UNIVERSITY OF CALIFORNIA, SAN DIEGO

Liquid Types

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

in

Computer Science

by

Patrick Rondon

Committee in charge:

Professor Ranjit Jhala, Chair
Professor Samuel R. Buss
Professor Sorin Lerner
Professor Jens Palsberg
Professor Geoffrey Voelker

2012

The dissertation of Patrick Rondon is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____
Chair

University of California, San Diego

2012

DEDICATION

For Mom, Dad, and Claudia.

EPIGRAPH

Given the pace of technology, I propose we leave math to the machines and go play outside.
— Calvin

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I owe a huge thanks to my advisor, Ranjit Jhala, for all the generous and patient guidance and support as well as the insight, inspiration, good humor, and, of course, food and coffee he's provided over the years.

Thanks to my committee members Sorin Lerner, Sam Buss, Geoff Voelker and Jens Palsberg for showing a keen interest in the work and firming up my efforts with their questions and insights.

I feel very fortunate to have spent the last few years with the incredibly talented and driven UCSD programming languages group. Thanks to all of you for hearing out my half-baked ideas, reading my half-written drafts, and sitting through my half-cocked talks; the other halves were always so much better for your input. Particular thanks are due to Sorin Lerner, who was always ready to dole out advice or lend an ear as needed.

I've been especially lucky to have Ming Kawaguchi, Ravi Chugh, and Alexander Bakst as collaborators and friends. Trying to keep up with them has always pushed me to go further and faster. Among non-collaborators, I owe particular thanks to Ross Tate and Zach Tatlock, who have been great friends and good or bad influences as appropriate (or inappropriate).

I'm lucky to have made a large number of friends at UCSD who have changed my life for the better in countless ways. I won't attempt an exhaustive list, for fear of missing someone or running out of pages; you know who you are. Thanks for everything!

I'm grateful for the lifelong support and encouragement of my "generalized parents": thanks, Mom, Dad, Uncle Ronnie, Tom, and Norah, for keeping me going. Thanks to Joseph, Vanessa, Aprille, Frank, Ryan, and Evan; if I turned out OK, it's largely because I grew up in such good company.

Finally, much of the credit for the actual completion of this work belongs to my wife and constant coffee shop companion, Claudia, who made the bad days bearable and the good days outstanding.

Published Works Adapted in This Dissertation

Chapter 2 contains material adapted from the following publications:

Patrick Rondon, Ming Kawaguchi, Ranjit Jhala. "Liquid Types", *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 159–169, 2008.

Ming Kawaguchi, Patrick Rondon, Ranjit Jhala. "DSolve: Safety Verification via Liquid Types", *Proceedings of Computer Aided Verification 2010 (CAV)*, pages 123–126, 2010.

The dissertation author was principal investigator on both publications.

Chapter 3 contains material adapted from the following publications:

Patrick Rondon, Ming Kawaguchi, Ranjit Jhala. "Low-Level Liquid Types", *Proceedings of the 2010 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 131–144, 2010.

Patrick Rondon, Alexander Bakst, Ming Kawaguchi, Ranjit Jhala. "CSolve: Verifying C with Liquid Types", *Proceedings of Computer Aided Verification 2012 (CAV)*, pages 744–750, 2012.

The dissertation author was principal investigator on both publications.

VITA

2006	B. S. in Computer Science, Pennsylvania State University
2009	M. S. in Computer Science, University of California, San Diego
2012	Ph. D. in Computer Science, University of California, San Diego

PUBLICATIONS

Patrick Rondon, Ming Kawaguchi, Ranjit Jhala, "Liquid Types", *Programming Language Design and Implementation*, 2008.

Ming Kawaguchi, Patrick Rondon, Ranjit Jhala, "Type-Based Data Structure Verification", *Programming Language Design and Implementation*, 2009.

Patrick Rondon, Ming Kawaguchi, Ranjit Jhala, "Low-Level Liquid Types", *Principles of Programming Languages*, 2010.

Ming Kawaguchi, Patrick Rondon, Ranjit Jhala, "DSolve: Verification via Liquid Types", *Computer-Aided Verification*, 2010.

Ravi Chugh, Patrick Rondon, Ranjit Jhala, "Nested Refinements: A Logic for Duck Typing", *Principles of Programming Languages*, 2012.

Ming Kawaguchi, Patrick Rondon, Alexander Bakst, Ranjit Jhala, "Deterministic Parallelism with Liquid Effects", *Programming Language Design and Implementation*, 2012.

Patrick Rondon, Alexander Bakst, Ming Kawaguchi, Ranjit Jhala, "CSolve: Low-Level Program Verification via Liquid Types", *Computer-Aided Verification*, 2012.

ABSTRACT OF THE DISSERTATION

Liquid Types

by

Patrick Rondon

Doctor of Philosophy in Computer Science

University of California, San Diego, 2012

Professor Ranjit Jhala, Chair

Because of our increasing dependence on software in every aspect of our lives, it is crucial that our software systems are reliable, safe, and correct — they must not crash, must be safe from attack, and must consistently compute the results we expect from them. As testing is insufficient to show the absence of errors and manual code review is tedious, costly, and error-prone, the only clear path to efficiently and reliably ensuring software quality is to develop automatic verification tools which require as little intervention from the programmer as possible.

In this dissertation, we present Liquid Types, an automated approach to software verification based on inferring and checking expressive refinement types, data types which are augmented with logical predicates, which can be used to express and verify sophisticated program invariants. We show how Liquid Types divides the task of program verification between type-based and logic-based reasoning to infer precise invariants of unboundedly-large data structures. Further, we show how the Liquid Types technique is suited both to high-level, pure functional languages and low-level, imperative languages with mutable state, allowing for the verification of programmings running the full range from applications to systems programs.

The Liquid Types technique has been implemented in type checkers for both the OCaml and C languages and applied to a number of challenging programs taken from the literature and from the wild. We highlight experimental results that show that the refinement type inference performed by Liquid Types can be used to verify crucial safety properties of real-world programs without imposing an undue verification-related overhead on the programmer.

Chapter 1

Introduction

In spite of advances in language design, development environments, run-time support, software engineering practices, and verification technology, the task of writing correct, reliable software remains tremendously difficult: it is still distressingly common for newly-developed programs to be susceptible to crashes, misbehaviors, and security vulnerabilities.

What makes writing reliable programs difficult, in spite of our decades of experience in the craft, is that it is both exacting and abstract. Programming is exacting in the sense that every detail of a program matters in determining how it will execute. Even the tiniest error — for example, reading one too many characters from an untrusted source into a buffer — can have catastrophic consequences — for example, complete takeover of a user's computer. Reasoning about all possible sources of errors requires programmers to be infallible experts on the semantics of their languages, libraries, and run-time environments.

Yet many factors influencing the behavior of a program are unknown until run-time. For example, we do not know until run time how threads will be scheduled or what inputs the user will provide. Programs that manipulate unboundedly-large data structures like linked lists also have infinite state spaces, making it impossible to reason about every concrete program state. (Of course, even a finite state space is typically far too large to admit case-by-case analysis, even using automated techniques.) Thus, programming is not just exacting, but also abstract, in that the programmer must use not concrete program states, but rather sets of possible program states, in reasoning about their programs.

In order to write correct and reliable programs, then, programmers are expected to have a flawless understanding of the semantics of their programming environments and complete omniscience with respect to the kinds of inputs their programs will receive and environments in which they will run. In order to safely maintain, extend, and update their programs, programmers must be able to carry out this reasoning not only flawlessly, but do so repeatedly and quickly. It is highly unrealistic to expect that fallible human programmers can reliably carry out such

detailed reasoning once, much less repeatedly and at a pace consistent with our demand for new and updated software, as people are incredibly ill-suited to such detailed reasoning.

The inescapable conclusion is that, if we wish to establish that our programs are correct and reliable, and we wish to ensure that they remain so as we upgrade and extend them, we must do so by automated means. Any tool for ensuring program quality must satisfy several criteria. First, it must be low-overhead: the cost of using the tool, in terms of programmer time invested in interacting with and understanding the tool, must be low, so that there is a net benefit to its use as compared with by-hand reasoning. Second, it must be precise: the tool should not give too many false positives, signaling to the programmer that their code is incorrect when in fact there is no error, again so that the overhead of sorting the wheat from the chaff does not negate the benefits of using the tool. Third, the tool should be expressive: the tool should be able to express and check a wide variety of program properties. In particular, to verify realistic programs, the tool must be able to reason precisely about the contents of unboundedly-large data structures like lists, arrays, and hash tables, which are ubiquitous in real-world programs.

1.1 Toward Automated Program Verification

Our criteria point us toward precise *automated* program verification tools which analyze programs and determine, with a limited amount of user intervention, whether they satisfy the desired criteria. Broadly, such tools work by automatically exploring the space of reachable program states and ensuring that any program state where a desired invariant is broken — for example, where an array index is out of bounds — is unreachable, thus verifying the absence of the error. Specific approaches to exploring the state space vary, but generally fall into the categories of *model checking* [16] and *abstract interpretation* [24]. (We consider type checking as a special case of abstract interpretation, and defer further discussion of type checking until later.) In model checking, the program's state space is explored systematically, beginning with the initial state and following all possible execution paths until all reachable states have been discovered or an error state is reached. This naïve approach to model checking is only applicable to programs that have finite state spaces; if the program manipulates unboundedly-large data structures like linked lists and trees, the set of possible program states will be infinite, and the state space exploration may not terminate. Thus, to effectively model check infinite-state programs, one generates an *abstraction* of the state space that breaks the infinite state space into subsets represented by a finite number of abstract representatives. One can then model check a corresponding finite-state *abstract* program which simulates the original program but operates on abstract, rather than concrete, states. The abstract interpretation approach is similar, but differs in the details: we take as our abstract state space a complete lattice whose elements are the abstract states. The concrete program statements are mapped to corresponding, monotone abstract transformers over the abstract state space. The program is executed with respect

to this abstract domain by evaluating the composition of the program's constituent abstract transformers to a fixed point; this is analogous to executing the abstract program in the model checking approach until all reachable states are discovered. Given a finite-height lattice, then, it's possible to use abstract interpretation to compute program invariants in a finite amount of time. In both approaches, after interpreting the program over the abstract state space, we check that no undesirable abstract states can be reached.

We note that the model checking and abstract interpretation approaches are more alike than they are different, and their central concerns are the same. First, we wish to construct an abstract domain which is *precise* in the following two senses: first, it is expressive enough to prove interesting program invariants, and second, it avoids using the same abstract state to represent both undesirable concrete program states and legitimate, valid program states which cause no trouble at runtime; doing so could result in a high number of "false alarms" where safe programs are erroneously reported as unsafe. Second, the operations on the abstract domain that we require to perform the analysis must be efficiently implementable, so that the analysis has reasonable performance. In particular, we must be able to efficiently decide inclusion between two abstract program states, in order to determine whether a given abstract program state includes an undesirable state, and, in the case of abstract interpretation, we must be able to efficiently compute a an abstract overapproximation of two abstract program states.

We thus note that an abstract domain is an enhanced *logic* of abstract program states: concrete and abstract program states are related by a modeling relation which determines which concrete states belong to a given abstract state; an entailment relation tells us when a model of one abstract state is also a model of another; and (most of) the operations on abstract states correspond to a proof theory in the logic of program states. Intuitively, it would seem that a natural choice for a logic of program states is first-order logic, augmented with theories suitable for program verification; indeed, the earliest work in manual program verification, based on Floyd-Hoare logic [39, 48, 29], represented program states using formulas of first-order logic. Representing program states with first-order logic comes with a considerable advantage: the availability of fast Satisfiability Modulo Theories (SMT) solvers for first-order logical formulas, which incorporate fast reasoning about such quantifier-free theories as arrays, integer linear arithmetic, and uninterpreted functions, makes it possible to automatically and efficiently reason about sophisticated quantifier-free first-order order facts about program states. Tools built on SMT technology automatically benefit from advances in the area, which incorporates SAT solving, theory-specific reasoning, and combination procedures for reasoning about facts which combine elements from several theories.

Unfortunately, unrestricted first-order logic suffers two problems that make it a poor choice for representing abstract states in an automated program verifier. First, unrestricted first-order logic is simultaneously not expressive enough to verify realistic programs with unbounded data structures and too expressive to be efficiently decidable. Second, first-order logic by itself

does not provide any sort of abstraction; the ability to precisely express arbitrary program states means that the set of abstract program states expressible in first-order logic simply *subsumes* the set of concrete program states. We review approaches to expressiveness and state abstraction in turn, then describe approaches to balancing abstraction and expressiveness.

A number of approaches have been developed to create logics which have better decidability or expressiveness properties than first-order logic. Logics which include transitive closure or reachability predicates have been developed for coping with linked data structures; a principal concern in the design of such logics is imposing sufficient restrictions on the allowed formulas so that validity remains decidable, ensuring that programmers do not need to provide explicit proofs, or providing semi-decision procedures which are sufficient for practical use. Examples of logics of this type include the Pointer Assertion Logic Engine [66] and the reachability logics of Chatterjee et al. [13] and Lahiri and Qadeer [59]. A related system is McPeak and Necula's logic based on local equality axioms [63], which provides a decision procedure for shape (rather than data) properties that constrain a fixed-size region around heap nodes. On the other hand, a number of higher-order logics have been developed for reasoning about programs, and corresponding verifiers have been developed. Example of systems of this type include NuPRL [21] and Coq [8], which support the development and verification of higher-order, pure functional programs; a number of systems have extended or incorporated these logics to accommodate writing and verifying imperative programs, among them Ynot, based on Hoare Type Theory [67], Bedrock [14], and Jahob [87]. Logical validity checking in such systems is undecidable, so that such systems approach automatic validity checking on a "best-effort" basis: both built-in and user-provided tactics are used to attempt to discharge proof obligations, with the user ultimately responsible for manually proving any obligations which the tactics are unable to discharge. As with first-order logic, we note that the logics discussed above do not inherently provide any sort of finite program state abstraction; these logics may satisfy our precision and expressiveness requirements, but do not themselves help with automating the verification process.

On the other hand, a number of abstract domains have been developed for automatic program analysis, based both on specialized state representations and on full first-order logic. These include a number of abstract domains specialized to properties of integer values, among them intervals [22], octagons [65], and polyhedra [25]. A number of abstract domains have also been developed for inferring shape, rather than data, properties of linked data structures. Much work in this area has focused on three-valued logic analysis [62] or separation logic [74, 49] (for example, [30, 86]); in the case of separation logic, the abstract domains are typically tailored to the particular data structures being analyzed, e.g., singly- or doubly-linked lists. In the same vein, a number of abstract domains have been developed for analyzing data-sensitive properties of specific data structures, among them arrays [42, 23] and singly-linked lists [10]. Such abstract domains tend to be efficient and expressive within their application domains, but these gains come at the cost of generality, trading expressiveness for efficiency.

The predicate abstraction domain [43] retains the full expressiveness of first-order logic and most of the automation of more specialized abstract domains. Elements of the predicate abstraction domain are Boolean combinations of user-provided first-order predicates. Thus, the domain is finite, but, in the limit, the expressiveness of the domain is the same as first-order logic, as the user may provide arbitrary predicates. Automation is still quite high because the user needs only to provide a set of relatively small predicates from which sophisticated program invariants can be constructed; further, sets of such predicates sufficient for performing verification can be often be guessed based on the syntax of the program and types of its identifiers, as in the Houdini annotation assistant [36]. Predicate abstraction forms the foundation of a number of successful automated program verification tools, among them SLAM [5], BLAST [47], MAGIC [12], and ESC/Java [37].

However, in spite of its advantages, predicate abstraction over first-order logic formulas suffers the same expressiveness and decidability limitations as first-order logic, and introduces new limitations of its own: in addition to the problems of deciding the validity of first-order formulas and the need for additional logical primitives to express invariants over linked structures, the user needs to explicitly provide any quantified facts in full as input predicates — predicate abstraction alone will not insert quantifiers where appropriate: for example, given a predicate stating that a value is nonzero, predicate abstraction-based techniques cannot infer that all elements of an array are nonzero without the user explicitly providing the entire quantified fact. Indexed predicate abstraction [37, 58] solves the latter problem by allowing the user to write predicates over both program variables and *index variables*, which will be quantified over when constructing predicates describing program states. A more sophisticated approach proposed by Srivastava and Gulwani [77] takes from the user both a set of predicates and a set of *templates* which contain variables ranging over conjunctions of the predicates; the user specifies the Boolean structure of the inferred invariants, including any desired quantifiers, as part of the template. Unfortunately, such approaches still leave the user with the burden of either deciding which variables within a given predicate should be quantified or deciding on the quantified structure of the invariants that should be inferred, and do not address the problems of automatically deciding the validity of universally-quantified facts.

1.2 Quantified Reasoning with Refinement Types

We now turn to a more restrictive abstract domain which supports efficient reasoning with quantified facts about unbounded data structures: types. In a type system, simple (non-compound) program values are classified according to their *types*, for example, `int` for integer values and `bool` for Boolean values. Complex data values, like lists of values of a particular type of value, are described with *type constructors*, which are simply functions from types to types: for example, the type constructor `list` can be applied to the type `int` to yield the type `int list`,

the type of a list whose elements are all integers; note that this type compactly represents a universally-quantified fact about the elements of a list, that all of its elements are integers. The syntaxes of both programs and types guide reasoning about such universally-quantified facts: an `int list` is only constructed by creating a new empty list or by applying the `cons` data constructor to a pair of an `int` value and an `int list` value; similarly, deconstructing an `int list` value into its components must always yield either an empty list or a pair of an `int` and an `int list`; and two program values have the same type only when all the components of their types are the same. Thus, types efficiently guide us in reasoning about quantified facts: in particular, they provide efficient syntax-directed methods for generalizing facts about individual data items to facts about entire data structures, for instantiating facts about data structures into facts about individual elements, and for deciding inclusion between abstract program states.

However, simple types like `int list` are not sufficiently expressive for the majority of program verification tasks. For example, such type systems are unable to express important program invariants like the fact that an integer value is nonzero or within the bounds of some array. To address this shortcoming, prior work has developed a number of *refinement type* systems [41, 27, 71], which enhance conventional types with formulas that allow for precise reasoning about data values. A refinement type is formed by combining a conventional type, like `int` or `bool`, with a logical predicate that further restricts the values that belong to the type. For example, the refinement type

$$\{\nu : \texttt{int} \mid \nu \neq 0\}$$

combines the base type `int` with the predicate $\nu \neq 0$. The special *value variable*, ν, is used to indicate the value which is described by this type; the refinement predicate $\nu \neq 0$ specifies that all values that have this type must be nonzero, so that the refinement type above specifies the set of nonzero integer values. Quantified reasoning with refinement types proceeds similarly to ordinary data types. The strategies for generalizing and instantiating universally-quantified data structure facts remain the same. To show that one quantified fact, expressed as a refinement type, implies another, we simply check pairwise implication between the components of the type. For example, we determine that the property of being a list of positive integers implies the property of being a list of list of nonzero integers, we simply check that the type of lists of positive integers,

$$\{\nu : \texttt{int} \mid \nu > 0\}\ \texttt{list},$$

is included in the type of lists of nonzero integers,

$$\{\nu : \texttt{int} \mid \nu \neq 0\}\ \texttt{list},$$

by verifying that $\nu > 0$ implies $\nu \neq 0$. Thus, refinement type checking reduces checking implications between universally-quantified facts about data structures, expressed as refinement types, to checking implications between quantifier-free formulas; such checks are easily discharged by off-the-shelf SMT solvers.

Refinement types have been shown to be broadly applicable and highly expressive program verification tools. Xi and Pfenning [83, 84] show that refinement types can be used to show the absence of array bounds violations in a number of higher-order ML programs. Dunfield [32] shows that refinement types can be used to verify the correctness of data structure implementations; he shows, for example, that an implementation of red-black tree operations maintains the required color invariant. Bengtson et al. [7] use refinement types to show the correctness of cryptographic protocol implementations. In each of the above, the verification was done by manually annotating each function in the program with refinement types, with annotation burdens of upwards of 10% of the total lines of source code. To make verification with refinement types practical, we will have to lower this burden considerably.

1.3 Liquid Types: A Method for Refinement Type Inference

Our key insight is that we can combine the quantified reasoning machinery of refinement type checking with the invariant inference machinery of predicate abstraction to yield an algorithm for refinement type inference, which will allow us to significantly lower the annotation burden associated with refinement type-based verification. We define a class of refinement types, called *liquid types*, whose refinement predicates are restricted to be conjunctions of instances of user-provided predicate templates. We then perform refinement type inference in three phases. First, we infer conventional data types like `int` and `bool` for each program expression. We then assign each inferred type a refinement predicate variable representing an unknown refinement predicate, and use the structure of the program and its inferred types to generate a set of logical constraints on the refinement predicate variables. We apply a fixed point procedure to solve for the refinement predicate variables as conjunctions of instances of the user-provided predicate templates such that, if a solution is found, replacing each refinement predicate variable with its solution yields a refinement typing for the program.

The resulting abstract domain neatly divides the invariant inference task between type-based and logic-based reasoning. Facts about values of base type are expressed by simple, quantifier-free formulas. These facts about individual data items are lifted to quantified facts over entire data structures by the type constructors used to form the types of unbounded collections, shifting the burden of quantified reasoning to the type system. The type system, in turn, uses straightforward, syntax-guided rules to reduce quantified reasoning to a set of quantifier-free implication checks which can be easily discharged by existing SMT solvers. The combination of type inference, predicate abstraction, and fast SMT solving leads to an automated approach to program verification which is precise, automatic, and scalable.

1.4 Other Approaches to Refinement Type Inference

Liquid types is not the first or only approach to refinement type inference for higher-order, functional programs. Knowles and Flanagan [53] present a type reconstruction algorithm for *generalized* refinement types, in which the refinement predicates are allowed to be arbitrary terms of the language being type checked. The authors use the power of generalized refinement type systems as leverage in solving a generalized type reconstruction problem: they present an algorithm which assigns types to program expressions in a way that preserves typability, in a manner roughly analogous to computing strongest postconditions and which takes advantage of the presence of fixed point combinators in the refinement type language to express loop invariants in the refinement type system.

The algorithm of Knowles and Flanagan only annotates expressions with types such that the original program is typable if and only if annotated program is typable. However, their algorithm does not — and cannot — decide if a program is typable, which is the essential step in using a type system to perform static program verification. Instead, type checking is deferred to a hybrid type checker [35, 44], which copes with the undecidability of type checking by deferring checks which are not statically decidable to runtime.

An alternative approach to verifying higher-order functional programs is to reduce such programs to higher-order recursion schemes and perform model checking on the result, as in [56]. Originally, such approaches were limited to verifying Boolean programs; recent work by Kobayashi et al. [57] has extended the reach of this approach to infinite-state programs by using predicate abstraction to generate an abstract Boolean program from a given infinite-state program. In the approach of Kobayashi et al., Counterexample-Guided Abstraction Refinement (CEGAR) is used to attempt to automatically discover a set of predicates sufficient to verify the program being analyzed. Terauchi [79] presents a similar approach to refinement type inference based on CEGAR. As is generally true for CEGAR-based approaches, the type inference methods outlined by Kobayashi et al. and Terauchi are incomplete: the CEGAR process may loop indefinitely, endlessly generating counterexamples but never finding an invariant strong enough to prove safety. In contrast, we provide an algorithm for deciding whether a program is typable using more restrictive liquid types over a particular set of predicate templates. Thus, we trade off the (typically quite small) cost of manually specifying a set of predicates over which to perform inference for the benefit of a sound and complete (relative to the provided predicates) inference system.

1.5 Low-Level Liquid Types

The first part of this dissertation presents liquid types as a suitable abstract domain for verifying the safety and correctness of programs written in a high-level, pure functional language.

The second part of the dissertation shows that the benefits of liquid types for performing quantified reasoning about unboundedly-large data structures can be extended to the setting of low-level languages which incorporate mutable state, unrestricted aliasing, and unrestricted casting and pointer arithmetic. To do so, we combine the basic liquid types technique with a number of other techniques, each aimed at solving a particular part of the problem of verifying low-level programs. We show that the resulting system, which we call *low-level liquid types*, is capable of inferring precise invariants and showing the memory safety of a variety of programs taken both from the wild and from the literature.

A principal concern in verifying programs with mutable state is coping with temporary invariant violation: when the fields of a data structure are updated separately, invariants that relate the values of the fields may be broken at intermediate points where not all fields have been updated yet. A seemingly straightforward solution is to simply use strong updates: as each field is updated, the type system updates the type of the field to precisely track the new value it was assigned. When the invariant is reestablished, it will be directly reflected in the type of the data structure, as the types of its fields precisely reflect the values they were assigned.

However, accommodating strong updates in a type system is complicated by two factors: unrestricted aliasing and unboundedly-large data structures. Unrestricted aliasing means that updating the type of a field is a non-trivial task: we must update the type of the field not only for the pointer that is being accessed directly, but also for any of its potential aliases, which may not even be in scope at the point where the update is performed, and thus not in the type environment, making their types inaccessible for the type system to update. The presence of unbounded collections makes strong updates difficult, as a single type must be used to describe multiple elements.

To allow strong updates in spite of aliasing, we adopt ideas from the alias types system of Walker and Morrisett [80], which adds a layer of indirection to the type system to allow strong updates to simultaneously update the type of all of a pointer's aliases. In the alias types discipline, the type of a pointer does not explicitly mention the type of its referent, but instead names a location in an abstract heap where the type of the referent is stored. Strong updates are then performed on the types of abstract heap locations, rather than on the types of the pointers themselves. Thus, by adding this layer of indirection, it becomes safe to perform strong updates in spite of unrestricted aliasing, since all aliases of a pointer will share the same location name and thus indirectly reference the same structure type.

As a side effect of adopting the alias types discipline, our type system automatically separates the heap into disjoint regions in the style of separation logic. Thus, our type system inherits some of the local reasoning capabilities of separation logic: updates to pointers affect only a restricted, statically-determined part of the heap's type, and our handling of function calls follows a frame rule-like discipline to allow us to preserve the types of heap locations present in the caller which are not accessed in the callee.

Adapting alias types to our setting solves our problems with reconciling aliasing with strong update, but still does not make strong update safe in the presence of unbounded collections. On the one hand, we wish to use strong updates to infer precise invariants in spite of temporary invariant violations. On the other hand, the presence of unbounded collections means that we must represent unbounded numbers of run-time objects using a bounded number of static types — in essence, by representing all elements of a collection with a single type. It would be unsound to strongly update the type of all elements of a collection when only a single element has been modified, but this does not make the need for strong updates any less necessary. To allow safe strong updates in the presence of unbounded collections, we adopt a *local non-aliasing* discipline, which we intuitively describe via a version control analogy. We begin in a state where all elements of the collection satisfy the same invariant, expressed as a type. At any point where we need to modify an element of the collection, we conceptually *check out* that element from the collection, giving it a type which is specific to that single element and which may thus be strongly updated safely. After the element has been updated and its invariant reestablished, as witnessed by the element's type, it may be *checked in* to the collection, restoring the property that all elements of the collection satisfy the same invariant. To preserve this property, we do not allow two elements from the same collection to be checked out simultaneously. Our local non-aliasing technique borrows from work on restrict [40, 3], adopt and focus [34], and thawing and freezing [2].

