shown to decrease after each substitution [84, 87].

3.11 TYPE INFERENCE RESULTS

Having defined both the constraint generator and the solver, and proved crucial soundness results, it is now possible to describe formally the behaviour of the entire type inference process.

THEOREM 3.8 (Soundness of Type Inference).

If we generate a constraint for an closed term, $\varepsilon \vdash e : \tau \sim G' \mid C$
then we run the solver on that constraint, $\wedge C \xrightarrow{\text{solv}} \overline{C'}; \mathcal{S}$
and the solver is successful, then $\wedge \overline{C'} \subseteq A$
that term is well typed under the solver's assignment. $\Rightarrow A; \varepsilon \vdash e : \mathcal{S}(\tau)$

Proof. Note that if $\overline{C'} \subseteq A$ then $A \vdash \wedge \overline{C'}$. Furthermore, if $A \vdash \wedge \overline{C'}$ and $\overline{C} \xrightarrow{\text{solv}} \overline{C'}; \mathcal{S}$, then by Theorem 3.7 (where \mathcal{S}' is the empty assignment), we can conclude that $A \vdash \mathcal{S}(C)$. Then, by Theorem 3.2, we have that $A; \varepsilon \vdash e : \mathcal{S}(\tau)$ as required. \square

While no other top-level results (such as completeness) can be proven, the theorems proven about each individual phase give us additional confidence that our algorithm is robust enough to be a workable foundation for future Cogent type system research.

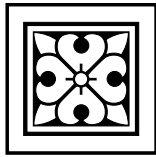
Testing on our implementation has identified cases in the implemented file systems where the type inference makes many user-provided signatures obsolete. With further work, we hope to make our inference system sufficiently robust that explicit type signatures become increasingly rare in Cogent code.



Well-typedness is a key assumption of our overall refinement theorem, ensuring the dynamic uniqueness property necessary to justify our compilation of pure functions to imperative C. Thus, the type system presented here is a vital component of our verification framework.

Our type inference algorithm, on the other hand, is not strictly necessary to establish the refinement theorems needed to certify compilation. It is, nonetheless, an important part of Cogent itself. Without type inference, developers would have to

painstakingly annotate types throughout their programs. This is not just a theoretical tedium: Prior to the introduction of type inference, Cogent users had to do precisely that, reporting it as the single greatest usability problem they observed in Cogent programming. Our algorithm therefore ameliorates this usability problem, improving the experience of Cogent users, and thereby reducing the effort required to develop software in Cogent.



CHAPTER 4

DYNAMIC SEMANTICS

Everyone, left to his own devices,
forms an idea about what goes on in
language which is very far from the
truth.

Ferdinand de Saussure

IT has long been understood that linear and uniqueness type systems can be used to provide a purely functional interface to mutable state and side-effects [132]. This intuition follows from the *uniqueness property* mentioned in Chapter 2, that each live mutable object is referenced by exactly one variable at a time: If a function has a reference to a mutable object, no other references must exist. Therefore, destructive update is indistinguishable from the traditional purely functional copy-update idiom, as no aliases exist to observe the change.

Despite this result, many languages with uniqueness types, such as Rust [118] or Vault [32], only make use of such type systems to reduce or eliminate the need for runtime memory management, and to facilitate informal reasoning about the provenance of pointers. The functional language Clean [10] makes use of uniqueness types to abstract over effects, but it still has need for a garbage collector, and it does not prove, on paper nor in a machine-checked proof script, the semantic coincidence that results from the type system.

The proof of this semantic coincidence is more than just a curiosity for Cogent, as it forms a key part of the compiler certificate used to show refinement from an Isabelle/HOL shallow embedding of the Cogent code all the way to an efficient C implementation, the details of which are discussed in Chapter 5.

Hofmann [59] first formalised this intuition by providing both a set-theoretic de-

notational semantics and a compilation to C for a functional language, and demonstrating that these two semantics coincide in a pen-and-paper proof. The language in question, however, was extremely minimal, and did not involve heap-allocated objects or pointers, merely mutable stack-allocated integers.

In this respect, the machine-checked proof of semantic coincidence for Cogent represents a significant advancement in the state of the art, as Cogent is a higher-order language with full support for compound types and heap-allocated objects, necessitating a more intricate formulation of the uniqueness property, outlined in Section 4.2. Cogent also integrates with C code called via the foreign function interface, which necessitates a formal treatment of the boundary between these languages. Specifically, we must characterise the obligations the C code must meet in order to maintain our uniqueness invariant (see Section 4.2.3).

Each of the theorems presented in this chapter are formalised and machine-checked in Isabelle/HOL, as they form a vital part of our overall refinement certificate. Each theorem includes the corresponding name (written in typewriter typeface) of the equivalent theorem in Isabelle/HOL formalisation of Cogent [24].

4.1 A TALE OF TWO SEMANTICS

As previously mentioned, we assign two dynamic semantics to Cogent terms. The first is the functional *value semantics*, which is suitable for equational reasoning, and can be easily connected to an Isabelle/HOL shallow embedding. The second, the *update semantics*, is more imperative in flavour, where values may take the form of *pointers* to a mutable *store*.

Figure 4.1 describes the syntax of values and their environments for our two dynamic semantics. Both semantics definitions are parameterised by a set of *abstract values*, α_v and α_u respectively, which denote values of abstract types defined in C. They are also parameterised by functional abstractions of any C foreign functions used in the Cogent code, manually written and supplied by the programmer. For an abstract function f , the value semantics abstraction $\llbracket f \rrbracket_v$ must be a pure function, and the update semantics abstraction $\llbracket f \rrbracket_u$ must be a refinement of $\llbracket f \rrbracket_v$ which respects the invariants of our type system. The exact proof obligations placed on these functions are outlined in Section 4.2.3. The Cogent refinement framework described in Chapter 5 is additionally parameterised by refinement proofs between these purely functional abstractions and their C implementation. If full end-to-end verification of all components of the system is desired, the user must additionally prove this refinement, and

compose this proof with our framework.

4.1.1 VALUE SEMANTICS

The rules for the value semantics are given in Figure 4.2. Specified as a big-step evaluation relation $V \vdash e \Downarrow v$, these rules describe the evaluation of an expression e to a single result value v with the environment V containing the values of all variables in scope. In many ways, these semantics are entirely typical of a λ -calculus or other purely functional language: all values are self-contained, there is no notion of sharing or references. Therefore, other than the values of all available variables, there is no need for any context to evaluate an expression. The rules can be viewed as an evaluation algorithm, as they are entirely syntax-directed — exactly one rule specifies the evaluation for each form of expression. Syntactic constructs which only exist to aid the uniqueness type system have no impact on the dynamic semantics. For example, the **let!** construct behaves identically to **let**.

Just as in Chapter 3, where we assumed the existence of a global type environment for top-level definitions called $typeOf(\cdot)$, we include a global definition environment $defnOf(\cdot)$ that, given a function name, provides either:

1. a transparent definition, written $\Lambda \vec{a}. \lambda x. e$, which denotes a Cogent function returning e , parametric for type variables \vec{a} and a single value argument x ; or
2. a black box (\blacksquare), which indicates that the function's definition is *abstract*, i.e. provided externally in C .

The rule VTAPP describes how non-abstract functions are evaluated to function values. As functions must be defined on the top-level, our function values $\langle\langle \lambda x. e \rangle\rangle$ consist only of an unevaluated expression parameterised by a value, evaluated when the function is applied, thereby supplying the argument value. There is no need to define closures or environment capture, as top-level functions cannot capture local bindings. Abstract function values, written $\langle\langle \mathbf{abs}. f \mid \vec{\tau} \rangle\rangle$, are passed indirectly, as a pair of the function name and a list of the types used to instantiate type variables. When an abstract function value $\langle\langle \mathbf{abs}. f \mid \vec{\tau} \rangle\rangle$ is applied to an argument, the user-supplied purely functional abstraction of the C semantics $\llbracket f \rrbracket_V$ is invoked—merely a mathematical function from the argument value to the output value.

Value Semantics	
value semantics values	$v ::= \ell$ <i>(literals)</i> $\quad \langle\langle \lambda x. e \rangle\rangle$ <i>(function values)</i> $\quad \langle\langle \mathbf{abs.} f \mid \bar{\tau} \rangle\rangle$ <i>(abstract functions)</i> $\quad K v$ <i>(variant values)</i> $\quad \{\mathbf{f} \mapsto v\}$ <i>(records)</i> $\quad a_v$ <i>(abstract values)</i>
environments	$V ::= \bar{x} \mapsto v$
abstract values	a_v
abstract function semantics	$\llbracket \cdot \rrbracket_v : f \rightarrow v \rightarrow v$
Update Semantics	
update semantics values	$u ::= \ell$ <i>(literals)</i> $\quad \langle\langle \lambda x. e \rangle\rangle$ <i>(function values)</i> $\quad \langle\langle \mathbf{abs.} f \mid \bar{\tau} \rangle\rangle$ <i>(abstract functions)</i> $\quad K u$ <i>(variant values)</i> $\quad \{\mathbf{f} \mapsto u\}$ <i>(records)</i> $\quad a_u$ <i>(abstract values)</i> $\quad p$ <i>(pointers)</i>
environments	$U ::= \bar{x} \mapsto u$
abstract values	a_u
pointers	p
sets of pointers	r, w
mutable stores	$\mu : p \rightarrow u$
abstract function semantics	$\llbracket \cdot \rrbracket_u : f \rightarrow \mu \times u \rightarrow \mu \times u$
primop semantics	$\llbracket r \rrbracket : v \times v \rightarrow v$
function defn. env.	$defnOf(\cdot) : f \rightarrow D$
function defn.	$D ::= \blacksquare$ <i>(abstract functions)</i> $\quad \Lambda \bar{d}. \lambda x. e$ <i>(function definitions)</i>

Figure 4.1: Syntax for both dynamic semantics interpretations.

$$\boxed{V \vdash e \Downarrow v}$$

$$\frac{x \mapsto v \in V}{V \vdash x \Downarrow v} \text{VVAR} \quad \frac{}{V \vdash \ell \Downarrow \ell} \text{VLIT} \quad \frac{V \vdash e_1 \Downarrow v_1 \quad V \vdash e_2 \Downarrow v_2}{V \vdash e_1 \ \ell \ e_2 \Downarrow v_1 \ \llbracket \ell \rrbracket \ v_2} \text{VOP}$$

$$\frac{V \vdash e \Downarrow \text{True} \quad V \vdash e_1 \Downarrow v}{V \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Downarrow v} \text{VIP-T} \quad \frac{V \vdash e \Downarrow \text{False} \quad V \vdash e_2 \Downarrow v}{V \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Downarrow v} \text{VIP-F}$$

$$\frac{V \vdash e_1 \Downarrow \langle \lambda x. e \rangle \quad V \vdash e_2 \Downarrow v \quad x \mapsto v \vdash e \Downarrow v'}{V \vdash e_1 \ e_2 \Downarrow v'} \text{VAPP} \quad \frac{V \vdash e_1 \Downarrow \langle \langle \text{abs. } f \mid \bar{\tau} \rangle \rangle \quad V \vdash e_2 \Downarrow v \quad v' = \llbracket f \rrbracket_v v}{V \vdash e_1 \ e_2 \Downarrow v'} \text{VAPP-A}$$

$$\frac{\text{defnOf}(f) = \Lambda \bar{a}_i. \lambda x. e}{V \vdash f[\bar{\tau}_i] \Downarrow \langle \langle \lambda x. e \left[\frac{\bar{\tau}_i}{\bar{a}_i} \right] \rangle \rangle} \text{VTAPP} \quad \frac{\text{defnOf}(f) = \blacksquare}{V \vdash f[\bar{\tau}_i] \Downarrow \langle \langle \text{abs. } f \mid \bar{\tau}_i \rangle \rangle} \text{VTAPP-A}$$

$$\frac{V \vdash e_1 \Downarrow v' \quad x \mapsto v', V \vdash e_2 \Downarrow v}{V \vdash \text{let } x = e_1 \text{ in } e_2 \Downarrow v} \text{VLET} \quad \frac{V \vdash e_1 \Downarrow v' \quad x \mapsto v', V \vdash e_2 \Downarrow v}{V \vdash \text{let! } (\bar{y}) \ x = e_1 \text{ in } e_2 \Downarrow v} \text{VLET!}$$

$$\frac{V \vdash e \Downarrow \{f \mapsto v, \bar{f}_i \mapsto v_i\}}{V \vdash e.f \Downarrow v} \text{VMEMBER} \quad \frac{\text{for each } i, V \vdash e_i \Downarrow v_i}{V \vdash \# \{f_i = e_i\} \Downarrow \{f_i \mapsto v_i\}} \text{VSTRUCT}$$

$$\frac{V \vdash e_1 \Downarrow \{f \mapsto v, \bar{f}_i \mapsto v_i\} \quad x \mapsto \{f \mapsto v, \bar{f}_i \mapsto v_i\}, y \mapsto v, V \vdash e_2 \Downarrow v'}{V \vdash \text{take } x \ \{f = y\} = e_1 \text{ in } e_2 \Downarrow v'} \text{VTAKE}$$

$$\frac{V \vdash e_1 \Downarrow \{f \mapsto v, \bar{f}_i \mapsto v_i\} \quad V \vdash e_2 \Downarrow v'}{V \vdash \text{put } e_1.f = e_2 \Downarrow \{f \mapsto v', \bar{f}_i \mapsto v_i\}} \text{VPUT}$$

$$\frac{V \vdash e \Downarrow v}{V \vdash K \ e \Downarrow K \ v} \text{VCON} \quad \frac{V \vdash e \Downarrow K \ v' \quad x \mapsto v', V \vdash e' \Downarrow v}{V \vdash \text{case } e \text{ of } K \ x. \ e' \Downarrow v} \text{VIRREF}$$

$$\frac{V \vdash e \Downarrow K \ v' \quad x \mapsto v', V \vdash e_1 \Downarrow v}{V \vdash \text{case } e \text{ of } K \ x. \ e_1 \text{ else } y. \ e_2 \Downarrow v} \text{VCASE-M}$$

$$\frac{V \vdash e \Downarrow K' \ v' \quad K \neq K' \quad y \mapsto K' \ v', V \vdash e_2 \Downarrow v}{V \vdash \text{case } e \text{ of } K \ x. \ e_1 \text{ else } y. \ e_2 \Downarrow v} \text{VCASE-N}$$

Figure 4.2: The value semantics evaluation rules.

$$\boxed{\mathbb{U} \vdash e \mid \mu \Downarrow u \mid \mu}$$

$$\frac{x \mapsto u \in \mathbb{U}}{\mathbb{U} \vdash x \mid \mu \Downarrow u \mid \mu} \text{UVAR} \quad \frac{}{\mathbb{U} \vdash \ell \mid \mu \Downarrow \ell \mid \mu} \text{ULIT} \quad \frac{\mathbb{U} \vdash e_1 \mid \mu_1 \Downarrow u_1 \mid \mu_2 \quad \mathbb{U} \vdash e_2 \mid \mu_2 \Downarrow u_2 \mid \mu_3}{\mathbb{U} \vdash e_1 \lambda e_2 \mid \mu_1 \Downarrow u_1 \llbracket \lambda \rrbracket u_2 \mid \mu_3} \text{UOP}$$

$$\frac{\mathbb{U} \vdash e \mid \mu_1 \Downarrow \text{True} \mid \mu_2 \quad \mathbb{U} \vdash e_1 \mid \mu_2 \Downarrow u \mid \mu_3}{\mathbb{U} \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \mid \mu_1 \Downarrow u \mid \mu_3} \text{UIF-T} \quad \frac{\mathbb{U} \vdash e \mid \mu_1 \Downarrow \text{False} \mid \mu_2 \quad \mathbb{U} \vdash e_2 \mid \mu_2 \Downarrow u \mid \mu_3}{\mathbb{U} \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \mid \mu_1 \Downarrow u \mid \mu_3} \text{UIF-F}$$

$$\frac{\mathbb{U} \vdash e_1 \mid \mu_1 \Downarrow \langle \lambda x. e \rangle \mid \mu_2 \quad \mathbb{U} \vdash e_2 \mid \mu_2 \Downarrow u \mid \mu_3 \quad x \mapsto u' \mid \mu_3 \Downarrow u' \mid \mu_4}{\mathbb{U} \vdash e_1 e_2 \mid \mu_1 \Downarrow u' \mid \mu_4} \text{UAPP} \quad \frac{\mathbb{U} \vdash e_1 \mid \mu_1 \Downarrow \langle \text{abs. } f \mid \bar{\tau} \rangle \mid \mu_2 \quad \mathbb{U} \vdash e_2 \mid \mu_2 \Downarrow u \mid \mu_3 \quad (u', \mu_4) = \llbracket f \rrbracket_{\mathbb{U}}(u, \mu_3)}{\mathbb{U} \vdash e_1 e_2 \mid \mu_1 \Downarrow u' \mid \mu_4} \text{UAPP-A}$$

$$\frac{\text{defnOf}(f) = \Lambda \bar{a}_i. \lambda x. e}{\mathbb{U} \vdash f[\bar{\tau}_i] \mid \mu \Downarrow \langle \lambda x. e[\bar{\tau}_i/\bar{a}_i] \rangle \mid \mu} \text{UTAPP} \quad \frac{\text{defnOf}(f) = \blacksquare}{\mathbb{U} \vdash f[\bar{\tau}_i] \mid \mu \Downarrow \langle \text{abs. } f \mid \bar{\tau}_i \rangle \mid \mu} \text{UTAPP-A}$$

$$\frac{\mathbb{U} \vdash e_1 \mid \mu_1 \Downarrow u' \mid \mu_2 \quad x \mapsto u', \mathbb{U} \vdash e_2 \mid \mu_2 \Downarrow u \mid \mu_3}{\mathbb{U} \vdash \text{let } x = e_1 \text{ in } e_2 \mid \mu_1 \Downarrow u \mid \mu_3} \text{ULET} \quad \frac{\mathbb{U} \vdash e_1 \mid \mu_1 \Downarrow u' \mid \mu_2 \quad x \mapsto u', \mathbb{U} \vdash e_2 \mid \mu_2 \Downarrow u \mid \mu_3}{\mathbb{U} \vdash \text{let! } (\bar{y}) x = e_1 \text{ in } e_2 \mid \mu_1 \Downarrow u \mid \mu_3} \text{ULET!}$$

$$\frac{\mathbb{U} \vdash e_1 \mid \mu_1 \Downarrow \{f \mapsto u, \bar{f}_i \mapsto \bar{u}_i\} \mid \mu_2 \quad x \mapsto \{f \mapsto u, \bar{f}_i \mapsto \bar{u}_i\}, y \mapsto u, \mathbb{U} \vdash e_2 \mid \mu_2 \Downarrow u' \mid \mu_3}{\mathbb{U} \vdash \text{take } x \{f = y\} = e_1 \text{ in } e_2 \mid \mu_1 \Downarrow u' \mid \mu_3} \text{UTAKE}$$

$$\frac{\mathbb{U} \vdash e_1 \mid \mu_1 \Downarrow \{f \mapsto u, \bar{f}_i \mapsto \bar{u}_i\} \mid \mu_2 \quad \mathbb{U} \vdash e_2 \mid \mu_2 \Downarrow u' \mid \mu_3}{\mathbb{U} \vdash \text{put } e_1.f = e_2 \mid \mu_1 \Downarrow \{f \mapsto u', \bar{f}_i \mapsto \bar{u}_i\} \mid \mu_3} \text{UPUT}$$

$$\frac{\mathbb{U} \vdash e \mid \mu_1 \Downarrow \{f \mapsto u, \bar{f}_i \mapsto \bar{u}_i\} \mid \mu_2}{\mathbb{U} \vdash e.f \mid \mu_1 \Downarrow u \mid \mu_2} \text{UMEM} \quad \frac{\mathbb{U} \vdash \bar{e}_i \mid \mu_1 \Downarrow^* \bar{u}_i \mid \mu_2}{\mathbb{U} \vdash \#\{\bar{f}_i = e_i\} \mid \mu_1 \Downarrow \{f_i \mapsto u_i\} \mid \mu_2} \text{USTRUCT}$$

$$\frac{\mathbb{U} \vdash e \mid \mu_1 \Downarrow u \mid \mu_2}{\mathbb{U} \vdash K e \mid \mu_1 \Downarrow K u \mid \mu_2} \text{UCON} \quad \frac{\mathbb{U} \vdash e \mid \mu_1 \Downarrow K u' \mid \mu_2 \quad x \mapsto u', \mathbb{U} \vdash e' \mid \mu_2 \Downarrow u \mid \mu_3}{\mathbb{U} \vdash \text{case } e \text{ of } K x. e' \mid \mu_1 \Downarrow u \mid \mu_2} \text{UIRRBP}$$

$$\frac{\mathbb{U} \vdash e \mid \mu_1 \Downarrow K u' \mid \mu_2 \quad x \mapsto u', \mathbb{U} \vdash e_1 \mid \mu_2 \Downarrow u \mid \mu_3}{\mathbb{U} \vdash \text{case } e \text{ of } K x. e_1 \text{ else } y. e_2 \mid \mu_1 \Downarrow u \mid \mu_3} \text{UCASE-M}$$

$$\frac{\mathbb{U} \vdash e \mid \mu_1 \Downarrow K' u' \mid \mu_2 \quad K \neq K' \quad y \mapsto K' u', \mathbb{U} \vdash e_2 \mid \mu_2 \Downarrow u \mid \mu_3}{\mathbb{U} \vdash \text{case } e \text{ of } K x. e_1 \text{ else } y. e_2 \mid \mu_1 \Downarrow u \mid \mu_3} \text{UCASE-N}$$