A further complication in type checking low-level programs written in C-like languages is the lack of an existing static type discipline: in C, types exist only to guide the compiler in mapping C's operations to machine operations, and arbitrary casts are permitted, so that the C type information need not accurately describe the actual data values manipulated by the program and is thus useless in building a refinement type system. In order to provide a solid foundation for building a refinement type system suitable for low-level programs, we begin by developing a *physical* type system which can accurately reflect the contents of memory. Our physical type system expresses the types of pointers as pairs of an abstract location and an offset into that location, expressed as the product of an interval, giving upper and lower bounds to the offset, and a congruence class, giving the "period" of the offset (e.g., the integers mod 4 for a pointer which may point to elements within an array of 4-byte integers); this representation allows us to precisely track the targets of pointers in spite of pointer arithmetic. Similarly, our heap locations are expressed as blocks composed of bindings to fixed and periodic offsets. Our structures for physical types and processes for performing physical type inference are thus similar to those adapted by [81, 64].

Finally, not all invariants we wish to express can be captured as refinement types which relate fields of the same structure; properties of linked data structures, like sortedness, require that we relate the values of fields within two linked structures. However, in the presence of uncontrolled mutation, refinement types that contain references between two structures are

unsound. To allow our system to express such relationships, we allow fields of a data structure to become *final*, i.e., immutable. We allow refinement predicates to contain pointer dereferences as long as they only refer to final fields, allowing a sound form of dereference in refinement types. One can then express invariants like sortedness by giving a refinement type that says, in effect, that the value of the data field in any linked list node is less than the value of the data field in the node pointed to by its `next` field. Our notion of final fields is inspired by Leino et al.'s frozen fields [61] and the handling of object properties in Nystrom et al.'s system of constrained types [70]; a key difference is that we infer which fields are final in our system and automatically infer properties over final fields, while the other systems mentioned require users to both manually annotate which fields are final and manually specify the invariants they expect to hold.

The combination of liquid types with the above techniques results in an effective automated technique for inferring precise invariants of low-level programs that manipulate unbounded data structures.

1.6 Related Approaches to Verifying Low-Level Programs

We have already placed liquid types in the general context of existing program analysis techniques. However, there are a few techniques which are especially closely related to low-level liquid types; we draw explicit comparisons to them below.

Gulwani et al. [45] give a method for constructing abstract domains of quantified facts from existing abstract domains which capture unquantified facts. In order to guide their analysis in discovering quantified facts, the user is required to provide indexed predicate-style templates: the analysis infers quantified facts that are instances of the user-provided templates, and the variables the user wishes to be universally quantified must be explicitly annotated. Thus, the user is ultimately responsible for guiding the analysis in inferring quantified facts. By contrast, our system allows the user to provide predicate templates which the system may apply equally well to either local variables or heap-allocated data; quantification is performed automatically, without the user's guidance.

The Boolean heaps abstract domain of Podelski and Wies [72] also addresses the problem of applying predicate abstraction to infer quantified invariants of heap-allocated data. In their approach, heap objects are abstracted as their evaluation under a set of a user-provided predicates; a Boolean heap is a set of such evaluations, and their abstract domain is taken to be a set of such heaps. Thus, the size of the abstract state may be doubly-exponential in the number of predicates provided. Further, because there is no notion of heap separation built in to their system, computing the abstract post state of a command in their system potentially involves analyzing the state of all abstract objects on the heap. Finally, facts about linked data structures must be expressed in their system using transitive closure or similar mechanisms. By contrast,

our low-level type system first divides the heap objects according to their may-alias sets; the states of the objects in each set are then abstracted by liquid types, which are essentially the objects' evaluations under the user-provided predicates. Abstract post states are computed by isolating a single element, performing strong updates, and performing a subtyping test. Finally, we rely on the type structure to express quantification over linked data structures rather than using transitive closure or reachability. Our system thus sacrifices some degree of expressiveness for increased scalability: our goal is whole-program refinement type inference, while the Boolean heaps approach has largely been applied to local shape inference, e.g., inferring loop invariants in functions which have already been annotated with pre- and post-conditions. A further difference between our approaches is that our low-level liquid types also ensure memory safety with respect to a basic, unrefined type system, preventing certain errors like partial reads and writes of data structure fields.

Both CCured [20] and Deputy [19] implement enhanced type systems for existing C programs with the aim of ensuring memory safety. The CCured system annotates pointers with kinds indicating whether they are used to reference a single item ("Safe" pointers), a sequence of items as in an array ("Seq" pointers), or are subject to arbitrary pointer arithmetic ("Wild" pointers). CCured then inserts run-time bounds checks for Seq and Wild pointers to ensure the safety of memory accesses in the presence of arbitrary pointer arithmetic. For unannotated programs, a whole-program pointer kind inference algorithm annotates each program with its kind, attempting to find as many Safe pointers as possible. The Deputy system implements a refinement type checker for the C programming language. In Deputy, a flow- and path-insensitive type system is used to insert dynamic safety checks into the program. The system then performs a static analysis to attempt to optimize away as many checks as possible, reducing the run-time penalty imposed for type safety. Both CCured and Deputy are *hybrid* type systems which insert dynamic type checks when a type obligation cannot be proven statically. In contrast to these hybrid type systems, we aim for full static verification, and do not insert dynamic checks. However, our system could easily be extended to handle the insertion of dynamic contract checks where type inference is unable to prove that a term has a particular required type. Similarly, our system could be used to discharge the assertions placed by a system like Deputy.

An alternate approach of Condit et al., implemented in the Havoc system [17], similarly combines logic- and type-based approaches to program verification, but comes at the problem from a complementary direction: rather than embedding logic into types, as in a refinement type system, their system begins with a Hoare-style verifier, then embeds type assertions into the logic. Type safety is proved by explicitly asserting and verifying a type safety predicate relating the contents of memory and their corresponding types at each step of evaluation. Combining type assertions with other properties makes their system highly expressive, at the cost of placing additional burdens on the underlying theorem prover. While the authors provide a decision procedure for discharging type assertions, they do not address the problems of invariant inference

and reasoning with general quantified invariants.

The aforementioned projects focus on bringing the benefits of static checking to C programs. However, in recent years, a number of new languages and accompanying type systems have been created to address the problem of safe low-level programming.

The Cyclone project of Jim et al. [51] aims to develop a type- and memory-safe C-like language with region-based memory management, polymorphic types, existential types, and without pointer arithmetic. Types are explicitly specified by the user. The Cyclone language ensures memory and type safety for valid programs, but does not include a refinement type system capable of verifying more general properties. Additionally, programs written in C must be ported to Cyclone — for example, to remove pointer arithmetic to use regions in place of manual memory management.

Similarly, the BitC project of Shapiro et al. [76] attempts to bring strong static type checking and inference for (non-refinement) types to a low-level language suitable for operating system development. The goal of BitC is essentially to build a low-level derivative of ML which can be used to write systems software, that is, one which allows the programmer to determine the representations of data types and which features a type system that thoroughly integrates polymorphism with mutable state. In contrast to BitC, our system aims at proving more general properties of data structures in existing C programs, but does not (yet) incorporate features like type polymorphism and effect types.

The ATS project of Xi et al. [88] combines type checking and theorem proving techniques to create a language suitable for systems programming. The techniques supported by the system enable the verification of a wide range of properties — for example, linear types can be used to verify correct resource and API usage. Strong updates and pointer arithmetic are handled through *stateful views*, in which proof terms witness the types of memory contents at particular addresses; proof terms are consumed and produced during set and get operations, in an approach based on linear logic. The stateful views approach is more general than ours: for example, it is possible in ATS to change the type of all elements in a data structure after an update, while our system only allows strong updates on single data structure elements to ensure data structure invariants hold for the entire execution of the program, in spite of temporary invariant violations. The generality of ATS comes at the cost of increased programmer annotation burden: even simple programs using stateful views require the programmer to explicitly manipulate proof terms to show that heap accesses are within bounds and that the data accessed have the expected types.

In comparison to all of the above projects, low-level liquid types is the only system to combine a refinement type system expressive enough to statically verify memory safety in existing C programs while also supporting type inference.

1.7 Contributions

This dissertation makes the following contributions:

- We present *liquid types*, an approach to automated program verification based on refinement type inference. We show how liquid types combines type checking with predicate abstraction to automatically infer precise, universally-quantified invariants about unboundedly-large data structures like lists, arrays, and trees, and permits simple reasoning about higher-order functions.
- We develop the basic liquid types technique in the context of a higher-order, pure functional language.
- We present *low-level liquid types*, which adapts liquid types to the setting of low-level programs with mutable state, pointer arithmetic, and unrestricted aliasing.
- We show, through a series of benchmarks taken from the literature and from the wild, that the liquid types approach to program verification can be used to show a variety of safety and correctness properties of realistic programs, both functional and imperative, while imposing an extremely low annotation burden on the programmer — under 3% of the total number of the program source lines are composed of input predicate templates used in refinement type inference.

In the following chapters, we show that liquid refinement type inference allows programmers to verify data-sensitive safety properties of real-world programs written in high-level, functional languages, as well as in low-level imperative languages, at the cost of an extremely small annotation burden on the programmer. We first develop the liquid types technique in the context of a high-level, functional language which is a subset of ML. Next, we adapt the liquid types technique to the setting of a low-level, C-like language with pointer arithmetic and mutable state. Throughout, we show, through a series of benchmarks taken both from the literature and the wild, that liquid types enables the verification of a crucial safety properties in real-world programs while imposing an annotation burden of at most 3% of the program size. Finally, we suggest a number of directions for future work in extending the reach of the liquid type inference technique and the expressiveness of refinement types in general.

Chapter 2

Liquid Types

In this chapter, we develop the liquid type inference technique in the setting of an ML-like, higher-order, functional language.

2.1 Overview

To start, we show how the liquid types algorithm works through a series of examples that demonstrate how liquid types enables precise data- and control flow-sensitive reasoning to prove the safety of array-manipulating benchmarks which take advantage of language features like recursion, higher-order functions, and polymorphism.

2.1.1 Refinement Types and Qualifiers

We begin our overview of the liquid types algorithm for refinement type inference by describing refinement types, logical qualifiers, and liquid types.

Refinement Types Following [4, 35], our system allows *base refinement types* of the form

$$\{\nu : t \mid \phi\},$$

where ν is a special *value variable* not appearing in the program, t is a *base type*, and ϕ is a logical predicate constraining the value variable called the *refinement predicate*. Intuitively, the refinement predicate specifies the set of values v of the base type t such that the predicate $\phi[\nu \mapsto v]$ is valid. For example, $\{\nu : \texttt{int} \mid 0 < \nu\}$ specifies the set of positive integers, and $\{\nu : \texttt{int} \mid \nu \leq \texttt{n}\}$ specifies the set of integers whose value is less than or equal to the value of the program variable `n`. We use the base refinement types to build up *dependent function types*, written $x : \tau_1 \rightarrow \tau_2$ (following [4, 35]). Here, τ_1 is the domain type of the function, and the formal parameter x may appear in the refinements of the range type τ_2.

Logical Qualifiers and Liquid Types A *logical qualifier* is a logical predicate over the program variables, the special value variable ν which is distinct from the program variables, and the special placeholder variable $\star$ that can be instantiated with program variables.

For the rest of this subsection, let us assume that $\mathbb{Q}$ is the set of logical qualifiers

$$\{0 \leq \nu,\ \star \leq \nu,\ \nu < \star,\ \nu < \texttt{len}\ \star\}.$$

In section 2.4 we describe a simple set of qualifiers for array bounds checking. We say that a qualifier q *matches* the qualifier q' if replacing some subset of the free variables in q with $\star$ yields q'. For example, the qualifier $x \leq \nu$ matches the qualifier $\star \leq \nu$. We write $\mathbb{Q}^\star$ for the set of all qualifiers not containing $\star$ that match some qualifier in $\mathbb{Q}$. For example, when $\mathbb{Q}$ is as defined as above, $\mathbb{Q}^\star$ includes the qualifiers

$$\{0 \leq \nu,\ \texttt{x} \leq \nu,\ \texttt{y} \leq \nu,\ \texttt{k} \leq \nu,\ \nu < \texttt{n},\ \nu < \texttt{len a}\}.$$

We write t as an abbreviation for $\{\nu : t \mid \texttt{true}\}$. Additionally, when the base type t is clear from the context, we abbreviate $\{\nu : t \mid \phi\}$ as $\{\phi\}$. For example,

$$\texttt{x} : \texttt{int} \rightarrow \texttt{y} : \texttt{int} \rightarrow \{\texttt{x} \leq \nu \wedge \texttt{y} \leq \nu\}$$

denotes the type of a (curried) function that takes two integer arguments x and y and returns an integer no less than x and y.

2.1.2 Liquid Type Inference by Example

Given a program and a set of qualifiers $\mathbb{Q}$, our liquid type inference algorithm proceeds in three steps:

Step 1: Hindley-Milner Type Inference First, our algorithm invokes Hindley-Milner [26] to infer types for each subexpression and the necessary type generalization and instantiation annotations. Next, our algorithm uses the computed ML types to assign to each subexpression a *template*, a dependent type with the same structure as the inferred ML type, but which has *liquid type variables* κ representing the unknown type refinements.

Step 2: Liquid Constraint Generation Second, we use the syntax-directed liquid typing rules to generate a system of constraints that capture the subtyping relationships between the templates that must be met for a liquid type derivation to exist.

Step 3: Liquid Constraint Solving Third, our algorithm uses the subtyping rules to split the complex template constraints into simple constraints over the liquid type variables. Our algorithm then solves these simple constraints using a fixpoint computation inspired by predicate abstraction [1, 43] to find, for each κ, the *strongest* conjunction of qualifiers from $\mathbb{Q}^\star$ that satisfies

```
let max x y =
  if x > y then x else y

let rec sum k =
  if k < 0 then 0 else
    let s = sum (k-1) in
    s + k

let foldn n b f =
  let rec loop i c =
    if i < n then loop (i+1) (f i c) else c in
  loop 0 b

let arraymax a =
  let am l m = max (sub a l) m in
  foldn (len a) 0 am
```

Figure 2.1: Example OCaml Program

all the constraints. Note that, for the final step, we need only consider the finite subset of $\mathbb{Q}^\star$ whose free variables belong to the program.

In the following, through a series of examples, we show how our type inference algorithm incorporates features essential for inferring precise dependent types — namely path sensitivity, recursion, higher-order functions, and polymorphism — and thus can statically prove the safety of array accesses.

Example 1: Path Sensitivity

Consider the `max` function, shown in Figure 2.1, as an OCaml program. We will show how our algorithm infers that `max` returns a value no less than both its arguments.

Step 1 HM infers that `max` has the type $\texttt{x} : \texttt{int} \rightarrow \texttt{y} : \texttt{int} \rightarrow \texttt{int}$. Using this type, we create a *template* for the liquid type of `max`, $\texttt{x} : \{\kappa_\texttt{x}\} \rightarrow \texttt{y} : \{\kappa_\texttt{y}\} \rightarrow \{\kappa_1\}$, where $\kappa_\texttt{x}$, $\kappa_\texttt{y}$, κ_1 are *liquid type variables* representing the unknown refinements for the formals `x` and `y` and the body of `max`, respectively.

Step 2 As the body is an `if` expression, our algorithm generates the following two constraints that stipulate that, under the appropriate branch condition, the `then` and `else` expressions, respectively `x` and `y`, have types that are *subtypes* of the entire body's type:

$$\texttt{x} : \{\kappa_\texttt{x}\}; \texttt{y} : \{\kappa_\texttt{y}\}; \texttt{x} > \texttt{y} \vdash \{\nu = \texttt{x}\} <: \{\kappa_1\} \tag{1.1}$$

$$\texttt{x} : \{\kappa_\texttt{x}\}; \texttt{y} : \{\kappa_\texttt{y}\}; \neg(\texttt{x} > \texttt{y}) \vdash \{\nu = \texttt{y}\} <: \{\kappa_1\} \tag{1.2}$$

Constraint 1.1 stipulates that when x and y have the types $\{\kappa_x\}$ and $\{\kappa_y\}$, respectively, and $x > y$, the type of the expression x, namely the set of all values equal to x, must be a subtype of the body's type, $\{\kappa_1\}$. Similarly, constraint 1.2 stipulates that when x and y have the types $\{\kappa_x\}$ and $\{\kappa_y\}$, respectively, and $\neg(x > y)$, the type of the expression y, namely the set of all values equal to y, must be a subtype of the body's type, $\{\kappa_1\}$.

Step 3 Since the program is "open", i.e., there are no calls to max, we assign κ_x and κ_y the predicate true, meaning that *any* integer arguments can be passed, and use a theorem prover to find the strongest conjunction of qualifiers in $\mathbb{Q}^\star$ that satisfies the subtyping constraints. The theorem prover deduces that when $x > y$ (respectively, $\neg(x > y)$) if $\nu = x$ (respectively, $\nu = y$) then $x \leq \nu$ and $y \leq \nu$. Hence, our algorithm infers that $x \leq \nu \wedge y \leq \nu$ is the strongest solution for κ_1 that satisfies the two constraints. By substituting the solution for κ_1 into the template for max, our algorithm infers

$$\texttt{max} : \texttt{x} : \texttt{int} \rightarrow \texttt{y} : \texttt{int} \rightarrow \{\nu : \texttt{int} \mid \texttt{x} \leq \nu \wedge \texttt{y} \leq \nu\}.$$

Example 2: Recursion

Next, we show how our algorithm infers that the recursive function sum from Figure 2.1 always returns a non-negative value greater than or equal to its argument k.

Step 1 HM infers that sum has the type $\texttt{k} : \texttt{int} \rightarrow \texttt{int}$. Using this type, we create a template for the liquid type of sum, $\texttt{k} : \{\kappa_k\} \rightarrow \{\kappa_2\}$, where κ_k and κ_2 represent the unknown refinements for the formal k and body, respectively. Due to the let rec, we use the created template as the type of sum when generating constraints for the body of sum.

Step 2 Again, as the body is an if expression, we generate constraints that stipulate that, under the appropriate branch conditions, the "then" and "else" expressions have subtypes of the body type $\{\kappa_2\}$. For the "then" branch, we get a constraint:

$$\texttt{sum} : \ldots ; \texttt{k} : \{\kappa_k\}; \texttt{k} < 0 \vdash \{\nu = 0\} <: \{\kappa_2\} \tag{2.1}$$

The else branch is a let expression. First, considering the expression that is locally bound, we generate a constraint

$$\texttt{sum} : \ldots ; \texttt{k} : \{\kappa_k\}; \neg(\texttt{k} < 0) \vdash \{\nu = \texttt{k} - 1\} <: \{\kappa_k\} \tag{2.2}$$

from the call to `sum` that forces the actual passed in at the callsite to be a subtype of the formal of `sum`. The locally bound variable `s` gets assigned the template corresponding to the output of the application, $\{\kappa_2[\mathtt{k} \mapsto \mathtt{k}-1]\}$, i.e., the output template of `sum` with the formal replaced with the actual argument, and we get the next constraint that ensures the "else" expression is a subtype of the body's type, $\{\kappa_2\}$:

$$\neg(\mathtt{k} < 0); \mathtt{s} : \{\kappa_2[\mathtt{k} \mapsto \mathtt{k}-1]\} \vdash \{\nu = \mathtt{s} + \mathtt{k}\} <: \{\kappa_2\}. \tag{2.3}$$

Step 3 Here, as `sum` is called, we try to find the strongest conjunction of qualifiers for $\kappa_\mathtt{k}$ and κ_2 that satisfies the constraints. To satisfy constraint 2.2, $\kappa_\mathtt{k}$ can only be assigned `true` (the empty conjunction), as when $\neg(\mathtt{k} < 0)$, the value of $\mathtt{k}-1$ can be negative, zero, or positive. On the other hand, κ_2 is assigned $0 \leq \nu \wedge \mathtt{k} \leq \nu$, the strongest conjunction of qualifiers in $\mathbb{Q}^\star$ that satisfies constraint 2.1 and constraint 2.3. Constraint 2.1 is trivially satisfied as the theorem prover deduces that when $\mathtt{k} < 0$, if $\nu = 0$ then $0 \leq \nu$ and $\mathtt{k} \leq \nu$. When κ_2 is assigned the above conjunction, the binding for `s` in the environment for constraint 2.3 becomes $\mathtt{s} : \{0 \leq \nu \wedge \mathtt{k} - 1 \leq \nu\}$. Thus, constraint 2.3 is satisfied, as the theorem prover deduces that when $\neg(\mathtt{k} < 0)$ and $(0 \leq \nu \wedge \mathtt{k} - 1 \leq \nu)[\nu \mapsto \mathtt{s}]$, if $\nu = \mathtt{s} + \mathtt{k}$ then $0 \leq \nu$ and $\mathtt{k} \leq \nu$. The substitution simplifies to $0 \leq \mathtt{s} \wedge \mathtt{k} - 1 \leq \mathtt{s}$, which effectively asserts to the solver the knowledge about the type of `s`, and crucially allows the solver to use the fact that `s` is non-negative when determining the type of $\mathtt{s} + \mathtt{k}$, and hence the output of `sum`. Thus, recursion enters the picture, as the solution for the output of the recursive call, which is bound to the type of `s`, is used in conjunction with the branch information to prove that the output expression is non-negative. Plugging the solutions for $\kappa_\mathtt{k}$ and κ_2 into the template, our system infers

$$\mathtt{sum} : \mathtt{k} : \mathtt{int} \rightarrow \{\nu : \mathtt{int} \mid 0 \leq \nu \wedge \mathtt{k} \leq \nu\}.$$

Example 3: Higher-Order Functions

Next, consider a program comprising only the higher-order accumulator `foldn` shown in Figure 2.1. We show how our algorithm infers that `f` is only called with arguments between 0 and `n`.

Step 1 HM infers that `foldn` has the polymorphic type

$$\Lambda\alpha.\mathtt{n} : \mathtt{int} \rightarrow \mathtt{b} : \alpha \rightarrow \mathtt{f} : (\mathtt{int} \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha.$$

From this ML type, we create the new template

$$\Lambda\alpha.\mathtt{n} : \{\kappa_\mathtt{n}\} \rightarrow \mathtt{b} : \alpha \rightarrow \mathtt{f} : (\{\kappa_3\} \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha$$

for `foldn`, where $\kappa_\mathtt{n}$ and κ_3 represent the unknown refinements for the formal `n` and the first parameter for the accumulation function `f` passed into `foldn`. This is a *polymorphic* template, as the

occurrences of α are preserved. This will allow us to *instantiate* α with an appropriate dependent type at places where `foldn` is called. HM infers that the type of `loop` is $\texttt{i} : \texttt{int} \rightarrow \texttt{c} : \alpha \rightarrow \alpha$, from which we generate a template $\texttt{i} : \kappa_{\texttt{i}} \rightarrow \texttt{c} : \alpha \rightarrow \alpha$ for `loop`, which we will use when analyzing the body of `loop`.

Step 2 First, we generate constraints inside the body of `loop`. As HM infers that the type of the body is α, we omit the trivial subtyping constraints on the "then" and "else" expressions. Instead, the two interesting constraints are:

$$\ldots;\texttt{i} : \{\kappa_{\texttt{i}}\};\texttt{i} < \texttt{n} \vdash \{\nu = \texttt{i} + 1\} <: \{\kappa_{\texttt{i}}\} \tag{3.1}$$

which stipulates that the actual passed into the recursive call to `loop` is a subtype of the expected formal, and

$$\ldots;\texttt{i} : \{\kappa_{\texttt{i}}\};\texttt{i} < \texttt{n} \vdash \{\nu = \texttt{i}\} <: \{\kappa_3\} \tag{3.2}$$

which forces the actual `i` to be a subtype of the first parameter of the higher-order function `f`, in the environment containing the critical branch condition. Finally, the application `loop` 0 yields

$$\ldots \vdash \{\nu = 0\} <: \{\kappa_{\texttt{i}}\} \tag{3.3}$$

forcing the type of the actual, 0, to be a subtype of the type of the formal, `i`.

Step 3 Here, as `foldn` is not called, we assign $\kappa_{\texttt{n}}$ the predicate `true` and try to find the strongest conjunction of qualifiers in $\mathrm{Q}^\star$ for $\kappa_{\texttt{i}}$ and κ_3. We can assign to $\kappa_{\texttt{i}}$ the predicate $0 \leq \nu$, which trivially satisfies constraint 3.3, and also satisfies constraint 3.1 as when $(0 \leq \nu)[\nu \mapsto \texttt{i}]$, if $\nu = \texttt{i} + 1$ then $0 \leq \nu$. That is, the theorem prover can deduce that if `i` is non-negative, then so is $\texttt{i} + 1$. To κ_3 we can assign the conjunction $0 \leq \nu \wedge \nu < \texttt{n}$ which satisfies constraint 3.2 as when $(0 \leq \nu)[\nu \mapsto \texttt{i}]$ and $\texttt{i} < \texttt{n}$, if $\nu = \texttt{i}$ then $0 \leq \nu$ and $\nu < \texttt{n}$. By plugging the solutions for κ_3 and $\kappa_{\texttt{n}}$ into the template our algorithm infers

$$\texttt{foldn} : \Lambda\alpha.\texttt{n} : \texttt{int} \rightarrow \texttt{b} : \alpha \rightarrow \texttt{f} : (\{0 \leq \nu \wedge \nu < \texttt{n}\} \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha.$$

Example 4: Polymorphism and Array Bounds Checking Consider the function `amax` that calls `foldn` with a helper that calls `max` to compute the max of the elements of an array and 0. Suppose there is a base type `array` representing arrays of integers. Arrays are accessed via a primitive function

$$\texttt{sub} : \texttt{a} : \texttt{array} \rightarrow \texttt{j} : \{\nu : \texttt{int} \mid 0 \leq \nu \wedge \nu < \texttt{len a}\} \rightarrow \texttt{int},$$

where the primitive function `len` returns the number of elements in the array. The `sub` function takes an array and an index that is between zero and the number of elements, and returns the integer at that index in the array. We show how our algorithm combines predicate abstraction,

function subtyping, and polymorphism to prove that (a) the array `a` is safely accessed at indices between 0 and `len a`, and (b) `amax` returns a non-negative integer.