(Continued in Figure 4.4)

$$\boxed{\mathbb{U} \vdash \bar{e} \mid \mu \Downarrow^* \bar{u} \mid \mu}$$

$$\frac{}{\mathbb{U} \vdash \varepsilon \mid \mu \Downarrow^* \varepsilon \mid \mu} \text{UNIL} \quad \frac{\mathbb{U} \vdash e_0 \mid \mu_1 \Downarrow u_0 \mid \mu_2 \quad \mathbb{U} \vdash \bar{e}_i \mid \mu_2 \Downarrow^* \bar{u}_i \mid \mu_3}{\mathbb{U} \vdash e_0 \bar{e}_i \mid \mu_1 \Downarrow^* u_0 \bar{u}_i \mid \mu_3} \text{UCONS}$$

Figure 4.3: The straightforward update semantics evaluation rules.

$$\boxed{\mathbb{U} \vdash e \mid \mu \Downarrow u \mid \mu}$$

$$\frac{\mathbb{U} \vdash e_1 \mid \mu_1 \Downarrow p \mid \mu_2 \quad \mu_2(p) = \{f \mapsto u, \overline{f_i \mapsto u_i}\} \quad x \mapsto p, y \mapsto u, \mathbb{U} \vdash e_2 \mid \mu_2 \Downarrow u' \mid \mu_3}{\mathbb{U} \vdash \mathbf{take} \ x \{f = y\} = e_1 \ \mathbf{in} \ e_2 \mid \mu_1 \Downarrow u' \mid \mu_3} \text{UTAKE-B}$$

$$\frac{\mathbb{U} \vdash e_1 \mid \mu_1 \Downarrow p \mid \mu_2 \quad \mu_2(p) = \{f \mapsto u, \overline{f_i \mapsto u_i}\} \quad \mathbb{U} \vdash e_2 \mid \mu_2 \Downarrow u' \mid \mu_3}{\mathbb{U} \vdash \mathbf{put} \ e_1.f = e_2 \mid \mu_1 \Downarrow p \mid \mu_3(p := \{f \mapsto u', \overline{f_i \mapsto u_i}\})} \text{UPUT-B}$$

$$\frac{\mathbb{U} \vdash e \mid \mu_1 \Downarrow p \mid \mu_2 \quad \mu_2(p) = \{f \mapsto u, \overline{f_i \mapsto u_i}\}}{\mathbb{U} \vdash e.f \mid \mu_1 \Downarrow u \mid \mu_2} \text{UMEM-B}$$

Figure 4.4: The update semantics evaluation rules concerning pointers.

4.1.2 UPDATE SEMANTICS

Similarly to the value semantics, the update semantics is specified as a big-step evaluation relation, however unlike the value semantics, a *mutable store* is included as an input to and output of an expression's evaluation, and values may be represented as pointers to locations in that mutable store. Written $\mathbb{U} \vdash e \mid \mu \Downarrow u \mid \mu'$, this evaluation relation specifies that, given an environment \mathbb{U} of values that may contain pointers into a mutable store μ , the evaluation of the expression e will result in the value u and a final store μ' . Figure 4.3 outlines the straightforward rules for this evaluation relation. The majority of these are very similar to their value-semantics equivalents, save that they thread the mutable store through the evaluation.

The mutable store is specified as a partial mapping from pointers (written p) to values. The exact content of pointer values is left abstract: our semantics merely requires that they be enumerable and comparable. In Chapter 5, we instantiate p to a concrete set to prove refinement to the C implementation.

Like in the value semantics, the semantics of foreign functions are provided externally, this time permitting modifications to the mutable store in addition to returning a value. Rules for variants and other primitive types are all analogous to the value semantics, however differences arise when it comes to record types. Unlike the value semantics, the update semantics distinguishes between *boxed* and *unboxed* records. For *unboxed* records, which are stack-allocated and passed by value, the rules for **take**, **put** etc. resemble their value-semantics counterparts. *Boxed* records, however,

are represented as a pointer — the rule for **take** must consult the heap, and the rule for **put** *mutates* the heap, destructively updating the record. These rules that involve the mutable heap are specified in Figure 4.4.

4.2 REFINEMENT AND TYPE PRESERVATION

To show that the update semantics refines the value semantics, the typical approach from data refinement, originally due to de Roever and Engelhardt [31], is to define a *forward simulation* or *refinement relation* R between values in the value semantics and states in the update semantics, and show that any update semantics evaluation has a corresponding value semantics evaluation that preserves this relation. When each semantics is viewed as a binary relation from initial states to final states (outputs), this requirement can be succinctly expressed as a commutative diagram. For example, with respect to an externally defined abstract function f , we would require that the user-provided value semantics $\llbracket f \rrbracket_v$ is indeed an abstraction of the user-provided update semantics $\llbracket f \rrbracket_u$:

$$R; \llbracket f \rrbracket_u \subseteq \llbracket f \rrbracket_v; R$$

(where $;$ is forward composition of relations)

Assuming that the relation holds initially, we can conclude from such a proof that any execution in our update semantics interpretation has a corresponding execution in our value semantics interpretation, and thus any functional correctness property we prove about all our value semantics executions applies also to our update semantics executions.

The relation R must relate value semantics values (v) to update semantics states ($u \times \mu$). A plausible definition would be as an abstraction function, which eliminates pointers from each update semantics value u in the state by following all pointers from the value u in the store μ , collapsing the pointer graph structure into a self-contained value v in the value semantics.

Such a relation, however, is not preserved by evaluation in the presence of aliasing of mutable data, as a destructive update (such as a **put**) to a location in the store aliased by two variables would affect the value of both variables in the update semantics, but only one of them in the value semantics. Therefore, the refinement relation must additionally encode the uniqueness property ensured by our type system,

which rules out not just *direct* aliasing, where two separate variables refer to the same data structure on the heap, but also *internal* aliasing, where a single data structure contains two or more aliasing pointers.

The rules in Figure 4.5 define our refinement relation, extended to take into account the type system and aliasing of pointers. Written $u \mid \mu : v : \tau \ [r * w]$, this judgement states that:

1. Transitively following all the pointers from u in the store μ results in the self-contained value v ,
2. Both u and v have the type τ ,
3. The set r contains all *read-only* pointers (according to the type τ) transitively accessible from u ,
4. The set w contains all *writable* pointers transitively accessible from u , and
5. The value u contains no internal aliasing of any of the writable pointers in w , whether by read-only or writable pointers.

We call the sets r and w the *footprint* of the value u . By annotating the relation in this way, we can insert the required non-aliasing constraints into the rules for compound values such as records. Read-only pointers may alias other read-only pointers, but writable pointers may not alias any other pointer, whether read-only or writable.

Because our relation relates both update semantics and value semantics to types, we can derive a value-typing relation for either semantics by creatively erasing part of the rules. Erasing all the update semantics parts (highlighted like `this`) leaves a value-typing relation definition for the value semantics, and erasing all the value semantics parts (highlighted like `this`) gives a state-typing relation definition for the update semantics. As we ultimately prove preservation for this refinement relation across evaluation, the same erasure strategy can be applied to the proofs to produce a typing *preservation* proof for either semantics — a key component of type safety.

POLYMORPHISM

As mentioned in Chapter 2, we implement parametric polymorphism by specialising code to avoid paying the performance penalties of other approaches such as boxing. This means that polymorphism in Cogent is restricted to predicative rank-1 quantifiers, in the style of ML. This allows us to specify dynamic objects, such as our

$$\begin{array}{c}
\boxed{u \mid \mu : v : \tau \quad [r * w]} \\
\frac{\ell < |T| \quad \varepsilon; x : \tau \vdash e : \tau'}{\ell \mid \mu : \ell : T \quad [\emptyset * \emptyset] \quad \langle \langle \lambda x. e \rangle \rangle \mid \mu : \langle \langle \lambda x. e \rangle \rangle : \tau \rightarrow \tau' \quad [\emptyset * \emptyset]} \text{RLIT} \quad \text{RFUN} \\
\frac{\text{typeOf}(f) = \forall \bar{a}_i. C \Rightarrow \tau}{\langle \langle \text{abs. } f \mid \bar{\tau}_i \rangle \rangle \mid \mu : \langle \langle \text{abs. } f \mid \bar{\tau}_i \rangle \rangle : \tau \left[\frac{\bar{\tau}_i}{\bar{a}_i} \right] \quad [\emptyset * \emptyset]} \text{RAFUN} \\
\frac{u \mid \mu : v : \tau \quad [r * w]}{K u \mid \mu : K v : \langle K^\circ \tau, K_i^u \bar{\tau}_i \rangle \quad [r * w]} \text{RVARIANT} \\
\frac{\mu(p) = a_v \quad a_v \mid \mu : \mathcal{A} \quad a_v : \mathcal{A} \quad A \quad \bar{\tau}_i \text{ } \textcircled{\text{R}} \quad [r * \emptyset]}{p \mid \mu : a_v : A \quad \bar{\tau}_i \text{ } \textcircled{\text{R}} \quad [\{p\} \cup r * \emptyset]} \text{RABSR} \quad \frac{\mu(p) = a_v \quad a_v \mid \mu : \mathcal{A} \quad a_v : \mathcal{A} \quad A \quad \bar{\tau}_i \text{ } \textcircled{\text{W}} \quad [r * w]}{p \mid \mu : a_v : A \quad \bar{\tau}_i \text{ } \textcircled{\text{W}} \quad [r * \{p\} \cup w]} \text{RABSW} \\
\frac{a_v \mid \mu : \mathcal{A} \quad a_v : \mathcal{A} \quad A \quad \bar{\tau}_i \text{ } \textcircled{\text{U}} \quad [r * w]}{a_v \mid \mu : a_v : A \quad \bar{\tau}_i \text{ } \textcircled{\text{U}} \quad [r * w]} \text{RABSU} \\
\frac{u = \{\bar{f}_i \mapsto \bar{u}_i, \bar{f}_k \mapsto \bar{u}_k\} \quad v = \{\bar{f}_i \mapsto \bar{v}_i, \bar{f}_k \mapsto \bar{v}_k\} \quad \text{for each } i, u_i \mid \mu : v_i : \tau_i \quad [r_i * w_i] \quad \text{for each } f_j \in \bar{f}_i \text{ where } i \neq j, w_i \cap (r_j \cup w_j) = \emptyset}{u \mid \mu : v : \{\bar{f}_i^\circ : \tau_i, \bar{f}_k^\bullet : \tau_k\} \textcircled{\text{U}} \quad [\bigcup_i r_i * \bigcup_i w_i]} \text{RRBCU} \\
\frac{\mu(p) = \{\bar{f}_i \mapsto \bar{u}_i, \bar{f}_k \mapsto \bar{u}_k\} \quad v = \{\bar{f}_i \mapsto \bar{v}_i, \bar{f}_k \mapsto \bar{v}_k\} \quad \text{for each } i, u_i \mid \mu : v_i : \tau_i \quad [r_i * \emptyset]}{p \mid \mu : v : \{\bar{f}_i^\circ : \tau_i, \bar{f}_k^\bullet : \tau_k\} \textcircled{\text{R}} \quad [\{p\} \cup \bigcup_i r_i * \emptyset]} \text{RRBCR} \\
\frac{\mu(p) = \{\bar{f}_i \mapsto \bar{u}_i, \bar{f}_k \mapsto \bar{u}_k\} \quad v = \{\bar{f}_i \mapsto \bar{v}_i, \bar{f}_k \mapsto \bar{v}_k\} \quad \text{for each } i, u_i \mid \mu : v_i : \tau_i \quad [r_i * w_i] \quad \text{for each } f_j \in \bar{f}_i \text{ where } i \neq j, w_i \cap (r_j \cup w_j) = \emptyset}{p \mid \mu : v : \{\bar{f}_i^\circ : \tau_i, \bar{f}_k^\bullet : \tau_k\} \textcircled{\text{W}} \quad [\bigcup_i r_i * \{p\} \cup \bigcup_i w_i]} \text{RRBCW} \\
\boxed{a_v \mid \mu : \mathcal{A} \quad a_v : \mathcal{A} \quad A \quad \bar{\tau}_i \text{ } s \quad [r * w]} \\
\text{(abstract types are user-provided)} \\
\boxed{U \mid \mu : V : \Gamma \quad [r * w]} \\
\text{for each } x_i : \tau_i \in \Gamma, \\
\frac{x_i \mapsto u_i \in U \quad x_i \mapsto v_i \in V \quad u_i \mid \mu : v_i : \tau_i \quad [r_i * w_i] \quad \text{for each } x_j \in \bar{x}_i \text{ where } i \neq j, w_i \cap (r_j \cup w_j) = \emptyset}{U \mid \mu : V : \Gamma \quad [r * w]} \text{RENV}
\end{array}$$

Figure 4.5: The value typing and update/value refinement rules.

values and their typing and refinement relations, in terms of simple monomorphic types, without type variables. Thus, to evaluate a polymorphic program, each type variable must first be instantiated to a monomorphic type. Theorem 3.6 shows that any valid instantiation of a well-typed polymorphic program is well-typed, which implies the monomorphic specialisation case when all variables are instantiated. Thus, our results about our refinement relation can safely assume the well-typedness of the monomorphic specialisation of the program which is being evaluated.

ENVIRONMENTS

Figure 4.5 also defines the refinement relation for environments and type contexts, written $\mathbb{U} \mid \mu : \mathbb{V} : \Gamma \ [r * w]$. Just as our original refinement relation enforces our uniqueness constraint inside a single value, the refinement relation for environments requires that the values of all variables in Γ meet the uniqueness constraints, such that no available variable will contain an alias of a writable pointer in any other available variable.

Because this relation is only concerned with *available* variables, we can show that the context-splitting relation (given in Figure 3.2), which partitions the available variables into two sub-contexts, also neatly bifurcates the associated pointer sets r and w , such that the same environment viewed through either of the sub-contexts does not alias the other sub-context's writable pointers:

LEMMA 4.1 (Splitting contexts splits footprints — `u_v_matches_split`).

<i>If an environment corresponds to a context,</i>	$\mathbb{U} \mid \mu : \mathbb{V} : \Gamma \ [r * w]$
<i>and that context is split in two, then</i>	$\wedge \ \varepsilon \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$
<i>the heap footprint may also be split into</i>	$\Rightarrow \ \exists r_1 \ w_1 \ r_2 \ w_2.$
<i>two sub-footprints, where each sub-footprint</i>	$\wedge \ \mathbf{r} = r_1 \cup r_2 \wedge \mathbf{w} = w_1 \cup w_2$
<i>does not contain any aliases to writable</i>	$\wedge \ r_1 \cap (w_2 \cup r_2) = \emptyset$
<i>pointers in the other footprint, and where</i>	$\wedge \ r_2 \cap (w_1 \cup r_1) = \emptyset$
<i>the refinement relation holds for each</i>	$\wedge \ \mathbb{U} \mid \mu : \mathbb{V} : \Gamma_1 \ [r_1 * w_1]$
<i>sub-footprint and sub-context respectively.</i>	$\wedge \ \mathbb{U} \mid \mu : \mathbb{V} : \Gamma_2 \ [r_2 * w_2]$

ABSTRACT VALUES

Because the representation of abstract values is defined externally to Cogent, the corresponding refinement relation $\mathbf{a}_v \mid \mu : \mathbb{A} \ \mathbf{a}_v : \mathbb{A} \ \tau_1^s \ [r * w]$ is defined externally also.

To ensure that our uniqueness invariant is maintained, certain constraints are placed on the pointer sets r and w in the user-supplied definition, depending on the sigil s :

- If the sigil s is \textcircled{r} , then the set w must be empty. This is because abstract, read-only values are assumed to be shareable in Cogent’s type system (see Figure 3.12), and therefore must not contain any writable pointers.
- If the sigil s is \textcircled{u} , then both sets must be empty. Abstract, unboxed values meet the **Share**, **Drop** and **Escape** constraints. Therefore, w must be empty to avoid violating uniqueness directly, and r must be empty to prevent uniqueness violations in **let!** expressions.
- If the sigil s is \textcircled{w} , then no constraints are placed on either set, as the type is considered linear.

These pointer sets need not include all pointers contained within the data structure, but merely those pointers to *Cogent values* that are accessible via the interface exposed to Cogent. This allows data structures that rely on sharing, or would otherwise violate the uniqueness property of the type system, to be safely imported and used by Cogent functions. Similarly, the requirements of the frame relation here only apply to those pointers accessible from the Cogent side. Thus, during the execution of an imported C function, the uniqueness and framing conditions need not be adhered to — only the *interface* with Cogent needs to satisfy these constraints. The exact requirements of the Cogent interface are summarised in Section 4.2.3.

4.2.1 FRAMING

If the inputs to a Cogent program have a footprint $[r * w]$, then it is reasonable to require that no live objects in the store other than those referenced in w (its *frame*) will be modified or affected by the evaluation of the program. In this way, two subprograms that affect different parts of the store may be evaluated independently. We formalise this requirement as a *framing* relation, which states exactly how evaluation may affect the mutable store.