Step 1 HM infers that (1) `amax` has the type $\texttt{a} : \texttt{array} \rightarrow \texttt{int}$, (2) `am` has the type $\texttt{l} : \texttt{int} \rightarrow \texttt{m} : \texttt{int} \rightarrow \texttt{int}$, and (3) `foldn` called in the body is a polymorphic instance where the type variable α has been instantiated with `int`. Consequently, our algorithm creates the following templates: (1) $\texttt{a} : \texttt{array} \rightarrow \{\kappa_4\}$ for `amax`, where κ_4 represents the unknown refinement for the output of `amax`, (2) $\texttt{l} : \kappa_{\texttt{l}} \rightarrow \texttt{m} : \kappa_{\texttt{m}} \rightarrow \{\kappa_5\}$ for `am`, where $\kappa_{\texttt{l}}$, $\kappa_{\texttt{m}}$, and κ_5 represent the unknown refinements for the parameters and output type of `am` respectively, and (3) $\{\kappa_6\}$ for the type that α is instantiated with, and so the template for the instance of `foldn` inside `amax` is the type computed in the previous example with $\{\kappa_6\}$ substituted for α, namely,

$$\texttt{n} : \texttt{int} \rightarrow \texttt{b} : \{\kappa_6\} \rightarrow \texttt{f} : (\{0 \leq \nu \wedge \nu < \texttt{n}\} \rightarrow \{\kappa_6\} \rightarrow \{\kappa_6\}) \rightarrow \{\kappa_6\}.$$

Step 2 First, for the application `sub a l`, our algorithm generates

$$\texttt{l} : \{\kappa_{\texttt{l}}\}; \texttt{m} : \{\kappa_{\texttt{m}}\} \vdash \{\nu = \texttt{l}\} <: \{0 \leq \nu \wedge \nu < \texttt{len a}\}, \tag{4.1}$$

which states that the argument passed into `sub` must be within the array bounds. For the application `max (sub a l) m`, using the type inferred for `max` in Example 1, we get

$$\texttt{l} : \{\kappa_{\texttt{l}}\}; \texttt{m} : \{\kappa_{\texttt{m}}\} \vdash \{\texttt{sub a l} \leq \nu \wedge \texttt{m} \leq \nu\} <: \{\kappa_5\}, \tag{4.2}$$

which constrains the output of `max` (with the actuals `(sub a l)` and `m` substituted for the parameters `x` and `y`, respectively), to be a subtype of the output type $\{\kappa_5\}$ of `am`. The call `foldn (len a) 0` generates

$$\ldots \vdash \{\nu = 0\} <: \{\kappa_6\}, \tag{4.3}$$

which forces the actual passed in for `b` to be a subtype of $\{\kappa_6\}$, the type of the formal `b` in this polymorphic instance. Similarly, the call `foldn (len a) 0 am` generates a constraint

$$\ldots \vdash \texttt{l} : \{\kappa_{\texttt{l}}\} \rightarrow \texttt{m} : \{\kappa_{\texttt{m}}\} \rightarrow \{\kappa_5\} <: \{0 \leq \nu \wedge \nu < \texttt{len a}\} \rightarrow \{\kappa_6\} \rightarrow \{\kappa_6\}, \tag{4.4}$$

forcing the type of the actual `am` to be a subtype of the formal `f` inferred in Example 1, with the curried argument `len a` substituted for the formal `n` of `foldn`, and

$$\ldots \vdash \{\kappa_6\} <: \{\kappa_4\}, \tag{4.5}$$

forcing the output of the `foldn` application to be a subtype of the body of `amax`. Upon simplification using the standard rule for subtyping function types, constraint 4.4 reduces to

$$\ldots \vdash \{0 \leq \nu \wedge \nu < \texttt{len a}\} <: \{\kappa_{\texttt{l}}\} \tag{4.6}$$

$$\ldots \vdash \{\kappa_6\} <: \{\kappa_{\texttt{m}}\} \tag{4.7}$$

$$\ldots \vdash \{\kappa_5\} <: \{\kappa_6\} \tag{4.8}$$

Step 3 The strongest conjunction of qualifiers from $\mathbb{Q}^\star$ that we can assign to $\kappa_{\texttt{m}}$, κ_4, κ_5 and κ_6 is the predicate $0 \leq \nu$. In essence, our algorithm infers that we can "instantiate" the type variable α with the dependent type $\{\nu : \texttt{int} \mid 0 \leq \nu\}$. This is sound because the base value 0 passed in is non-negative, so that constraint 4.3 is satisfied, and the accumulation function passed in (`am`), is such that if its second argument (`m` of type $\{\kappa_{\texttt{m}}\}$) is non-negative then the output (of type $\{\kappa_5\}$) is non-negative, so that constraint 4.2 is satisfied. Plugging the solution into the template, our algorithm infers

$$\texttt{amax} : \texttt{array} \rightarrow \{\nu : \texttt{int} \mid 0 \leq \nu\}.$$

The strongest conjunction over $\mathbb{Q}^\star$ we can assign to $\kappa_{\texttt{l}}$ is $0 \leq \nu \wedge \nu < \texttt{len a}$, which trivially satisfies constraint 4.6. Moreover, with this assignment, we have satisfied the "bounds check" constraint 4.1, i.e., we have inferred an assignment of dependent types to all the program expressions that proves that all array accesses occur within bounds.

This concludes our broad overview of the liquid types refinement type inference technique. In section 2.2, we outline the basic ML-like language, λ_L, on which we build the liquid types algorithm, and give its refinement typing rules. We give the details of the constraint-based liquid types algorithm in section 2.3, and prove its correctness in Appendix A. We report benchmark results in section 2.4.

2.2 The λ_L Language and Type System

We first present the syntax and static semantics of our core language λ_L, a variant of the λ-calculus with ML-style polymorphism extended with liquid types. We begin by describing the elements of λ_L, including expressions, types, and environments (Section 2.2.1). Next, we present the type judgments and derivation rules and state a soundness theorem which relates the static type system with the operational semantics (Section 2.2.2). We conclude by describing how the design of our type system enables automatic refinement type inference (Section 2.2.3).

2.2.1 Elements of λ_L

The syntax of values, expressions, and types for λ_L is summarized in Figure 2.2. λ_L values include variables and special constants which include integers, arithmetic operators and other primitive operations described below. λ_L expressions include values, λ-abstractions, and function applications. In addition, λ_L includes as expressions the common constructs `if-then-else` and `let`, which the liquid type inference algorithm exploits to generate precise types.

Types and Schemas We use t to denote base types such as `bool` or `int`. λ_L has a system of refined base types, dependent function types, and ML-style polymorphism via type variables

v	::=		**Values**
		x	variable
	|	`c`	constant
e	::=		**Expressions**
		v	value
	|	$\lambda x.e$	abstraction
	|	$v_1\ v_2$	application
	|	**if** v **then** e_1 **else** e_2	if-then-else
	|	**let** $x = e_1$ **in** e_2	let binding
	|	$\Lambda\alpha.e$	type abstraction
	|	$e[\dot{\tau}]$	type instantiation
Q	::=		**Liquid Refinements**
		`true`	true
	|	q	logical qualifier in $\mathbb{Q}^\star$
	|	$Q_1 \wedge Q_2$	conjunction of qualifiers
t	::=		**Base Types**
		`int`	base type of integers
	|	`bool`	base type of Booleans
	|	α	type variable
$F(R)$	::=		**Type Skeletons**
		$\{v : t \mid R\}$	base
	|	$x : F(R) \rightarrow F(R)$	function
$S(R)$	::=		**Type Schema Skeletons**
		$F(R)$	monotype
	|	$\Lambda\alpha.S(R)$	polytype
$\dot{\tau}$, $\dot{\sigma}$	::=	$F(\cdot)$, $S(\cdot)$	**Types, Schemas**
τ, σ	::=	$F(\phi)$, $F(\phi)$	**Refinement Types, Schemas**
$\hat{\tau}$, $\hat{\sigma}$	::=	$F(Q)$, $S(Q)$	**Liquid Types, Schemas**

Figure 2.2: Syntax of λ_L expressions and types

that are universally quantified at the outermost level to yield polymorphic type schemas. We write $\dot{\tau}$ and $\dot{\sigma}$ for ML types and schemas, τ and σ for refinement types and schemas, and $\hat{\tau}$ and $\hat{\sigma}$ for liquid types and schemas. Our refinement predicates ϕ are drawn from EUFA, the decidable logic of equality, uninterpreted functions, and linear arithmetic [69].

Environments and Well-formedness A *type environment* Γ is a sequence of *type bindings* $x : \hat{\sigma}$ and *guard predicates* ϕ. The former are standard; the latter capture constraints about the if-then-else branches under which an expression is evaluated, which is required to make the system path sensitive (Section 2.2.3). A type is considered *well-formed* with respect to an environment if all the free variables appearing in the refinement predicates of the type are bound in the environment. An environment is considered *well-formed* if, in each type binding, the type is well-formed with respect to the preceding (prefix) environment.

Shapes The *shape* of the refinement type $\hat{\tau}$, denoted by $\text{Shape}(\hat{\tau})$, is the ML type obtained by erasing all refinement predicates. We lift Shape to type schemas $\hat{\sigma}$ in the natural way. We lift Shape to type environments by applying it to each type binding and eliminating the guard predicates.

Constants As in [71, 35], the basic units of computation are the constants $\mathtt{c}$ built into λ_L, each of which has a dependent type $\text{ConstType}(\mathtt{c})$ that precisely captures the semantics of the constants. These include *basic constants*, corresponding to integers and Boolean values, and *primitive functions*, which encode various operations. The set of constants of λ_L includes:

$$
\begin{array}{rcl}
\texttt{true} & : & \{\nu : \texttt{bool} \mid \nu\} \\
\texttt{false} & : & \{\nu : \texttt{bool} \mid \neg\nu\} \\
\Leftrightarrow & : & \texttt{x}:\texttt{bool} \rightarrow \texttt{y}:\texttt{bool} \rightarrow \{\nu : \texttt{bool} \mid \nu \Leftrightarrow (\texttt{x} \Leftrightarrow \texttt{y})\} \\
3 & : & \{\nu : \texttt{int} \mid \nu = 3\} \\
= & : & \texttt{x}:\texttt{int} \rightarrow \texttt{y}:\texttt{int} \rightarrow \{\nu : \texttt{bool} \mid \nu \Leftrightarrow \texttt{x} = \texttt{y}\} \\
+ & : & \texttt{x}:\texttt{int} \rightarrow \texttt{y}:\texttt{int} \rightarrow \{\nu : \texttt{int} \mid \nu = \texttt{x} + \texttt{y}\} \\
\texttt{fix} & : & \Lambda\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha \\
\texttt{len} & : & \texttt{a}:\texttt{array} \rightarrow \{\nu : \texttt{int} \mid 0 \leq \nu \wedge \nu = \texttt{len a}\} \\
\texttt{sub} & : & \texttt{a}:\texttt{array} \rightarrow \texttt{i}:\{\nu : \texttt{int} \mid 0 \leq \nu \wedge \nu < \texttt{len a}\} \rightarrow \texttt{int}
\end{array}
$$

It may seem that the types of some constants are defined in terms of themselves — for example, in the return type of $+$. This is simply an artifact of using the same symbol, $+$, to represent both addition in the logic we use to express refinement predicates and as a name for the ML function which performs addition; the entities referenced by the symbol are completely distinct.

For clarity, we will use infix notation for constants like $+$. To simplify the exposition, we assume there is a special base type that encodes integer arrays in λ_L. The length of an array

Well-Formed Types $\boxed{\Gamma \vDash \sigma}$

$$\frac{\phi \text{ well-sorted in } \Gamma; \nu : t}{\Gamma \vDash \{\nu : t \mid \phi\}} \text{ WT-BASE} \qquad \frac{}{\Gamma \vDash \alpha} \text{ WT-VAR}$$

$$\frac{\Gamma; x : \tau_x \vDash \tau}{\Gamma \vDash x : \tau_x \rightarrow \tau} \text{ WT-FUN} \qquad \frac{\Gamma \vDash \sigma}{\Gamma \vDash \Lambda\alpha.\sigma} \text{ [WT-POLY]}$$

Figure 2.3: Rules for Liquid Type Well-Formedness

value is obtained using `len`. To access the elements of the array, we use `sub`, which takes as input an array `a` and an index `i` that must be within the bounds of `a`, i.e., non-negative and less than the length of the array.

2.2.2 Liquid Type Checking Rules

We now describe the key ingredients of the type system: the typing judgments and derivation rules summarized in Figure 2.4.

Our system has three kinds of judgments relating environments, expressions, and types.

Well-formedness Judgment $\Gamma \vDash \tau$: This judgment states that the dependent type schema τ is *well-formed* under the type environment Γ. Intuitively, a type is well-formed with respect to an environment if its base refinements are well-sorted, Boolean-valued predicates which refer only to variables in the environment. We take our refinement logic to have three basic sorts, `int`, `bool`, and `array`, and define well-sortedness with respect to an environment straightforwardly.

Subtype Judgment $\Gamma \vdash \sigma_1 <: \sigma_2$: This judgment states that refinement type schema σ_1 is a subtype of schema σ_2 in environment Γ.

Liquid Type Judgment $\Gamma \vdash_{\mathbb{Q}} e : \sigma$: This judgment states that, using the logical qualifiers $\mathbb{Q}$, the expression e has the type schema τ under the type environment Γ.

Soundness of Liquid Type Checking We note that our liquid typing judgment $\Gamma \vdash_{\mathbb{Q}} e : \sigma$ is a refinement of a general refinement typing judgment $\Gamma \vdash e : \sigma$, that is, any valid liquid type derivation is automatically a derivation in a more general refinement type system, as the only difference between the two is that our liquid type system enforces a particular structure on the refinements that may occur in certain types. The soundness of our type system follows from the soundness of similar refinement type systems, treated at length by Bengtson et al. [7], Knowles and Flanagan [55], and Belo et al. [6], among others; since our focus is on refinement inference

Liquid Type Checking $\boxed{\Gamma \vdash_{\mathrm{Q}} e : \sigma}$

$$\frac{\Gamma \vdash_{\mathrm{Q}} e : \sigma_1 \qquad \Gamma \vdash \sigma_1 <: \sigma_2 \qquad \Gamma \vDash \sigma_2}{\Gamma \vdash_{\mathrm{Q}} e : \sigma_2}\ \text{LT-Sub}$$

$$\frac{\Gamma(x) = \{\nu : t \mid \phi\}}{\Gamma \vdash_{\mathrm{Q}} x : \{\nu : t \mid \nu = x\}}\ \text{LT-Var} \qquad \frac{\Gamma(x) \text{ not a base type}}{\Gamma \vdash_{\mathrm{Q}} x : \Gamma(x)}\ \text{LT-Var}$$

$$\frac{}{\Gamma \vdash_{\mathrm{Q}} \mathtt{c} : \mathrm{ConstType}(\mathtt{c})}\ \text{LT-Const}$$

$$\frac{\Gamma; x : \hat{\tau}_x \vdash_{\mathrm{Q}} e : \tau \qquad \Gamma \vDash x : \hat{\tau}_x \ \to\ \tau}{\Gamma \vdash_{\mathrm{Q}} \lambda x.e : x : \hat{\tau}_x \ \to\ \tau}\ \text{LT-Fun}$$

$$\frac{\Gamma \vdash_{\mathrm{Q}} v_1 : x : \tau_x \ \to\ \tau \qquad \Gamma \vdash_{\mathrm{Q}} v_2 : \tau_x}{\Gamma \vdash_{\mathrm{Q}} v_1\ v_2 : \tau[x \mapsto v_2]}\ \text{LT-App}$$

$$\frac{\Gamma \vdash_{\mathrm{Q}} v : \mathtt{bool} \qquad \Gamma; v \vdash_{\mathrm{Q}} e_1 : \hat{\sigma} \qquad \Gamma; \neg v \vdash_{\mathrm{Q}} e_2 : \hat{\sigma} \qquad \Gamma \vDash \hat{\sigma}}{\Gamma \vdash_{\mathrm{Q}} \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2 : \hat{\sigma}}\ \text{LT-If}$$

$$\frac{\Gamma \vdash_{\mathrm{Q}} e : \sigma \qquad \alpha \text{ not free in } \Gamma}{\Gamma \vdash_{\mathrm{Q}} \Lambda\alpha.e : \Lambda\alpha.\sigma}\ \text{LT-Gen}$$

$$\frac{\Gamma \vdash_{\mathrm{Q}} e : \Lambda\alpha.\sigma \qquad \Gamma \vDash \hat{\tau} \qquad \mathrm{Shape}(\hat{\tau}) = \dot{\tau}}{\Gamma \vdash_{\mathrm{Q}} e[\dot{\tau}] : \sigma[\alpha \mapsto \hat{\tau}]}\ \text{LT-Inst}$$

Subtyping $\boxed{\Gamma \vdash \sigma_1 <: \sigma_2}$

$$\frac{\Gamma \vDash \phi_1 \Rightarrow \phi_2}{\Gamma \vdash \{\nu : t \mid \phi_1\} <: \{\nu : t \mid \phi_2\}}\ <:\text{-Base}$$

$$\frac{\Gamma \vdash \tau'_x <: \tau_x \qquad \Gamma; x : \tau'_x \vdash \tau <: \tau'}{\Gamma \vdash x : \tau_x \ \to\ \tau <: x : \tau'_x \ \to\ \tau'}\ <:\text{-Fun}$$

$$\frac{}{\Gamma \vdash \alpha <: \alpha}\ <:\text{-Var} \qquad \frac{\Gamma \vdash \sigma_1 <: \sigma_2}{\Gamma \vdash \Lambda\alpha.\sigma_1 <: \Lambda\alpha.\sigma_2}\ <:\text{-Poly}$$

Figure 2.4: Rules for Liquid Type Checking

rather than the underlying refinement type system, we simply assume the soundness of the underlying refinement type system. We formalize this assumption as follows:

Assumption 1. *(Overapproximation of Liquid Type Judgment) If* $\Gamma \vdash_{\mathbf{Q}} e : \sigma$ *then* $\Gamma \vdash e : \sigma$.

Let $\hookrightarrow$ describe the single evaluation step relation for λ_L expressions and $\hookrightarrow *$ describe the reflexive, transitive closure of $\hookrightarrow$. We formalize our assumption of a sound underlying refinement type system as follows:

Assumption 2. *(Refinement Type Soundness) If* $\Gamma \vdash e : \sigma$, *then either* $e \hookrightarrow *v$ *for some value* v *or the evaluation of* e *does not terminate.*

The preceding two assumptions combine to give the safety of the liquid types system:

Theorem 1. *(Liquid Type Safety) If* $\emptyset \vdash_{\mathbf{Q}} e : \sigma$, *then either* $e \hookrightarrow *v$ *for some value* v *or the evaluation of* e *does not terminate.*

We conclude that if an expression is well-typed in our type system then we are guaranteed that evaluation does not get "stuck", i.e., at run-time, every primitive operation receives valid inputs. Thus, if a program type checks we are guaranteed that every call to `sub` gets an index that is within the array's bounds. Arbitrary safety properties (e.g., divide-by-zero errors) can be expressed by using suitable types for the appropriate primitive constant (e.g., requiring the second argument of $(/)$ to be non-zero or requiring the argument to the `assert` function to evaluate to `true`).

2.2.3 Features of the Liquid Type System

Next, we describe some of the features unique to the design of the liquid type system and how they contribute to automatic type inference and verification.

1. Value and Path Sensitivity Our type system is both value and path sensitive: its reasoning incorporates both the values of variables, as determined through the refinement type bindings in the environment, and information about the branches under which an expression is evaluated, determined by the guard predicates in the environment. Both value and path sensitivity are crucial to proving properties like the safety of array accesses within a program. For example, our type system uses the branch information in the `sum` example of section 2.1 to infer that the occurrence of `k` inside the `else` expression is non-negative, since the `else` expression is evaluated only when $\texttt{k} \geq 0$. Further, our system uses the fact that `s` is non-negative, expressed as the refinement type $\{\nu : \texttt{int} \mid \nu \geq 0\}$ bound to `s`, to determine that the expression $\texttt{s} + \texttt{k}$ returns a non-negative value.

Note that value and path sensitivity are especially important in performing static array bounds checking, as programmers often compare an array index to some some other variable that is known to be smaller than the array length (e.g., in `amax` from section 2.1), and only perform the array access under the appropriate guard; verifying the safety of array accesses in this situation

requires knowing not only what values variables may take on, captured in their refinement types, but also which branch conditions are in effect.

Checking that a value has an expected type — for example, that an array index is within bounds — is done by performing a subtyping check, while type inference is performed by generating subtyping constraints. Thus, the crucial place where value and path information must be accounted for in our reasoning is in subtyping. By the rules in Figure 2.4, all subtyping checks are reduced to subtyping checks over base (i.e., non-function) refinement types; these subtyping checks are further reduced to checking logical implications between refinement predicates, given the value and path assumptions in the environment. Thus, in order to use the information from the environment in implication checks, we must embed the type environment into the logic. We write

$$[\![\Gamma]\!] \equiv \bigwedge\{\phi \mid \phi \in \Gamma\} \wedge \bigwedge\{\phi[\nu \mapsto x] \mid x : \{\nu : t \mid \phi\} \in \Gamma\}$$

as the embedding for the environment into our refinement logic. Notice that we use the guard predicates and base type bindings in the environment to strengthen the antecedent of the implication. However, we substitute all occurrences of the value variable ν in the refinements from Γ with the actual variable being refined, thereby asserting in the antecedent that the program variable satisfies the base refinement predicate. Thus, in the embedded formula, all occurrences of ν refer to the two types that are being checked for subtyping. For example, for the `then` expression in `max` from section 2.1, the subtyping relation: $\texttt{x} : \texttt{int}; \texttt{y} : \texttt{int}; \texttt{x} > \texttt{y} \vdash \{\nu = \texttt{x}\} <: \{\texttt{x} \leq \nu \wedge \texttt{y} \leq \nu\}$ holds as the following implication is valid in EUFA: $(\texttt{true} \wedge \texttt{true} \wedge \texttt{x} > \texttt{y}) \wedge (\nu = \texttt{x}) \Rightarrow \texttt{x} \leq \nu \wedge \texttt{y} \leq \nu$.

2. Recursion via Polymorphism To handle polymorphism, our type system incorporates type generalization and instantiation annotations, which are over ML type variables α and monomorphic ML types $\dot{\tau}$, respectively, and thus can be reconstructed via a standard type inference algorithm. The rule LT-INST allows a type schema to be instantiated with an arbitrary liquid type $\hat{\tau}$ of the same shape as $\dot{\tau}$, the monomorphic ML type used for instantiation. We use polymorphism to encode recursion via the polymorphic type given to `fix`. That is, `let rec` bindings are syntactic sugar: `let rec f = e in e'` is internally converted to `let f = fix (fun f -> e) in e'`. The expression type checks if there is an appropriate liquid type that can be instantiated for the α in the polymorphic type of `fix`; this liquid type corresponds to the type of the recursive function `f`.

3. The Liquid Type Restriction The most critical difference between the rules for liquid type checking and other refinement type systems is that our rules stipulate that certain kinds of expressions have liquid types. In essence, these expressions are the key points where appropriate refinement types must be inferred. By forcing the types to be liquid, we bound the space of possible solutions, thus making inference efficiently decidable.

LT-INST For polymorphic instantiation, also the mechanism for handling recursion, the liquid type restriction enables efficient inference by making the set of candidate refinement types finite.

LT-FUN For λ-abstractions, we impose the restriction that the input and output be liquid to ensure the types remain small, thereby making algorithmic checking and inference efficient. This is analogous to procedure "summarization" for first-order programs.

LT-IF For conditional expressions we impose the liquid restriction and implicitly force the `then` and `else` expressions to be subtypes of a fresh liquid type, instead of an explicit "join" operator as in dataflow analysis. We do so as the expression may have a function type and, with a join operator, input type contravariance would introduce disjunctions into the refinement type which would have unpleasant algorithmic consequences.

LT-LET For let expressions, we impose the liquid restriction as a means of eliminating the locally bound variable from the refinement type of the whole expression (as the local variable goes out of scope). The antecedent $\Gamma \vDash \tau$ requires that the liquid type be well-formed in the outer environment, which, together with the condition, enforced via *alpha*-renaming, that each variable is bound only once in the environment, is essential for ensuring the soundness of our system (Appendix A). An alternative would be to existentially quantify the bound variable in the let expression's type, in the style of Knowles and Flanagan [54]; we eschew this option for simplicity.

4. Placeholder Variables and α-Renaming We use the placeholder variables $\star$ in our refinement predicates instead of "hard-coded" program variables to make our type system robust to α-renaming. If $\mathbb{Q}$ is $\{x < \nu\}$, then $\emptyset \vdash_{\mathbb{Q}} \lambda x.x + 1 : x : \texttt{int} \rightarrow \{x < \nu\}$ a valid judgment, but $\emptyset \vdash_{\mathbb{Q}} \lambda y.y + 1 : y : \texttt{int} \rightarrow \{y < \nu\}$ is not, as $y < \nu$ is not in $\mathbb{Q}^\star$. If instead $\mathbb{Q}$ is $\{\star < \nu\}$, then $\mathbb{Q}^\star$ includes $\{x < \nu,\ y < \nu\}$ and so both of the above are valid judgments. In general, our type system is robust to renaming in the following sense: if $\Gamma \vdash_{\mathbb{Q}} e_1 : \sigma_1$ and e_1 is α-equivalent to e_2 and the free variables of $\mathbb{Q}$ are bound[1] in Γ, then for some σ_2 that is α-equivalent to σ_1, we have $\Gamma \vdash_{\mathbb{Q}} e_2 : \sigma_2$.

2.3 Liquid Type Inference

We now turn to the heart of our system: the algorithm Liquid, shown in Figure 2.6, that takes as input a type environment Γ, an expression e, and a finite set of logical qualifiers $\mathbb{Q}$ and determines whether e is well-typed over $\mathbb{Q}$, i.e., whether there exists some σ such that $\Gamma \vdash_{\mathbb{Q}} e : \sigma$. Our algorithm proceeds in three steps. First, we observe that the dependent type for any expression must be a refinement of its ML type, and so we invoke Hindley-Milner (HM) to infer the types of subexpressions, and use the ML types to generate *templates* representing the

[1] Recall that variables are bound at most once in any environment

unknown refinement types for the subexpressions (Section 2.3.1). Second, we use the syntax-directed liquid typing rules from Figure 2.4 to build a system of constraints that capture the subtyping relationships between the templates that must hold for a liquid type derivation to exist (Section 2.3.2). Third, we use Q to solve the constraints using a technique inspired by predicate abstraction (Section 2.3.3).

2.3.1 ML Types and Templates

Our type inference algorithm is based on the observation that the liquid type derivations are refinements of the ML type derivations, and hence the refinement types for all subexpressions are *refinements* of their ML types.