DEFINITION 4.1 (Framing Relation). Given an input set of writable pointers w_i to a store μ_i , and an output set of writable pointers w_o to a store μ_o , the framing relation $w_i \mid \mu_i \text{ frame } w_o \mid \mu_o$ ensures three properties for any pointer p :

- *Inertia*: Any value outside the frame is unaffected, i.e. if $p \notin w_i \cup w_o$ then $\mu_i(p) = \mu_o(p)$.
- *Leak freedom*: Any value removed from the frame must be freed, i.e. if $p \in w_i$ and $p \notin w_o$, then $\mu_o(p) = \perp$.
- *Fresh allocation*: Any value added to the frame must not overwrite anything else, i.e. if $p \notin w_i$ and $p \in w_o$ then $\mu_i(p) = \perp$.

If a program's evaluation meets the requirements specified in the framing relation, we can directly prove that our refinement relation is unaffected by any updates to the store outside the footprint:

LEMMA 4.2 (Unrelated updates — `upd_val_rel_frame`).

Assuming two unrelated pointer sets, where one set is part of a value's footprint, and the other is the frame of a computation; then the refinement relation is re-established for the resultant store of that computation.

$$\begin{array}{l} w \cap w_1 = \emptyset \\ u \mid \mu : v : \tau \ [r * w] \\ w_1 \mid \mu \text{ frame } w_2 \mid \mu' \\ \Rightarrow u \mid \mu' : v : \tau \ [r * w] \end{array}$$

This result also generalises smoothly to our refinement relation for environments and contexts:

LEMMA 4.3 (Unrelated updates for environments — `upd_val_rel_frame_env`).

Assuming two unrelated pointer sets, where one set is part of an environment's footprint, and the other is the frame of a computation; then the refinement relation is re-established for the resultant store of that computation.

$$\begin{array}{l} w \cap w_1 = \emptyset \\ U \mid \mu : V : \Gamma \ [r * w] \\ w_1 \mid \mu \text{ frame } w_2 \mid \mu' \\ \Rightarrow U \mid \mu' : V : \Gamma \ [r * w] \end{array}$$

The frame relation allows us to address the well known *frame problem* in verification and logic. Using these results along with Lemma 4.1, we can show (in the proof of Theorem 4.1) that evaluating one sub-expression does not affect any part of the store other than those mentioned in the heap footprint for the corresponding sub-context, and therefore that the refinement relation is preserved for the evaluation of subsequent sub-expressions.

4.2.2 PROVING REFINEMENT

To prove our desired refinement statement, we must show that every evaluation in the update semantics has a corresponding evaluation in the value semantics that preserves our refinement relation. We decompose this into two main theorems: one to show general *preservation* of the refinement relation, and one to show *upward-propagation* of evaluation.

As previously mentioned, the *preservation* theorem can, with the right kind of selective vision, be viewed as a type preservation theorem for either semantics. Viewed in its entirety, it states that our refinement relation is preserved by any pair of evaluations for a well typed expression:

THEOREM 4.1 (Preservation of Refinement/Typing Relation — correspondence).

<i>For a well-typed expression which evaluates</i>	$A; \Gamma \vdash e : \tau$
<i>in the value semantics from environment V,</i>	$\wedge V \vdash e \Downarrow v$
<i>and in the update semantics from U:</i>	$\wedge U \vdash \mu \mid e \Downarrow u \mid \mu'$
<i>If V and U correspond with some footprint,</i>	$\wedge U \mid \mu : V : \Gamma \ [r * w]$
<i>then there exists another footprint</i>	$\Rightarrow \exists r' \subseteq r. \exists w'.$
<i>which results from the initial footprint,</i>	$w \mid \mu \text{ frame } w' \mid \mu'$
<i>such that the result values correspond.</i>	$\wedge u \mid \mu : v : \tau \ [r' * w']$

Proof. By rule induction on the update semantics evaluation. For expressions which involve more than one sub-expression, we use Lemma 4.1 to establish that each sub-expression has a non-overlapping footprint. Then, from the inductive hypothesis, we know that the frame relation holds for each of these footprints. Then we use Lemma 4.2 and Lemma 4.3 to demonstrate that the evaluation of the first expression still preserves the refinement relation for the unrelated second expression.

To obtain this inductive hypothesis, we must additionally prove for each case that the frame relation holds for each evaluation. This is relatively simple, as the constraints of the frame relation are all straightforward consequences of our uniqueness type system. \square

Because it assumes the existence of an evaluation on both the update *and* value semantics levels, this preservation theorem is not sufficient to show refinement by itself. We still need to show that the value semantics evaluates whenever the update semantics does. This is where our *upward propagation* theorem comes in, proven by straightforward rule induction:

THEOREM 4.2 (Upward evaluation propagation — `val_executes_from_upd_executes`).

For a well-typed expression e which evaluates in the update semantics from \mathbb{U} , if \mathbb{U} has a corresponding value semantics environment, then e also evaluates in the value semantics.

$$\begin{array}{ll} A; \Gamma \vdash e : \tau & \\ \wedge \mathbb{U} \vdash \mu \mid e \Downarrow_{\mathbb{U}} u \mid \mu' & \\ \wedge \mathbb{U} \mid \mu : V : \Gamma [r * w] & \\ \Rightarrow \exists v. V \vdash e \Downarrow_{\mathbb{V}} v & \end{array}$$

With this result, the overall refinement of the value semantics to the update semantics is a simple corollary:

THEOREM 4.3 (Value \rightarrow Update refinement).

If a typed expression e , under environment \mathbb{U} , evaluates to u in the update semantics, and \mathbb{U} corresponds to environment V , then e evaluates to some v under V in the value semantics, and there exists a footprint that results from the original footprint such that u corresponds to v .

$$\begin{array}{ll} A; \Gamma \vdash e : \tau & \\ \wedge \mathbb{U} \vdash \mu \mid e \Downarrow_{\mathbb{U}} u \mid \mu' & \\ \wedge \mathbb{U} \mid \mu : V : \Gamma [r * w] & \\ \Rightarrow \exists v. V \vdash e \Downarrow_{\mathbb{V}} v & \\ \wedge \exists r' \subseteq r. \exists w'. & \\ \quad w \mid \mu \text{ frame } w' \mid \mu' & \\ \wedge \mathbf{u} \mid \mu : \mathbf{v} : \tau [r' * w'] & \end{array}$$

This theorem forms an essential component of our overall compiler certificate, the construction of which is outlined in Chapter 5.

4.2.3 FOREIGN FUNCTIONS

Each of the above theorems makes certain assumptions about the semantics given to abstract functions, $\llbracket \cdot \rrbracket_{\mathbb{U}}$ and $\llbracket \cdot \rrbracket_{\mathbb{V}}$. Specifically, we must assume that the two semantics are *coherent*, in that they evaluate in analogous ways; that they respect the constraints of the frame relation to maintain our memory invariants; and that they do not introduce any observable aliasing, which would violate the uniqueness constraint of our type system.

These three properties are ensured by an assumption similar in format to the two lemmas used for the proof of refinement, Theorems 4.1 and 4.2. Specifically, we assume for a foreign function f of type $\tau \rightarrow \rho$ that, given input values u and v that correspond, i.e. $\mathbf{u} \mid \mu : \mathbf{v} : \tau [r * w]$,

1. If the update semantics evaluates, i.e. $\llbracket f \rrbracket_{\mathbb{U}}(\mu, u) = (\mu', u')$, then the value semantics evaluates, i.e. $\llbracket f \rrbracket_{\mathbb{V}}(v) = v'$;
2. Their results correspond, i.e. $\mathbf{u}' \mid \mu' : \mathbf{v}' : \rho [r' * w']$ for some $r' \subseteq r$ and w' ; and

3. The frame relation holds, i.e. $w \mid \mu \text{ frame } w' \mid \mu'$.

These assumptions directly satisfy any obligations about foreign functions that arise in the proofs of Theorems 4.1 and 4.2, thus providing all the necessary ingredients to prove refinement in the presence of foreign functions.



The refinement theorem between our two semantic interpretations, vital to our overall framework, is only possible because Cogent is a significantly restricted language, disallowing aliasing of writable pointers. This *semantic shift* refinement has been proven in a mechanical theorem prover, definitively confirming the intuition of Wadler [132], and extending existing pen-and-paper theoretical work [59] to apply to real-world languages with heap-allocated objects and pointers.

CHAPTER 5

REFINEMENT FRAMEWORK

Translation is the art of failure.

Umberto Eco

THE refinement proof from value to update semantics presented in Chapter 4 is only one piece, albeit a crucial one, of the overall refinement chain from the Isabelle embedding of the Cogent code down to the generated C.

Both below and above this semantic shift, specialised tactics in Isabelle/HOL generate numerous refinement proofs, which mirror each transformation made by the Cogent compiler. These refinement proofs are combined into a proof of a top-level refinement theorem that connects the semantics of the C code with a higher-order logic (HOL) embedding of the Cogent code. The proof structure of this refinement framework is outlined in Figure 5.1, and involves a number of different embeddings: *shallow embeddings*, where the program is represented as a semantically equivalent HOL term, and also *deep embeddings*, where the program is represented as an abstract syntax tree in HOL.

As shallow embeddings have a direct semantic interpretation in HOL, they are easier to reason about concretely: that is, individual shallowly embedded programs are mere mathematical functions, and are therefore amenable to verification using standard theorem prover definitions and tactics. This is why the more abstract embeddings at the top of the chain are all shallow, as these embeddings are used for further functional correctness verification, connecting to a higher level abstract specification written specifically for the program under examination.

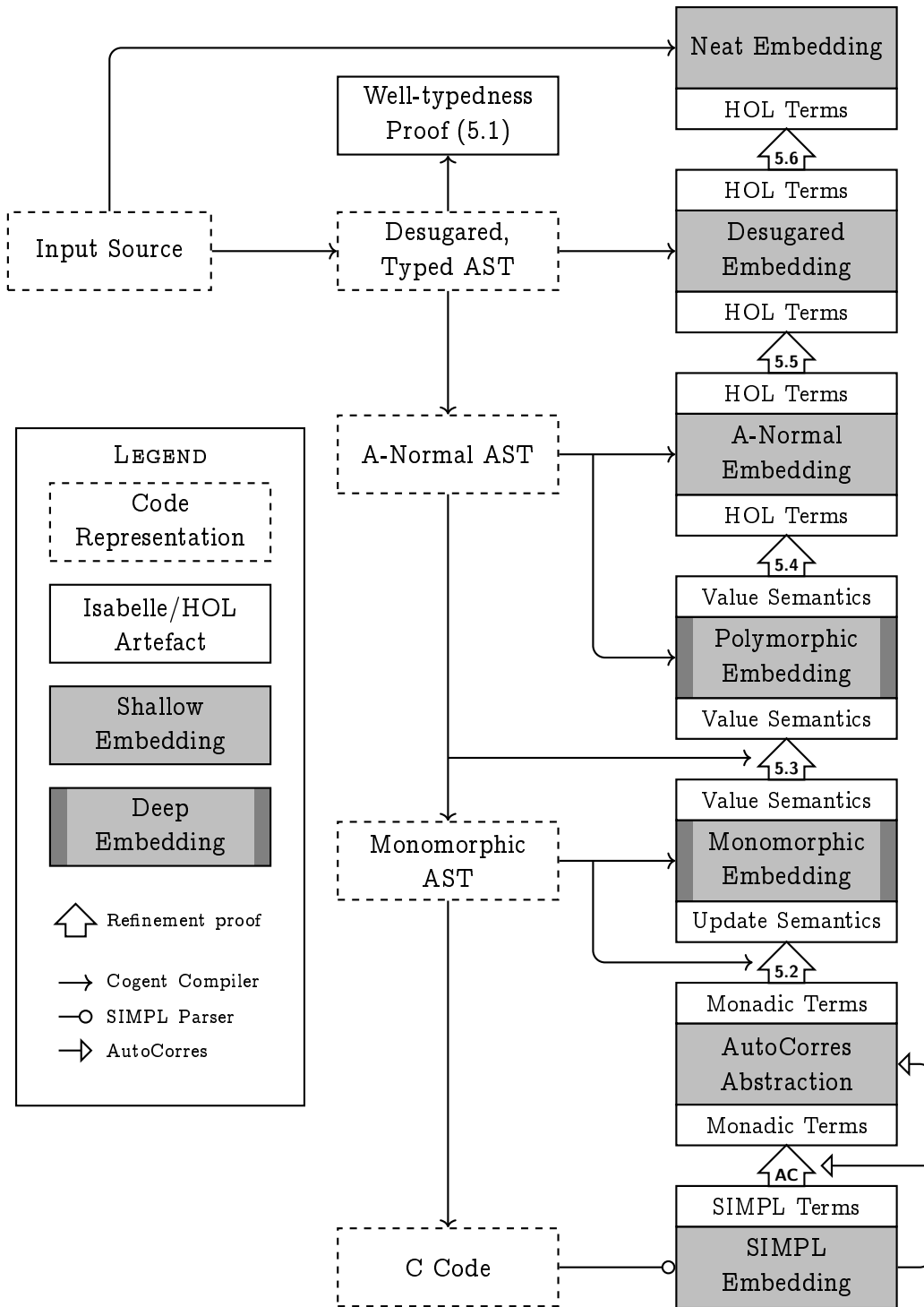
On the other hand, shallow embeddings make it very difficult to prove results for *all* programs, such as the refinement theorem between update and value semantics

in Chapter 4. In such situations, *deep* embeddings are preferred, where the program terms are represented as an abstract syntax tree, and separate evaluation relation(s) are defined to provide semantics, such as those in Chapter 4. This allows us to perform induction on program terms, exhaustively verifying a property for *every program*. Furthermore, this decoupling of term structure and semantics allows us to define multiple semantic interpretations for the same set of terms. We need both of these advantages to prove theorems like Theorem 4.3, which justify the semantic shift from value semantics to update semantics. The embeddings in the middle of the refinement chain are all therefore deep, as this is where Theorem 4.3 is used.

The lower-level embeddings closer to C code are also shallowly-embedded. This is because the Cogent verification framework builds on two existing mature verification tools for C software in Isabelle/HOL: The $C \rightarrow \text{SIMPL}$ Parser used in the seL4 project, and the automatic C abstraction tool AutoCorres [49, 48]. As both of these are designed for manual verification of specific C programs, they choose to represent C code using shallow embeddings, suitable for human consumption. The C Parser imports C code into the Isabelle-embedded language SIMPL [121] extended with the memory model of Tuch, Klein, and Norrish [128]; while AutoCorres abstracts this SIMPL code into HOL terms involving the *non-deterministic state monad* first described in Cock, Klein, and Sewell [23].

Each of the refinement proofs presented in Figure 5.1 is established via *translation validation* [109]. That is, rather than *a priori* verification of phases of the compiler, specialised Isabelle tactics and proof generators are used to establish a refinement proof *a posteriori*, relating the input and output of each compiler phase after the compiler has executed. For the most part, this is because these refinement stages involve shallow embeddings, which do not allow the kind of term inspection needed to directly model a compiler phase and prove it correct. It also has the advantage of allowing us some flexibility in implementation, as the *post-hoc* generated refinement proof is not dependent on the exact implementation of the compiler.

This approach is not without its drawbacks, however. Chief among these is the lack of a completeness guarantee: While we know that the compiler acted correctly if Isabelle validates the generated refinement proof, there is no way to establish any formal guarantee that Isabelle will always validate the generated proof if the compiler acts correctly. In a verified compiler, proofs need only to be checked once, thus indicating that the compiler is trustworthy; but with translation validation, proofs must be checked after each compilation due to this lack of certainty about completeness.



(numbers on arrows are theorem numbers)

Figure 5.1: Refinement phases of Cogent

5.1 REFINEMENT AND FORWARD SIMULATION

As mentioned in Chapter 4, each of our refinement proofs is based on the *forward simulation* technique for data refinement, an idea independently discovered by many people but crystallised by de Roever and Engelhardt [31]. This technique involves defining a *refinement relation* R that connects *abstract* states (for example in the HOL embedding) to corresponding *concrete* states (for example in the C code). Then, assuming R holds for initial states, we must prove that every possible concrete evaluation can be matched by a corresponding abstract execution, resulting in final states for which R is re-established:

$$R; \langle \text{abstract} \rangle \subseteq \langle \text{concrete} \rangle; R$$

(where $;$ is forward composition of relations)

These relation preservation proofs only imply refinement given the assumption that the relation R holds initially. A similar assumption is made for the verification of seL4 [75]. Bridging this remaining gap in the verification chain must be made on a case-by-case basis, and is the subject of further research.

5.2 WELL-TYPEDNESS PROOF

The refinement theorems concerning the monomorphic deep embedding, such as our semantic shift refinement relation in Chapter 4, assume that the Cogent program is well-typed. Therefore, it is necessary to prove in Isabelle/HOL that the generated monomorphic deep embedding is well-typed.

Specifically, the compiler will generate Isabelle/HOL definitions of the *defnOf*(\cdot) and *typeOf*(\cdot) environments (described in Chapter 3) for the monomorphised version of the Cogent program, and then prove the following theorem via a custom Isabelle tactic:

GENERATED THEOREM 5.1 (Typing). *Let f be the name of a monomorphic Cogent function, where $\text{defnOf}(f) = \lambda x. e$ and $\text{typeOf}(f) = \tau \rightarrow \rho$. Then, $x : \tau \vdash e : \rho$.*

Because these typing rules are not algorithmic, we require additional information from the Cogent compiler to produce an efficient deterministic algorithm that synthesises a proof of this theorem. There are a number of sources of non-determinism in these typing rules:

1. The use of the context-splitting relation in the typing rules means that a naïve algorithm for proof synthesis could necessitate traversing over every sub-expression to determine which variables are used in each split. The compiler eliminates the need for this by emitting a table of *hints* that informs the proof-synthesis tactic on how each context is split, indicating which variables are used in each sub-expression.
2. As the subsumption rule of subtyping is not syntax-directed, it could potentially be used at any point in the typing derivation. To eliminate non-determinism resulting from such potential upcasts, the compiler includes special **promote** syntax nodes in the generated deep embedding, which indicate precisely where in the syntax tree subsumption has been used.
3. Integer literals are overloaded in the Cogent syntax, which can make their typing ambiguous. The compiler resolves this simply by annotating all literals with their precise inferred type in the generated deep embedding.

Armed with this additional information from the compiler, our proof synthesis tactic proceeds by merely applying each of the non-algorithmic typing rules from Chapter 3 as introduction rules. The choice of which rule to apply, and which instantiations of schematic variables to use, is now entirely unambiguous.

Because HOL is a *proof-irrelevant* logic, once we prove the top-level typing theorem for a function, we lose access to the typing lemmas for each of the sub-expressions that make up the function’s body. As theorems do not contain any information or structure beyond their truth, we cannot precisely extract these lemmas from the theorem. As we will see in Section 5.3.1, our synthesised refinement proof from the monomorphic deep embedding to the AutoCorres embedding needs access to all of these typing lemmas. For this reason, our tactic remembers each intermediate typing derivation in a tree structure as it proves the top-level typing theorem. This tree structurally matches the derivation tree for the typing theorem itself: each node contains the intermediate theorem for that part of the typing derivation.

5.3 REFINEMENT PHASES

The only synthesised proof artefacts in our framework aside from the proof of well-typedness are the six refinement theorems presented in Figure 5.1. While they are all refinement theorems proven by translation validation, the exact structure of the theorem and the mechanism used to prove them differs in each case.

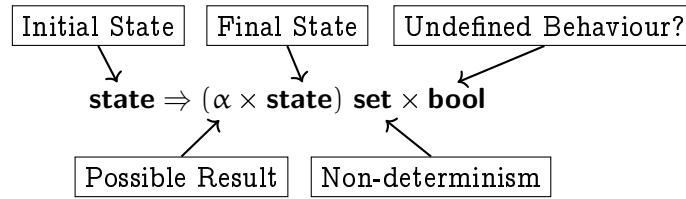
Each of the generated embeddings correspond to the parts of the program written in Cogent. As mentioned in Chapter 2, many functions in Cogent software are *foreign*, i.e. written externally in C. Each of the refinement certificates presented here assume similar refinement statements for each of the foreign functions. Therefore, to fully verify Cogent software, a proof engineer must provide manually-written abstractions of C code, and manually prove the refinement theorems that are automatically generated for Cogent code. As demonstrated in Chapter 2, these foreign functions tend to be reusable library functions. Thus, the cost in terms of verification effort these functions can be amortised by re-using these manually-verified libraries in multiple systems.

5.3.1 SIMPL AND AUTOCORRES

As previously mentioned, we assign a formal semantics to C code using the $C \rightarrow \text{SIMPL}$ parser also used in the verification of seL4 and other projects. SIMPL is an imperative language embedded in Isabelle/HOL with straightforward semantics designed by Schirmer [121], intended for use with program logics such as Hoare Logic for software verification. The language semantics is parameterised by a type used to model all mutable state used in the program. The $C \rightarrow \text{SIMPL}$ parser instantiates this parameter with a generated Isabelle record type containing a field for each local variable in the program, along with a special field for the C heap using the memory model of Tuch, Klein, and Norrish [128].

While we could, in principle, work with the SIMPL code directly, its memory model treats the heap essentially as a large collection of bytes: It does not make use of any of the information from C's type system to automatically abstract heap data structures. This is, in part, due to the nature of manually-written C code, where programmers often subvert the type system using potentially unsafe casts, reinterpreting memory based on dynamic information. Because our code is automatically generated, and does not rely on dynamically reinterpreting memory, we can abstract away from the bits and bytes of the C heap to a higher-level, typed representation — this is where AutoCorres comes in.

As mentioned in Chapter 1, AutoCorres [49, 48] is a tool intended to reduce the cost of manually verifying C programs in Isabelle/HOL. It works by automatically abstracting the SIMPL interpretation of the C code into a shallow embedding using the non-deterministic state monad of Cock, Klein, and Sewell [23]. In this monad, computations are represented using the following HOL type:



Here, **state** represents all the global state of the C program, including any global variables, and a set of *typed heaps*, one for each C type used on the heap in the C program. A typed heap for a particular type τ is modelled as a function $\tau \text{ ptr} \Rightarrow \tau$.

Given an input **state**, the computation will produce a set called *results*, consisting of the possible return value and final **state** pairs, as well as a flag called *failed*, which indicates when undefined behaviour is possible.

In the generated embedding, each access to a typed heap is protected by a *guard* that ensures that the given pointer is valid, to ensure that the heap function is defined for that particular input. Proving that these guards always hold is therefore essential for showing that the program is free of undefined behaviour. When proving refinement from Cogent code, we discharge these obligations by appealing to a globally-invariant *state relation* that implies the validity of all pointers in scope.

Figure 5.3 shows a very simple Cogent program that negates the boolean interpretation of an unsigned integer inside a boxed record. To simplify code generation to C, the Cogent compiler first transforms the program into *A-normal form*, an intermediate representation first developed by Sabry and Felleisen [120]. This form ensures that a unique variable binding is made for each step of the computation, making it easier to convert an expression-oriented language like Cogent to a statement-oriented language like C. This A-normal form also simplifies the refinement tactic used to connect the AutoCorres-abstracted C code to the Cogent deep embedding, described in the next section. As shown in Figure 5.3, the monadic embedding of the C code has a strong resemblance to the A-normal form of the Cogent program. Figure 5.2 describes the notation used in HOL for the monadic embedding, inspired by the **do**-notation of Haskell [85]. Because AutoCorres is designed for human-guided verification, it includes a number of context-sensitive rules to simplify the resulting monadic embedding. For example, it includes features which can simplify reasoning about machine words into reasoning about natural numbers, if it can prove that no overflow occurs. Because we are using AutoCorres as part of an automated framework, most of these abstraction and simplification features are disabled to give highly predictable output. The only significant feature used is the abstraction to the typed heap model.

As can be seen in Figure 5.1, AutoCorres synthesises a refinement proof, showing

do \dots ; \dots od	sequence of statements
$x \leftarrow P$	monadic binding
condition c P_1 P_2	run P_1 if c is true, else run P_2
return v	monadic return
gets f	return the part of the state given by f
modify h	update the state using function h
guard g	program fails if g is false
$P \gg= Q$	monadic bind (desugared)

Figure 5.2: The monadic embedding do-notation.

that the monadic embedding is a true abstraction of the imported SIMPL code. While this refinement proof forms a part of our overall compiler certificate, this proof is entirely internal to AutoCorres, and the SIMPL embedding is not exposed. Therefore, our combined refinement theorem, documented in Section 5.3.6, treats the AutoCorres-generated monadic shallow embedding as the most concrete representation in our overall refinement statement.

5.3.2 AUTOCORRES AND COGENT

While AutoCorres provides some much-needed abstraction on top of C code, the monadic embedding still resembles the generated C code far more than the Cogent code from which it was generated. We still need a technique to validate the code generation phase of the compiler, and synthesise a refinement proof to connect the semantics of Cogent to this monadic embedding.

The C code generation phase of the compiler proceeds relatively straightforwardly, and does not perform global optimisations or code transformations. Transformations such as the aforementioned A-normalisation occur in earlier compiler phases and are verified at a higher level in the overall refinement certificate. As all terms are in A-normal form at this stage, nested sub-expressions are replaced with explicit variable bindings. The refinement framework consists of a series of compositional rules designed to prove refinement in a syntax-directed way, one for each A-normal expression.

REFINEMENT RELATIONS

While our high level view of refinement from de Roever and Engelhardt [31] defines just a single *refinement relation* R that relates abstract and concrete states, three relations must be defined when proving refinement from the Cogent deep embedding (with

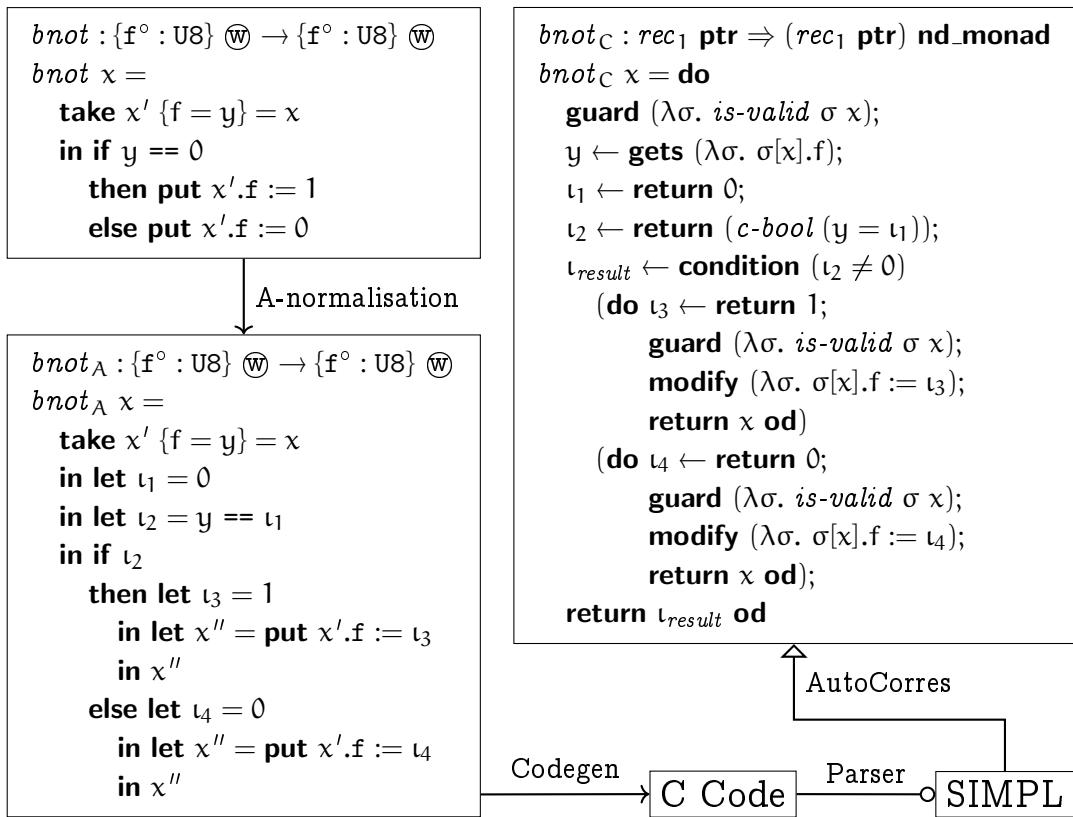


Figure 5.3: An example program, its A-normalisation, and monadic embedding.

representations $\delta ::=$	T	<i>(primitives)</i>
	Fun	<i>(functions)</i>
	Abstract A	<i>(abstract types)</i>
	$\{\overline{f} : \delta\}$	<i>(records)</i>
	$\langle \overline{K} \delta \rangle$	<i>(variants)</i>
	Ptr δ	<i>(boxed types)</i>

$$\boxed{\text{erase}(\cdot) : \tau \rightarrow \delta}$$

$\text{erase}(\mathbf{T})$	=	T
$\text{erase}(\tau \rightarrow \rho)$	=	Fun
$\text{erase}(\mathbf{A} \overline{\tau}_i \mathbb{U})$	=	Abstract A
$\text{erase}(\mathbf{A} \overline{\tau}_i \mathbb{I})$	=	Ptr (Abstract A)
$\text{erase}(\mathbf{A} \overline{\tau}_i \mathbb{W})$	=	Ptr (Abstract A)
$\text{erase}(\{\overline{f}_i^u : \tau_i\} \mathbb{U})$	=	$\{\overline{f}_i : \text{erase}(\tau_i)\}$
$\text{erase}(\{\overline{f}_i^u : \tau_i\} \mathbb{I})$	=	Ptr $\{\overline{f}_i : \text{erase}(\tau_i)\}$
$\text{erase}(\{\overline{f}_i^u : \tau_i\} \mathbb{W})$	=	Ptr $\{\overline{f}_i : \text{erase}(\tau_i)\}$
$\text{erase}(\langle \overline{K}_i^u \tau_i \rangle)$	=	$\langle \overline{K}_i \text{erase}(\tau_i) \rangle$

Figure 5.4: Partial type erasure to determine C representation

the update semantics) to the AutoCorres monadic embedding. The Cogent compiler generates each of these relations after obtaining the monadic shallow embedding and the definitions of its typed heaps from AutoCorres:

1. A *value relation*, written \mathcal{R}_{val} , that relates Cogent update-semantics values (defined in Figure 4.1) to monadic C values. Because AutoCorres generates separate Isabelle types for each C type, this value relation is defined for each generated type using Isabelle's ad-hoc overloading features. Morally, this relation asserts the equality of the two values. For example, the record type in the example in Figure 5.3 would cause the following definitions to be generated:

$$\begin{aligned} (\ell, \quad v_c :: \mathbf{8 \ word}) &\in \mathcal{R}_{\text{val}} \Leftrightarrow (\ell = v_c) \\ (\{f \mapsto u\}, \quad v_c :: \mathbf{rec_1}) &\in \mathcal{R}_{\text{val}} \Leftrightarrow (u, v_c.f) \in \mathcal{R}_{\text{val}} \\ (\mathbf{p}, \quad v_c :: \mathbf{rec_1 \ ptr}) &\in \mathcal{R}_{\text{val}} \Leftrightarrow (\mathbf{p} = v_c) \end{aligned}$$

Note that the definition for the C structure type $\mathbf{rec_1}$ depends on the definition for 8-bit words. The compiler always outputs these definitions in dependency order to ensure that this does not pose a problem.

2. A *type relation*, written $\mathcal{R}_{\text{type}}$, which allows us to determine which AutoCorres

heap to select for a given Cogent type. As with the value relation, the type relation is defined using ad-hoc overloading. It does not relate Cogent types directly to AutoCorres-generated types, but rather a Cogent *representation*, as defined in Figure 5.4. A representation, written as δ , is a partially-erased Cogent type, which contains all the necessary information to determine which C type is used to represent it. Therefore, the usage tags on taken fields and constructors, type parameters in abstract values, the read-only status of sigils, and other superfluous information is discarded. The function $erase(\cdot)$ describes how to convert a type to its representation.

The reasoning behind the decision to relate *representations* instead of Cogent types to C types is quite subtle: Unlike in C, for a Cogent value to be well-typed, all accessible pointers in the value must be valid (i.e. defined in the store μ) and the values those pointers reference must also, in turn, be well-typed. For taken fields of a record, however, no typing obligations are required for those values, as they may include invalid pointers (see the update semantics erasure of the rules in Figure 4.5). In C, however, taken fields must still be well-typed, and values can be well-typed even if they contain invalid pointers. Therefore, it is impossible to determine from a Cogent value alone what C type it corresponds to, making the overloading used for these relations ambiguous.

To remedy this, we additionally include the representation of a value's type inside each update-semantics value u in our formalisation, although this detail is not shown in Figure 4.1. This means that we can determine which C type corresponds to a Cogent value simply by extracting the relevant representation, without requiring recursive descent into the heap or unnecessary restrictions on taken fields.

3. A *state relation*, written \mathcal{R} , which relates a Cogent store μ to a collection of AutoCorres heaps σ . We define $(\mu, \sigma) \in \mathcal{R}$ if and only if for all pointers p in the domain of μ , there exists a value v in the appropriate heap of σ (selected by \mathcal{R}_{type}) at location p such that $(\mu(p), v) \in \mathcal{R}_{val}$.

The state relation cannot be overloaded in the same way as \mathcal{R}_{val} and \mathcal{R}_{type} , because it relates the heaps for every type simultaneously. We introduce an intermediate state relation, \mathcal{R}_{heap} , which relates a particular typed heap with a portion of the Cogent store. Like the other relations, this intermediate relation can make use of type-based overloading. We define \mathcal{R}_{heap} for each C type τ_C

that appears on the heap as follows:

$$\begin{aligned} (\mu, \sigma_{\tau_C}) \in \mathcal{R}_{\text{heap}} &\Leftrightarrow \forall p. \mu(p) = u \wedge (\text{repr}(u), \tau_C) \in \mathcal{R}_{\text{type}} \\ &\Rightarrow \text{is-valid } \sigma_{\tau_C} p \wedge (u, \sigma_{\tau_C}[p]) \in \mathcal{R}_{\text{val}} \end{aligned}$$

where *repr* gives the representation for a value, and *is-valid* σp is true iff the pointer p points to a valid object in the heap σ . The state relation \mathcal{R} over all typed heaps is defined to be merely the conjunction of every $\mathcal{R}_{\text{heap}}$ for each C type used in the program:

$$(\mu, \sigma) \in \mathcal{R} \Leftrightarrow (\mu, \sigma_{\tau_1}) \in \mathcal{R}_{\text{heap}} \wedge (\mu, \sigma_{\tau_2}) \in \mathcal{R}_{\text{heap}} \wedge \dots$$

CORRESPONDENCE

We define refinement generically between a monadic C computation P and a Cogent expression e , evaluated under the update semantics. We denote refinement with a predicate **corres**, similar to the refinement calculus of Cock, Klein, and Sewell [23]. The state relation \mathcal{R} changes for each Cogent program, so we parameterise **corres** by an arbitrary state relation \mathcal{R} . It is additionally parameterised by the typing context Γ and the environment U , as well as by the initial update semantics store μ and typed heaps σ :

DEFINITION 5.1 (Cogent $\rightarrow C$ correspondence).

$$\begin{aligned} \mathbf{corres} \ \mathcal{R} \ e \ P \ U \ \Gamma \ \mu \ \sigma &= (\exists r \ w. U \mid \mu : \Gamma \ [r * w]) \wedge (\mu, \sigma) \in \mathcal{R} \\ &\Rightarrow \neg \text{failed} (P \ \sigma) \\ &\wedge \quad \forall (v_C, \sigma') \in \text{results} (P \ \sigma). \\ &\quad \exists \mu' \ u. \ U \vdash \mu \mid e \Downarrow u \mid \mu' \\ &\quad \wedge \quad (\mu', \sigma') \in \mathcal{R} \wedge (u, v_C) \in \mathcal{R}_{\text{val}} \end{aligned}$$

This definition states that, for well-typed stores μ where the state relation \mathcal{R} holds initially, the monadic embedding of the C program P will not exhibit any undefined behaviour and, moreover, for all executions of P there must exist a corresponding execution under the update semantics of the expression e such that the final states are related by the state relation \mathcal{R} , and the returned values are related by the value relation \mathcal{R}_{val} .

AutoCorres proves that if *failed* is false for a given program, then the C code is type and memory-safe, and is free of undefined behaviour [49]. We prove non-failure as a side-condition of the refinement statement, essentially using Cogent's type system to

$$\begin{array}{c}
\frac{(x \mapsto u) \in \mathbb{U} \quad (u, v_C) \in \mathcal{R}_{\text{val}}}{\mathbf{corres} \mathcal{R} x (\mathbf{return} v_C) \cup \Gamma \mu \sigma} \text{C-VAR} \\
\\
\frac{\begin{array}{c} \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad \Gamma_1 \vdash e_1 : \tau \quad \mathbf{corres} \mathcal{R} e_1 P_1 \cup \Gamma_1 \mu \sigma \\ \forall u v_C \mu' \sigma'. (u, v_C) \in \mathcal{R}_{\text{val}} \Rightarrow \mathbf{corres} \mathcal{R} e_2 (Q v_C) (x \mapsto u, \mathbb{U}) (x : \tau, \Gamma_2) \mu' \sigma' \end{array}}{\mathbf{corres} \mathcal{R} (\mathbf{let} x = e_1 \mathbf{in} e_2) (P \gg= Q) \cup \Gamma \mu \sigma} \text{C-LET} \\
\\
\frac{\begin{array}{c} \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \quad \Gamma_1 \vdash x : \text{Bool} \quad (x \mapsto \ell) \in \mathbb{U} \quad (\ell = \text{True}) \Leftrightarrow (c \neq 0) \\ \mathbf{corres} \mathcal{R} e_1 P_1 \cup \Gamma_2 \mu \sigma \quad \mathbf{corres} \mathcal{R} e_2 P_2 \cup \Gamma_2 \mu \sigma \end{array}}{\mathbf{corres} \mathcal{R} (\mathbf{if} x \mathbf{then} e_1 \mathbf{else} e_2) (\mathbf{condition} c P_1 P_2) \cup \Gamma \mu \sigma} \text{C-IF}
\end{array}$$

Figure 5.5: Some example **corres** rules

guarantee C memory safety during execution. The **corres** predicate can compose with itself sequentially: it both assumes and shows the relation \mathcal{R} , and the additional typing assumptions are preserved thanks to update-semantics type preservation corollary of Theorem 4.1.