ML Type Inference Oracle Let HM be an ML type inference oracle, which takes an ML type environment Γ and an expression e and returns the ML type schema $\dot{\sigma}$ if and only if, using the classical ML type derivation rules [26], there exists a derivation $\Gamma \vdash e : \dot{\sigma}$. The liquid type derivation rules are refinements of the ML type derivation rules. That is, if $\Gamma \vdash_{\mathbb{Q}} e : \sigma$ then $\text{HM}(\text{Shape}(\Gamma), e) = \text{Shape}(\sigma)$. Moreover, we assume that the ML type derivation oracle has "inserted" suitable type generalization ($\Lambda\alpha.e$) and instantiation ($e[\tau]$) annotations. Thus, the problem of refinement type inference reduces to inferring appropriate refinements of the ML types.

Templates Let $\mathbb{K}$ be a set of *liquid type variables* κ, used to represent unknown type refinement predicates. A *template* T is a refinement type schema described via the grammar shown below, where some of the refinement predicates are replaced with liquid type variables with *pending substitutions*. A *template environment* is a map Γ from variables to templates.

$$\theta \quad ::= \quad \epsilon \mid [x \mapsto v]; \theta \qquad \text{(Pending Substitutions)}$$

$$T \quad ::= \quad S(\phi \cup \theta\kappa) \qquad \text{(Templates)}$$

Variables with Pending Substitutions A *sequence of pending substitutions* θ is defined using the grammar above. To understand the need for θ, consider rule LT-APP from Figure 2.4, which specifies that the type of a function application is obtained by substituting all occurrences of the formal argument x in the output type of v_1 with the actual expression v_2 passed in at the application. When generating the constraints, the output type of v_1 is unknown and is represented by a template containing liquid type variables. Suppose that the type of v_1 is $x : t \rightarrow \{\nu : t \mid \kappa\}$, where κ is a liquid type variable. In this case, we will assign the application $v_1\ v_2$ the type $\{\nu : t \mid \kappa[x \mapsto v_2]\}$, where $\kappa[x \mapsto v_2]$ is a variable with a *pending* substitution [53]. Note that substitution can be "pushed inside" type constructors, e.g., $\theta(\{\kappa_1\} \rightarrow \{\kappa_2\})$ is the same as $\{\theta\kappa_1\} \rightarrow \{\theta\kappa_2\}$ and so it suffices to apply the pending substitutions only to the root of the template.

$$
\begin{array}{l}
\text{ConsGen}(\Gamma,\ e) = \textbf{match}\ e\ \textbf{with} \\
\quad \mid\ x \rightarrow \textbf{if}\ \text{HM}(\text{Shape}(\Gamma), e) = t\ \textbf{then}\ \ (\{\nu :\ t\ \mid\ \nu = x\}, \emptyset)\ \textbf{else}\ \ (\Gamma(x), \emptyset) \\
\quad \mid\ \mathtt{c} \rightarrow (\text{ConstType}(\mathtt{c}), \emptyset) \\
\quad \mid\ v_1\ v_2 \rightarrow \\
\qquad \textbf{let}\ (x : T_x\ \rightarrow\ T, \mathbb{C}_1)\ =\ \text{ConsGen}(\Gamma,\ v_1) \\
\qquad \textbf{let}\ (T'_x, \mathbb{C}_2)\ =\ \text{ConsGen}(\Gamma,\ v_2) \\
\qquad (T[x \mapsto v_2], \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\Gamma \vdash T'_x <: T_x\}) \\
\quad \mid\ \lambda x.e \rightarrow \\
\qquad \textbf{let}\ x : T_x\ \rightarrow\ T\ =\ \text{Fresh}(\text{HM}(\text{Shape}(\Gamma), \lambda x.e)) \\
\qquad \textbf{let}\ (T', \mathbb{C})\ =\ \text{ConsGen}(\Gamma; x : T_x,\ e) \\
\qquad (x : T_x\ \rightarrow\ T, \mathbb{C} \cup \{\Gamma \vDash x : T_x\ \rightarrow\ T\} \cup \{\Gamma; x : T_x \vdash T' <: T\}) \\
\quad \mid\ \textbf{if}\ v\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rightarrow \\
\qquad \textbf{let}\ T\ =\ \text{Fresh}(\text{HM}(\text{Shape}(\Gamma), e)) \\
\qquad \textbf{let}\ (\_, \mathbb{C}_1)\ =\ \text{ConsGen}(\Gamma,\ v) \\
\qquad \textbf{let}\ (T_2, \mathbb{C}_2)\ =\ \text{ConsGen}(\Gamma; v,\ e_2) \\
\qquad \textbf{let}\ (T_3, \mathbb{C}_3)\ =\ \text{ConsGen}(\Gamma; \neg v,\ e_3) \\
\qquad (T, \mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \{\Gamma \vDash T\} \cup \\
\qquad\quad \{\Gamma; v \vdash T_2 <: T\} \cup \{\Gamma; \neg v \vdash T_3 <: T\}) \\
\quad \mid\ \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \rightarrow \\
\qquad \textbf{let}\ T\ =\ \text{Fresh}(\text{HM}(\text{Shape}(\Gamma), e)) \\
\qquad \textbf{let}\ (T_1, \mathbb{C}_1)\ =\ \text{ConsGen}(\Gamma,\ e_1) \\
\qquad \textbf{let}\ (T_2, \mathbb{C}_2)\ =\ \text{ConsGen}(\Gamma; x : T_1,\ e_2) \\
\qquad (T, \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\Gamma \vDash T\} \cup \{\Gamma; x : T_1 \vdash T_2 <: T\}) \\
\quad \mid\ \Lambda\alpha.e \rightarrow \\
\qquad \textbf{let}\ (T, \mathbb{C})\ =\ \text{ConsGen}(\Gamma,\ e) \\
\qquad (\Lambda\alpha.T, \mathbb{C}) \\
\quad \mid\ e[\dot{\tau}] \rightarrow \\
\qquad \textbf{let}\ T\ =\ \text{Fresh}(\dot{\tau}) \\
\qquad \textbf{let}\ (\Lambda\alpha.T', \mathbb{C})\ =\ \text{ConsGen}(\Gamma,\ e) \\
\qquad (T'[\alpha \mapsto T], \mathbb{C} \cup \{\Gamma \vDash T\})
\end{array}
$$

Figure 2.5: Constraint Generation from λ_L Programs

2.3.2 Constraint Generation

We now describe how our algorithm generates constraints over templates by traversing the expression in the syntax-directed manner of a type checker, generating fresh templates

$\text{Refine}(C,\ A) =$ **match** C **with**
$\quad \mid \Gamma \vDash \{\nu :\ t\ \mid\ \theta\kappa\} \rightarrow A[\kappa \mapsto \{q \mid q \in A(\kappa)$ **and** θq well-sorted in $\text{Shape}(\Gamma); \nu : t\}]$
$\quad \mid \Gamma \vdash \{\nu :\ t\ \mid\ p\} <: \{\nu :\ t\ \mid\ \theta\kappa\} \rightarrow A[\kappa \mapsto \{q \mid q \in A(\kappa)$ **and** $A(\Gamma) \vDash A(p) \Rightarrow \theta q\}]$
$\quad \mid \_ \rightarrow \bot$

$\text{Solve}(\mathbb{C},\ A) =$
$\quad$ **if** exists $C \in \mathbb{C}$ such that $A(C)$ is not valid **then** $\text{Solve}(\mathbb{C},\ \text{Refine}(C,\ A))$ **else** A

$\text{Liquid}(\Gamma, e, \mathbb{Q}) =$
$\quad$ **let** $(T, \mathbb{C})\ =\ \text{ConsGen}(\Gamma,\ e)$
$\quad$ **let** $A\ =\ \text{Solve}(\text{Split}(\mathbb{C}),\ \lambda\kappa.\, \text{QInst}(\Gamma, e, \mathbb{Q}))$
$\quad A(T)$

Figure 2.6: Liquid Type Inference Algorithm

for unknown types, constraints that capture the relationships between the types of various subexpressions, and well-formedness requirements. The generated constraints are such that they have a solution if and only if the expression has a valid liquid type derivation. Our inference algorithm uses two kinds of constraints over templates.

Well-formedness Constraints Constraints of the form $\Gamma \vDash T$, where Γ is template environment, and T is a template, ensure that the types inferred for each subexpression are over program variables that are in scope at that subexpression.

Subtyping Constraints Constraints of the form $\Gamma \vdash T_1 <: T_2$ where Γ is a template environment and T_1 and T_2 are two templates of the same shape, ensure that the types inferred for each subexpression can be combined using appropriate subsumption relationships to yield a valid type derivation.

Our constraint generation algorithm, ConsGen, shown in Figure 2.5, takes as input a template environment Γ and an expression e that we wish to infer the type of and returns as output a pair of a type template T, which corresponds to the unknown type of e, and a set of constraints $\mathbb{C}$. Intuitively, ConsGen mirrors the type derivation rules and generates constraints $\mathbb{C}$ which capture exactly the relationships that must hold between the templates of the subexpressions in order for e to have a valid type derivation over $\mathbb{Q}$. To understand how ConsGen works, notice that the expressions of λ_L can be split into two classes: those whose types are constructible from the environment and the types of subexpressions, and those whose types are not.

Expressions with Constructible Types The first class of expressions are variables, constants, function applications and polymorphic generalizations, whose types can be immediately con-

structed from the types of subexpressions or the environment. For such expressions, ConsGen recursively computes templates and constraints for the subexpressions and appropriately combines them to form the template and constraints for the expression.

As an example, consider ConsGen(Γ, v_1 v_2). First, ConsGen is called recursively to obtain the templates and constraints for the subexpressions v_1 and v_2. If a valid ML type derivation exists, then v_1 must be a function type with some formal x. The returned template is the result of pushing the pending substitution of x with the actual argument v_2 into the "leaves" of the template corresponding to the return type of v_1. The returned constraints are the union of the constraints for the subexpressions and a subtyping constraint ensuring that the type of the argument v_2 is a subtype of the argument type of v_1.

Expressions with Liquid Types The second class are expressions whose types cannot be derived as above, as the subsumption rule is required to perform some kind of "over-approximation" of their concrete semantics. These include λ-abstractions, if-then-else expressions, let-bindings, and polymorphic instantiations (which includes recursive functions). We use two observations to infer the types of these expressions. First, the shape of the refinement type is the same as the ML type of the expression. Second, from the liquid type restriction, we know that the refinement predicates for these expressions are conjunctions of logical qualifiers from $\mathbb{Q}^\star$ (see rules LT-LET, LT-FUN, LT-IF, and LT-INST of Figure 2.4). Thus, to infer the types of these expressions, we invoke HM to determine the ML type of the expression, use Fresh to generate a template with the same shape as the ML type but with fresh liquid type variables representing the unknown refinements, and generate subtyping constraints which are satisfied if and only if there is an assignment to the predicate variables in the type template that makes the expression well-typed.

As an example, consider ConsGen(Γ, **if** v **then** e_1 **else** e_2). First, a fresh template is generated using the ML type of the expression. Next, ConsGen recursively generates templates and constraints for the `then` and `else` subexpressions. Note that, for the `then` subexpression, the environment is extended with v, while, for the `else` subexpression, the environment is extended with $\neg v$, as in the type derivation rule LT-IF from Figure 2.4. The fresh template is returned as the template for the whole expression. The constraints returned are the union of those for the subexpressions, a well-formedness constraint for the whole expression's template, and subtyping constraints forcing the templates for the `then` and `else` subexpressions to be subtypes of the whole expression's template.

Example Constraints The well-formedness constraint $\emptyset \vDash \texttt{x} : \{\kappa_{\texttt{x}}\} \rightarrow \texttt{y} : \{\kappa_{\texttt{y}}\} \rightarrow \kappa_1$ is generated for the fresh template for `max` from Figure 2.1. The constraint ensures that the inferred type for `max` only contains program variables that are in scope at the point where `max` is bound. The `if` expression that is the body of `max` is an expression with liquid type. For this expression, a

fresh template $\kappa_{1'}$ is generated, and the subtyping constraints

$$\mathtt{x} : \{\kappa_{\mathtt{x}}\}; \mathtt{y} : \{\kappa_{\mathtt{y}}\}; \mathtt{x} > \mathtt{y} \vdash \{\nu = \mathtt{x}\} <: \{\kappa_{1'}\}$$
$$\mathtt{x} : \{\kappa_{\mathtt{x}}\}; \mathtt{y} : \{\kappa_{\mathtt{y}}\}; \neg(\mathtt{x} > \mathtt{y}) \vdash \{\nu = \mathtt{y}\} <: \{\kappa_{1'}\}$$
$$\mathtt{x} : \{\kappa_{\mathtt{x}}\}; \mathtt{y} : \{\kappa_{\mathtt{y}}\} \vdash \{\kappa_{1'}\} <: \{\kappa_1\}$$

are generated, capturing the relationships between the `then` and the `if` expression, the `else` and the `if` expression, and the `if` and the output expression, respectively. (Constraints 1.1 and 1.2 are the above constraints, simplified for exposition.) The recursive application `sum (k-1)` from Figure 2.1 is an expression with a constructible type. For this expression the subtyping constraint 2.2 is generated, forcing the actual to be a subtype of the formal. The output of the application, i.e., the output type κ_2 of `sum`, with the pending substitution of the formal `k` with the actual $(\mathtt{k} - 1)$ is shown bound to `s` in constraint 2.3.

2.3.3 Constraint Solving

Next, we describe our two-step algorithm for solving the constraints, i.e., assigning liquid refinement predicates to all variables κ such that all constraints are satisfied. In the first step, the algorithm uses the well-formedness and subtyping rules to split the complex constraints, which may contain function types, into simple constraints over refinement predicate variables with pending substitutions. In the second step, the algorithm computes a solution to the simplified constraints by beginning with a trivial assignment, in which each refinement predicate variable is assigned the conjunction of all logical qualifiers, and iteratively weakening the solution until it find the least fixpoint solution for all the simplified constraints or determines that the constraints have no solution. In the following, we formalize the notion of a solution and then describe the two-step algorithm that computes solutions.

Satisfying Liquid Assignments A *Liquid Assignment over* $\mathbb{Q}$ is a map A from refinement predicate variables κ to sets of qualifiers from $\mathbb{Q}^\star$. Assignments can be lifted to maps from templates T to refinement type schemas $A(T)$ and template environments Γ to environments $A(\Gamma)$, by substituting each refinement predicate variable κ with $\bigwedge A(\kappa)$ and then applying the pending substitutions. The Liquid Assignment A *satisfies* a constraint C if $A(C)$ is valid. That is, A satisfies a well-formedness constraint $\Gamma \vDash T$ if $A(\Gamma) \vDash A(T)$, and a subtyping constraint $\Gamma \vdash T_1 <: T_2$ if $A(\Gamma) \vdash A(T_1) <: A(T_2)$. Liquid Assignment A is a *solution* for a set of constraints $\mathbb{C}$ if it satisfies every constraint in $\mathbb{C}$.

Step 1: Splitting into Simple Constraints. First, the algorithm calls Split, which uses the rules for well-formedness and subtyping (Figure 2.4) to convert all the constraints over complex types (e.g., function types) into simple constraints over base types. An assignment is a solution for $\mathbb{C}$ if and only if it is a solution for $\text{Split}(\mathbb{C})$.

Example: Splitting The well-formedness constraint $\emptyset \vDash \texttt{x} : \{\kappa_{\texttt{x}}\} \rightarrow \texttt{y} : \{\kappa_{\texttt{y}}\} \rightarrow \{\kappa_1\}$ splits into the three simple constraints: $\emptyset \vDash \{\kappa_{\texttt{x}}\}$, $\texttt{x} : \{\kappa_{\texttt{x}}\} \vDash \{\kappa_{\texttt{y}}\}$ and $\texttt{x} : \{\kappa_{\texttt{x}}\}; \texttt{y} : \{\kappa_{\texttt{y}}\} \vDash \{\kappa_1\}$, which ensure that: the parameter `x` must have a refinement over only constants and the value variable ν, as the first constraint's environment is empty; the parameter `y` must have a refinement over only `x` and ν; and the output type's refinement can refer to both parameters `x`, `y` and the value variable. The function subtyping constraint generated by the call `foldn (len a) 0 am` (constraint 4.4) splits into the simple subtyping constraints 4.6, 4.7, and 4.8. Notice how substitution and contravariance combine to cause the flow of the bounds information into input parameter $\{\kappa_1\}$ (constraint 4.6) thus allowing the system to statically check the array access.

Step 2: Iterative Weakening Due to the well-formedness constraints, any solution over $\mathbb{Q}$ must map the liquid type variables to sets of qualifiers whose free variables are either the value variable ν or the variables in the input environment Γ, written $\text{Vars}(\Gamma)$, or the variables in the input expression e, written $\text{Vars}(e)$. That is, any solution maps the liquid variables to a set of qualifiers contained in $\text{QInst}(\Gamma, e, \mathbb{Q})$ which is defined as

$$\{q \mid q \in \mathbb{Q}^\star \text{ and } \text{FV}(q) \subseteq \{\nu\} \cup \text{Vars}(\Gamma) \cup \text{Vars}(e)\}$$

where $\text{Vars}(\Gamma)$ and $\text{Vars}(e)$ are the set of variables in Γ and e, respectively. Notice that if $\mathbb{Q}$ is finite, then $\text{QInst}(\Gamma, e, \mathbb{Q})$ is also finite, as the placeholder variables can only be instantiated with the finitely many variables from Γ and e. Thus, to solve the constraints, we call the procedure Solve, shown in Figure 2.6, with the split constraints and a trivial initial assignment that maps each liquid type variable to $\text{QInst}(\Gamma, e, \mathbb{Q})$.

Solve repeatedly picks a constraint that is not satisfied by the current assignment and calls Refine to remove the qualifiers that prevent the constraint from being satisfied. For unsatisfied constraints of the form $\Gamma \vDash \{\nu : t \mid \theta\kappa\}$, Refine removes from the assignment for κ all the qualifiers q such that θq (the result of applying the pending substitutions θ to q) would not be well-sorted in the environment $\text{Shape}(\Gamma); \nu : t$. For unsatisfied constraints of the form $\Gamma \vdash \{\nu : t \mid p\} <: \{\nu : t \mid \theta\kappa\}$, where p is either a refinement predicate or a refinement predicate variable with pending substitutions, Refine removes from the assignment for κ all the logical qualifiers q such that the implication $[\![A(\Gamma)]\!] \wedge A(p) \Rightarrow \theta q$ is not valid in EUFA. For unsatisfied constraints of the form $\Gamma \vdash \{\nu : t \mid p\} <: \{\nu : t \mid \phi\}$, Refine, and therefore Solve, returns $\bot$, indicating that the constraints have no solution, as there is no way to weaken p to satisfy the constraint.

Correctness of Solve For two assignments A and A', we say that $A \leq A'$ if, for all κ, the set of logical qualifiers $A(\kappa)$ *contains* the set of logical qualifiers $A'(\kappa)$. We can prove that if a set of constraints has a solution over $\mathbb{Q}$ then it has a unique minimum solution with respect to $\leq$. Intuitively, we invoke Solve with the least possible assignment that maps each liquid variable to all the possible qualifiers. Solve then uses Refine to iteratively weaken the assignment until

the unique minimum solution is found. The correctness of Solve follows from the following invariant about the iterative weakening: if $A^\star$ is the minimum solution for the constraints, then in each iteration, the current assignment A satisfies $A \leq A^\star$. Thus, if Solve returns a solution, then it must be the minimum solution for $\mathbb{C}$ over $\mathbb{Q}$. If at some point a constraint $\Gamma \vdash \{\nu : t \mid p\} <: \{\nu : t \mid \phi\}$ is unsatisfied, subsequent weakening cannot make it satisfied. Thus, if Solve returns $\bot$, then $\mathbb{C}$ has no solution over $\mathbb{Q}$.

By combining the steps of constraint generation, splitting and solving, we obtain our dependent type inference algorithm, Liquid, shown in Figure 2.6. The algorithm takes as input an environment Γ, an expression e and a finite set of logical qualifiers $\mathbb{Q}$, and determines whether there exists a valid liquid type derivation over $\mathbb{Q}$ for e in the environment Γ. The correctness properties of Liquid are stated in the theorem below, whose proof is in Appendix A. From Theorems 1 and 2, we conclude that, if $\text{Liquid}(\emptyset, e, \mathbb{Q}) = \sigma$, then every primitive operation invoked during the evaluation of e succeeds.

Theorem 2. *Liquid Type Inference*

1. $\text{Liquid}(\Gamma, e, \mathbb{Q})$ *terminates,*
2. *If* $\text{Liquid}(\Gamma, e, \mathbb{Q}) = \sigma$ *then* $\Gamma \vdash_{\mathbb{Q}} e : \sigma$*, and,*
3. *If* $\text{Liquid}(\Gamma, e, \mathbb{Q}) = \bot$ *then there is no* σ *such that* $\Gamma \vdash_{\mathbb{Q}} e : \sigma$*.*

Running Time Most of the time taken by Liquid is spent in Solve, which asymptotically dominates the time taken to generate constraints. Solve returns the same output regardless of the order in which the constraints are processed; for efficiency, we implement Solve in two phases. First, Solve makes a (linear) pass that solves the well-formedness constraints, thus rapidly pruning away irrelevant qualifiers. Second, Solve uses a standard worklist-based algorithm that solves the subtyping constraints. The time taken in the first phase is asymptotically dominated by the time taken in the second. Let Q be the maximum number of qualifiers that any refinement predicate variable is mapped to after the first well-formedness pass, V be the number of variables in the program e that have a base type, and D be the size of the ML type derivation for e in the environment Γ. A constraint is sent to Refine only when the antecedent of its implication changes, i.e., at most $V \times Q$ times. There are at most $O(D)$ constraints and so Refine is called at most $O(D \times V \times Q)$ times. Each call to Refine makes at most Q calls to the theorem prover. Thus, in all the running time of Liquid is $O(D \times V \times Q^2)$ assuming each theorem prover call takes unit time. Of course, D can be exponential in the program size (but tends to be linear in practice), and the size of each theorem prover query is $O(V \times Q)$. Though validity checking in EUFA is NP-Hard, several solvers for this theory exist which are very efficient for the simple queries that arise in our context [28, 33].

2.3.4 Features of Liquid Type Inference

We now discuss some features of the inference algorithm.

1. Type Variables and Polymorphism There are two kinds of type variables used during inference: ML type variables α obtained from the ML types returned by HM, and refinement predicate variables κ introduced during liquid constraint generation to stand for unknown liquid refinement predicates. Our system is monomorphic in the refinement predicate variables. Polymorphism only enters via the ML type variables as fresh liquid type variables are created at each point where an ML type variable α is instantiated with a monomorphic ML type.

2. Whole Program Analysis and Non-General Types Due to the above, the types we obtain for function inputs are the strongest liquid supertype of all the arguments passed into the function. This is in contrast with ML type inference, which infers the most general type of the function independent of how the function is used. For example, consider the function `neg` defined as `fun x -> (-x)`, and suppose that $\mathbb{Q} = \{0 \leq \nu, 0 \geq \nu\}$. In a program comprising only the above function, i.e., where the function is never passed arguments, our algorithm infers $\texttt{neg} : \{0 \leq \nu \wedge 0 \geq \nu\} \rightarrow \{0 \leq \nu \wedge 0 \geq \nu\}$ which is useless but sound. If `neg` is only called with (provably) non-negative arguments, the algorithm infers the type $\texttt{neg} : \{0 \leq \nu\} \rightarrow \{0 \geq \nu\}$, while if `neg` is only called with (provably) non-positive arguments, the algorithm infers the type $\texttt{neg} : \{0 \geq \nu\} \rightarrow \{0 \leq \nu\}$. If `neg` is called with arbitrary arguments, the algorithm infers $\texttt{neg} : \texttt{int} \rightarrow \texttt{int}$ and not a more general intersection of function types. We found this design choice greatly simplified the inference procedure by avoiding expensive "case splits" on all possible inputs [41] while still allowing us to prove the safety of challenging benchmarks. Moreover, we can represent the intersection type in our system as: $\texttt{x} : \texttt{int} \rightarrow \{(0 \leq \texttt{x} \Rightarrow 0 \geq \nu) \wedge (0 \geq \texttt{x} \Rightarrow 0 \leq \nu)\}$, and so, if needed, we can recover the precision of intersection types by using qualifiers containing implications.

3. A-Normalization Recall the `sum` example from section 2.1. Our system as described would fail to infer that the output type of

```
let rec sum k = if k < 0 then 0 else (s + sum (k-1))
```

was non-negative, as it cannot use the fact that `sum (k-1)` is non-negative when inferring the type of the `else` expression. This is solved by *A-Normalizing*[38] the program so that intermediate subexpressions are bound to temporary variables, thus allowing us to use information about types of intermediate expressions, as in the original `sum` implementation.

2.4 Implementation and Evaluation

To validate the utility of the liquid types refinement type inference technique as applied to high-level, functional programming languages, we have built DSOLVE, which infers liquid types for OCaml programs. While refinement types can be used to statically prove a variety of properties, as shown by Kawaguchi et al. [52], Bengtson et al. [7], and Dunfield [31], among others, in our evaluation we focus on the canonical problem of proving the safety of array accesses. We use a diverse set of challenging benchmarks that were previously annotated in the DML project [83] to demonstrate that DSOLVE, together with a simple set of array bounds checking qualifiers, can prove safety completely automatically for many programs. For the few programs where these bounds checking qualifiers are insufficient, the programmer typically only needs to specify one or two extra qualifiers. Even in these rare cases, the refinement types DSOLVE infers using only the bounds checking qualifiers help the programmer to rapidly identify the relevant extra qualifiers. We show that, over all the benchmarks, DSOLVE reduces the manual annotation required to prove safety from 17%, by number of lines, to under 1%. Finally, we describe a case study where DSOLVE was able to pinpoint an error in an open source OCaml bit vector library implementation, in a function that contained an explicit, but insufficient, safety check.

2.4.1 DSOLVE: Liquid Types for OCaml

We begin with a description of our implementation of liquid type inference in the tool DSOLVE, which infers liquid types for OCaml programs.

Architecture DSOLVE is built on top of OCaml: DSOLVE uses the OCaml parser and type inference engine to implement the oracle HM, and uses OCaml's facilities for outputting type annotations that can be viewed by the user using an external tool. Type inference in DSOLVE is divided into the following three phases: First, the OCaml compiler's parser and type checker are used to translate the input program to a typed AST; this phase also parses the module's refinement type specification. Second, the typed AST is traversed to generate a set of subtyping constraints over templates that represent the potentially-unknown refinement types of the program expressions. Third, the constraints are solved using predicate abstraction over a finite set of predicates generated from user-provided logical qualifiers. This pass uses the Z3 SMT solver [28] to discharge logical implications corresponding to the subtyping constraints. If the constraints can be satisfied, the program is deemed safe. Otherwise, DSOLVE reports a type error and the lines in the original source program that yielded the unsatisfiable constraints.