Figure 5.5 shows some of the simpler **corres** rules used by our Isabelle tactic to automatically prove refinement. The rule C-VAR for variables, relating them to a monadic **return** operation; the rule C-LET for **let** bindings, relating them to the monadic bind operator $\gg=$; and the rule C-IF for conditional expressions, relating them to the **condition** operation from Figure 5.2. Note that in the rule C-IF, we can assume that the condition expression x is a variable, as the Cogent code is already in A-normal form. In our Isabelle formalisation, we have defined many **corres** rules which validate the entire Cogent language, however they all follow the same basic format as the rules presented in Figure 5.5. The assumptions for these rules fall into three main groups:

1. Each rule for compound expressions includes well-typedness assumptions about some sub-expressions. Theorem 4.1, used to discharge value-typing assumptions in the **corres** definition, also has well-typedness assumptions. Our automated tactic therefore needs access to all of the typing derivations used to construct the overall typing theorem for a program. A mere top-level well-typedness theorem is not sufficient to discharge these obligations. This is why we store each intermediate typing theorem as a tree in Isabelle/ML, as previously mentioned in Section 5.2.
2. Expressions which interact with the heap, such as **take** and **put** for boxed records,

must maintain the relation \mathcal{R} between the Cogent store and the AutoCorres typed heaps. Because the definitions of the typed heaps and the definition of *is-valid* are not provided until after we import the C program, we define these rules generically, parameterised by these AutoCorres-provided definitions. Then, after importing the C program, our framework automatically generates and proves *specialised* versions of the rule for the specific program at hand. This specialisation technique is documented in detail by Rizkallah et al. [114].

3. Expressions such as **take** and **let** which are not made into leaves of the syntax tree by A-normalisation typically have recursive **corres** assumptions for each sub-expression, resolved by recursively applying our tactic. Because each rule is defined for exactly one A-normal Cogent expression, these proofs are syntax-directed and can be resolved by recursive descent without ambiguity or backtracking.

Cogent is a total language and does not permit recursion, so we have, in principle, a well-ordering on function calls in any program. Therefore, our tactic proceeds by starting at the leaves of the call graph, proving **corres** theorems bottom-up until refinement is proven for the entire program.¹

GENERATED THEOREM 5.2 (Update Semantics \sqsubseteq Monadic Embedding). *Let f be the name of a monomorphic and A-normal Cogent function, where $\text{defnOf}(f) = \lambda x. e$ and $\text{typeOf}(f) = \tau \rightarrow \rho$. Let P be the monadic shallow embedding derived from the generated C code for f . Then, for any corresponding arguments u and v_C of the appropriate type, we have:*

$$\forall \mu \sigma. (u, v_C) \in \mathcal{R}_{\text{val}} \Rightarrow \mathbf{corres} \ \mathcal{R} \ e \ (P \ v_C) \ (x \mapsto u) \ (x : \tau) \ \mu \ \sigma$$

This picture is complicated somewhat by the presence of higher order functions in Cogent, which are commonly used for loops and iteration. When higher order functions are involved, the call graph is no longer so clear, as it cannot be strictly determined syntactically. Our framework supports second-order functions by first proving **corres** for all argument functions (e.g. the loop body) before establishing **corres** for the second-order function (e.g. the loop combinator). We could straightforwardly extend this framework to any higher-order functions, but second-order functions were sufficient to cover our case-study file system implementations [4].

¹There are options to achieve this in the presence of recursion. Primitive or structural recursion *a la* Coquand and Paulin [26] is one such option.

5.3.3 MONOMORPHISATION

The next refinement step that is established by translation validation is monomorphisation. The monomorphisation proof shows that the supplied polymorphic Cogent program is an abstraction of the monomorphised equivalent produced by the compiler. At this point, we can operate freely in value semantics without concern for mutable state, as the semantic shift occurs on the monomorphic deep embedding, justified by Theorem 4.3.

The Cogent compiler converts polymorphic programs into monomorphic ones by generating monomorphic specialisations of polymorphic functions based on each type argument used in the program, *a la* Harper and Morrisett [53]. Inside our framework, the compiler generates a renaming function θ that, for a polymorphic function name f_p and types $\vec{\tau}$, yields a specialised monomorphic function name f_m . Just as we assume that foreign functions are correctly implemented in C, we also assume that their behaviour remains consistent under θ . We write two main Isabelle/HOL functions to simulate this compiler monomorphisation phase, each defined in terms of an arbitrary renaming function θ :

1. An *expression* monomorphisation function, $\mathcal{M}_\theta(\cdot)$, which applies θ to any type applications in the expression.
2. A *value* monomorphisation function, $\mathcal{M}_\theta^V(\cdot)$ which applies the expression monomorphisation function \mathcal{M}_θ to each expression inside a value (i.e. in a function value).

Then, we generate a proof which shows that the monomorphised program the Isabelle function produces is identical to that produced by the compiler. If the programs are not structurally identical, this indicates a bug in the compiler.

GENERATED THEOREM 5.3 (Monomorphisation). *Let θ be the generated renaming function and f be a polymorphic function where $\text{defnOf}(f) = \lambda x. e$. Let f_m be a monomorphised version of f generated by the compiler. Then, $\text{defnOf}(f_m) = \lambda x. \mathcal{M}_\theta(e)$.*

Then, it remains to prove that the monomorphic program is a refinement of the polymorphic one:

THEOREM 5.1 (Monomorphisation Refinement). *Let f be a (polymorphic) Cogent function and $\text{defnOf}(f) = \lambda x. e$. Let v be an appropriately-typed argument for f . Let θ be any renaming function. Then for any v' , if $(x \mapsto \mathcal{M}_\theta^V v) \vdash \mathcal{M}_\theta e \Downarrow \mathcal{M}_\theta^V v'$, then $(x \mapsto v) \vdash e \Downarrow v'$.*

Proof. This is proven once and for all by rule induction over the value semantics relation, with appropriate assumptions being made about foreign functions. Typing assumptions are discharged via Theorem 3.6. \square

5.3.4 A NORMAL AND DEEP EMBEDDINGS

Above the semantic shift and monomorphisation stages of our refinement chain, we no longer have any use for deep embeddings. As we are now in the value semantics, shallow embeddings are preferred, as Isabelle’s simplifier can work wonders on pure HOL terms. Therefore, as with Section 5.3.1, we must connect a shallow embedding to a deep embedding. However, this time the deep embedding is the *bottom* of the refinement, and the shallow embedding is comprised of simple pure functions, rather than procedures in a state monad.

This shallow embedding is still in A-normal form and is produced by the compiler: For each Cogent type, the compiler generates a corresponding Isabelle/HOL type definition, and for each Cogent function, a corresponding Isabelle/HOL constant definition. We erase usage tags, sigils and other type system features used for uniqueness type checking, converting the Cogent program to a simple pure term in the fragment of System \mathcal{F} [45, 113] supported by Isabelle. As we have already made use of the type system to justify our semantic shift, we no longer need these type system features in the value semantics.

In addition to these definitions, we automatically prove a theorem that each generated HOL function refines to its corresponding deeply embedded polymorphic Cogent term under the value semantics. Refinement is formally defined here by the predicate **scorres**, which relates a shallowly embedded expression s to a deeply embedded one e when evaluated under the environment V :

DEFINITION 5.2 (Shallow \rightarrow Deep correspondence).

$$\mathbf{scorres} \ s \ e \ V = \forall v. V \vdash e \Downarrow v \Rightarrow (s, v) \in \mathcal{R}_S$$

Here, \mathcal{R}_S is a *value relation*, much like the value relation \mathcal{R}_{val} for **corres** refinement, connecting HOL and Cogent values. Just as with the **corres** refinement, the relation \mathcal{R}_S is defined incrementally, using Isabelle’s ad-hoc overloading mechanism. The automated tactic for **scorres** theorems is substantially simpler than the tactic for **corres**, as **scorres** rules do not require well-typedness, nor do they involve any mutable state or the state relation \mathcal{R} . The tactic proceeds simply by applying specially-crafted intro-


```

record  $\alpha$  T =
  f ::  $\alpha$ 

definition
  bnot :: ( $\delta$  word) T  $\Rightarrow$  ( $\delta$  word) T
where
  bnot x =
    let (x', y) = takef x
    in if (y = 0)
      then x' (| f = 1 |)
      else x' (| f = 0 |)

```

Figure 5.6: Neat embedding of the program from Figure 5.3.

duction rules one by one, which correspond exactly to each form of A-normal Cogent syntax.

The program-specific refinement theorem produced by our tactic is:

GENERATED THEOREM 5.4 (Shallow to Deep refinement). *Let f be the name of an A-normal Cogent function where $\text{defnOf}(f) = \lambda x. e$ and let s be the shallow embedding of f . Then, for any $(v_s, v) \in \mathcal{R}_S$, we have **scorres** $(s\ v_s)\ e\ (x \mapsto v)$. The definition of \mathcal{R}_S ensures that v_s and v are of matching types.*

5.3.5 DESUGARED AND NEAT EMBEDDINGS

Figure 5.6 depicts the top-level *neat* embedding for the example presented previously in Figure 5.3. As can be seen, the Isabelle definitions use the same names and structure as the original Cogent program, making it easy for the user to reason about. In addition to the neat embedding, the compiler also produces a *desugared* shallow embedding, which does not resemble the input program as closely. For example, pattern matching is split into a series of binary **case** expressions. Lastly, the compiler also produces an A-normal shallow embedding, which resembles the A-normal intermediate representation of the code, as seen in Figure 5.3.

Because we are now on the level of purely functional shallow embeddings, the proofs connecting the neat embedding to desugared embedding, and the desugared embedding to the A-normal equivalent, are significantly stronger than refinement — Instead, we prove equality. In Isabelle/HOL, equality is defined based on $\alpha\beta\eta$ -equivalence, which means that this notion of equality admits the principle of functional extensionality.

GENERATED THEOREM 5.5 (Neat and A-Normal equality). *Let s_D be the desugared shallow embedding and s_A be the A-normal shallow embedding of a Cogent function. Then $s_D \stackrel{\alpha\beta\eta}{=} s_A$.*

GENERATED THEOREM 5.6 (Neat and Desugared equality). *Let s_N be the neat shallow embedding and s_D be the desugared shallow embedding of a Cogent function. Then $s_N \stackrel{\alpha\beta\eta}{=} s_D$.*

The proofs of these theorems are simple to generate. Since we can now use equational reasoning with Isabelle's powerful rewriter, we just unfold definitions on both sides, apply extensionality, and the rest of the proof is automatic given the right congruence rules and equality theorems for functions lower in the call graph.

5.3.6 COMBINED PREDICATE FOR FULL REFINEMENT

To show that the top-level neat shallow embedding is a valid abstraction of the C code, the individual refinement certificates presented in the previous sections (Generated Theorems 5.2, 5.3, 5.4, 5.5 and 5.6) are not sufficient. We must also show that the individual refinement relations for each of these stages compose together, producing an overall proof of refinement across the entire chain.

We define our combined predicate **correspondence** connecting a top-level shallow embedding s , a monomorphic deep embedding e of type τ , and a AutoCorres-produced monadic embedding P . It is also parameterised by the C state relation \mathcal{R} , the monomorphism renaming function θ , the update and value semantics environments U and V for the deeply embedded expression e , as well as its typing context Γ , the Cogent store μ and the AutoCorres state σ .

DEFINITION 5.3 (Correspondence).

$$\begin{aligned}
 \text{correspondence } \theta \mathcal{R} s e \tau P U V \Gamma \mu \sigma = & \\
 (\exists r w. U \mid \mu : V : \Gamma \ [r * w]) \wedge (\mu, \sigma) \in \mathcal{R} & \\
 \Rightarrow \neg \text{failed } (P \sigma) \wedge \forall (v_C, \sigma') \in \text{results } (P \sigma). & \\
 \exists \mu' u v. U \vdash e \mid \mu \Downarrow_u u \mid \mu' & \\
 \wedge V \vdash e \Downarrow_v \mathcal{M}_\theta^v v & \\
 \wedge (\mu', \sigma') \in \mathcal{R} \wedge (u, v_C) \in \mathcal{R}_{\text{val}} & \\
 \wedge (\exists r w. u \mid \mu' : \mathcal{M}_\theta^v v : \tau \ [r * w]) & \\
 \wedge (s, v) \in \mathcal{R}_S &
 \end{aligned}$$

Observe that this definition is essentially the combination of our semantic shift preservation theorem (i.e. Theorem 4.1) with the refinement predicates **corres** (Definition 5.1) and **scorres** (Definition 5.2).

Intuitively, our top-level theorem states that for related input values, all programs in the refinement chain evaluate to related output values, propagating up the chain according to the intuitive forward-simulation method of de Roever and Engelhardt [31]. This can of course be used to deduce that there exist intermediate programs through which the C code and its shallow embedding are directly related. The user does not need to care what those intermediate programs are.

GENERATED THEOREM 5.7 (Overall Refinement). *For a Cogent function f , let $\text{defnOf}(f) = \lambda x. e$ and s be the shallow embedding of f . Let f_m be the monomorphised version of f according to renaming function θ , where $\text{typeOf}(f_m) = \tau \rightarrow \rho$, and P is the monadic embedding of the generated C for f_m .*

*Then, we can show that for related input values v_S, v, u and v_C for the pure shallow embedding, value semantics, update semantics and monadic embedding respectively, our **correspondence** predicate holds:*

$$\begin{aligned} & \forall \mu \sigma. (v_S, v) \in \mathcal{R}_S \\ & \quad \wedge (\exists r w. u \mid \mu : \mathcal{M}_\theta^\nu v : \tau \ [r * w]) \\ & \quad \wedge (u, v_C) \in \mathcal{R}_{\text{val}} \\ & \Rightarrow \mathbf{correspondence} \ \theta \ \mathcal{R} \ (s \ v_S) \ (\mathcal{M}_\theta \ e) \ \rho \ (P \ v_C) \ (x \mapsto u) \ (x \mapsto v) \ (x : \tau) \ \mu \ \sigma \end{aligned}$$

The automatic proof of this theorem is straightforward, merely unfolding the definitions of **corres** and **scorres** in Generated Theorems 5.2 and 5.4, applying Generated Theorem 5.3 to establish the equivalence of the definition of f_m with $\mathcal{M}_\theta \ e$, and applying Theorem 4.3 to connect the value and update semantics.

Generated Theorems 5.6 and 5.5 show equality, not mere refinement, and thus they implicitly apply to our overall theorem, extending it to cover these high-level embeddings.

As previously mentioned, this theorem assumes that abstract foreign functions adhere to their specification and their behaviour is unchanged when monomorphised.

5.4 CONNECTING TO ABSTRACT SPECIFICATIONS

Generated Theorem 5.7 shows that, assuming the the refinement relation holds initially, that the C functions are appropriately verified, and that our SIMPL C semantics

accurately capture the semantics of the executed code, any functional correctness property we prove about the neat shallow embedding applies just as well to our C implementation. We stipulate *functional correctness* properties here, as other properties, such as security or timing properties, are not necessarily preserved by refinement.

To prove functional correctness, we must first define a functional correctness specification. This specification can take a variety of forms, but must essentially capture the externally observable correctness requirements of the program, without concern for implementation details or performance. Typically, this specification is highly *non-deterministic*, to allow for abstraction from operational details of the program. For example, the seL4 refinement proof contains a number of layers of specification, where non-determinism increases in each layer up the refinement chain [75]. The Cogent file system verification of Amani et al. [4] specifies each file system operation as a program in a set monad to model this non-determinism.

While they establish a similar property, the manually-written functional correctness proofs for these Cogent file system implementations were significantly easier than the corresponding proofs for seL4. Both proofs involved establishing and maintaining a data invariant about the state of the system, but the data invariant for seL4 included a number of properties that are simply not necessary in Cogent, such as the alignment of objects in memory, the validity of pointers, and the absence of aliasing between heap objects. All of these properties are ensured automatically by Cogent's type system, and justified by Generated Theorem 5.7.

When proving that the file system correctly maintains its invariants to show functional correctness, proof engineers can reason about a significantly simpler embedding with Cogent than they would with, say, a C program. Because our neat shallow embedding consists only of pure functions, our specifications and embeddings need not deal with any mutable state. Unlike in seL4, there is no need to resort to cumbersome machinery like separation logic [112] to show that data invariants are maintained: each function can be reasoned about by simply unfolding its definition, and separation between objects x and y follows trivially from x and y being separate variables.

5.5 EVALUATION

Our decision to write the Cogent compiler tool-chain in Haskell but the refinement framework and proof tactics in Isabelle/ML allows the Cogent tool-chain to be used outside the theorem prover, while still allowing our refinement framework to build on the existing C and AutoCorres framework available in Isabelle/HOL.

On the other hand, this choice leads to some complexity in designing the interface between these components. This is illustrated by the well-typedness proof in Section 5.2, where the Cogent compiler generates a certificate tree with the necessary type derivation hints. Initially, a naïve format consisting of the entire derivation tree was used, resulting in gigabyte-sized certificates. Various compression techniques reduced this to a reasonable size (a few megabytes), but these certificates still take some time to process. It would be possible to avoid these certificates entirely by duplicating the entire type inference algorithm presented in Chapter 3 in Isabelle/ML, but this would increase the code maintenance burden significantly.

The use of pre-existing mature tools to give C code a semantics in Isabelle/HOL, namely the $C \rightarrow \text{SIMPL}$ parser and AutoCorres, is a pragmatic choice aimed at reducing the effort required to build our refinement framework, ensuring that our C semantics lines up with other large-scale C verification projects, and enabling integration with the seL4 verification specifically. Unfortunately, however, these tools are particularly time-consuming when processing Cogent-generated C code. For the file system implementations of Amani et al. [4], these tools take anywhere from 12 to 32 CPU hours to generate the monadic embedding of the generated C code. While the time taken to establish our refinement certificate does not endanger the trustworthiness of Cogent software, it does make our automatic verification framework less useful as a debugging tool. As discussed in Chapter 6, future work involves integrating robust specification-based testing tools to Cogent, to improve turn-around time for debugging and to allow verification to be attempted only after developers are confident that the code is indeed correct.

Klein et al. [75] report that approximately one third of the overall verification effort for seL4 went into the second refinement step, connecting the intermediate executable specification to the C code. This estimation is not including the effort that went into developing re-usable libraries and frameworks. Our Generated Theorem 5.7 encompasses this step and more, because, as previously discussed, our intermediate executable specification (the neat embedding) is significantly more high-level. Therefore, we can confidently predict that, where Cogent can be used to implement a system, our refinement framework will reduce the effort of verifying that system by at least a third, relative to existing C verification techniques. Because our neat embedding is higher-level than the intermediate executable specification of seL4, the savings are possibly even greater.

In the course of their verification of two file system operations, Amani et al. [4]

found six defects in their already-tested file system implementations. The effort for verifying the complete file system component chain for these operations was roughly 9.25 person months, and produced roughly 13,000 lines of proof for the 1,350 lines of Cogent code. This compares favourably with traditional C-level verification as for instance in seL4, which spent 12 person years with 200,000 lines of proof for 8,700 source lines of C code. Roughly 1.65 person months per 100 C source lines in seL4 are reduced to ≈ 0.69 person months per 100 Cogent source lines with our framework. In future, we plan to implement a data-description language as an extension to Cogent (see Chapter 6), which will automate the functional correctness verification of approximately 850 lines of deserialisation and serialisation code in these file system implementations. These 850 lines of Cogent code required ≈ 4000 lines of proof to verify, taking approximately 4.5 person months. With this added automation, the cost of verification can be reduced even further.



Functional programmers have long recognised, and advocated for, the benefits afforded by reasoning over pure functions. For the first time, our refinement framework allows these benefits to be enjoyed by proof engineers verifying low-level operating system components, without enlarging the trusted computing base. Building on the key refinement theorem given to us by our uniqueness type system (Theorem 4.3), our refinement framework makes use of multiple translation validation techniques to establish a long refinement chain. This allows engineers to reason about Cogent code on a high level in Isabelle/HOL and have confidence that their reasoning applies just as well to the C implementation we generate.


Cogent not only allows non-experts in formal verification to write provably-safe code, it is also a key step towards lowering the effort and complexity for the full mechanical verification of operating system components against high-level formal specifications.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

“Begin at the beginning,” the King said, very gravely, “and go on till you come to the end: then stop.”

Lewis Carroll, *Alice in Wonderland*

UR work has already shown promising results, both in terms of performance (see Chapter 2) and verification (Chapter 5), however the file system implementations and verification conducted as a case-study [2, 4] bring several opportunities into focus for future improvements to our framework.

While the performance of the file systems is comparable to C in terms of I/O microbenchmarks, macrobenchmarks show that the Cogent implementations have some significant performance overheads. Adding various optimisation passes to the Cogent compiler would improve performance, but presents a substantial verification challenge. We discuss compiler optimisations in Section 6.1. Certain performance issues resulting from expensive marshalling and unmarshalling of operating system data structures can also be resolved by our proposed data description language extension to Cogent, described in Section 6.3.

With regards to verification, our overall refinement certificate could be extended in a number of ways, to eliminate assumptions in our framework, and to enable more automatic verification of more properties and more code. A natural future extension would be to extend the certificate *down* below C to the binary itself, eliminating one of the main assumptions of our framework. We discuss binary verification in Section 6.2. The certificate can also be extended *upward* to allow certified code to be written at an even higher level, reducing the effort required to connect Cogent to a high-level

specification. One method to accomplish this would be to increase the specification power of our type system, to enable more domain-specific and high-level properties to be automatically checked at compile time. We discuss options for type system extension in Section 6.4. The data description language extension described in Section 6.3 also serves as a highly abstract specification of serialisation and deserialisation code in particular.

Our framework could also be extended *outward*, removing limitations of the language to enable more kinds of code to be written and verified with Cogent. One of the most glaring limitations of Cogent is the intentional absence of recursion, to ensure that all Cogent programs terminate. We discuss relaxing or removing this limitation in Section 6.5. Another clear avenue for extension is support for concurrency, discussed in Section 6.6.

Lastly, Section 6.7 introduces our plan to improve the development methodology of Cogent, reducing turn-around time for debugging by integrating property-based testing with verification.

6.1 OPTIMISATIONS

Currently, the Cogent compiler relies primarily on the underlying C compiler for optimisations. Generated code displays patterns which are uncommon in handwritten code and therefore might not be picked up by the C optimiser, even if they are trivial to optimise. For example, due to the A-normal representation used by the Cogent compiler, the generated C code is already quite close to single static assignment (SSA) form used internally by C compilers `gcc` and `clang`, however these compilers do not always recognise this and optimise accordingly. Generating a compiler's SSA representation directly, such as LLVM IR, may eliminate these problems, and projects to verify subsets of LLVM IR exist for us to target [135], however this would imply significant changes to our verification infrastructure.

If we were to add optimisation passes to the Cogent compiler itself, we would of course have to verify such optimisations, or at least establish the soundness of any code transformation via translation validation. Adding significant optimisations to the Cogent-to-C stage of our framework would complicate the syntax-directed correspondence approach described in Chapter 5. Cogent-to-Cogent optimisations, however, are straightforward — the ease of proving A-normalisation correctness over the shallow embedding via rewriting suggests that this is the right approach in our context. Many optimisations are described as equational rewrites for functional languages (e.g. stream

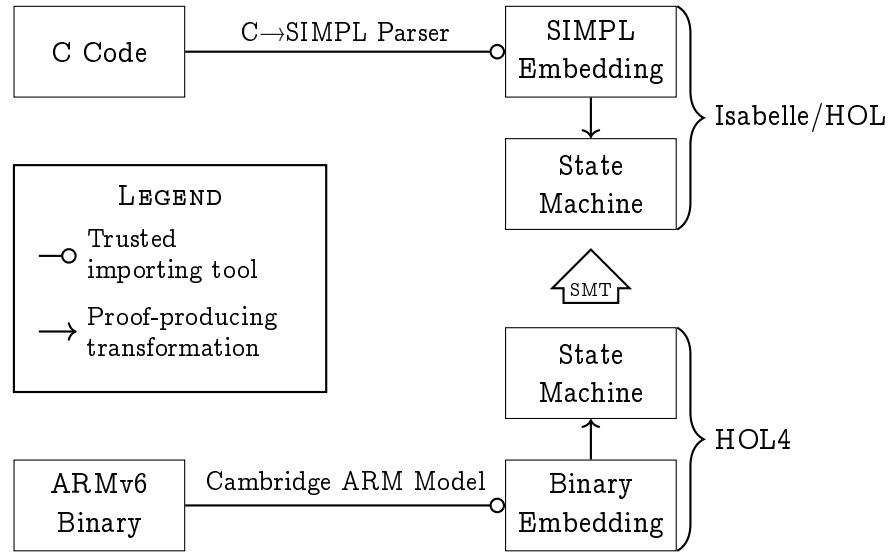


Figure 6.1: The binary verification framework of Sewell, Myreen, and Klein [122]

fusion [27]). In particular, some of the source-to-source optimisations discussed by Chlipala [20] seem promising for Cogent.

6.2 BINARY VERIFICATION

Extending our refinement down to the binary level is an obvious next step, ensuring that the code that executes on hardware faithfully implements the semantics of our generated C and therefore of the Cogent code itself.

Our generated C code can be compiled with the verified C compiler CompCert [81], but, as previously mentioned, CompCert uses a different C semantics to our C→SIMPL parser, and as CompCert is implemented in Coq and our parser in Isabelle, connecting the two would require either the Cogent framework (including AutoCorres) to be re-implemented in Coq or the CompCert C semantics to be re-implemented in Isabelle, both of which would require a large investment of engineering effort. A similar effort would be required if we were to target the Coq formalisation of LLVM IR [135].

A more directly useful means of binary assurance for us would be the binary verification framework of Sewell, Myreen, and Klein [122], which establishes refinement via translation validation from the SIMPL C semantics to an imported ARMv6 binary using the Cambridge ARM model [41]. As shown in Figure 6.1, this framework works by simulating both the SIMPL code and the binary, represented as state machines, and generating refinement obligations for each execution step that are discharged by

an automatic SMT solver. It has been successfully applied to the seL4 microkernel compiled with the gcc compiler with local optimisations enabled (-O2).

Preliminary testing on binaries produced from Cogent-generated C code indicate that the large structs that Cogent sometimes generates triggers an edge case that causes this refinement tool to fail. Such incompleteness notwithstanding, Cogent code also depends heavily on the C compiler's optimiser for performance. Establishing refinement with this tool in the presence of heavy code optimisations remains a significant challenge.

While this framework allows us to remove the C compiler and C→SIMPL parser from our *trusted computing base*, the set of software and hardware for which we must assume correctness, it also adds an SMT solver and the Cambridge ARM model. While still complex pieces of software, these components are smaller than an optimising C compiler and our C→SIMPL parser. The risks to trustworthiness can be further ameliorated here by using multiple different SMT solver implementations.

6.3 DATA DESCRIPTION LANGUAGE

As mentioned in Chapter 5, the exact structure used to represent states is a *parameter* to the definition of SIMPL, and therefore our C→SIMPL parser is free to choose a structure that mirrors the C code closely. Typically, the state structure is an Isabelle/HOL record with fields for each of the stack-allocated local variables used in the code, along with a field for the heap, represented using the memory model of Tuch, Klein, and Norrish [128]. A consequence of this representation is that, while the heap memory model used allows for pointer arithmetic, unions, and type-casting of heap memory, the view of the stack is significantly abstracted. With this state definition, it is not well-defined to take a pointer to a stack-allocated variable, nor to reinterpret stack memory as a different type. C code that performs such operations is rejected by the parser.

To accommodate these restrictions, the Cogent compiler chooses very straightforward memory layouts to represent algebraic data types. For record (product) types, each field is laid out in memory as a C struct. For variant types, a special value called the *tag*, which indicates the constructor used for the variant, is stored in a struct along with several sub-structures for the constructor parameters, only one of which will contain meaningful data.

Other components of the system that are implemented directly in C have significantly more freedom in how they lay out data. The Linux kernel, for example, typically

chooses much more exotic data representations, using techniques such as:

- `BITFIELDS` — Several boolean flags are very often represented using the individual bits of a machine word.
- `TYPE TAGS` — The value of one part of a data structure can determine how to interpret/typecast another part of a data structure, for example with tagged unions or tagged void pointers.
- `CONTAINER PATTERN` — Kernel-defined structures are often nested within component-specific structures at offset zero. This means that a component-specific object can be safely cast to the more general kernel-defined type for use within the kernel, and then cast back to the component-specific type when returned to the component by the kernel.
- `PADDING AND ALIGNMENT` — Often, blank space is left in the object intentionally to account for architecture-specific alignment considerations.
- `DYNAMICALLY SIZED OBJECTS` — Kernel data types often contain values that determine the size of other objects. For example, an array is often paired with its length as an integer.

Seeing as none of these techniques can currently be used by the Cogent compiler when laying out types in memory, C types such as these are typically modelled as abstract types in Cogent. Conversion functions between these data structures and Cogent data types must be manually written in C and painstakingly verified in Isabelle/HOL. These functions are very tedious to write, frustratingly error prone, and have a negative impact on performance. The bulk of the significant performance overheads observed in the file system macro-benchmarks presented in Chapter 2 can be attributed to this code.

Therefore, we aim to extend the compiler and verification framework of Cogent to better support these kinds of memory layouts, transparently representing them as algebraic data types.

To this end, we are in the process of designing a data description language, called Dargent, that describes how a Cogent algebraic data type may be laid out in memory, down to the bit level. An initial proposal paper on Dargent was published at ISoLA in 2018 [98]. Data descriptions in this language will influence the definition of the refinement relation to C code generated by the compiler. Eventually, we hope to make

sizes	s	$::=$	$nB \mid nb \mid nB + mb$	
layout expressions	l	$::=$	s	(block of memory)
			$ L$	(another layout)
			$ l \text{ at } s$	(offset operator)
			$ \mathbf{record} \{f_i: l_i\}$	(records)
			$ \mathbf{variant} (l) \{K_i (n_i): l_i\}$	(variants)
declarations	d	$::=$	$\mathbf{layout} L = l$	
layout names	L			
numbers	n, m	\in	\mathbb{N}	

Figure 6.2: The grammar of our Dargent prototype.

this language flexible enough to accommodate any conventional representation of an algebraic data type in memory.

When fully realised, our framework will allow the programmer to write code as normal, manipulating ordinary algebraic data types, and after compilation the generated C code will manipulate kernel data structures directly, without copying and synchronisation at run-time. This will improve performance by eliminating redundant work, dramatically simplify the process of integrating C and Cogent code, and make it possible to verify more code with Cogent, rather than using cumbersome C verification frameworks.

There are numerous data description languages and calculi which allow for automatic synthesis of serialisation and deserialisation code [37, 38, 8, 89, 134, 9], some of which integrate with functional languages [39, 83]. These languages, unlike Dargent, focus entirely on parsing and pretty printing, and often for file formats rather than in-memory data structures. By contrast, Dargent is primarily intended for transparent data refinement, where marshalling code is avoided entirely by accessing the data directly in its bespoke format.

Figure 6.2 gives a grammar for the syntax of Dargent descriptions, which are made of one or more **layout** declarations, which give names to particular *layout expressions*. Each such expression describes how to lay out a data type in memory. Primitive types such as integer types, pointers, and booleans are laid out as a contiguous block of memory of a particular size. For example, a layout consisting of a single four byte block would only be appropriate to describe Cogent types that occupy four contiguous bytes of memory, such as U32 or pointer types.

layout *FourBytes* = 4B

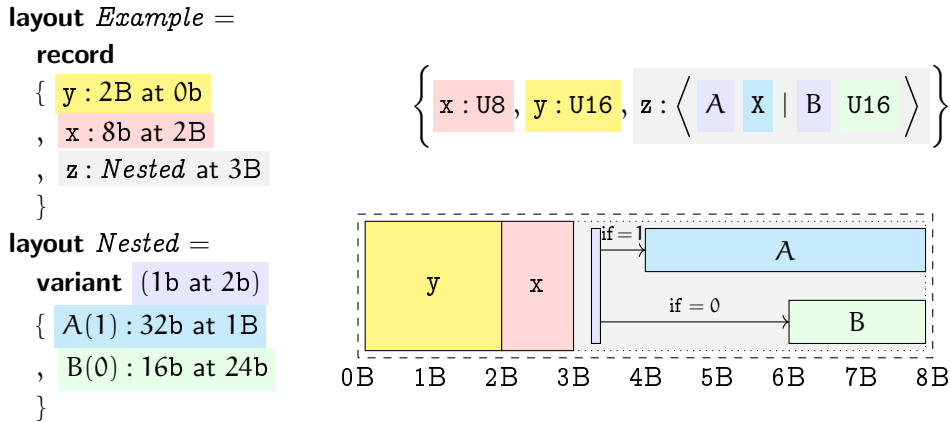


Figure 6.3: A Cogent type laid out according to a Dargent specification.

Layouts for record types use the **record** construct, which contains sub-expressions for the memory layout of each field. Seeing as we can specify memory blocks down to individual bits, we can naturally represent records of boolean values as a bitfield:

layout Bitfield = record {**x** : 1b,**y** : 1b at 1b,**z** : 1b at 2b}

Here the at operator is used to place each field at a different bit offset, so that they do not overlap. If two record fields reserve overlapping blocks, the description is rejected by the compiler.

Layouts for variant types use the **variant** construct, which firstly requires a layout expression for the *tag* data, which encodes the constructor in the variant which is being used. Then, for each constructor in the variant, a specific tag value is given as well as a layout expression for any additional data provided in the variant type for that constructor.

Figure 6.3 gives an illustrative example of a Dargent description in our current prototype. We describe a memory layout for a Cogent record type containing two numbers and a variant, $\{x : U8, y : U16, z : \langle A \ X \ | \ B \ U16 \rangle\}$, where X is a boxed abstract type. As can be seen from the ordering of the fields y and x , fields may be placed in any order and at any location. This allows us to accommodate data layouts where certain parts of the data type must appear at particular offsets, such as with the previously-mentioned container pattern. It also makes it possible to leave unreserved space in between fields, accommodating data layouts which do this to respect padding or alignment constraints in the architecture.

The variant field z is represented according to the *Nested* description, offset by

three bytes. That *Nested* description reserves the third bit of the first byte (the fourth byte of the original object) to determine which of the two constructors A and B is active. If the bit is 1, the constructor A is active, with the additional X pointer payload stored at a one-byte offset (the fifth to eighth bytes of the original object). If the bit is 0, the constructor B is active, with the U16 payload stored at a three byte offset (the seventh and eighth byte of the original object).

A Cogent type τ represented on the heap using a particular Dargent layout L_1 will result in different C code than the same type τ represented using a different layout L_2 . While they are identical on the abstract level, on the concrete level they are not interchangeable. Therefore, we must extend the Cogent type system such that identical types represented differently are distinguished, by *tagging* types with their representation.

Seeing as our Dargent descriptions only apply to objects allocated on the heap, and all Cogent types represented on the heap have *boxed* sigils (i.e. \textcircled{W} or \textcircled{F}), the natural place to add these tags is on the sigils themselves. We will add an additional representation parameter l to the sigil. For example, a writable record type might be written $\{f : \textcircled{U} U32, g : \textcircled{U} U8\} (\textcircled{W} L)$, associating the layout L to the type. The type checker is then responsible for ensuring two properties:

1. That the layout is well-formed, i.e. that it does not reserve overlapping blocks of memory, and that it does not reference any unknown **layout** declarations.
2. That the type can actually be represented according to the layout description. For example, a 32-bit word U32 or a pointer on a 32-bit architecture could both be represented by our earlier description *FourBytes*, but a U64 value could not.

These descriptions would then be used to generate the correspondence relation for each boxed type used in the program, forming the basis of the refinement proof from Cogent to C.

For unboxed types, we are still constrained by the restrictions on stack memory layouts from our $C \rightarrow \text{SIMPL}$ parser. Therefore, data types with the \textcircled{U} unboxed sigil, which are stack allocated, do not carry an associated layout and are represented using the simple layouts described earlier.

This extension already represents a dramatic overhaul of the C code generation and therefore of the refinement process described in Chapter 5. Once we have developed an initial proof of concept, we plan to extend Dargent in a number of ways, including tighter integration with C compilers to inform data layouts, specification of constraints

and automatic generation of data validation procedures, support for endianness annotations and error checking protocols to enable parsing wire formats, support for dynamically-sized data by enriching Cogent's type system (see Section 6.4), and allowing functions to be defined parametrically on layouts to cleanly separate concrete layout concerns from abstract logic. Full details on these planned extensions are documented in our separate proposal [98].

6.4 RICHER TYPE SYSTEM

While Cogent already automates at least a third of a typical functional correctness refinement theorem to C, as described in Chapter 5, proving full functional correctness still requires a significant amount of manual proof effort. Enriching the Cogent type system is a natural avenue to further reduce this effort. The more properties we can encode in the type system and check automatically, the less will have to be manually established by a proof engineer in post-hoc verification.

Dependent types in the spirit of Martin-Löf [86], such as those implemented in Agda [95], Idris [15], and ATS [17] offer nearly unlimited specification power, and computational type theories such as those in NuPRL [25] lift even the restriction of decidable type checking, but such theories still require manual (albeit often tool-assisted) proof effort to show that the given programs meet the specifications in their types.

The subset of dependent types where such refinement theorems are automatically established are called *refinement types* [42]. These type systems allow types to be annotated with *refinements*, assertions that denote a subset of the annotated type. These predicates can range from basic data invariants to full specifications of functional correctness. For example, a function to give the midpoint of two natural numbers might be defined with a full specification like:

$$\textit{midpoint} : \{a : \mathbb{N}\} \rightarrow \{b : \mathbb{N} \mid a \leq b\} \rightarrow \{x : \mathbb{N} \mid x = a + \frac{1}{2}(b - a)\}$$

Or an over-approximation of the specification that may be easier to automatically verify:

$$\textit{midpoint} : \{a : \mathbb{N}\} \rightarrow \{b : \mathbb{N} \mid a \leq b\} \rightarrow \{x : \mathbb{N} \mid a \leq x \leq b\}$$

During type checking, *verification conditions*, i.e. independent logical propositions that must hold in order for the refinements to hold, are gathered and dispatched to an external automated theorem prover such as an SMT solver. Tools such as Liquid-Haskell [130] are successful examples of this technique being applied to languages in

wide use. The verification power of this approach is demonstrated by the language F^* [124], which has been successfully used to verify cryptographic algorithms with refinement types [111]. Automated theorem provers have been used to great effect to solve verification conditions in imperative languages too, such as in Dafny [119] and in Whiley [105], as well as in the VCC tool for C programs [28].