DSOLVE is conservative: if an error is reported, it may be because the program is unsafe, or because the set of qualifiers provided was insufficient, or because the invariants needed to prove safety cannot be expressed within our refinement type system.

Input DSOLVE takes as input a source (`.ml`) file containing an OCaml program, an interface (`.mlq`) file containing a refinement type specification for the interface functions of the `.ml` file, and a qualifier (`.hquals`) file containing a set of logical qualifiers. DSOLVE combines the qualifiers from the `.hquals` file with some scraped from the specification `.mlq` file and a standard qualifier library to obtain the set of logical qualifiers used to infer liquid types.

Output DSOLVE produces as output a refinement type for each program expression in a standard OCaml type annotation (`.annot`) file. The user can view the inferred refinement types using standard tools like `Emacs`, `Vim`, and `Caml2HTML`. If all the constraints are satisfied, the program is reported as safe. Otherwise, DSOLVE outputs warnings indicating the potentially unsafe expressions in the program.

Modular Checking DSOLVE verifies one module at a time. If a module depends on another module, it can be checked against that module's `.mlq` file; the other module's source code is not required.

Abstract Modules It is possible to create a `.mlq` file which defines types, axioms (background predicates), and uninterpreted functions, without a corresponding `.ml` file. Such "abstract modules" allow the user to extend DSOLVE with reasoning about mathematical structures which do not appear directly in the program. For example, an abstract module `Set.mlq` might contain a type which represents a polymorphic set collection, along with an appropriate refined interface and axioms which build a set theory. This set theory can be used in another module's type refinements; for example, it may be used in a sorting module to verify that the sets of elements in the input and output lists of a sorting function are equal.

2.4.2 Benchmark Results

To show the real-world applicability of the liquid types approach, we applied DSolve to a number of benchmarks from the DML project [82] (ported to OCaml) that were previously annotated with dependent types with the goal of statically proving the safety of array accesses [83]. The benchmarks are listed in the first column of Table 2.1. The second column indicates the size of the benchmark (ignoring comments and whitespace). The benchmarks include OCaml implementations of the Simplex algorithm for linear programming (`simplex`), the fast Fourier transform (`fft`), Gaussian elimination (`gauss`), matrix multiplication (`matmult`), binary search in a sorted array (`bsearch`), computing the dot product of two vectors (`dotprod`), insertion sort (`isort`), the n-queens problem (`queen`), the Towers of Hanoi problem (`tower`), a fast byte copy routine (`bcopy`), and heap sort (`heapsort`). The above include all DML array benchmarks except `quicksort`, whose invariants remain unclear to us. In addition, we ran DSOLVE on a simplified Quicksort routine from OCaml's Sort module (`qsort-o`), a version ported from the

Table 2.1: **Experimental Results: LOC** is the number of lines of program text (without annotation) after removing whitespace and comments from the code. **DML** is the number of lines of manual annotation required in the DML versions of the benchmarks. **DSOLVE** is the amount of manual annotation required by DSOLVE, i.e., number of lines of qualifiers not in $\mathbb{Q}_{BC}$. **Time** is the time taken by DSOLVE to infer refinement types.

Program	LOC	DML (LOC)	DSOLVE (LOC)	Time (s)
`dotprod`	7	3 (30%)	0%	0.31
`bcopy`	8	3 (27%)	0%	0.15
`bsearch`	24	3 (11%)	0%	0.46
`queen`	30	7 (19%)	0%	0.70
`isort`	33	12 (27%)	0%	0.88
`tower`	36	8 (18%)	1 (2%)	3.33
`matmult`	43	10 (19%)	0%	1.79
`heapsort`	84	11 (12%)	0%	0.53
`fft`	107	13 (11%)	1 (1%)	9.13
`simplex`	118	33 (22%)	0%	7.73
`gauss`	142	22 (13%)	1 (1%)	3.17
TOTAL	633	125 (17%)	3(1%)	
`qsort-o`	62		0 (0%)	1.89
`qsort-d`	112		5 (5%)	18.28
`bitv`	426		65 (15%)	63.11

DML benchmark (`qsort-d`) where one optimization is removed, and BITV, an open source bit vector library (`bitv`).

Array Bounds Checking Qualifiers To automate static array bounds checking with DSOLVE, we observe that the safety of array accesses typically depends on the *relative ordering* of integer expressions. Thus, to statically prove the safety of array accesses, we use the mechanically-generated set of array bounds checking qualifiers $\mathbb{Q}_{BC}$ defined as

$$\mathbb{Q}_{BC} \triangleq \{\nu \bowtie X \mid \bowtie \in \{<, \leq, =, \neq, >, \geq\} \wedge X \in \{0, \star, \texttt{len}\ \star\}\}.$$

Next, we show experimental results demonstrating that liquid type inference over $\mathbb{Q}_{BC}$ suffices to prove the safety of most array accesses. Even when DSOLVE needs extra qualifiers, the types inferred using $\mathbb{Q}_{BC}$ help the programmer quickly identify the relevant extra qualifiers.

Array Bounds Checking Results As shown in column **DSOLVE** of Table 2.1, DSOLVE needs no manual annotations for most programs; that is, the qualifiers $\mathbb{Q}_{BC}$ suffice to automatically prove the safety of all array accesses. For some of the examples, e.g., `tower`, we do need to provide extra qualifiers. However, even in this case, the annotation burden is typically just a few qualifiers. For example, in `tower`, we require a qualifier which is analogous to $\nu = \texttt{n} - \texttt{h1} - \texttt{h2}$, which describes the height of the "third" tower, capturing the invariant that the height is the total number of rings `n` minus the rings in the first two towers. Similarly, in `bitv`, one qualifier states the key invariant

relating a bit vector's length to the length the underlying data structure used to store the bit vector, an array of integers. The time for inference is robust to the number of qualifiers, as most qualifiers are pruned away by the well-formedness constraints. In our prototype implementation, the time taken for inference is reasonable even for non-trivial benchmarks like `simplex`, `fft` and `gauss`.

Case Study: Bit Vectors We applied DSOLVE to verify the open source BITV bit vector library (version `0.6`). A bit vector in BITV consists of a record with two fields: `length`, the number of bits stored, and `bits`, the actual data, stored within an array of integers. If b is the number of bits stored per array element, `length` and `bits` are related by

$$(\texttt{len bits} - 1) \cdot b < \texttt{length} \leq (\texttt{len bits}) \cdot b.$$

The executed code, and hence refinement types, are different for 32- and 64-bit machines. Thus, to verify the code using our conjunctive types, we fixed the word size to 32 bits. We were able to verify the array safety of 58 of BITV's 65 bit vector creation, manipulation, and iteration functions, which contain a total of 30 array access operations.

There are three kinds of manual annotation needed for verification: *extra qualifiers* (14 lines), *trusted assumptions* (8 lines), and *interface specifications* (43 lines). The trusted assumptions (which are akin to dynamic checks) are needed due to current limitations of our system. These include the conservative way in which modular arithmetic is embedded into EUFA, the lack of refinements for type variables and recursive datatypes, and the conservative handling of control flow. The interface specifications are needed because BITV is a library, i.e., an open program. Thus, for verification, we need to specify that the API functions are called with valid input vectors that satisfy invariants like the one described above. The interface specifications, by far the largest category of annotations, are unavoidable. The extra qualifiers and expressiveness limitations are directions for future work.

DSOLVE was able to locate a serious bounds checking error in BITV. The error occurs in BITV's `blit` function, which copies `c` bits from `v1`, starting at bit `offset1`, to `v2`, starting at bit `offset2`. This function first checks that the arguments passed are safe, and then calls a fast but unsafe internal function, `unsafe_blit`:

```
let blit v1 offset1 v2 offset2 c =
  if c < 0 || offset1 < 0 || offset1 + c > v1.length
              offset2 < 0 || offset2 + c > v2.length
  then invalid_arg "Bitv.blit";
  unsafe_blit v1.bits offset1 v2.bits offset2 c
```

`unsafe_blit` immediately accesses the bit at `offset1` in `v1`, regardless of the value of `c`. When the parameters are such that: `offset1` $=$ `v1.length` and `v1.length` mod $b = 0$ and `c` $= 0$,

`unsafe_blit` attempts to access the bit at index `v1.length`, which must be located in

```
v1.bits[v1.length / b];
```

but this is

```
v1.bits[len v1.bits],
```

which is out of bounds and can cause a crash, as we verified with a simple input. The problem is that `blit` does not verify that the starting offset is within the bounds of the bit vectors. This is fixed by adding the test `offset1 >= v1.length` (and `offset2 >= v2.length` for similar reasons). DSOLVE successfully type checks the corrected version.

Acknowledgements

This chapter contains material adapted from the following publications:

Patrick Rondon, Ming Kawaguchi, Ranjit Jhala. "Liquid Types", *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 159–169, 2008.

Ming Kawaguchi, Patrick Rondon, Ranjit Jhala. "DSolve: Safety Verification via Liquid Types", *Proceedings of Computer Aided Verification 2010 (CAV)*, pages 123–126, 2010.

Chapter 3

Low-Level Liquid Types

Static verification is a crucial last line of defense at the lowest levels of the software stack, as at those levels we cannot fall back on dynamic mechanisms to protect against bugs, crashes, or malicious attacks. Recent years have seen significant progress on automatic static verification tools for systems software. These tools employ abstract interpretation [9, 50] or software model checking [5, 46, 12, 85] to infer path-sensitive invariants over program variables like status flags and counters and thereby verify control-sensitive safety properties. Unfortunately, these approaches have been proven insufficient for verifying data-sensitive properties of values stored in lists, trees, etc., as this requires the precise inference of invariants of data values stored within unbounded collections of heap-allocated cells.

In the previous chapter, we introduced liquid types, a refinement type system for OCaml that marries the ability of OCaml types to infer coarse invariants of data structures (and higher-order functions) with the ability of predicate abstraction and SMT solvers to infer path-sensitive invariants of individual variables. We demonstrated that this symbiotic combination enables the highly automated verification of complex data-sensitive properties of high-level, functional programs. Unfortunately, the very nature of low-level, imperative code, typically written in C, makes the translation of the liquid types program verification strategy to the setting of systems software verification extremely challenging.

Lack of Types First, due to the presence of casts and pointer arithmetic, low-level systems code is essentially untyped: C's type system is designed only to allow the compiler to determine the number of bytes that should be read or written by each instruction, and hence, unlike the type systems of higher-level languages, C's types provide no invariants about data values.

Mutation Second, mutation makes the very notion of type refinement problematic. The key idea in refinement types is to adorn the basic underlying types with refinement predicates over program variables. For example, in an OCaml program, the refinement type $\{\nu : \texttt{int} \mid \texttt{x} < \nu\}$

describes an integer that is greater than the program variable x. However, this type is meaningless if the value of x can change over time.

Unbounded Collections Third, even if we could meaningfully track mutation, we cannot always uniquely identify the object being mutated. In particular, the presence of unbounded collections means that we must represent many elements of a collection by a single type. This makes it impossible to strongly update the type of an element in the collection, creating a major loss of precision in the presence of the temporary invariant violations common in low-level programs.

In this chapter, we develop *low-level liquid types*, a static refinement type system for C that enables the precise verification and inference of data-sensitive properties of low-level software. Low-level liquid types tackles the above challenges via a three-tiered design.

First, low-level liquid types is founded on a new physical type system that classifies values and heaps. A value is either a *datum* of a given size, e.g., a 4-byte integer or a 1-byte character, or a *reference* corresponding to a pair of a heap *location* and an *offset* within the location. Intuitively, an offset corresponds to a field (respectively, cell) of the structure (respectively, array) resident at the location. A heap is a map from locations to a sequence of offset-value bindings that define the contents of the given location. By precisely tracking arithmetic on offsets, physical types provide coarse invariants about the basic shapes of data values.

Second, each physical type is refined with a predicate that captures precise properties of the values defined by the type. Low-level liquid types makes a clear separation between immutable state, which is tracked using a traditional type environment, and mutable state, which is tracked in a flow-sensitive heap. We ensure soundness by restricting the refinements to pure predicates that refer only to immutable values. Of course, in C all entities are mutable. We recover precision for stack-allocated variables by first carrying out an SSA renaming, which creates different (immutable) versions of each variable at different program points.

Third, we recover precision for heap-allocated locations by using the physical type information to strongly update the types of the heap's contents on writes through pointers. Since such strong updates are unsound when several physical heap locations are represented by a single type, low-level liquid types distinguishes between *abstract* locations which summarize a collection of physical memory locations and *concrete* locations which describe exactly one physical memory location. Low-level liquid types enables strong updates by enforcing the requirement that all pointer reads and writes are to concrete locations, and by employing two mechanisms to account for aliasing, which we explain via a version control metaphor. The *unfold* operation "checks out" a concrete reference to a particular location from the set described by an abstract location, obtaining a pointer to a single run-time location whose type may be strongly updated soundly. The dual to the unfold operation, *fold*, "commits" the changes made to the particular location back into the abstract location after ensuring that the particular location satisfies the invariants of the abstract location. Together, the automatically-inserted fold and

unfold annotations ensure that the invariants for an abstract location soundly apply to *all* the elements that correspond to that location, while simultaneously allowing strong updates. This is crucial, as strong updates are essential for both establishing invariants in spite of field-at-a-time, incremental object initialization and reasoning in spite of the temporary violations of the invariants that are ubiquitous in low-level code.

To demonstrate the utility of low-level liquid types, we have implemented our type system in CSOLVE, a prototype static verifier for C. CSOLVE takes as input a C program and a set of logical predicate templates and returns as output the inferred refinement types of functions, local variables, and heap contents, along with a report of any type errors that occurred. Through a set of challenging case studies, we show how the combination of types and predicate abstraction enables the precise, path-sensitive verification and inference of control-sensitive properties of individual variables *and* data-sensitive properties of aggregate structures manipulated in low-level programs.

3.1 Overview

We begin with a high-level overview of low-level liquid types, and then, via a sequence of examples, we illustrate how they enable the precise static verification and inference of program invariants in the presence of challenging low-level programming constructs, including pointer arithmetic, memory allocation, temporary invariant violations, aliasing, and unbounded data structures.

3.1.1 Physical and Refinement Types and Heaps

We begin our overview of the low-level liquid types system by explaining the physical base types that our system uses to represent data in low-level programs. We then explain how these types are combined into heap types that represent the contents of the mutable run-time store. Finally, we explain how refinement and liquid types and heaps are formed.

Physical Types Our system is based on a physical type system for C where every program variable is either a *basic data value* of some size, e.g., a 4-byte integer, denoted by `int`, or a *reference* comprising a *location* and an *index* within the location, denoted by $\texttt{ref}(\ell, i)$, where ℓ is the location and i the index within the location. For the purposes of this section, an index is either a natural number n, which is a singleton offset used to model pointers to specific fields of a structure, or of the form n^{+m}, which is a sequence of offsets $\{n + km\}_{k=0}^{\infty}$ used to model pointers into an array of items, each of size m, that starts at offset n. Thus, $\texttt{ref}(\ell, 4)$ is a (possibly null) pointer that refers to a location ℓ at (field) offset 4, while $\texttt{ref}(\ell, 0^{+4})$ is a (possibly null) pointer that refers to a location within an array of 4-byte integers.

Heaps To ensure the soundness of types in the presence of mutation, our representation of the program state is partitioned into an *environment*, which is a standard sequence of type bindings for immutable variables, and a *heap*, which is a mapping from locations ℓ to a set of index-type pairs that describe the contents of the location, called a *block*. For example, the heap

$$\ell^1 \mapsto 0 : \texttt{int},\ 4 : \texttt{int}$$
$$\ell^2 \mapsto 0^{+1} : \texttt{char}$$

has two locations. The first, ℓ^1, contains a structure with two integer fields, one at offset 0 and one at offset 4. The second, ℓ^2, contains an array of one-byte characters (denoted `char`).

Refinement Types and Heaps In low-level liquid types, as in the liquid types system of the previous chapter, program invariants are captured via *refinement types*, denoted by $\{\nu : t \mid \phi\}$, where t is the physical type being refined, ν is a special *value variable* that denotes the value being described, and ϕ is the *refinement predicate*, a logical predicate containing the value variable. A *refinement heap* is a heap where each location is mapped to a sequence of offset-refinement-type pairs. For example, $\ell_1 \mapsto 0 : \{\nu : \texttt{int} \mid 0 \leq \nu\}$ is a heap with a location ℓ_1 which contains a non-negative integer at offset 0.

Liquid Types The definitions of logical qualifiers and liquid types were given in section 2.1; we review them here for convenience. A *logical qualifier* is a Boolean-valued expression over the program variables, the value variable ν, and a placeholder variable $\star$. We say that a qualifier q matches the qualifier q' if replacing some subset of the free variables in q with $\star$ yields q'. For example, the qualifier $\nu \leq x + y$ matches the qualifier $\nu \leq \star + \star$. We write $\mathbb{Q}^\star$ for the set of all qualifiers not containing $\star$ that match some qualifier in $\mathbb{Q}$. In the rest of this section, let $\mathbb{Q}$ be the set

$$\{0 \leq \nu, \nu = \star + \star, \nu = \text{BBegin}(\nu), \text{BBegin}(\nu) = \text{BBegin}(\star), \text{BEnd}(\nu) = \text{BBegin}(\nu) + \star\}.$$

The terms $\text{BBegin}(\cdot)$ and $\text{BEnd}(\cdot)$ are uninterpreted function applications denoting the start and end addresses of memory blocks; we will explain these shortly. A *liquid type over* $\mathbb{Q}$ (abbreviated to just liquid type) is a refinement type where the refinement predicates are conjunctions of qualifiers from $\mathbb{Q}^\star$. Our system enables inference by requiring that the certain entities, e.g., loop-modified variables, functions, and blocks in aggregate structures, have liquid types.

3.1.2 Low-Level Liquid Types By Example

In the following, we give a high-level, example-driven overview of how low-level liquid types is used to infer properties of low-level programs that manipulate stack-allocated scalar data, in-heap array data, and unbounded heap data structures.

```
char *make_string (int n) {
   char *res;
   char *str;

1: if (n < 0) return NULL;

2: res = (char *) malloc (n * sizeof (char));
3: str = res;
4: for (int i = 0; i < n; i++) {
5:   *str++ = '\0';
   }

6: return res;
}
```

Figure 3.1: Example: `make_string`

Local Invariants

We begin by showing how our system uses local refinement types for individual program variables to verify the safety of the pointer dereferences in the `make_string` function shown in Figure 3.1. The function takes an integer parameter `n`, allocates a new block of memory of size `n`, iterates over the block using `str` to initialize it, and returns a reference, `res`, to the start of the block.

Physical Types First, we describe the physical types computed for each variable. The function calls `malloc` to create a new heap location ℓ^1 and returns a pointer to the location with offset 0. Thus, `res` gets the physical type $\texttt{ref}(\ell^1, 0)$. `str` is initialized with `res`, but is updated inside the loop with an increment of 1. Hence, `res` gets assigned the physical type $\texttt{ref}(\ell^1, 0^{+1})$. The loop index `i` gets the physical type `int`.

Pointer Allocation and Arithmetic To specify when it is safe to dereference a pointer, we refine the output type of `malloc` so that it contains information about the size of the allocated block. In particular, in our system `malloc` returns a value of type

$$\{\nu : \texttt{ref}(\ell, 0) \mid \text{BLen}(\nu, \texttt{n})\}$$

where `n` is the size argument passed to `malloc` and BLen is the following *block length predicate*:

$$\text{BLen}(\nu, n) \triangleq \text{BBegin}(\nu) = \nu \wedge \text{BEnd}(\nu) = \nu +_p n$$

The refinement states that the return value is equal to the start of the allocated region in which it points, denoted $\text{BBegin}(\nu)$, and that the end of the allocated region, denoted $\text{BEnd}(\nu)$, is `n` bytes from the beginning; the operator $+_p$ represents pointer arithmetic in our refinement logic. We

```
typedef struct {
  int   len;
  char *str;
} string;

string *new_string (int n, char c) {
   string *s;
   char   *str;

0: if (n < 0) return NULL;

1: s      = (string *) malloc (sizeof (string));
2: s->len = n;
3: str    = make_string (n);
4: s->str = str;
5: init_string (s,c);

   return s;
}

void init_string (string *s, char c) {
  for (int i = 0; i < s->len; i++) {
    s->str[i] = c;
  }
}
```

Figure 3.2: Example: `new_string`

adopt a *logical* model of memory where allocated blocks are considered to be infinitely far apart. Finally, to specify the safety of pointer dereferences, we stipulate that whenever a pointer x is dereferenced for reading or writing, it has the *bounds-safe* type

$$\{\nu : \texttt{ref}(\ell, 0^{+1}) \mid \text{BBegin}(\nu) \leq \nu \wedge \nu < \text{BEnd}(\nu)\}.$$

Safety Verification To verify that the pointer dereference on line 5 is safe, we must verify that `str` has the bounds-safe type; this will require determining that $\texttt{str} = \texttt{res} +_p \texttt{i}$. This is challenging for a type system, as both `str` and `i` are mutated by the loop. Our system addresses this problem by using SSA renaming to compute different types for the different versions of mutated variables. In the sequel, let $\texttt{x}_\texttt{j}$ be the SSA name of `x` at line `j`. Thus, from the `malloc` at line 2 our system deduces that $\texttt{res}_2$ has type

$$\{\nu : \texttt{ref}(\ell^1, 0) \mid \text{BLen}(\nu, \texttt{n})\}, \tag{a}$$

```
typedef struct _slist {
  struct _slist *next;
  string        *s;
} slist;

slist *new_strings (int n) {
   string *s;
   slist  *sl, *t;

1: sl = NULL;

2: for (int i = 1; i < n; i++) {
3:   s       = (string *) malloc (sizeof (string));
4:   s->len  = i;
5:   s->str  = make_string (i);

6:   t       = (slist *) malloc (sizeof (slist));
7:   t->s    = s
8:   t->next = sl;
9:   sl      = t;
   }

   return sl;
}
```

Figure 3.3: Example: `new_strings`

i.e., that `res` is a pointer to the start of a new location ℓ^1 whose size is `n` bytes. This same type is assigned to $\texttt{str}_3$. Next, our system uses liquid type inference over the qualifiers $\mathbb{Q}$ to infer that, at line 5, $\texttt{i}_5$ and $\texttt{str}_5$ have the respective types

$$\{\nu : \texttt{int} \mid 0 \leq \nu < \texttt{n}\}$$
$$\{\nu : \texttt{ref}(\ell^1, 0^{+1}) \mid \nu = \texttt{res}_2 +_p \texttt{i}_5\}.$$

Notice that these types are loop invariants. They hold the first time around the loop as initially `i` is 0 and `str` is equal to `res`. The types are inductive invariants, as each loop iteration increments `i` and `res`. Thus, our system uses liquid type inference to combine the above facts with (a) and deduce that at line 5

$$\text{BBegin}(\texttt{str}_5) \leq \texttt{str}_5 \wedge \texttt{str}_5 < \text{BEnd}(\texttt{str}_5),$$

i.e., that $\texttt{str}_5$ has the bounds-safe type and hence the pointer dereferences at line 5 of `make_string` are safe.

Function Types Finally, note that `make_string` returns the pointer `res` (i.e., $\texttt{res}_2$) on line 6. Thus, using the type from (a) and the fact that the location ℓ^1 was freshly generated via `malloc`, our

system concludes that `make_string` has the type:

$$\forall \ell^1.(\texttt{n} : \texttt{int})/\emptyset \rightarrow \{\nu : \texttt{ref}(\ell^1, 0) \mid \text{BLen}(\nu, \texttt{n})\}/\ell^1 \mapsto 0^{+1} : \texttt{char} \quad \text{(b)}$$

That is, the function takes an integer `n` and an empty heap (that is, the function does not touch any pre-existing heap contents) and returns a pointer to the start of a new `char` array of size `n`.

Heap-block Invariants

Next, we show how our system uses refinements to verify safety properties of blocks of data residing in the heap. Consider the `new_string` function shown in Figure 3.2. This function takes a parameter, `n`, and produces a `string` structure representing a string of length `n`. The `string` structure has two fields: `len`, the length of the string, and `str`, a pointer to the contents of the string. The programmer intends that the fields obey the following two invariants:

(I_1) the `len` field is non-negative, and

(I_2) the `str` field points to a `char` array of size `len`.

Note that these invariants do not hold at all points during the lifetime of the structure; instead, the programmer establishes them on lines 1-4, and then calls the procedure `init_string` to fill in the string with the supplied character `c`.

Next, we show how our system precisely tracks updates to the structure, tolerating the early stages in which the invariant does not hold, in order to verify the safety of the pointer dereferences within `init_string`.

First, the `malloc` in line 1 creates a new location on the heap, ℓ^2, and gives `s` the type $\texttt{ref}(\ell^2, 0)$, stating that it points into this location at offset 0. Initially, this location contains an 8-byte block (the size of the `string` structure), and so at line 2 the heap is

$$\ell^2 \mapsto \text{ uninitialized 8-byte block .}$$

In line 2, we assign `n` to the `len` field of `s`, which creates a new binding in the heap for ℓ^2 at the offset corresponding to the field `len`, namely 0, since `len` is the first element of the structure. Thus, at line 3 the heap is

$$\ell^2 \mapsto 0 : \{\nu : \texttt{int} \mid \nu = \texttt{n}\}, \text{ uninitialized 4-byte block.}$$

Next, in line 3, the call to `make_string` creates a new location and assigns to `str` a pointer to the new location, with the type shown in (b) (and (a)). Thus, at line 4 the heap contains two locations

$$\ell^1 \mapsto 0^{+1} : \texttt{char}$$
$$\ell^2 \mapsto 0 : \{\nu : \texttt{int} \mid \nu = \texttt{n}\}, \text{ uninitialized 4-byte block.}$$

In line 4, the value of `str` is assigned to `s->str`, which creates a binding at the corresponding offset in ℓ^2, namely 4, as the first field, `len`, was an `int` which is 4 bytes long. Thus, at line 5 the heap is

$$\ell^1 \mapsto 0^{+1} : \texttt{char}$$
$$\ell^2 \mapsto 0 : \{\nu : \texttt{int} \mid \nu = \texttt{n}\}, 4 : \{\nu : \texttt{ref}(\ell^1, 0) \mid \nu = \texttt{str}\}.$$

Finally, at line 5 we have the call to `init_string`. At the callsite, our system uses the qualifiers in $\mathbb{Q}$, and the type of `str` to infer that the previously shown heap binding for ℓ^2 is subsumed by

$$\ell^2 \mapsto 0 : \{\nu : \texttt{int} \mid \nu = \texttt{n}\}, 4 : \{\nu : \texttt{ref}(\ell^1, 0) \mid \text{BLen}(\nu, \texttt{n})\}.$$

As the value at offset 0 equals `n`, the above block is subsumed by

$$\ell^2 \mapsto 0 : \{\nu : \texttt{int} \mid \nu = \texttt{n}\}, 4 : \{\nu : \texttt{ref}(\ell^1, 0) \mid \text{BLen}(\nu, @0)\},$$

where `n` is replaced by @0, a name that denotes the value within the same block at offset 0. Finally, our system uses the test at line 0 to deduce that `n` is non-negative at the callsite, so `init_string` is called with the heap h defined as

$$h \triangleq \ell^2 \mapsto 0 : \{\nu : \texttt{int} \mid 0 \leq \nu\}, 4 : \{\nu : \texttt{ref}(\ell^1, 0) \mid \text{BLen}(\nu, @0)\},$$
$$\ell^1 \mapsto 0^{+1} : \texttt{char}.$$

Note that, as the `len` field of a `string` structure is located at offset 0 and its `str` field is located at offset 4, the bindings for ℓ^2 capture exactly the structure invariants $\mathsf{I}_1, \mathsf{I}_2$ intended by the programmer. Moreover, even though the invariants don't hold everywhere, our system is able to use strong updates to establish them at function call boundaries. Thus, our system infers that the function `init_string` has the type

$$\forall \ell^1, \ell^2.(\texttt{s} : \texttt{ref}(\ell^2, 0))/h \rightarrow \texttt{void}/h,$$

and, via reasoning analogous to that for `make_string`, our system verifies the safety of array accesses in `init_string`.

Data Structure Invariants

In `new_string`, `s` pointed to exactly one heap location, ℓ^1, throughout the execution of the function. Thus, we could soundly perform strong updates to the block describing the contents of ℓ^1; this allowed us to determine that the strings built by `new_string` satisfied the desired invariants. Unfortunately, we cannot soundly use strong updates when dealing with *collections* of locations.

Consider the function `new_strings` shown in Figure 3.3. This function takes an integer parameter, `n`, and creates a list of strings of lengths from 1 to `n`, all of which satisfy the invariants $\mathsf{I}_1, \mathsf{I}_2$. This is accomplished by looping from 1 to `n`, allocating memory for a new `string` and assigning the pointer to this memory to `s` (line 3), initializing it as in `new_string` (lines 4,5), and inserting `s` into a list of strings (lines 6,7,8).

Note that `s` points to many different concrete locations over the course of executing the function; this is in contrast to the previous functions, in which pointers only pointed to a single concrete location while the function was executed. We formalize this distinction by saying that `s` points to an *abstract location* $\widetilde{\ell}$. That is, in our system, `s` has the physical type $\texttt{ref}(\widetilde{\ell}, 0)$, which states that it refers to the offset 0 within one of many possible locations.

Observe that it is not sound to perform strong updates to an abstract location's type. To see why, suppose that we had strongly updated $\widetilde{\ell}$ as we did when analyzing `new_string`. Then we would assign $\widetilde{\ell}$ a block type as follows:

$$\widetilde{\ell} \mapsto 0 : \{\nu : \texttt{int} \mid \nu = \texttt{i}\}, \ldots$$

The problem with this type is that every string has a different length, and the above type only ascribes a single length, `i`, to all strings. Thus, while we need strong updates to establish the desired invariants for each string, we clearly cannot soundly perform strong updates on the types of abstract locations.

We solve this problem with the following crucial observation. Suppose that the code uses a pointer `s` to access a collection of locations $\widetilde{\ell}$. As long as we do not modify `s` or use other pointers to $\widetilde{\ell}$, only one particular concrete location from the set represented by $\widetilde{\ell}$ can be modified at a time. Thus, when a pointer to $\widetilde{\ell}$ is *first* used, we can *unfold* the abstract location into a fresh concrete location, ℓ_j, which inherits $\widetilde{\ell}$'s invariant. As long as $\widetilde{\ell}$ is only accessed by a pointer to ℓ_j, we can soundly perform strong updates on ℓ_j's type. However, as soon as another pointer to $\widetilde{\ell}$ is used, the possibility of aliasing means we can no longer rely on ℓ_j's type to be accurate. Thus, *before* we access an abstract location via another pointer of type $\widetilde{\ell}$, we *fold* the concrete location ℓ_j back into the collection by verifying that ℓ_j satisfies $\widetilde{\ell}$'s invariants and removing it from the heap. The other pointer then gets its own unfolded copy of the location, and can strongly update it, until it gets folded back into the collection, and so on. Our system automatically places folds and unfolds in the code in a manner that ensures that: (1) every heap access occurs via a reference to a concrete location, and (2) every abstract location has at most one corresponding concrete location unfolded in the heap at any point in time. In this way, our system can soundly establish invariants about unbounded data structures in spite of temporary invariant violation even in the presence of aliasing.

We now illustrate the above mechanism using the code in Figure 3.3. We will say that, within the body of the loop, `s` points to some concrete location, ℓ_j, which is an instance of $\widetilde{\ell}$. We will use strong updates, as in the previous examples, to verify that ℓ_j has the desired invariants,

i.e., that

$$\ell_j \mapsto 0 : \{\nu : \texttt{int} \mid 0 \leq \nu\},\ 4 : \{\nu : \texttt{ref}(\ell_2, 0) \mid \text{BLen}(\nu, @0)\}.$$

Finally, at the end of the loop — i.e., before we access another pointer into $\widetilde{\ell}$ in the next iteration — we *fold* the concrete location ℓ_j into the collection by ensuring that it satisfies $\widetilde{\ell}$'s invariants, i.e, by stipulating that at the end of of the loop, the block ℓ_j is a subtype of the block $\widetilde{\ell}$. In this manner, our system performs strong updates *locally* and infers using $\mathbb{Q}$ that at the end of the `new_strings`, the heap is of the form

$$\begin{aligned}
&\widetilde{\ell} \mapsto 0 : \texttt{ref}(\widetilde{\ell}, 0),\ 4 : \texttt{ref}(\widetilde{\ell^1}, 0)\\
&\widetilde{\ell^1} \mapsto 0 : \{\nu : \texttt{int} \mid 0 \leq \nu\},\ 4 : \{\nu : \texttt{ref}(\widetilde{\ell^2}, 0) \mid \text{BLen}(\nu, @0)\}\\
&\widetilde{\ell^2} \mapsto 0^{+1} : \texttt{char}.
\end{aligned}$$

Thus, our system infers that the function returns a list ($\widetilde{\ell}$) of pointers to `string` structures (ℓ^1) each of which satisfy invariants I_1 and I_2.

This concludes our overview of low-level liquid types. Next we formalize our core language and static type system (section 3.2). We then describe our experimental evaluation via a set of challenging case studies (section 3.5). The operational semantics of our core language appears in Appendix B, while a proof that our type system is sound with respect to the semantics is given in Appendix C.

3.2 The NANOC Language and Type System

In this section, we present the syntax and types of NANOC, a simple C-like language with integers and pointers, then present its type checking judgments.

3.2.1 Syntax

The syntax of NANOC is shown in Figure 3.4. We give an overview of the language's features below.

Values The set of NANOC values includes variables and integer constants. Integer constants have the form $n_{|w|}$, where w is the size, in bytes, of the value and n is the numerical integer value. When it does not cause ambiguity, we will sometimes write 0 for the constant $0_{|W|}$, where W is the length in bytes of a machine word. (Throughout this dissertation, we often assume a machine word is 4 bytes long for illustrative purposes; this does not reflect a limitation of our system.)

Pure Expressions We distinguish the *pure* expressions of NANOC, which do not access the heap, from its potentially *impure* expressions. The pure expressions of NANOC, denoted by a, include

v	::=		**Values**
		x	variable
	$\mid$	$n_{\lvert w \rvert}$	integer
a	::=		**Pure Expressions**
		v	value
	$\mid$	$v_1 \circ v_2$	arithmetic operation
	$\mid$	$v_1 +_p v_2$	pointer arithmetic
	$\mid$	$v_1 \bowtie v_2$	relation
e	::=		**Expressions**
		a	pure expression
	$\mid$	$*_n v$	heap read of n bytes
	$\mid$	$*v_1 := v_2$	heap write
	$\mid$	**if** v **then** e_1 **else** e_2	if-then-else
	$\mid$	$f(\overline{v})[\overline{\ell_f} \mapsto \overline{\ell}]$	function call
	$\mid$	**malloc**(v)	memory allocation
	$\mid$	**let** $x = e_1$ **in** e_2	let binding
	$\mid$	**letu** $x =$ **unfold** v **in** e	location unfold
	$\mid$	**fold** ℓ	location fold
F	::=	$f\ (\overline{x_i})\ \{\ e\ \}$	**Function Declarations**
P	::=	$\overline{F}\ e$	**Programs**

Figure 3.4: Syntax of NANOC programs

values, integer and pointer arithmetic expressions, and integer and pointer comparisons. We use the symbol $\circ$ to stand for the arithmetic operators $+$, $*$, and $/$, and the symbol $+_p$ to denote the binary operation that adds an integer offset to a pointer value. We use the symbol $\bowtie$ to stand for the Boolean-valued binary relations $<$, $\leq$, $=$, $\neq$, $\geq$, and $>$. NANOC uses the C convention that nonzero values represent truth and all other values represent falsehood.

Expressions The impure expressions of NANOC, denoted by e, include the pure expressions, as well as if-then-else expressions, let bindings, reads from and writes to memory, memory allocation, location folding and unfolding, and function calls. Note that all bindings are to immutable variables — all mutation is factored into the heap. Next, we examine location

unfolding and folding and function calls in more detail.

Location Fold and Unfold Our goal is to verify invariants of in-memory data structures. These invariants are represented as types associated with *abstract* heap locations, each of which may represent several *concrete* (actual, run-time) heap locations. Verifying properties of the data at these abstract locations in the presence of temporary invariant violation would seem to require performing strong updates on the types of abstract locations. Unfortunately, this would be unsound, since a single abstract location can represent several concrete locations.

However, at run-time a reference will only point to a single concrete location at a time. Thus, operations on abstract locations through a single reference will only affect a single concrete location. Intuitively, if we can get access to this concrete location, we can soundly perform strong updates on it.

Our intuition follows a version control metaphor. Before using a pointer, we can "check out a copy" of its abstract location, giving a concrete location for the pointer which has the same type as the abstract location — a "working copy". As long as the abstract location is accessed only through this pointer to the working copy, it will be sound to perform strong updates on the type of the new concrete location. Finally, if it becomes necessary to use another pointer to the same abstract location, we "check in" the concrete location by checking that it satisfies the same invariant as the corresponding abstract location. The concrete location is then discarded so that no further modification can be made to the working copy.

The "check out" operation is implemented via the **letu** $x = \textbf{unfold}\ v\ \textbf{in}\ e$ construct, where v is a reference to abstract location $\widetilde{\ell}$. The expression creates a new concrete location corresponding to $\widetilde{\ell}$; a reference to this new location is bound to x in e. The "check in" operation is implemented via the **fold** ℓ expression, which verifies that the concrete location corresponding to $\widetilde{\ell}$ satisfies the same invariant as $\widetilde{\ell}$. These procedures and the distinction between abstract and concrete locations are discussed in more detail in the context of their static typing rules in section 3.2.

Function Calls The expression $f(\overline{x})[\overline{\ell_f} \mapsto \overline{\ell}]$ calls function f with parameters $\overline{x}$, and instantiates the quantified locations $\overline{\ell_f}$ in f's type with the locations $\overline{\ell}$. We discuss function calls in more detail in section 3.2.

Programs A NANOC program, denoted by P, is a sequence of function definitions followed by an expression. The program is evaluated by evaluating the expression using the provided function definitions.

3.2.2 Types

The syntax of NANOC types is shown in Figure 3.5. NANOC has a system of refined base types, τ, dependent heaps, h, and dependent function schemas, σ.

t	$::=$		**Base Types**
		$\texttt{int}(n, i)$	integer
	$\mid$	$\texttt{ref}(\ell, i)$	pointer
τ	$::=$	$\{\nu : t \mid \phi\}$	**Refinement Types**
i	$::=$		**Indices**
		n	constant
	$\mid$	$[l,\ u]^c_m$	bounded congruence class
b	$::=$	$\overline{i} : \overline{\tau}$	**Blocks**
ℓ	$::=$		**Heap Locations**
		$\widetilde{\ell}$	abstract location
	$\mid$	ℓ_j	concrete location
h	$::=$		**Heap Types**
		$\emptyset$	empty heap
	$\mid$	$h * \ell \mapsto b$	extended heap
σ	$::=$	$(\overline{x_i} : \overline{\tau})/h_1 \ \rightarrow\ \tau/h_2$	**Function Schemes**

Figure 3.5: Syntax of NANOC types

Locations and References The NANOC *locations*, ℓ, denote areas of the heap. We use $\widetilde{\ell}$ to denote an *abstract location*; abstract locations cannot be read from or written to. We use ℓ_j to denote a *concrete location*; only concrete locations can be read from or written to. Every concrete location ℓ_j corresponds to some abstract location $\widetilde{\ell}$, and we require for soundness that there is at most one concrete location corresponding to a particular abstract location at any given program point. We call references to abstract locations *abstract references* and references to concrete locations *concrete references*. We refer to the ℓ of an abstract location $\widetilde{\ell}$ or concrete location ℓ_j as the location's *name*. When it is unambiguous from the context, we will simply use ℓ to refer to the base name of either an abstract location $\widetilde{\ell}$ or concrete location ℓ_j.

Indices The integer and reference types of NANOC make use of *indices*, i, which are a shorthand notation for single integers and bounded congruence classes of integers. The index n represents

the singleton set $\{n\}$. The index $[l,\ u]^c_m$ represents the sequence of integers

$$\{k \mid l \leq k \leq u \wedge k \equiv c \bmod m\},$$

where: $l < u$; $0 < m$; $l, u \equiv c \bmod m$; and $0 \leq c < m$. We use the notation n^{+m} as shorthand for the index $[n,\ \infty]^{n \bmod m}_m$:

$$\begin{aligned} n^{+m} &= [n,\ \infty]^{n \bmod m}_m \\ &= \{k \mid n \leq k \leq \infty \wedge k \equiv n \bmod m\}. \end{aligned}$$

We use the notation $\top$ as shorthand for $[-\infty,\ \infty]^0_1$:

$$\begin{aligned} \top &= [-\infty,\ \infty]^0_1 \\ &= \{k \mid -\infty \leq k \leq \infty \wedge k \equiv 0 \bmod 1\} \\ &= \mathbb{Z}. \end{aligned}$$

We refer to indices of the form n as *singleton* indices, and all other indices as *sequence* indices. We use i^+ as a metavariable representing a sequence index. We write $[\![i]\!]$ for the set of integers represented by index i.

Base Types The base types, t, of NANOC include integer and reference types. We use $\texttt{int}(w, i)$ to denote the type of w-byte integers n such that $n \in [\![i]\!]$. We use $\texttt{ref}(\ell, i)$ to denote the type of references to location ℓ at an offset $n \in [\![i]\!]$ within that location.

Refined Types As in the previous chapter, a refined type, τ, has the form $\{\nu :\ t \mid \phi\}$, where t is a base type and ϕ is the refinement predicate. As before, we take as our language of refinement predicates the quantifier-free formulas in the (decidable) theory of equality, linear arithmetic and uninterpreted functions (EUFA).

Our refinement logic is augmented with expressions of the form $@n$. When the expression $@n$ is used in the refinement type of a value stored in the heap, the expression refers to the value stored at index $@n$ within the same heap location. Thus, this expression form allows our type system to express dependencies between the fields of a heap-allocated structure.

We use the following type abbreviations: `int` abbreviates $\texttt{int}(W, \top)$, `char` abbreviates $\texttt{int}(1, \top)$, and `void` abbreviates $\texttt{int}(0, 0)$. As before, when it is unambiguous from the context, we use t to abbreviate the type $\{\nu :\ t \mid \texttt{true}\}$. Similarly, when the base type t is clear from the context, we use $\{\phi\}$ to abbreviate $\{\nu :\ t \mid \phi\}$.

Blocks A block type, b, statically represents the contents of a run-time heap location. The types of the block's contents at various offsets are given by bindings $i : \tau$ which state that the values at the offsets in i have the type τ. Within a block, no two index bindings may overlap.

Heaps A heap type, h, represents the contents of the run-time heap, giving a block type to each location in the heap. The contents of heap location ℓ are given by a binding to a block b, written $\ell \mapsto b$. We can form the concatenation of two heaps h_1 and h_2 as $h_1 * h_2$; the resulting heap contains all bindings present in either h_1 or h_2. Our heaps enjoy the following properties: (1) no location may be bound twice in a heap, (2) every abstract location in the heap has at most one corresponding concrete location in the heap, and (3) every concrete location in the heap has exactly one corresponding abstract location in the heap. We say that a run-time store *satisfies* a heap type if every value in the heap has the type specified by the corresponding heap type binding.

Function Schemas We combine refined base types and heap types to form dependent function schemas σ. A dependent function schema consists of an input and output portion. The input portion of a dependent function is a pair $(\overline{x_i} : \overline{\tau_i})/h$ of a dependent tuple giving the parameter types and the input heap, i.e., the heap contents required to call the function. The output portion of a dependent function is a pair τ/h, called a *world*, containing the return type of the function and the output heap, i.e., the heap contents after the function returns. The types in the output world of a dependent function type may refer to variables bound in the input tuple.

Since functions can take reference parameters, they may operate on arbitrary heap locations. Thus, we assume that all function schemas are implicitly quantified over the names of their heap locations.

3.2.3 Typing Rules

In this section, we give the refinement type checking rules of and discuss liquid type inference for NANOC. We begin with a description of NANOC's type environments, rules for type well-formedness, and subtyping. We then discuss several of the most interesting typing rules.

Environments Our typing rules make use of two types of environments: *local environments* and *global environments*. A local environment, Γ, is a sequence of *type bindings* $x : \tau$ and *guard predicates* ϕ. The former are standard; guard predicates capture the results of conditional guards under which an expression is evaluated. A global environment, G, is a sequence of bindings $f : \sigma$ mapping functions to their type schemas.

We assume that suitable renaming has been performed so that no name is bound twice in an environment. An environment is well-formed if each bound type is well-formed in the prefix of the environment that precedes the binding.

$$\Gamma \quad ::= \quad \epsilon \mid x : \tau; \Gamma \mid \phi; \Gamma \qquad \text{(Local Environment)}$$

$$G \quad ::= \quad \epsilon \mid f : \sigma; G \qquad \text{(Global Environment)}$$

Type Well-Formedness $\boxed{\Gamma \vDash \tau}$

$$\frac{\phi \text{ well-sorted in } \Gamma; \nu : t}{\Gamma \vDash \{\nu : t \mid \phi\}} \text{ WF-TYPE}$$

Dependent Block Well-Formedness $\boxed{\Gamma \vDash_@ b}$

$$\frac{\begin{array}{c}\text{DisjointOffsets}(n : \tau,\ b) \\ x \notin \Gamma,\ \text{FV}(\overline{\tau_j}) \qquad \Gamma \vDash \tau \qquad \Gamma, x : \tau \vDash_@ b[@n \mapsto x]\end{array}}{\Gamma \vDash_@ n : \tau,\ b} \text{ WF-DBLOCK-SINGLE}$$

$$\frac{\Gamma \vDash \overline{i_j^+} : \overline{\tau_j}}{\Gamma \vDash_@ \overline{i_j^+} : \overline{\tau_j}} \text{ WF-DBLOCK-SEQUENCE}$$

Non-Dependent Block Well-Formedness $\boxed{\Gamma \vDash b}$

$$\frac{\text{DisjointOffsets}(\overline{i_j} : \overline{\tau_j}) \qquad \forall j. \Gamma \vDash \tau_j}{\Gamma \vDash \overline{i_j} : \overline{\tau_j}} \text{ WF-NDBLOCK}$$

Heap Type Well-Formedness $\boxed{\Gamma \vDash h}$

$$\frac{}{\Gamma \vDash \emptyset} \text{ WF-HEMPTY}$$

$$\frac{\Gamma \vDash h_0 \qquad \widetilde{\ell} \notin \text{dom}(h_0) \qquad \Gamma \vDash_@ b}{\Gamma \vDash h_0 * \widetilde{\ell} \mapsto b} \text{ WF-HABSTRACT}$$

$$\frac{\Gamma \vDash h_0 \qquad \Gamma \vDash b \qquad \widetilde{\ell} \in \text{dom}(h_0) \qquad \ell_k \notin \text{dom}(h_0)}{\Gamma \vDash h_0 * \ell_j \mapsto b} \text{ WF-HCONCRETE}$$

$$\frac{\Gamma \vDash \tau \qquad \Gamma \vDash h}{\Gamma \vDash \tau / h} \text{ WF-WORLD}$$

Function Schema Well-Formedness $\boxed{\vDash \sigma}$

$$\frac{\Gamma = \overline{x_i} : \overline{\tau_i} \qquad \Gamma \vDash \overline{\tau_i} \qquad \Gamma \vDash h_1 \qquad \Gamma \vDash \tau / h_2 \qquad \overline{\tau_i}, h_1, \tau, h_2 \text{ abstract}}{\vDash (\overline{x_i} : \overline{\tau_i}) / h_1 \ \rightarrow \ \tau / h_2} \text{ WF-FUNSCHEME}$$

Figure 3.6: Well-formedness rules for NANOC

Well-Formedness Judgments The judgments of Figure 3.6 ensure that types, heaps, worlds, and function type schemas are *well-formed* in local environments Γ and heaps h. A type is well-formed in a local environment Γ if its refinement predicate ϕ is a well-sorted Boolean formula in Γ.

A block is well-formed if no two index bindings overlap and each type is well-formed with respect to the local environment and preceding indices. We distinguish between *concrete blocks*, bound to concrete heap locations, which must have refinements over immutable variables bound in the environment, and *abstract blocks*, bound to abstract heap locations, which have refinements which may additionally use offset names (e.g., @0) to refer to values at other offsets within the block. We disallow offset names in the refinements for concrete blocks for two reasons. First, they are unnecessary, as we can use names bound in the environment to precisely describe a particular location. Second, they are problematic, as the values at the offsets can be changed by strong updates, thus invalidating the refinements.

To ensure that no block bindings overlap, the block well-formedness rules make use of an auxiliary function, DisjointOffsets, which determines whether all the bindings in a block refer to disjoint offsets within the block. Before we can define DisjointOffsets, we must first define a function, OffsetsOf, which returns the block offsets occupied by a particular binding. Formally, the function $\text{OffsetsOf}(i, \tau)$ yields the set of sets of offsets occupied by a value of type τ located at offsets $k \in [\![i]\!]$, where each set represents the offsets occupied by the contiguous bytes of a value starting at some offset k. For example, $\text{OffsetsOf}(0, \texttt{int}(4,0))$ produces the set $\{\{0,1,2,3\}\}$, i.e., the set of offsets occupied by a 4-byte integer located at offset 0. As another example, $\text{OffsetsOf}(0^{+4}, \texttt{int}(4,0))$ produces the set $\{\{0,1,2,3\},\{4,5,6,7\},\ldots\}$. Finally, the function DisjointOffsets is defined as:

$$\text{DisjointOffsets}(\overline{i_j} : \overline{\tau_j}) = \forall \mathbb{S}, \mathbb{T} \in \biguplus_j \text{OffsetsOf}(i_j, \tau_j).\ \mathbb{S} \cap \mathbb{T} = \emptyset,$$

where $\uplus$ is the multiset union operator — we must use multiset rather than traditional union because, if two bindings were to map to the same set of indices, the traditional union would keep only a single copy of that set of indices, and DisjointOffsets would thus not be able to detect the overlap.

A heap is well-formed if each block is well-formed, no location is bound twice, each abstract location has at most one corresponding concrete location, and each concrete location has a corresponding abstract location. Note that we check the well-formedness of blocks bound to abstract and concrete locations using separate well-formedness rules; only blocks bound to abstract locations are allowed to contain types with dependent field references.

A function schema is well-formed if all parameters are well-formed with respect to the previous parameters and the input heap, the input heap is well-formed with respect to the parameters, and the output world is also well-formed with respect to the parameters.

Subtyping Judgments The subtyping judgments of NANOC are shown in Figure 3.7. The rules

Base Subtyping $\boxed{\Gamma \vdash \tau_1 <: \tau_2}$

$$\frac{i_1 \mathrel{\widetilde{\subseteq}} i_2 \qquad \Gamma \vDash \phi_1 \Rightarrow \phi_2}{\Gamma \vdash \{\nu : \texttt{int}(w, i_1) \mid \phi_1\} <: \{\nu : \texttt{int}(w, i_2) \mid \phi_2\}} \text{ <:-INT}$$

$$\frac{i_1 \mathrel{\widetilde{\subseteq}} i_2 \qquad \Gamma \vDash \phi_1 \Rightarrow \phi_2}{\Gamma \vdash \{\nu : \texttt{ref}(\ell, i_1) \mid \phi_1\} <: \{\nu : \texttt{ref}(\ell, i_2) \mid \phi_2\}} \text{ <:-REF}$$

$$\frac{}{\Gamma \vdash \{\nu : \texttt{ref}(\ell_j, i) \mid \phi\} <: \{\nu : \texttt{ref}(\widetilde{\ell}, i) \mid \phi\}} \text{ <:-ABSTRACT}$$

$$\frac{}{\Gamma \vdash \{\nu : \texttt{int}(W, 0) \mid \nu = 0\} <: \{\nu : \texttt{ref}(\ell, i) \mid \nu = 0\}} \text{ <:-NULL}$$

$$\frac{\Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3} \text{ <:-TRANS}$$

Block Subtyping $\boxed{\Gamma \vdash b_1 <: b_2}$

$$\frac{\Gamma \vdash \tau_1 <: \tau_2 \qquad x \notin \Gamma \qquad \Gamma; x : \tau_1 \vdash b_1[@n \mapsto x] <: b_2[@n \mapsto x]}{\Gamma \vdash n : \tau_1,\ b_1 <: n : \tau_2,\ b_2} \text{ <:-SINGLE}$$

$$\frac{\Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma \vdash b_1 <: b_2}{\Gamma \vdash i^+ : \tau_1,\ b_1 <: i^+ : \tau_2,\ b_2} \text{ <:-SEQUENCE}$$

Heap Subtyping $\boxed{\Gamma \vdash h_1 <: h_2}$

$$\frac{}{\Gamma \vdash \emptyset <: \emptyset} \text{ <:-EMPTY-HEAP}$$

$$\frac{\Gamma \vdash b_1 <: b_2 \qquad \Gamma \vdash h_1 <: h_2}{\Gamma \vdash h_1 * \ell \mapsto b_1 <: h_2 * \ell \mapsto b_2} \text{ <:-HEAP}$$

World Subtyping $\boxed{\Gamma \vdash \tau_1 / h_1 <: \tau_2 / h_2}$

$$\frac{\Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma \vdash h_1 <: h_2}{\Gamma \vdash \tau_1 / h_1 <: \tau_2 / h_2} \text{ <:-WORLD}$$

Figure 3.7: Subtyping rules for NANOC

Subindex Relationship $\boxed{i_1 \mathrel{\widetilde{\subseteq}} i_2}$

$$
\begin{aligned}
n &\mathrel{\widetilde{\subseteq}} n \\
n &\mathrel{\widetilde{\subseteq}} [l,\ u]^c_m \text{ if } l \le n \le u, n \equiv c \bmod m \\
[l_1,\ u_1]^{c_1}_{m_1} &\mathrel{\widetilde{\subseteq}} [l_2,\ u_2]^{c_2}_{m_2} \\
&\qquad \text{if } l_2 \le l_1, u_1 \le u_2, m_2 \mid m_1, c_1 \equiv c_2 \bmod m_2
\end{aligned}
$$

Figure 3.8: Subindex relation

use set-theoretic inclusion checks between arithmetic sequences represented by indices and logical implication checks over the refinement predicates. As in the previous chapter, we embed environments into the refinement logic as follows:

$$[\![\Gamma]\!] \equiv \bigwedge\{\phi \mid \phi \in \Gamma\} \wedge \bigwedge\{\phi[\nu \mapsto x] \mid x : \{\nu : t \mid \phi\} \in \Gamma\}.$$

Most of the rules in Figure 3.7 are straightforward. Rule <:-NULL is used to coerce the integer value $0_{|W|}$ into an arbitrary pointer type, allowing the use of null pointers. Rule <:-ABSTRACT allows a concrete pointer to be treated as abstract. The rules for subtyping integers and pointers, <:-INT and <:-REF, make use of the index subsumption relationship, $\widetilde{\subseteq}$, defined in Figure 3.8.