Adding a feature such as refinement types to Cogent has the potential to drastically reduce verification effort, depending on the expressive power of the refinements, as well as potentially reducing debugging turn-around time, as SMT push-button verification is faster than manual proof in an interactive theorem prover, although less powerful.

Of course, relying exclusively on such automated theorem provers increases the size of our trusted computing base, as we must now assume the correctness of the SMT solver to have confidence in the theorems they prove. Furthermore, SMT solvers can be unreliable, failing to prove theorems that could be easily shown to hold by hand. Both of these problems could be remedied by formalising our verification condition generator in an interactive proof assistant such as Isabelle/HOL, and requiring the user to supply proofs of each condition in an interactive proof script. Note that such proof scripts may themselves invoke SMT solvers [14], but their certificates are replayed in the proof system of Isabelle/HOL and therefore do not add to the size of the trusted computing base. The proof assistants Coq [64] and PVS [103] support variants of refinement types where, rather than relying on an SMT solver, verification conditions can be deferred for manual proof *after* the program structure is defined [123, 117], which may provide inspiration for the formalisation of a verification condition generator for Cogent.

Another potential complication is from the integration of refinement and uniqueness types. More generally, the combination of dependent and linear types is a fraught research landscape, with earlier attempts [16, 77, 129] strictly bifurcating the context of *intuitionistic* variables, on which types may depend, and *linear* variables, which may not be used in a dependent type. Such a restriction is much too onerous for Cogent, where we would often wish to specify refinements that use linear variables. Fortunately, recent work from McBride [88] and Atkey [7] offers a means to unite linear and dependent types in a more satisfactory way, by distinguishing between use of a variable in the context of a computation and use in the context of a specification.

6.5 RECURSION AND NON-TERMINATION

A significant amount of the code in our case studies that must still be implemented in C consists of higher order functions for iteration over data structures which are

abstract on the Cogent side. Moving this code into Cogent, by extending Cogent to support recursion, promises to reduce verification effort by reducing the amount of C code that must be verified.

To be able to reason equationally about Cogent shallow embeddings in Isabelle, however, we must still ensure totality. Thus, general recursion is still not an option for Cogent. Existing total functional languages such as Agda [95] include a termination checker, where only *structural recursion* on inductive types (and productive corecursion on coinductive types) is permitted. It is possible to convert such recursions into code in terms of eliminators, corresponding to the induction principle of a type [26].

Certain processes such as drivers, and indeed operating systems themselves, are also *not supposed to terminate*, but rather to run continuously while the machine is on, responding to events from running software or hardware. At the moment, such processes are implemented in Cogent with a C shell, which awaits events in a loop and executes a Cogent function whenever an event occurs. These are clearly better specified as productive corecursive programs. Extending Cogent to support corecursion will likely be ultimately needed in order to support moving these particular C loops into Cogent. Fortunately, Isabelle also supports corecursive shallow embeddings, providing us with a direct translation target.

6.6 CONCURRENCY

So far, Cogent has only been used to implement individual processes running on a system, such as a specific file system server or hardware driver, where sequential semantics have been sufficient. Extending Cogent to support a whole systems, however, necessitates a semantics that models multiple concurrently executing processes which may interact with each other. Adding support for concurrency to Cogent presents a number of interesting research opportunities. There are many type systems based on linear logic and linear types that are intended to describe concurrent systems, such as session types [34], which could be cleanly integrated into Cogent's existing linear type system.

Verifying a concurrent Cogent would also necessitate a verified concurrent semantics for each level in our refinement chain, including for C. There is a concurrent version of SIMPL used for the verification of the eChronos real-time operating system [5] called COMPLX [3]. It is intended for verification of very low-level, potentially racy code that interacts with shared memory, where concurrency abstractions are not needed. Therefore its verification condition generator is based on the foundational Owicki/Gries

method [102] rather than more modern techniques such as rely/guarantee [67] or concurrent separation logic [101]. In contrast, the kind of concurrent systems described by session types and more generally process algebras [58, 92, 11] are usually significantly higher level, relying on abstractions such as message passing. Formally connecting these two in our refinement certificate is therefore a very significant challenge.

6.7 TESTING FRAMEWORKS

Property-based testing, in the style of QuickCheck [22], is a promising technique for reducing the cost of verification. Similar to formal verification, property-based testing uses a specification of the desired properties of a unit under test. From this specification it automatically generates test cases to search for counter-examples. Hughes [62] and Arts et al. [6] show that property-based testing is effective for detecting bugs and finding inconsistencies in specifications. In their work on secure information flow, Hritcu et al. [61] observe that property-based testing is especially valuable in the context of formal verification, as it can eliminate the wasted effort of trying to prove a faulty or ill-specified system correct.

Furthermore, as we have previously argued [19], the costs and downsides of using property-based testing apply to a much lesser degree in the context of verification. Design considerations such as modularity and reduced coupling, essential for property-based testing, are also essential for verification.

When maintaining already-verified systems, a proof might require significant changes whenever the code changes, e.g. when optimising algorithms, whereas property-based testing only requires developer input when the specification changes. Therefore, testing can provide quick feedback on the likely correctness of the change, reducing code maintenance cost. Furthermore, all verification projects depend on some assumptions such as correctness of hardware or external software. Some of these assumptions can be tested to increase our confidence in the correctness of the overall system.

In large scale verification projects such as seL4, the proof engineers and systems engineers are usually in separate teams. Property-based testing provides a way for systems programmers, who may not be well versed in formal methods, to use and manipulate specifications in the form of tests.

For all these reasons, we intend to implement a property-based testing framework for Cogent. We describe the design of our prototype testing framework in our paper at PLOS 2017 [19], as well as the challenges we expect to face. In this proposal, we describe property-based testing for refinement properties much like the properties used

to verify functional correctness. This allows the same set of properties to be used for both verification and testing.



Even without all of the above-mentioned extensions, Cogent has achieved its stated goal: To reduce the cost of formally verifying functional correctness of low-level operating systems components. It achieves this by allowing users to write code at a high level of abstraction, in the native language of interactive proof assistants — purely functional programming.

So as not to compromise on efficiency, Cogent uses a uniqueness type system, allowing purely functional code to be compiled to efficient destructive updates, eliminating the need for a garbage collector, and ensuring memory safety. To make this type system usable without verbose type annotations, we include a powerful type inference algorithm in the Cogent compiler.

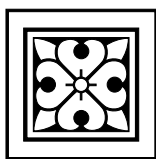
To ensure the validity of functional correctness theorems proven about Cogent code, the compiler generates a certificate to show that all functional correctness properties proven about the Cogent code also apply to the generated C implementation. This generation process relies on a number of key theorems, including the vital proof that the imperative interpretation of Cogent is a refinement of the functional interpretation.

To interact with existing systems and to enable greater expressivity, we include a foreign function interface that allows the programmer to mix Cogent code with C code. It is possible to verify C code and compose these proofs with the proofs generated by the Cogent framework.

Our two case study file systems serve to validate our approach, with key operations of one file system verified for functional correctness. The results from these studies confirm our hypothesis: the language ensures a greater degree of reliability by default compared to C programming, verification effort is reduced by at least one third, and the performance of the generated code is, while slower, still comparable to native C implementations, and acceptable for realistic file system implementations.

As outlined in this chapter, there are many opportunities for improvement in many aspects. Nonetheless, Cogent as it is now is already a significant milestone, bringing the grand goal of affordable, verified, high-assurance software one step closer to reality.





IF you will permit some vain pontification on the last page of my thesis, I would like to reflect on this undertaking, and on the dramatic effect it has had on my thinking. My once-co-supervisor Toby Murray said that all graduate students enter into a valley of despair where they no longer believe in the value of their work. Certainly I am no counter-example. I do not even know if I successfully found my way out of it.

I think that this process of self-doubt is universal because it is fundamentally the experience of education. Before we know a thing, we cannot know what it means, we only have an impression, a signifier. This impression may purport to be revolutionary, exciting, or simply interesting. But, once we truly learn the thing, the impression always seems like a lie. By learning, we have seen through the signifier and to the signified: Now we know the full detail of the thing, all of its darker sides and well-hidden skeletons. The experience of education, then, is one of disillusionment. If we are perennially disappointed by the failure of our interest to meet our expectation, then surely all education leads to doubt and to despair.

How, then, does one escape from the trap? What meaning or motive can I find in this work I have done? To quote from my favourite novel [36]:

The more I reread this list the more I am convinced it is the result of chance and contains no message. But these incomplete pages have accompanied me through all the life that has been left me to live since then; I have often consulted them like an oracle. . .

Even at the height of my self-doubt, when I no longer believe what I have written and I no longer positively appraise my own work, this thesis still means one thing to me: *I did this*. This has been my life for some years. And, throughout that time, I have learned — I have been educated. My doubt is what tells me so.

I think most people who have written a significant body of work will be the first to criticise it. They know the most about it, and in knowing it, they have lost the ability to evaluate it. I can't speak for the value of my work. It is too tied up with my own self-worth that I will never be sure of it.

It is cold in the scriptorium, my thumb aches. I leave this manuscript, I do not know for whom; I no longer know what it is about: *stat rosa pristina nomine, nomina nuda tenemus*.

BIBLIOGRAPHY

- [1] The High Assurance Systems Programming Project (HASP). *The Habit Programming language: The revised preliminary report*. Nov. 2010. URL: <http://hasp.cs.pdx.edu/habit-report-Nov2010.pdf>.
- [2] Sidney Amani. “A methodology for trustworthy file systems”. PhD thesis. University of New South Wales, 2016.
- [3] Sidney Amani, June Andronick, Maksym Bortin, Corey Lewis, Christine Rizkallah, and Joey Tuong. “COMPLX: a verification framework for concurrent imperative programs”. In: *Certified Programs and Proofs*. Paris, France: ACM, Jan. 2017, pp. 138–150. DOI: <https://doi.org/10.1145/3018610.3018627>.
- [4] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O’Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. “Cogent: Verifying High-Assurance File System Implementations”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta, GA, USA, Apr. 2016, pp. 175–188. DOI: [10.1145/2872362.2872404](https://doi.org/10.1145/2872362.2872404).
- [5] June Andronick, Corey Lewis, Daniel Matichuk, Carroll Morgan, and Christine Rizkallah. “Proof of OS scheduling behavior in the presence of interrupt-induced concurrency”. In: *International Conference on Interactive Theorem Proving*. Ed. by Jasmin Christian Blanchette and Stephan Merz. Nancy, France: Springer, Aug. 2016, pp. 52–68.
- [6] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. “Testing AUTOSAR software with QuickCheck”. In: *International Conference on Software Testing, Verification and Validation Workshops*. Apr. 2015, pp. 1–4. DOI: [10.1109/ICSTW.2015.7107466](https://doi.org/10.1109/ICSTW.2015.7107466).

-
- [7] Robert Atkey. “Syntax and Semantics of Quantitative Type Theory”. In: *Logic in Computer Science*. LICS '18. Oxford, United Kingdom: ACM, 2018, pp. 56–65. ISBN: 978-1-4503-5583-4. DOI: 10.1145/3209108.3209189.
- [8] Godmar Back. “DataScript: A Specification and Scripting Language for Binary Data”. In: *Generative Programming and Component Engineering*. Springer-Verlag, 2002, pp. 66–77. ISBN: 3-540-44284-7.
- [9] Julian Bangert and Nickolai Zeldovich. “Nail: A Practical Tool for Parsing and Generating Data Formats”. In: *Operating Systems Design and Implementation*. Broomfield, CO, 2014, pp. 615–628. ISBN: 978-1-931971-16-4.
- [10] Erik Barendsen and Sjaak Smetsers. “Conventional and Uniqueness Typing in Graph Rewrite Systems”. In: *Foundations of Software Technology and Theoretical Computer Science*. Vol. 761. Lecture Notes in Computer Science. 1993, pp. 41–51.
- [11] J. A. Bergstra. *Handbook of Process Algebra*. Ed. by A. Ponse and Scott A. Smolka. New York, NY, USA: Elsevier Science Inc., 2001. ISBN: 0444828303.
- [12] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. “Verified Correctness and Security of OpenSSL HMAC”. In: *Security Symposium*. Washington, DC, US: USENIX, Aug. 2015, pp. 207–221. ISBN: 978-1-931971-232.
- [13] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. “Linear Haskell: Practical Linearity in a Higher-order Polymorphic Language”. In: *Principles of Programming Languages*. ACM, Dec. 2017, 5:1–5:29. DOI: 10.1145/3158093.
- [14] Sascha Böhme and Tobias Nipkow. “Sledgehammer: Judgement Day”. In: *International Conference on Automated Reasoning*. Edinburgh, UK: Springer-Verlag, 2010, pp. 107–121. ISBN: 3-642-14202-8, 978-3-642-14202-4. DOI: 10.1007/978-3-642-14203-1_9.
- [15] Edwin Brady. “Idris, a general-purpose dependently typed programming language: Design and implementation”. In: *Journal of Functional Programming* 23 (05 Sept. 2013), pp. 552–593. DOI: 10.1017/S095679681300018X.
- [16] Iliano Cervesato and Frank Pfenning. “A Linear Logical Framework”. In: *Information and Computation* 179.1 (Nov. 2002), pp. 19–75. ISSN: 0890-5401. DOI: 10.1006/inco.2001.2951.

- [17] Chiyan Chen and Hongwei Xi. “Combining Programming with Theorem Proving”. In: *International Conference on Functional Programming*. ACM, Sept. 2005, pp. 66–77. DOI: 10.1145/1090189.1086375.
- [18] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nikolai Zeldovich. “Using Crash Hoare Logic for Certifying the FSCQ File System”. In: *Symposium on Operating Systems Principles*. Monterey, CA: ACM, Oct. 2015, pp. 18–37.
- [19] Zilin Chen, Liam O’Connor, Gabriele Keller, Gerwin Klein, and Gernot Heiser. “The Cogent Case for Property-Based Testing”. In: *Workshop on Programming Languages and Operating Systems*. PLOS’17. Shanghai, China: ACM, 2017, pp. 1–7. ISBN: 978-1-4503-5153-9. DOI: 10.1145/3144555.3144556.
- [20] Adam Chlipala. “An Optimizing Compiler for a Purely Functional Web application Language”. In: *International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: ACM, 2015, pp. 10–21. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784741.
- [21] Alonzo Church. “A Formulation of the Simple Theory of Types”. In: *Journal of Symbolic Logic* 5 (1940), pp. 56–68.
- [22] Koen Claessen and John Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: *International Conference on Functional Programming*. ICFP ’00. ACM, 2000, pp. 268–279. ISBN: 1-58113-202-6. DOI: 10.1145/351240.351266.
- [23] David Cock, Gerwin Klein, and Thomas Sewell. “Secure Microkernels, State Monads and Scalable Refinement”. In: *International Conference on Theorem Proving in Higher Order Logics*. Vol. 5170. Springer-Verlag, 2008, pp. 167–182.
- [24] *Cogent source code and Isabelle/HOL formalisation*. <https://github.com/NICTA/cogent>.
- [25] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986. ISBN: 0-13-451832-2.

- [26] Thierry Coquand and Christine Paulin. “Inductively defined types”. In: *International Conference on Computer Logic*. 1988, pp. 50–66. DOI: 10.1007/3-540-52335-9_47.
- [27] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. “Stream Fusion: From Lists to Streams to Nothing at All”. In: *International Conference on Functional Programming*. ICFP '07. Freiburg, Germany: ACM, 2007, pp. 315–326. ISBN: 978-1-59593-815-2. DOI: 10.1145/1291151.1291199.
- [28] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. “VCC: Contract-based modular verification of concurrent C”. In: *International Conference on Software Engineering*. May 2009, pp. 429–430. DOI: 10.1109/ICSE-COMPANION.2009.5071046.
- [29] Luis Damas and Robin Milner. “Principal Type-schemes for Functional Programs”. In: *Principles of Programming Languages*. Albuquerque, NM, USA, Jan. 1982.
- [30] Jared Davis and Magnus O. Myreen. “The Reflective Milawa Theorem Prover is Sound (Down to the Machine Code that Runs it)”. In: *Journal of Automated Reasoning* 55.2 (2015), pp. 117–183. DOI: 10.1007/s10817-015-9324-6.
- [31] Willem P. de Roever and Kai Engelhardt. *Data Refinement: Model-oriented Proof Theories and their Comparison*. Vol. 46. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998. ISBN: 0-521-64170-5.
- [32] Robert DeLine and Manuel Fähndrich. “Enforcing High-level Protocols in Low-level Software”. In: *Programming Language Design and Implementation*. PLDI '01. New York, NY, USA, 2001, pp. 59–69.
- [33] Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. “Running the Manual: An Approach to High-assurance Microkernel Development”. In: *Workshop on Haskell*. Portland, Oregon, USA: ACM, 2006, pp. 60–71. ISBN: 1-59593-489-8. DOI: 10.1145/1159842.1159850.
- [34] Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. “Sessions and Session Types: An Overview”. In: *Web Services and Formal Methods*. Ed. by Cosimo Laneve and Jianwen Su. Springer Berlin Heidelberg, 2010, pp. 1–28. ISBN: 978-3-642-14458-5.
- [35] Stephen Dolan and Alan Mycroft. “Polymorphism, Subtyping, and Type Inference in MLsub”. In: *Principles of Programming Languages*. Paris, France, Jan. 2017.

- [36] Umberto Eco. *The Name of the Rose*. tr. by William Weaver from *Il nome della rosa* (1980). Harcourt, 1983.
- [37] Kathleen Fisher and Robert Gruber. “PADS: A Domain-specific Language for Processing Ad Hoc Data”. In: *Programming Language Design and Implementation*. Chicago, IL, USA: ACM, 2005, pp. 295–304. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065046.
- [38] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. “The Next 700 Data Description Languages”. In: *Principles of Programming Languages*. Charleston, South Carolina, USA: ACM, 2006, pp. 2–15. ISBN: 1-59593-027-2. DOI: 10.1145/1111037.1111039.
- [39] Kathleen Fisher and David Walker. “The PADS Project: An Overview”. In: *International Conference on Database Theory*. Uppsala, Sweden: ACM, 2011, pp. 11–17. ISBN: 978-1-4503-0529-7. DOI: 10.1145/1938551.1938556.
- [40] Robert W. Floyd. “Assigning Meanings to Programs”. In: *Program Verification: Fundamental Issues in Computer Science*. Dordrecht: Springer Netherlands, 1993, pp. 65–81. ISBN: 978-94-011-1793-7. DOI: 10.1007/978-94-011-1793-7_4.
- [41] Anthony Fox. “Formal Specification and Verification of ARM6”. In: *Theorem Proving in Higher Order Logics*. Ed. by David Basin and Burkhart Wolff. Springer, 2003, pp. 25–40. ISBN: 978-3-540-45130-3.
- [42] Tim Freeman and Frank Pfenning. “Refinement Types for ML”. In: *Programming Language Design and Implementation*. Toronto, Ontario, Canada: ACM, 1991, pp. 268–277. ISBN: 0-89791-428-7. DOI: 10.1145/113445.113468.
- [43] Benedict Gaster and Mark P. Jones. “A Polymorphic Type System for Extensible Records and Variants”. In: *University of Nottingham Technical Report, NOTTCS-TR-96-3* (Jan. 1997).
- [44] Jean-Yves Girard. “Linear Logic”. In: *Journal of Theoretical Computer Science* 50 (1987), pp. 1–102. DOI: 10.1016/0304-3975(87)90045-4.
- [45] Jean-Yves Girard. “Une extension de l’interprétation de Gödel à l’analyse, et son application à l’élimination de coupures dans l’analyse et la théorie des types”. In: *Scandinavian Logic Symposium*. North-Holland, 1971, pp. 63–92.
- [46] Georges Gonthier. “Formal Proof — The Four-Color Theorem”. In: *Notices of the American Mathematical Society* 55.11 (2008), pp. 1382–1393.