Pure Typing Judgments The typing judgments for pure expressions are shown in Figure 3.9. The rules are quite standard [71, 35, 75, 7]. Note that the refinement predicates for these expressions precisely track the value of the expression.

The rules T-ARITH and T-PTRARITH use index operators $\mathrel{\widetilde{\circ}}$ and $\mathrel{\widetilde{+}}$, which are binary operations on indices which approximate arithmetic operations. The definitions of these operations are given in Figure 3.10; in the case of the commutative operators $\mathrel{\widetilde{+}}$ and $\mathrel{\widetilde{\cdot}}$, we only show one argument order for each definition. We note that our logic's pointer arithmetic operator, $+_p$, satisfies the following two laws, so that adding an offset to a pointer always yields a pointer into the same block:

$$
\begin{aligned}
\text{BBegin}(v_1 +_p v_2) &= \text{BBegin}(v_1) \\
\text{BEnd}(v_1 +_p v_2) &= \text{BEnd}(v_1)
\end{aligned}
$$

Typing Judgments The typing judgments for expressions and programs are shown in Figure 3.11, Figure 3.12, and Figure 3.13. The program typing rules are straightforward. The expression typing judgment $G,\ \Gamma,\ h \vdash e : \tau / h'$ states that, in global environment G and local environment Γ, if the heap initially satisfies heap type h, then evaluating e produces a value of type τ and a heap

Pure Typing Rules $\boxed{\Gamma \vdash a : \tau}$

$$\frac{\Gamma(x) = \{\nu : t \mid \phi\}}{\Gamma \vdash x : \{\nu : t \mid \nu = x\}}\ \text{T-VAR}$$

$$\frac{}{\Gamma \vdash n_{|w|} : \{\nu : \texttt{int}(w, n) \mid \nu = n_{|w|}\}}\ \text{T-INT}$$

$$\frac{\Gamma \vdash a_1 : \texttt{int}(n, i_1) \qquad \Gamma \vdash a_2 : \texttt{int}(n, i_2)}{\Gamma \vdash a_1 \circ a_2 : \{\nu : \texttt{int}(n, i_1 \mathbin{\widetilde{\circ}} i_2) \mid \nu = a_1 \circ a_2\}}\ \text{T-ARITH}$$

$$\frac{\Gamma \vdash a_1 : \texttt{ref}(\ell, i_1) \qquad \Gamma \vdash a_2 : \texttt{int}(W, i_2)}{\Gamma \vdash a_1 +_p a_2 : \{\nu : \texttt{ref}(\ell, i_1 \mathbin{\widetilde{+}} i_2) \mid \nu = a_1 +_p a_2\}}\ \text{T-PTRARITH}$$

$$\frac{\Gamma \vdash a_1 : \tau \qquad \Gamma \vdash a_2 : \tau}{\Gamma \vdash a_1 \bowtie a_2 : \{\nu : \texttt{int}(W, [0,\ 1]_1^0) \mid \text{if } a_1 \bowtie a_2 \text{ then } \nu = 1_{|W|} \text{ else } \nu = 0_{|W|}\}}\ \text{T-RELATION}$$

$$\frac{\Gamma \vdash a : \tau_1 \qquad \Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma \vDash \tau_2}{\Gamma \vdash a : \tau_2}\ \text{T-PURESUB}$$

Figure 3.9: Typing rules for pure NANOC expressions

satisfying h'. The majority of the rules are straightforward; the most interesting rules are those that deal with memory access.

Type Checking Memory Operations

Below, we discuss the rules for memory allocation, heap operations, function calls, and location unfolding. The key idea that enables our system to verify and infer invariants about in-memory data structures in the presence of temporary invariant violation is our distinction between *concrete locations* and *abstract locations*. Thus, to better understand the rules for memory operations, we begin with a more thorough description of abstract and concrete locations.

Concrete Locations Concrete locations are names that refer to *exactly one* physical memory location. For example, a single item in a linked list has one physical location and thus can be identified with a concrete location. The block bound to a concrete location describes the current state of the contents of exactly one physical location.

Abstract Locations Abstract locations are names that refer to *zero or more* concrete locations. For example, all items in a linked list may share the same abstract location, although each item is at a different concrete location. The block bound to an abstract location is an invariant that applies to all elements which share that abstract location.

Index Addition $\boxed{i_1 \mathbin{\tilde{+}} i_2}$

$$n \mathbin{\tilde{+}} m = n + m$$
$$n \mathbin{\tilde{+}} [l,\ u]^c_m = [l + n,\ u + n]^{c+n \bmod m}_m$$
$$[l_1,\ u_1]^{c_1}_{m_1} \mathbin{\tilde{+}} [l_2,\ u_2]^{c_2}_{m_2} = [l_1 + l_2,\ u_1 + u_2]^0_{\gcd(m_1,m_2,c_1,c_2)}$$

Index Multiplication $\boxed{i_1 \mathbin{\tilde{\cdot}} i_2}$

$$n \mathbin{\tilde{\cdot}} m = nm$$
$$n \mathbin{\tilde{\cdot}} [l,\ u]^c_m = [nl,\ nu]^{nc}_{nm}$$
$$i_1^+ \mathbin{\tilde{\cdot}} i_2^+ = [-\infty,\ \infty]^0_1$$

Index Division $\boxed{i_1 \mathbin{\tilde{/}} i_2}$

$$n \mathbin{\tilde{/}} m = n/m \text{ if } m \neq 0$$
$$i \mathbin{\tilde{/}} i = [-\infty,\ \infty]^0_1$$

Figure 3.10: Index arithmetic operators

Since we wish to verify data structure invariants in spite of temporary invariant violation, we will allow memory to be accessed only through concrete locations. This will enable our type system to perform strong updates to the types of concrete locations, providing robustness with respect to temporary invariant violation. Because we wish to verify properties of unbounded collections, which are represented using abstract locations, we need a strategy to handle pointers to abstract locations.

Strategy for Collections We employ a two-pronged strategy for handling pointers to abstract locations, and thereby collections. First, as long as only a single pointer to an abstract location is used, we can be assured that only one corresponding concrete location is being accessed. We will use our *location unfold* operation to obtain a concrete location corresponding to a pointer's referent. As long as the abstract location is only accessed through this "unfolded" pointer, we can safely perform strong updates on the new concrete location. Second, if we must use another pointer to access the abstract location, we can no longer be assured that a single concrete location will be updated. When this happens, we will use the *location fold* operation to ensure that the contents of the concrete location created earlier meet the abstract location's invariant, disallow further use of the unfolded pointer (without another unfold), and allow the new pointer to be soundly unfolded.

Standard Typing Rules $\boxed{G,\ \Gamma,\ h \vdash e : \tau / h_2}$

$$\frac{\Gamma \vdash a : \tau}{G,\ \Gamma,\ h \vdash a : \tau / h}\ \text{T-PURE}$$

$$\frac{G,\ \Gamma,\ h \vdash e : \tau_1 / h_1 \qquad \Gamma \vdash \tau_1 / h_1 <: \tau_2 / h_2 \qquad \Gamma \vDash \tau_2 / h_2}{G,\ \Gamma,\ h \vdash e : \tau_2 / h_2}\ \text{T-SUB}$$

$$\frac{\Gamma \vdash v : \mathtt{int}(W, i) \qquad G,\ \Gamma; v \neq 0,\ h \vdash e_1 : \hat{\tau} / \hat{h}' \qquad G,\ \Gamma; v = 0,\ h \vdash e_2 : \hat{\tau} / \hat{h}'}{G,\ \Gamma,\ h \vdash \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \hat{\tau} / \hat{h}'}\ \text{T-IF}$$

$$\frac{G,\ \Gamma,\ h \vdash e_1 : \tau_1 / h_1 \qquad G,\ \Gamma; x : \tau_1,\ h_1 \vdash e_2 : \hat{\tau}_2 / \hat{h}_2 \qquad \Gamma \vDash \hat{\tau}_2 / \hat{h}_2}{G,\ \Gamma,\ h \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \hat{\tau}_2 / \hat{h}_2}\ \text{T-LET}$$

Figure 3.11: Typing rules for standard NANOC expressions

In the following, we describe the typing rules for the key operations of location unfolding and folding and demonstrate how they allow us to soundly perform strong updates. We then describe the remaining heap-accessing operations: memory allocation, heap read and write, and function calls.

Unfolding The expression **letu** $x =$ **unfold** v **in** e, which "acquires" a concrete pointer to the location $\widetilde{\ell}$ that v points to, is typed by rule T-UNFOLD. The rule first type checks v in Γ to determine where it points. The block b bound to this location is located in the initial heap, h, to find the invariant satisfied by the abstract location. With some modification, this same block is bound to a new concrete location, ℓ_j, to ensure that this concrete location initially satisfies the same invariants as the abstract location did.

The modification consists of a sequence of substitutions. The block b may contain types which reference previous elements by their indices (i.e., may contain types containing names like $@i$). Such types only have meaning in the context of the block where the indices are bound; if these types are extracted from the block — by typing a read operation, for example — they will be meaningless, since the indices are not bound to types in the environment. To give these types meaning outside of the block, we create fresh variable names x_i for each sequence index i and extend the environment with appropriately-substituted bindings for these names. Each concrete location has a "selfified" refinement stating that the value at each index i is equal to the corresponding name x_i. Note that sequence indices are *not* bound to selfified types, because a sequence index binding represents multiple data values.

Finally, a pointer to ℓ_j is bound to x in the body e. Well-formedness checks ensure that no other concrete location corresponding to $\widetilde{\ell}$ exists and that the new bindings do not escape the

Heap Read/Write Typing Rules $\boxed{G,\ \Gamma,\ h \vdash e : \tau / h_2}$

$$\frac{\Gamma \vdash v : \{\nu :\ \texttt{ref}(\ell_j, i)\ \mid\ \text{Safe}(\nu, n)\} \qquad h = h_1 * \ell_j \mapsto \ldots, i : \tau, \ldots \qquad \text{SizeOf}(\tau) = n}{G,\ \Gamma,\ h \vdash *_n v : \tau / h} \text{ T-READ}$$

$$\frac{\begin{array}{c} h = h_1 * \ell_j \mapsto \ldots, n : \tau_1, \ldots \qquad \Gamma \vdash v_1 : \{\nu :\ \texttt{ref}(\ell_j, n)\ \mid\ \text{Safe}(\nu, \text{SizeOf}(\tau_2))\} \\ \Gamma \vdash v_2 : \tau_2 \qquad \text{SizeOf}(\tau_2) = \text{SizeOf}(\tau_1) \qquad h' = h_1 * \ell_j \mapsto \ldots, n : \tau_2, \ldots \end{array}}{G,\ \Gamma,\ h \vdash *v_1 := v_2 : \texttt{void} / h'} \text{ T-SUPD}$$

$$\frac{\begin{array}{c} \Gamma \vdash v_1 : \{\nu :\ \texttt{ref}(\ell_j, n^{+m})\ \mid\ \text{Safe}(\nu, \text{SizeOf}(\hat{\tau}_2))\} \\ \Gamma \vdash v_2 : \hat{\tau} \qquad h = h_1 * \ell_j \mapsto \ldots, n^{+m} : \hat{\tau}, \ldots \end{array}}{G,\ \Gamma,\ h \vdash *v_1 := v_2 : \texttt{void} / h} \text{ T-WUPD}$$

Figure 3.12: Typing rules for NANOC heap reads and writes

scope of the body.

Note that the pointer being unfolded must be non-null. Because null pointers are treated as references to arbitrary, possibly uninhabited, abstract locations with arbitrary invariants, allowing a null pointer to be unfolded would allow the introduction of arbitrary predicates into the environment, leading to unsoundness. By allowing only non-null pointers to be unfolded, we ensure that we only unfold pointers to concrete locations which had previously been allocated, initialized, and folded. Such pointers are guaranteed to genuinely satisfy the invariants of their corresponding abstract locations and so there is no risk of unsoundness in unfolding them.

Folding The expression **fold** ℓ, which "releases" the concrete location currently assigned to $\widetilde{\ell}$, is typed by rule T-FOLD. The rule uses subtyping to check that the concrete location ℓ_j satisfies the invariant specified by its corresponding abstract location $\widetilde{\ell}$ and removes concrete location ℓ_j from the output heap, preventing further use of pointers to ℓ_j.

Memory Allocation The expression **malloc**(v) is typed by rule T-MALLOC, which creates a new concrete location corresponding to newly-allocated memory. The new concrete location corresponds to abstract location $\widetilde{\ell}$, which is mapped to block b, giving the desired invariant for the new concrete location. This invariant is not yet established for the concrete location, which represents freshly-allocated memory; thus, the concrete location is mapped to b^0, defined as

$$\frac{\begin{array}{c}\Gamma \vdash v : \{\nu : \texttt{ref}(\widetilde{\ell}, i_y) \mid \nu \neq 0\} \qquad h = h_0 * \widetilde{\ell} \mapsto \overline{n_k} : \overline{\tau_k}, \overline{i^+} : \overline{\tau^+} \\ \overline{x_k} \text{ disjoint} \qquad \overline{x_k} \notin \Gamma,\ e,\ \mathrm{FV}(h) \qquad \theta = [\overline{@n_k} \mapsto \overline{x_k}] \\ \Gamma_1 = \Gamma; \overline{x_k} : \overline{\theta\tau_k} \qquad \ell_j \text{ fresh} \qquad h_1 = h * \ell_j \mapsto \overline{n_k} : \overline{\{\nu = x_k\}}, \overline{i^+} : \overline{\theta\tau^+} \\ G,\ \Gamma_1; x : \{\nu : \texttt{ref}(\ell_j, i_y) \mid \nu = v\},\ h_1 \vdash e : \hat{\tau}_2 / \hat{h}_2 \qquad \Gamma_1 \vDash h_1 \qquad \Gamma \vDash \hat{\tau}_2 / \hat{h}_2\end{array}}{G,\ \Gamma,\ h \vdash \textbf{letu}\ x = \textbf{unfold}\ v\ \textbf{in}\ e : \hat{\tau}_2 / \hat{h}_2}\ \text{T-UNFOLD}$$

$$\frac{h = h_0 * \widetilde{\ell} \mapsto \hat{b}_1 * \ell_j \mapsto b_2 \qquad \Gamma \vdash b_2 <: \hat{b}_1}{G,\ \Gamma,\ h \vdash \textbf{fold}\ \ell : \texttt{void} / h_0 * \widetilde{\ell} \mapsto \hat{b}_1}\ \text{T-FOLD}$$

$$\frac{\begin{array}{c}\Gamma \vDash h_m \qquad \Gamma \vDash h_u \qquad G(f) = (\overline{x_j} : \overline{\tau_j}) / h_f \ \rightarrow\ \tau' / h'_f \\ \theta = [\overline{x_j} \mapsto \overline{v_j}][\overline{\ell_f} \mapsto \overline{\ell}] \qquad \Gamma \vDash h_u * \theta h_f \qquad \Gamma \vdash \overline{v_j} : \overline{\theta\tau_j} \qquad \Gamma \vdash h_m <: \theta h_f\end{array}}{G,\ \Gamma,\ h_u * h_m \vdash f(\overline{v_j})[\overline{\ell_f} \mapsto \overline{\ell}] : \theta\tau' / h_u * \theta h'_f}\ \text{T-CALL}$$

$$\frac{\ell_j \text{ fresh} \qquad h = h_0 * \widetilde{\ell} \mapsto b \qquad \Gamma \vDash h * \ell_j \mapsto b \qquad \Gamma \vdash v : \{\nu : \texttt{int}(W, i) \mid \nu \geq 0\}}{G,\ \Gamma,\ h \vdash \textbf{malloc}(v) : \{\nu : \texttt{ref}(\ell_j, 0) \mid \mathrm{Allocated}(\nu, v)\} / h * \ell_j \mapsto b^0}\ \text{T-MALLOC}$$

Figure 3.13: Non-standard typing rules for NANOC expressions

follows:

$$\begin{aligned}(n : \{\nu : \texttt{int}(w, i) \mid \phi\}, b)^0 &= n : \{\nu : \texttt{int}(w, 0) \mid \nu = 0_{|w|}\}, b^0 \\ (n : \{\nu : \texttt{ref}(\ell, i) \mid \phi\}, b)^0 &= n : \{\nu : \texttt{ref}(\widetilde{\ell}, 0) \mid \nu = 0\}, b^0 \\ (i^+ : \{\nu : \texttt{int}(w, i) \mid \phi\}, b)^0 &= i^+ : \{\nu : \texttt{int}(w, i \sqcup 0) \mid \phi \vee \nu = 0_{|w|}\}, b^0 \\ (i^+ : \{\nu : \texttt{ref}(\ell, i) \mid \phi\}, b)^0 &= i^+ : \{\nu : \texttt{ref}(\widetilde{\ell}, i \sqcup 0) \mid \phi \vee \nu = 0\}, b^0\end{aligned}$$

Thus, b^0 is the type of a block with the same shape as b, where all the contents are set to zero. (Note that we change the types of arrays to indicate that their values are either zero or satisfy the refinement predicate specified in b. If we did not use disjunction here, it would only be possible to write zeroes into freshly-allocated arrays, which is sound but useless.) The expression returns a reference to the beginning of the concrete location (index 0). The refinement on the returned reference states that the reference is a non-null pointer to the start of a block of size v; we denote this with the abbreviation Allocated, defined as

$$\mathrm{Allocated}(v_1, v_2) \triangleq v_1 \neq 0 \wedge \mathrm{BLen}(v_1, v_2).$$

The uniqueness of concrete location bindings within the heap is ensured using heap well-formedness; i.e., if there is an active concrete location corresponding to the abstract location being allocated, it must be "folded up" before **malloc** is invoked.

Pointer Read The expression $*_n v$, which reads n bytes from the address pointed to by v, is typed by rule T-READ. This rule ensures that the pointer is *safe*, i.e., that v is non-null and points to a region which contains at least n bytes. This is ensured by requiring that v satisfies the predicate $\text{Safe}(v, n)$, defined as

$$\text{Safe}(v, n) \triangleq v \neq 0 \wedge \text{BBegin}(v) \leq v \wedge v + n \leq \text{BEnd}(v).$$

If v is non-null and within bounds, the type of the read is given by the type bound in the heap at the reference's location, index pair. The rule additionally ensures that the dereference reads exactly one value — that is, that it does not read part of a single value or a single value plus part of another; this is ensured using the SizeOf type operator, defined in Appendix B. The heap is left unaltered.

Pointer Write The expression $*v_1 := v_2$ is typed by rules T-SUPD and T-WUPD. If the reference's type identifies exactly one location within a block — i.e., it has a singleton index n — the rule T-SUPD can be used to return a new, strongly-updated heap where the type of the referent has been updated to the type of the value being assigned. We require that the value written through the reference has the same size as the value that was previously at that location; this ensures that the block type remains well-formed after the strong update.

If the reference's type only indicates that it potentially points to one of several locations within a block — i.e., it has a sequence index n^{+m} — a strong update is unsound; in this case, the rule T-WUPD is used to ensure that the new value has the same type as the previous value. Note that we could introduce fold and unfold operations on arrays to allow strong updates to array elements, but we eschew this for simplicity. Both rules ensure that the dereferenced pointer is safe.

Function Call The expression $f(\overline{v_j})[\overline{\ell_f} \mapsto \overline{\ell}]$ is typed by rule T-CALL, which is inspired by the modular "footprint"-based frame rule from separation logic. This rule splits the initial heap into two portions: h_m, the portion of the heap which is modified by the function, and h_u, the portion of the heap which is left unmodified by the function. To ensure soundness, we check that h_m and h_u are individually well-formed; this prevents placing a concrete location in h_u and its corresponding abstract location in h_m, allowing the function to unsoundly unfold an already-unfolded location. The rule also generates a substitution mapping formal (location) parameters to actual (location) parameters. This substitution is used to check that the actual parameters and heap are subtypes of the formal parameters and heap. The result of the call is the return type and the function's output heap, both with the actual parameters substituted for the formals. The resultant output heap is joined with the unmodified portion of the input heap to obtain the caller's heap after the function returns.

Program Typing Programs are typed using rules T-FUN and T-MAIN. Rule T-FUN is used to type function definitions, and is straightforward.

Program Typing $\boxed{G \vdash P : \tau / h}$

$$\frac{\models \hat{\sigma} \qquad \hat{\sigma} = (\overline{x_j : \hat{\tau}_j})/\hat{h} \;\rightarrow\; \hat{\tau}'/\hat{h}' \qquad G; \mathtt{f} : \hat{\sigma}, e,\ \overline{x_j : \hat{\tau}_j},\ \hat{h} \vdash e : \hat{\tau}'/\hat{h}' \qquad G; \mathtt{f} : \hat{\sigma} \vdash P : \tau_p / h_p}{G \vdash \mathtt{f}\ (\overline{x_j})\ \{\ e\ \}\ P : \tau_p / h_p}\ \text{[T-FUN]}$$

$$\frac{h \text{ abstract} \qquad G,\ \emptyset,\ h \vdash e : \tau / h'}{G \vdash e : \tau / h'}\ \text{T-MAIN}$$

Figure 3.14: Program Typing

Rule T-MAIN, which type checks the expression that makes up the "main" function of the program, ensures the initial heap contains only abstract locations: since at the beginning of execution no locations have been allocated and no invariants established, the initial heap cannot contain concrete locations. It may, however, contain abstract locations, since they need not describe the contents of any concrete locations.

3.3 Data Structure Verification with Final Fields

While the system presented so far is adept at specifying, verifying, and inferring invariants that express relationships between local variables or between two fields of one structure, it cannot express invariants which relate fields in one structure to fields in another, different structure, and thus cannot express or verify properties of linked, mutable data structures.

To understand why, consider expressing the following invariant as a type:

$$\texttt{p->size} = \texttt{p->next->size}$$

We might consider a naïve approach which would give p a type which says that it is a reference to a location ℓ of type

$$\ell \mapsto \texttt{size} : \texttt{int},\ \texttt{next} : \{\nu : \texttt{ref}(\ell, 0) \mid \nu\texttt{->size} = \texttt{size}\},$$

which says that the value obtained by dereferencing the `next` pointer to obtain its `size` field yields a value equal to the `size` field of the current structure, `p`. However, giving `p` such a type would be unsound if the value of its `size` field can change as the program executes. Nonetheless, if we are to verify the safety and correctness of realistic programs, we must be able to reason about inter-link invariants like this one.

In this section, we augment our type system to express and verify such precise inter-link invariants of mutable linked data structures. To do so, we wish to retain the expressive power afforded by allowing dereferences inside type refinements while avoiding the unsoundness usually

associated with them. Our key observation is that the problem with allowing references inside refinement predicates is *uncontrolled mutation*: in our example, at any point in the program, the `size` field of either structure may change, invalidating the refinement and causing unsoundness. If, however, we can be sure that the `size` field is immutable after the current program point, it is completely safe to allow dereferences that access `size` within refinements. We call such fields *final*, and our type system soundly allows refinement predicates to contain dereferences to such final fields.

This extension considerably increases the expressiveness of our type system. For example, our system is able to use flow- and path-sensitive reasoning, along with the refinements expressible using final fields, to determine automatically that library code which destructively updates lists in order to add or remove elements maintains the lists' sortedness invariants.

3.3.1 Final Fields Example: Memory Allocation

We now illustrate by example how the addition of final fields to our system enables the verification and inference of invariants of mutable linked data structures.

Figure 3.15 shows an implementation of a `malloc`-like memory manager whose free list is divided into allocation "pools" of type `pool` which consist of an allocation size and the free list for that size, which is a linked list of `region` structures, each of which consists of an 8-byte header containing the region's size and link to the next free region, followed by an array of characters containing the allocated memory itself. The pools themselves are linked together in a list in increasing order of allocation size.

Link Invariants We begin by verifying that each pointer produced by `new_region` points to the `mem` field of an allocated location which contains exactly as many bytes as specified by the region header, plus space for the header itself. Ideally, we would give the return value the type

$$\{\nu : \texttt{ref}(\ell, 8) \mid \nu = \text{BBegin}(\nu) + 8 \wedge \text{BEnd}(\nu) = \nu + *\,\text{BBegin}(\nu)\}.$$

Note the term $*\,\text{BBegin}(\nu)$ dereferences the pointer to access its `size` field. Thus, this type states that the return value is a pointer to the `mem` field (eighth byte) of its location and that the end of the block comes `size` bytes beyond the referenced location. Unfortunately, it is not generally sound to use dereferences in types; if the `size` field of the returned structure is mutated at any point, the refinement may be invalidated, and the type system will be unsound.