- [47] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. “A Machine-Checked Proof of the Odd Order Theorem”. In: *International Conference on Interactive Theorem Proving*. Vol. 7998. Lecture Notes in Computer Science. 2013, pp. 163–179. DOI: 10.1007/978-3-642-39634-2_14.
- [48] David Greenaway, June Andronick, and Gerwin Klein. “Bridging the Gap: Automatic Verified Abstraction of C”. In: *International Conference on Interactive Theorem Proving*. Princeton, New Jersey: Springer, Aug. 2012, pp. 99–115. ISBN: 978-3-642-32346-1.
- [49] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. “Don’t Sweat the Small Stuff: Formal Verification of C Code Without the Pain”. In: *Programming Language Design and Implementation*. Edinburgh, UK: ACM, June 2014, pp. 429–439. DOI: 10.1145/2594291.2594296.
- [50] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. “CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels”. In: *Symposium on Operating Systems Design and Implementation*. ACM, Nov. 2016.
- [51] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. “A formal proof of the Kepler conjecture”. In: *Computing Research Repository* abs/1501.02155 (2015). URL: <http://arxiv.org/abs/1501.02155>.
- [52] Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. “A Revision of the Proof of the Kepler Conjecture”. In: *Discrete & Computational Geometry* 44.1 (2010), pp. 1–34. DOI: 10.1007/s00454-009-9148-4.
- [53] Robert Harper and Greg Morrisett. “Compiling Polymorphism Using Intensional Type Analysis”. In: *Principles of Programming Languages*. San Francisco, California, USA: ACM, 1995, pp. 130–141. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199475.

- [54] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. “IronFleet: proving practical distributed systems correct”. In: *Symposium on Operating Systems Principles*. Monterey, CA, USA: ACM, Oct. 2015, pp. 1–17. DOI: 10.1145/2815400.2815428.
- [55] Gernot Heiser. *Can Truly Dependable Systems Be Affordable?* Keynote at Asia-Pacific Workshop on Systems. Singapore, July 2013.
- [56] Gernot Heiser, June Andronick, Kevin Elphinstone, Gerwin Klein, Ihor Kuz, and Leonid Rzhzyk. “The Road to Trustworthy Systems”. In: *Workshop on Scalable Trusted Computing*. Chicago, IL, USA, Oct. 2010, pp. 3–9.
- [57] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.
- [58] C. A. R. Hoare. *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985. ISBN: 0-13-153271-5.
- [59] Martin Hofmann. “A Type System for Bounded Space and Functional In-Place Update”. In: *European Symposium on Programming*. Vol. 1782. Lecture Notes in Computer Science. 2000, pp. 165–179.
- [60] S. Holmström. “A linear functional language”. In: *Workshop on Implementation of Lazy Functional Languages*. Chalmers University of Technology, 1988.
- [61] Cătălin Hritcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. “Testing Noninterference, Quickly”. In: *International Conference on Functional Programming*. Boston, Massachusetts, USA: ACM, 2013, pp. 455–468. ISBN: 978-1-4503-2326-0. DOI: 10.1145/2500365.2500574.
- [62] John Hughes. “Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane”. In: *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. 2016, pp. 169–186. DOI: 10.1007/978-3-319-30936-1_9.
- [63] John Hughes. “Why Functional Programming Matters”. In: *Computer Journal* 32.2 (1989), pp. 98–107.
- [64] INRIA. *The Coq Proof Assistant Reference Manual*. 2009.

- [65] *IOzone Filesystem Benchmark*. <http://www.iozone.org/>.
- [66] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. “Cyclone: A Safe Dialect of C”. In: *USENIX Annual Technical Conference*. ATEC '02. USENIX, 2002, pp. 275–288. ISBN: 1-880446-00-6.
- [67] C. B. Jones. “Tentative Steps Toward a Development Method for Interfering Programs”. In: *ACM Transactions in Programming Languages and Systems* 5.4 (Oct. 1983), pp. 596–619. ISSN: 0164-0925. DOI: 10.1145/69575.69577.
- [68] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. “Rust-Belt: Securing the Foundations of the Rust Programming Language”. In: *Principles of Programming Languages*. ACM, Dec. 2017, 66:1–66:34. DOI: 10.1145/3158154.
- [69] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”. In: *Journal of Functional Programming* 28 (2018). DOI: 10.1017/S0956796818000151.
- [70] Sudeep Kanav, Peter Lammich, and Andrei Popescu. “A Conference Management System with Verified Document Confidentiality”. In: *International Conference on Computer Aided Verification*. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 167–183.
- [71] Jeffrey Katcher. *PostMark: A New File System Benchmark*. Tech. rep. TR-3022. NetApp, Oct. 1997.
- [72] Gabriele Keller, Toby Murray, Sidney Amani, Liam O'Connor, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. “File Systems Deserve Verification Too!” In: *Workshop on Programming Languages and Operating Systems*. Farmington, Pennsylvania, USA, Nov. 2013, pp. 1–7. DOI: 10.1145/2525528.2525530.
- [73] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. “Comprehensive Formal Verification of an OS Microkernel”. In: *Transactions on Computer Systems* 32.1 (Feb. 2014), 2:1–2:70. ISSN: 0734-2071. DOI: 10.1145/2560537.

- [74] Gerwin Klein, June Andronick, Gabriele Keller, Daniel Matichuk, Toby Murray, and Liam O'Connor. "Provably Trustworthy Systems". In: *Philosophical Transactions of the Royal Society A* 375 (2104 Sept. 2017), pp. 1–23. DOI: 10.1098/rsta.2015.0404.
- [75] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: Formal Verification of an OS Kernel". In: *Symposium on Operating Systems Principles*. Big Sky, Montana, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596.
- [76] Gerwin Klein, Thomas Sewell, and Simon Winwood. "Refinement in the Formal Verification of the seL4 Microkernel". In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Ed. by David S. Hardin. Springer US, 2010, pp. 323–339. ISBN: 978-1-4419-1538-2. DOI: 10.1007/978-1-4419-1539-9_11.
- [77] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. "Integrating Linear and Dependent Types". In: *Principles of Programming Languages*. POPL '15. Mumbai, India: ACM, 2015, pp. 17–30. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2676969.
- [78] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. "Self-Formalisation of Higher-Order Logic — Semantics, Soundness, and a Verified Implementation". In: *Journal of Automated Reasoning* 56.3 (2016), pp. 221–259. DOI: 10.1007/s10817-015-9357-x.
- [79] Ramana Kumar, Magnus Myreen, Michael Norrish, and Scott Owens. "CakeML: A Verified Implementation of ML". In: *Principles of Programming Languages*. Ed. by Peter Sewell. San Diego: ACM, Jan. 2014, pp. 179–191. DOI: 10.1145/2535838.2535841.
- [80] Yves Lafont. "The Linear Abstract Machine". In: *Journal of Theoretical Computer Science* 59.1-2 (July 1988), pp. 157–180. ISSN: 0304-3975. DOI: 10.1016/0304-3975(88)90100-4.
- [81] Xavier Leroy. "A Formally Verified Compiler Back-end". In: *Journal of Automated Reasoning* 43.4 (Dec. 2009), pp. 363–446. ISSN: 0168-7433. DOI: 10.1007/s10817-009-9155-4.

- [82] Ben Lippmeier. “Type inference and optimisation for an impure world”. PhD thesis. Australian National University, 2009.
- [83] Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernandez, and Artem Gleyzer. “PADS/ML: A Functional Data Description Language”. In: *Principles of Programming Languages*. Nice, France: ACM, 2007, pp. 77–83. ISBN: 1-59593-575-4. DOI: 10.1145/1190216.1190231.
- [84] Zohar Manna and Richard Waldinger. “Deductive synthesis of the unification algorithm”. In: *Science of Computer Programming 1.1* (1981), pp. 5–48. ISSN: 0167-6423. DOI: [https://doi.org/10.1016/0167-6423\(81\)90004-6](https://doi.org/10.1016/0167-6423(81)90004-6).
- [85] Simon Marlow. *Haskell 2010 Language Report*. Ed. by Simon Marlow. 2010.
- [86] Per Martin-Löf. *Intuitionistic type theory. Notes by Giovanni Sambin*. Vol. 1. Studies in Proof Theory. Bibliopolis, 1984, pp. iv+91. ISBN: 88-7088-105-9.
- [87] Conor McBride. “First-order Unification by Structural Recursion”. In: *Journal of Functional Programming* 13.6 (Nov. 2003), pp. 1061–1075. ISSN: 0956-7968. DOI: 10.1017/S0956796803004957.
- [88] Conor McBride. “I Got Plenty o’ Nuttin’”. In: *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. 2016, pp. 207–233. DOI: 10.1007/978-3-319-30936-1_12.
- [89] Peter J. McCann and Satish Chandra. “Packet Types: Abstract Specification of Network Protocol Messages”. In: *Applications, Technologies, Architectures, and Protocols for Computer Communication*. Stockholm, Sweden: ACM, 2000, pp. 321–333. ISBN: 1-58113-223-9. DOI: 10.1145/347059.347563.
- [90] John McCarthy and Patrick J. Hayes. “Some Philosophical Problems from the Standpoint of Artificial Intelligence”. In: *Machine Intelligence 4*. Edinburgh University Press, 1969, pp. 463–502.
- [91] Andrew McCreight, Tim Chevalier, and Andrew Tolmach. “A Certified Framework for Compiling and Executing Garbage-collected Languages”. In: *International Conference on Functional Programming*. ACM, 2010, pp. 273–284.
- [92] R. Milner. *A Calculus of Communicating Systems*. Berlin, Heidelberg: Springer-Verlag, 1982. ISBN: 0387102353.
- [93] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997.

- [94] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN: 3-540-43376-7.
- [95] Ulf Norell. “Dependently Typed Programming in Agda”. In: *Types in Language Design and Implementation*. Savannah, GA, USA: ACM, 2009, pp. 1–2. ISBN: 978-1-60558-420-1. DOI: 10.1145/1481861.1481862.
- [96] *POSIX Filesystem Test Project*. <http://sourceforge.net/p/ntfs-3g/pjd-fstest/ci/master/tree/>. Retrieved Jan., 2016.
- [97] Liam O’Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. “Refinement Through Restraint: Bringing Down the Cost of Verification”. In: *International Conference on Functional Programming*. Nara, Japan, Sept. 2016.
- [98] Liam O’Connor, Zilin Chen, Partha Susaria, Christine Rizkallah, Gerwin Klein, and Gabriele Keller. “Bringing Effortless Refinement of Data Layouts to Co-gent”. In: *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. Limassol, Cyprus: Springer, Nov. 2018, pp. 134–149. DOI: 10.1007/978-3-030-03418-4.
- [99] Martin Odersky. “Observers for Linear Types”. In: *European Symposium on Programming*. London, UK, UK: Springer-Verlag, 1992, pp. 390–407. ISBN: 3-540-55253-7.
- [100] Martin Odersky, Martin Sulzmann, and Martin Wehr. “Type inference with constrained types”. In: *Theory and Practice of Object Systems 5.1* (1999), pp. 35–55.
- [101] Peter W. O’Hearn. “Resources, Concurrency, and Local Reasoning”. In: *Theoretical Computer Science* 375.1-3 (Apr. 2007), pp. 271–307. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2006.12.035.
- [102] Susan Owicki and David Gries. “An axiomatic proof technique for parallel programs”. In: *Acta Informatica* 6.4 (Dec. 1976), pp. 319–340. ISSN: 1432-0525. DOI: 10.1007/BF00268134.
- [103] Sam Owre, John. Rushby, and Natarajan Shankar. “PVS: A prototype verification system”. In: *Conference on Automated Deduction*. Ed. by Deepak Kapur. Springer Berlin Heidelberg, 1992, pp. 748–752. ISBN: 978-3-540-47252-0.

- [104] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. “Faults in Linux: Ten Years Later”. In: *Architectural Support for Programming Languages and Operating Systems*. ACM, 2011, pp. 305–318. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950401.
- [105] David J. Pearce and Lindsay Groves. “Whiley: A Platform for Research in Software Verification”. In: *Software Language Engineering*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Cham: Springer International Publishing, 2013, pp. 238–248. ISBN: 978-3-319-02654-1.
- [106] Benjamin C. Pierce, ed. *Advanced Topics in Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2004.
- [107] Benjamin C. Pierce. “Bounded Quantification Is Undecidable”. In: *Information and Computation* 112.1 (1994), pp. 131–165.
- [108] Benjamin C. Pierce and David N. Turner. “Local Type Inference”. In: *Transactions on Programming Languages and Systems* 22.1 (Jan. 2000), pp. 1–44.
- [109] Amir Pnueli, Ofer Shtrichman, and Michael Siegel. “Translation Validation for Synchronous Languages”. In: *International Colloquium on Automata, Languages and Programming*. ICALP '98. Springer-Verlag, 1998, pp. 235–246. ISBN: 3-540-64781-3.
- [110] François Pottier and Didier Rémy. In: *Advanced Topics in Types and Programming Languages*. Ed. by Benjamin C. Pierce. Cambridge, MA, USA: MIT Press, 2004. Chap. The Essence of ML Type Inference.
- [111] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hrițcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. “Verified Low-level Programming Embedded in F*”. In: *Proceedings of the ACM on Programming Languages* 1.ICFP (Aug. 2017), 17:1–17:29. ISSN: 2475-1421. DOI: 10.1145/3110261.
- [112] John C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. In: *Symposium on Logic in Computer Science*. IEEE Computer Society, 2002, pp. 55–74. ISBN: 0-7695-1483-9.
- [113] John C. Reynolds. “Towards a Theory of Type Structure”. In: *Programming Symposium, Proceedings Colloque Sur La Programmation*. London, UK, UK: Springer-Verlag, 1974, pp. 408–423. ISBN: 3-540-06859-7.

- [114] Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. "A Framework for the Automatic Formal Verification of Refinement from Cogent to C". In: *International Conference on Interactive Theorem Proving*. Nancy, France, Aug. 2016.
- [115] J. A. Robinson. "Computational logic: The unification computation". In: *Machine intelligence* 6.63-72 (1971), pp. 10–1.
- [116] C. Ruggieri and T. P. Murtagh. "Lifetime Analysis of Dynamically Allocated Objects". In: *Principles of Programming Languages*. San Diego, California, USA: ACM, 1988, pp. 285–293. ISBN: 0-89791-252-7. DOI: 10.1145/73560.73585.
- [117] John Rushby, Sam Owre, and Natarajan Shankar. "Subtypes for specifications: predicate subtyping in PVS". In: *IEEE Transactions on Software Engineering* 24.9 (Sept. 1998), pp. 709–720. ISSN: 0098-5589. DOI: 10.1109/32.713327.
- [118] Rust. *The Rust Language*. <http://rust-lang.org>. Accessed: 2014-10-05. 2014.
- [119] K. Rustan and M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: 978-3-642-17511-4.
- [120] Amr Sabry and Matthias Felleisen. "Reasoning About Programs in Continuation Passing Style." In: *Conference on LISP and Functional Programming*. San Francisco, California, USA: ACM, 1992, pp. 288–298. ISBN: 0-89791-481-3. DOI: 10.1145/141471.141563.
- [121] Norbert Schirmer. "A verification environment for sequential imperative programs in Isabelle/HOL". In: *Logic for Programming, AI, and Reasoning*. Springer, 2005, pp. 398–414.
- [122] Thomas Sewell, Magnus Myreen, and Gerwin Klein. "Translation Validation for a Verified OS Kernel". In: *Programming Language Design and Implementation*. Seattle, Washington, USA: ACM, June 2013, pp. 471–481.
- [123] Matthieu Sozeau. "Subset Coercions in Coq". In: *Types for Proofs and Programs*. TYPES'06. Nottingham, UK: Springer-Verlag, 2007, pp. 237–252. ISBN: 3-540-74463-0, 978-3-540-74463-4.

- [124] Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. “Dependent Types and Multi-monadic Effects in F*”. In: *Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: ACM, 2016, pp. 256–270. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837655.
- [125] Swift Team. *Ownership*. <https://github.com/apple/swift/blob/d84048/docs/OwnershipManifesto.md>.
- [126] Mads Tofte and Jean-Pierre Talpin. “Implementation of the Typed Call-by-value λ -calculus Using a Stack of Regions”. In: *Principles of Programming Languages*. Portland, Oregon, USA: ACM, 1994, pp. 188–201. ISBN: 0-89791-636-0. DOI: 10.1145/174675.177855.
- [127] Jesse A. Tov and Riccardo Pucella. “Practical Affine Types”. In: *Principles of Programming Languages*. Austin, Texas, USA: ACM, 2011, pp. 447–458. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926436.
- [128] Harvey Tuch, Gerwin Klein, and Michael Norrish. “Types, Bytes, and Separation Logic”. In: *Principles of Programming Languages*. Nice, France: ACM, 2007, pp. 97–108. ISBN: 1-59593-575-4. DOI: 10.1145/1190216.1190234.
- [129] Matthijs Vákár. “A Categorical Semantics for Linear Logical Frameworks”. In: *Foundations of Software Science and Computation Structures*. Ed. by Andrew Pitts. Springer Berlin Heidelberg, 2015, pp. 102–116. ISBN: 978-3-662-46678-0.
- [130] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. “Refinement Types for Haskell”. In: *International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: ACM, 2014, pp. 269–282. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628161. URL: <http://doi.acm.org/10.1145/2628136.2628161>.
- [131] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. “OUTSIDEIN(X): Modular Type Inference with Local Assumptions”. In: *Journal of Functional Programming* 21.4-5 (Sept. 2011), pp. 333–412.
- [132] Philip Wadler. “Linear Types Can Change the World!” In: *Programming Concepts and Methods*. 1990.

-
- [133] David Walker. In: *Advanced Topics in Types and Programming Languages*. Ed. by Benjamin C. Pierce. Cambridge, MA, USA: MIT Press, 2004. Chap. Substructural Type Systems.
- [134] Yan Wang and Verónica Gaspes. “An Embedded Language for Programming Protocol Stacks in Embedded Systems”. In: *Partial Evaluation and Program Manipulation*. Austin, Texas, USA: ACM, 2011, pp. 63–72. ISBN: 978-1-4503-0485-6. DOI: 10.1145/1929501.1929511.
- [135] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. “Formalizing the LLVM Intermediate Representation for Verified Program Transformations”. In: *Symposium on Principles of Programming Languages*. POPL ’12. Philadelphia, PA, USA: ACM, 2012, pp. 427–440. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103709.