Final Fields On the other hand, a straightforward analysis of the program shows that, after its initialization on line 6, the `size` field of the returned structure is never mutated. We say that such a field is *final*, and note that it is sound for refinements to dereference such fields so long as they remain final. In refinement predicates, we syntactically distinguish dereferences to final fields from dereferences to potentially mutable fields by writing references to final fields as $!v$.

```
struct region {
    int     size;
    region *next;
    char    mem[0];
};

struct free_pool {
    int        size;
    region    *free;
    free_pool *next;
};

// Assume fl is initialized and non-null
free_pool *fl = ...;

void init (int size, char *m) {
 1: while (size--)
 2:     *m++ = 0;
}

char *new_region (int sz) {
 3: region *r =
     (region *) sbrk (sizeof (*r) + sz);
 4: init (sz, &r->mem);
 5: r->next = NULL;
 6: r->size = sz;

 7: return &r->mem;
}

char *pool_alloc (free_pool *p) {
 8: if (p->free) {
 9:     region *r = p->free;
10:     p->free   = r->next;

11:     return &r->mem;
    }

12: return new_region (p->size);
}
```

```
char *alloc (int size) {
13: if (size <= 0) return NULL;

14: free_pool *p;
15: for (p = fl;
16:      p->size < size && p->next != NULL;
17:      p = p->next) ;

18: if (p->size >= size) return pool_alloc (p);

19: free_pool *np  =
      (free_pool *) sbrk (sizeof (*np));
20: np->size       = 2 * p->size;
21: np->free       = NULL;
22: np->next       = NULL;
23: p->next        = np;

24: return pool_alloc (np);
}

void dealloc (char *mem) {
25: if (mem == NULL) return;

26: region *r = (region *) mem - 1;
27: init (r->size, &r->mem);

28: free_pool *p = fl;
29: while (p->size != r->size) {
30:     p = p->next;
31:     if (p == NULL) return;
    }

32: r->next = p->free;
33: p->free = r;
}
```

Figure 3.15: Final fields example: memory management

We now return to `new_region`. At line 3, the function calls `sbrk` to expand the heap, returning a pointer to the beginning of a new location ℓ_j. From the refined type of `sbrk` and the fact that `sizeof (region)` is 8, we have that `r`'s type is

$$\texttt{r} : \{\nu : \texttt{ref}(\ell_j, 0) \mid \text{Allocated}(\nu, 8 + \texttt{sz})\}.$$

Line 4 initializes the `mem` field, i.e., the data block returned to the user. Line 5 initializes the `next` field to `NULL`.

Line 6 initializes the `size` field to the requested size, `sz`. This is the final assignment made to the `size` field for the lifetime of this structure; thus, after this point, the `size` field is final. Our analysis will soundly introduce the assumption

$$!\mathtt{r} = \mathtt{sz}$$

into the environment when type checking the remainder of the function, reflecting the immutable value of `r`'s `size` field.

We compute the type of the returned pointer as

$$\mathtt{\&r->mem} : \{\nu : \mathtt{ref}(\ell_j, 8) \mid \nu = \mathtt{r} +_p 8\}.$$

Together, this type, the assumption $!\mathtt{r} = \mathtt{sz}$, and the type of `r` allow our system to deduce that

$$\mathrm{BEnd}(\mathtt{\&r->mem}) = \mathtt{\&r->mem} + !\,\mathrm{BBegin}(\mathtt{\&r->mem}).$$

By subsumption, then, we are able to give the return value of `new_region` the type

$$\{\nu : \mathtt{ref}(\widetilde{\ell}, 8) \mid \nu \neq 0 \wedge \nu = \mathrm{BBegin}(\nu) + 8 \wedge \mathrm{BEnd}(\nu) = \nu + !\,\mathrm{BBegin}(\nu)\}$$

which soundly dereferences the final field `size` to express the invariant that the returned pointer references an allocated block of memory that is as long as the region header plus the number of bytes specified in the `size` field of the header. Our system is able to verify all the invariants shown so far using the qualifier set $\mathbb{Q}_2$ defined as

$$\begin{aligned}\mathbb{Q}_2 = \{&\nu \neq 0,\ \nu > 0,\ \nu \geq 0,\\ &\mathrm{BEnd}(\nu) = \nu +_p \star,\\ &\nu \neq 0 \Rightarrow \nu \geq \mathrm{BBegin}(\nu),\\ &\nu \neq 0 \Rightarrow \nu = \mathrm{BBegin}(\nu) + 8,\\ &\nu \neq 0 \Rightarrow \mathrm{BEnd}(\nu) = \nu + !\,\mathrm{BBegin}(\nu)\}.\end{aligned}$$

3.3.2 Linked Structure Invariants

At least as important as the invariants that hold *for each* region and pool of the allocator of Figure 3.15 are the invariants that hold *between* these objects. For example:

1. The list of pools must be sorted in ascending order by `size` to avoid allocating new regions when a sufficiently large free region is already available.

2. The elements of the free list of a pool must have a `size` field that agrees with the `size` field of the pool.

3. All regions within the free list of a pool must have the same value for `size`.

These invariants are challenging because they are not properties of single elements within the pool and free list structures, but rather relationships between them. Nevertheless, we can take advantage of the fact that certain fields are final to express, verify, and infer these invariants.

Sorted Free List We begin by showing how our system verifies that the list of pools is sorted in increasing order of their `size` fields. This amounts to showing that the linked list insertion in `alloc` maintains the order of the free list `fl`.

Block Invariants We first note that the heap has the following type at entry to `alloc`:

$$\begin{aligned}
\widetilde{\ell^1} \mapsto\ & 0 : \text{final } \{\nu : \texttt{int} \mid \nu > 0\}, \\
& 4 : \texttt{ref}(\widetilde{\ell^1}, 0), \\
& 8^{+1} : \texttt{char} \\
\widetilde{\ell^2} \mapsto\ & 0 : \text{final } \texttt{int}, \\
& 4 : \texttt{ref}(\widetilde{\ell^1}, 0), \\
& 8 : \{\nu : \texttt{ref}(\widetilde{\ell^2}, 0) \mid \nu \neq 0 \Rightarrow @0 < !\nu\}
\end{aligned}$$

Here, $\widetilde{\ell^1}$ represents the collection of `region` structures, while $\widetilde{\ell^2}$ represents the collection of `pool` structures. The qualifier "final" attached to a field's type indicates that the field is final, i.e., it will not be mutated after the current program point. The type $\{\nu : \texttt{ref}(\widetilde{\ell^2}, 0) \mid \nu \neq 0 \Rightarrow @0 < !\nu\}$ indicates that the `next` field of each `pool` is either null or points to a `pool` structure with a strictly greater `size` — in other words, that the list of pools is sorted by size.

We demonstrate how our system verifies that this type is maintained through execution of the `alloc` function. If the test on line 13 passes, the function returns immediately without touching the heap. Otherwise, the loop of lines 14–17 iterates over the free list `fl` until either a pool of regions of sufficient size is found or we reach the last pool without finding any that contains large enough regions. Line 18 checks whether the loop exited because a large enough pool was found; if so, the function exits. Otherwise, a new pool is allocated and appended to the free list in lines 19–23.

Verifying Insertion At line 18, `p` is unfolded to a concrete location $\ell^2_{\mathtt{p}}$:

$$\begin{aligned}
\ell^2_{\mathtt{p}} \mapsto\ & 0 : \{\nu : \text{final } \texttt{int} \mid \nu = \mathtt{x}_1\}, \\
& 4 : \{\nu : \texttt{ref}(\widetilde{\ell^1}, 0) \mid \nu = \mathtt{x}_2\}, \\
& 8 : \{\nu : \texttt{ref}(\widetilde{\ell^2}, 0) \mid \nu = \mathtt{x}_3\}
\end{aligned}$$

The fresh variables $\mathtt{x}_1$, $\mathtt{x}_2$, $\mathtt{x}_3$ are created to represent the initial values of `p`'s fields and bound in the environment; their types are not important, except to note that we know that $\mathtt{x}_1$ is positive.

At line 19, `np` is assigned the result of an `sbrk` call which allocates new memory; the effect of this call is to create a concrete location pair, $\ell^2_{\texttt{np}}$, on the heap and bind `np` to a pointer to $\ell^2_{\texttt{np}}$:

$$\begin{aligned}\ell^2_{\texttt{np}} \mapsto 0 : \texttt{int},\\ 4 : \texttt{ref}(\widetilde{\ell^1}, 0),\\ 8 : \texttt{ref}(\widetilde{\ell^2}, 0)\end{aligned}$$

The type of $\ell^3_{\texttt{np}}$ contains no refinements, reflecting the fact that nothing is known about the uninitialized data at that location. Note that we have unfolded both `p` and `np`, which share the same abstract location; this is sound, as the freshly-allocated pointer `np` cannot possibly alias with any existing pointers.

At line 20, the assignment to `np->size` changes the type of `np`'s `size` field:

$$\begin{aligned}\ell^2_{\texttt{np}} \mapsto 0 : \text{final } \{\nu : \texttt{int} \mid \nu = 2 * \texttt{x}_1\},\\ 4 : \texttt{ref}(\widetilde{\ell^1}, 0),\\ 8 : \texttt{ref}(\widetilde{\ell^2}, 0)\end{aligned}$$

Recall that $\texttt{x}_1$ is the fresh variable corresponding to `p->size`. The `size` field is marked final to indicate that it is no longer mutated after this assignment, and our system soundly adds the guard

$$!\texttt{np} = 2 * \texttt{x}_1$$

to the environment.

Lines 21 and 22 alter the remaining fields:

$$\begin{aligned}\ell^2_{\texttt{np}} \mapsto 0 : \text{final } \{\nu : \texttt{int} \mid \nu = 2 * \texttt{x}_1\},\\ 4 : \{\nu : \texttt{ref}(\widetilde{\ell^1}, 0) \mid \nu = 0\},\\ 8 : \{\nu : \texttt{ref}(\widetilde{\ell^2}, 0) \mid \nu = 0\}\end{aligned}$$

Finally, we update the type of `p->next` to reflect the assignment of `np`:

$$\begin{aligned}\ell^2_{\texttt{p}} \mapsto 0 : \text{final } \{\nu : \texttt{int} \mid \nu = \texttt{x}_1\},\\ 4 : \{\nu : \texttt{ref}(\widetilde{\ell^1}, 0) \mid \nu = \texttt{x}_2\},\\ 8 : \{\nu : \texttt{ref}(\widetilde{\ell^2}, 0) \mid \nu = \texttt{np}\}\end{aligned}$$

By subtyping and the guard $!\mathtt{np} = 2 * \mathtt{x}_1$, we have

$$\begin{aligned}\ell^2_{\mathrm{p}} \mapsto 0 : \text{final } \{\nu : \mathtt{int} \mid \nu = \mathtt{x}_1\},\\ 4 : \{\nu : \mathtt{ref}(\overset{\sim}{\ell^1}, 0) \mid \nu = \mathtt{x}_2\},\\ 8 : \{\nu : \mathtt{ref}(\overset{\sim}{\ell^2}, 0) \mid \mathtt{x}_1 < !\nu\}\end{aligned}$$

This type is subsumed by the type of $\overset{\sim}{\ell^2}$, and so we have successfully verified that appending a new pool to the end of the memory pool list preserved the sortedness of the list.

Our system is able to verify all the invariants shown so far using the qualifier set $\mathbb{Q}_3$, defined as

$$\mathbb{Q}_3 = \mathbb{Q}_2 \cup \{\nu \neq 0 \Rightarrow \star < !\nu\}.$$

Size Agreement The remaining invariants, that the elements of the free list for each `pool` have `size` fields that match the `size` field of the pool itself and that each `region` in a free list has the same value for its `size` fields, are proved similarly. Thus, using the qualifier set

$$\mathbb{Q}_4 = \mathbb{Q}_3 \cup \{\nu \neq 0 \Rightarrow !\nu = \star\},$$

our system is able to prove all the invariants of this section by inferring that the heap has the following type:

$$\begin{aligned}\overset{\sim}{\ell^1} \mapsto 0 : \text{final } \mathtt{int},\\ 4 : \{\nu : \mathtt{ref}(\overset{\sim}{\ell^1}, 0) \mid @0 = !\nu\},\\ 8^{+1} : \mathtt{char}\\ \overset{\sim}{\ell^2} \mapsto 0 : \text{final } \mathtt{int},\\ 4 : \{\nu : \mathtt{ref}(\overset{\sim}{\ell^1}, 0) \mid @0 = !\nu\},\\ 8 : \{\nu : \mathtt{ref}(\overset{\sim}{\ell^2}, 0) \mid \nu \neq 0 \Rightarrow @0 < !\nu\}\end{aligned}$$

3.3.3 Formal Changes to the NANOC Type System

In this section, we describe the changes to the NANOC type system that are needed to accommodate final fields.

Dereference Expressions in Refinement Predicates We augment our language of refinement predicates with the expression form $!v$, which represents the value obtained by dereferencing the pointer v. To ensure that our type system is sound, our well-formedness rules only allow such dereferences to appear in refinement predicates when v is a pointer to a final field. Formally, this

u	$::=$		**Field Qualifiers**
	$\mid$	ϵ	mutable field (no qualifier)
	$\mid$	final	final field
b	$::=$		**Blocks**
	$\mid$	$\overline{i} : \overline{u\ \tau}$	block

Figure 3.16: Additions to NANOC types to support final fields

Refinement Dereference Well-Formedness $\boxed{\Gamma, h \vDash_! v}$

$$\frac{h = h_0 * \ell \mapsto \ldots,\ i : \text{final}\ \tau,\ \ldots \qquad \Gamma \vdash v : \texttt{ref}(\ell, i)}{\Gamma, h \vDash_! v}\ \text{WF-DEREF}$$

Predicate Well-Formedness $\boxed{\Gamma, h \vDash \phi}$

$$\frac{\phi \text{ well-sorted in } \Gamma, h \qquad !v \in \phi \Rightarrow \Gamma, h \vDash_! v}{\Gamma, h \vDash \phi}\ \text{WF-PRED}$$

Figure 3.17: Determining well-formedness of refinement predicates

obligation is shown with the judgment $\Gamma, h \vDash_! v$ of Figure 3.17, which states that it is acceptable to dereference v under environment Γ and heap h exactly when Γ and h indicate that v is a pointer to a final field within h.

Field Qualifiers A field qualifier u is used to indicate whether a field within a heap-allocated structure may be modified. If a field has no qualifier, it may be modified; a field with the modifier final cannot be modified.

Block Types We modify the form of blocks so that each field consists of a triple of its index, which gives its location within the block, its field qualifier, which determines whether the field may be mutated, and its type. The syntaxes of field qualifiers and block types augmented with field qualifiers are shown in Figure 3.16.

Type Well-Formedness The updated type well-formedness rules of NANOC are given in Figure 3.18. The changes are largely straightforward: because the well-formedness of refinement predicates now depends on the field qualifiers contained in the heap, it is necessary to thread heaps through the well-formedness rules.

Type Well-Formedness $\boxed{\Gamma, h \vDash \tau}$

$$\frac{\Gamma, h \vDash \phi}{\Gamma, h \vDash \{\nu : t \mid \phi\}} \text{ WF-TYPE}$$

Dependent Block Well-Formedness $\boxed{\Gamma, h \vDash_@ b}$

$$\frac{\begin{array}{c}\text{DisjointOffsets}(n : \tau,\ b) \\ x \notin \Gamma,\ \text{FV}(\overline{\tau_j}) \qquad \Gamma, h \vDash \tau \qquad \Gamma, x : \tau, h \vDash_@ \overline{i_j} : \overline{\tau_j[@n \mapsto x]}\end{array}}{\Gamma, h \vDash_@ n : \tau,\ \overline{i_j} : \overline{\tau_j}} \text{ WF-DBLOCK-SINGLE}$$

$$\frac{\Gamma, h \vDash \overline{i_j^+} : \overline{\tau_j}}{\Gamma, h \vDash_@ \overline{i_j^+} : \overline{\tau_j}} \text{ WF-DBLOCK-SEQUENCE}$$

Non-Dependent Block Well-Formedness $\boxed{\Gamma, h \vDash b}$

$$\frac{\text{DisjointOffsets}(\overline{i_j} : \overline{\tau_j}) \qquad \forall j.\Gamma, h \vDash \tau_j}{\Gamma, h \vDash \overline{i_j} : \overline{\tau_j}} \text{ WF-NDBLOCK}$$

Heap Type Well-Formedness $\boxed{\Gamma, h_1 \vDash h_2}$

$$\frac{}{\Gamma, h \vDash \emptyset} \text{ WF-HEMPTY} \qquad \frac{\Gamma, h \vDash h_0 \qquad \widetilde{\ell} \notin \text{dom}(h_0) \qquad \Gamma, h \vDash_@ b}{\Gamma, h \vDash h_0 * \widetilde{\ell} \mapsto b} \text{ WF-HABSTRACT}$$

$$\frac{\Gamma, h \vDash h_0 \qquad \Gamma, h \vDash b \qquad \widetilde{\ell} \in \text{dom}(h_0) \qquad \ell_k \notin \text{dom}(h_0)}{\Gamma, h \vDash h_0 * \ell_j \mapsto b} \text{ WF-HCONCRETE}$$

World Well-Formedness $\boxed{\Gamma \vDash \tau/h}$

$$\frac{\Gamma, h \vDash \tau \qquad \Gamma, h \vDash h}{\Gamma \vDash \tau/h} \text{ WF-WORLD}$$

Function Schema Well-Formedness $\boxed{\vDash \sigma}$

$$\frac{\Gamma = \overline{x_i} : \overline{\tau_i} \qquad \Gamma, h_1 \vDash \overline{\tau_i} \qquad \Gamma \vDash h_1 \qquad \Gamma \vDash \tau/h_2 \qquad \overline{\tau_i}, h_1, \tau, h_2 \text{ abstract}}{\vDash (\overline{x_i} : \overline{\tau_i})/h_1 \ \rightarrow\ \tau/h_2} \text{ WF-FUNSCHEME}$$

Figure 3.18: Rules for well-formedness of NANOC types with final fields

Type Checking The changed and updated type checking rules for NANOC with final fields are shown in Figure 3.19. The rules T-READ-MUTABLE and T-READ-FINAL are straightforward

Expression Typing Rules $\boxed{G,\ \Gamma,\ h \vdash e : \tau / h_2}$

$$\frac{\Gamma \vdash v : \{\nu :\ \texttt{ref}(\ell_j, i)\ \mid\ \text{Safe}(\nu, n)\} \qquad h = h_1 * \ell_j \mapsto \ldots, i : \tau, \ldots \qquad \text{SizeOf}(\tau) = n}{G,\ \Gamma,\ h \vdash *_n v : \tau / h} \text{ T-READ-MUTABLE}$$

$$\frac{\Gamma \vdash v : \{\nu :\ \texttt{ref}(\ell_j, i)\ \mid\ \text{Safe}(\nu, n)\} \qquad h = h_1 * \ell_j \mapsto \ldots, i : \text{final}\ \{\nu :\ t\ \mid\ \phi\}, \ldots \qquad \text{SizeOf}(\tau) = n}{G,\ \Gamma,\ h \vdash *_n v : \{\nu :\ t\ \mid\ \nu = !v \wedge \phi\} / h} \text{ T-READ-FINAL}$$

$$\frac{G,\ \Gamma,\ h_0 * \ell \mapsto b_1,\ i : \text{final}\ \tau_i,\ b_2 \vdash e : \tau_e / h_e \qquad h_e = h' * \ell \mapsto b_1',\ i : \text{final}\ \tau_i',\ b_2' \qquad h_2 = h' * \ell \mapsto b_1',\ i : u\ \tau_i',\ b_2' \qquad \Gamma \vDash \hat{\tau} / \hat{h}_2}{G,\ \Gamma,\ h_0 * \ell \mapsto b_1,\ i : \tau_i,\ b_2 \vdash e : \hat{\tau} / \hat{h}_2} \text{ T-FINALIZE}$$

$$\frac{h = h_0 * \ell_j \mapsto \ldots, i : \{\nu :\ t\ \mid\ \phi\}, \ldots \qquad n \mathrel{\widetilde{\subseteq}} i \qquad \Gamma \vdash v : \texttt{ref}(\ell_j, n) \qquad G,\ \Gamma; \phi[\nu \mapsto !v],\ h \vdash e : \tau / h'}{G,\ \Gamma,\ h \vdash e : \tau / h'} \text{ T-ASSUME-FINAL}$$

$$\frac{\ell_j \text{ fresh} \qquad h = h_0 * \widetilde{\ell} \mapsto b \qquad \Gamma \vDash h * \ell_j \mapsto b \qquad \Gamma \vdash v : \{\nu :\ \texttt{int}(W, i)\ \mid\ \nu \geq 0\} \qquad \tau = \{\nu :\ \texttt{ref}(\ell_j, 0)\ \mid\ \text{Allocated}(\nu, v)\}}{G,\ \Gamma,\ h \vdash \textbf{malloc}(v) : \tau / h * \ell_j \mapsto b_!^0} \text{ T-MALLOC}$$

Figure 3.19: Rules for type checking NANOC expressions with final fields

extensions of the rule T-READ, applied to fields which are mutable or final, respectively. Rule T-READ-MUTABLE is identical to T-READ. Rule T-READ-FINAL strengthens the refinement predicate of the value read from the heap to indicate that it is identical to the value obtained by dereferencing the pointer v.

Rule T-FINALIZE is used to add the field qualifier final to the field at offset i within location ℓ when type checking expression e, rendering that field immutable within the expression and allowing refinement predicates to soundly contain dereferences of pointers to that field. After type checking the body expression e, the field may either be left final, or restored to mutability; this choice is represented by the variable u attached to the field, which indicates that any field qualifier may be used for the field at offset i within ℓ. In the case where the field is made mutable again after type checking the body, soundness requires that we ensure that there are no refinement predicates in scope that may dereference pointers to the field. Since the field was mutable before this expression, such dereferences cannot be contained in Γ or the input heap, and so may only occur in τ or h_2; thus, it suffices to check that the output type and heap are well-formed when the field is made mutable again.

For simplicity of implementation, we generally use T-FINALIZE to finalize a field for the remainder of a function; that is, we do not make a field mutable again after it has been finalized. The sole exception is when a function whose heap indicates a field is mutable calls a function which does not mutate that field, and so whose function type assigns the final qualifier to that field. In this case, we wrap the call using T-FINALIZE to make the field immutable solely for the duration of the call, restoring the field to mutability as soon as the callee returns.

Rule T-ASSUME-FINAL is used to introduce assertions about the values obtained by dereferencing pointers into the environment when type checking an expression. Note that the assertion added to the environment is over a value v which points to a *concrete* location in the heap, guaranteeing that it refers to a single (non-null) run-time location.

Rule T-MALLOC changes in two significant ways. First, the block returned is $b^0_!$, defined as

$$
\begin{aligned}
(n : u\ \{\nu :\ \texttt{int}(w,i)\ \mid\ \phi\}, b)^0_! &= n : \{\nu :\ \texttt{int}(w,0)\ \mid\ \nu = 0_{|w|}\}, b^0_! \\
(n : u\ \{\nu :\ \texttt{ref}(\ell,i)\ \mid\ \phi\}, b)^0_! &= n : \{\nu :\ \texttt{ref}(\widetilde{\ell},0)\ \mid\ \nu = 0\}, b^0_! \\
(i^+ : u\ \{\nu :\ \texttt{int}(w,i)\ \mid\ \phi\}, b)^0_! &= i^+ : \{\nu :\ \texttt{int}(w,i \sqcup 0)\ \mid\ \phi \vee \nu = 0_{|w|}\}, b^0_! \\
(i^+ : u\ \{\nu :\ \texttt{ref}(\ell,i)\ \mid\ \phi\}, b)^0_! &= i^+ : \{\nu :\ \texttt{ref}(\widetilde{\ell},i \sqcup 0)\ \mid\ \phi \vee \nu = 0\}, b^0_!
\end{aligned}
$$

The key difference from b^0 is that all field qualifiers are removed from the fields, meaning that every field in the newly-allocated location is mutable. This does not cause any problems with soundness, as the location is newly-allocated; to be unsound, there would have to be an existing pointer whose refinement type contains a dereference targeting one of the fields in the newly-allocated region, which is impossible.

Second, we omit the well-formedness requirement, which allows us to have an existing pointer and any number of freshly-allocated pointers all unfolded simultaneously. This is sound, as it is impossible for a freshly-allocated pointer to alias with any existing pointer. (While this change is independent of the changes related to final fields, we have deferred it to this point because it complicates the theory and we do not prove it sound.)

Soundness While we believe the rules presented to be sound, we do not make any formal claims of their soundness. In particular, the soundness proof of Appendix C does not consider the rules presented in this section.

3.4 Type Inference

Next, we give a brief overview of type inference in NANOC. Type inference occurs in three phases. The first infers physical types for each function in the program. The second inserts location fold and unfold operations where necessary. The third and final step infers refinement types using liquid type inference.

3.4.1 Physical Type Inference

In the previous chapter, we based our type inference technique on the rich type information provided by ML's type system. Because C programs are essentially untyped, we first use a basic type inference pass to assign rich physical types to local variables and expressions and to discover the types of the heap's contents.

We first use the declared C types of all functions in the program to generate a corresponding physical type schema for each function. This process is largely automatic and rarely requires annotations to be added. The generated function schemas are then used to infer physical types for local variables and the heap. This process occurs in two phases: First, our algorithm infers physical types for local variables, assuming that the types of values stored in the heap conform to the declared types of the pointers used to access the values. Physical type inference for local variables works by first assigning each local variable a physical type with an index variable representing its as-yet-unknown index. The algorithm then traverses the body of the function in a syntax-directed manner and emits subtyping constraints capturing the flow of data within the function. The algorithm then uses a fixpoint algorithm to solve the constraints over the index domain to find an assignment of index variables to indices so that the function is typable given the assumptions made about the heap.

After the types of local variables have been inferred, the function's (abstract) heap type is inferred by traversing the function and incrementally building the heap type as heap read and write instructions are encountered. This phase also performs a field-sensitive may alias analysis to determine which locations exist in the heap and to assign location names to values of reference type.

The type assumptions made in the first phase are checked in an assume-guarantee fashion by emitting appropriate deferred type checks that are enforced by the refinement type system.

3.4.2 Fold and Unfold Inference

After physical type inference, our system automatically inserts location fold and unfold expressions in order to ensure that every dereference is on a concrete pointer and that only one concrete location is unfolded at a time, as required by our typing rules. The appropriate spots to insert folds and unfolds are determined by a forward dataflow analysis that works as follows: At the beginning of the function, all heap locations are in a folded state, i.e., there are no concrete locations in the heap. The algorithm traverses each block in the control flow graph in order, inserting an unfold before any access to a pointer which has not already been unfolded, and inserting any necessary folds if another pointer which may alias the one being unfolded is already unfolded. The analysis tracks which pointers are currently unfolded at the exit of each block; if two different unfolded pointers to the same abstract location can reach the entry of a

block — for example, if two different pointers are unfolded along the `then` and `else` branches of an `if` statement — the pointers are folded at the exits of their respective blocks. The algorithm iterates until it reaches a fixed point.

3.4.3 Final Field Inference

Type inference with final fields has two components. First, we must determine which fields within the heap are final at each program point. Second, we must determine where to use the non-syntax-directed rules T-FINALIZE and T-ASSUME-FINAL.

Determining Final Fields Per Program Point We use an interprocedural dataflow analysis to discover, at each program point, which fields in the heap are *final*, i.e., will not be mutated from that program point on. The intraprocedural component is a straightforward backwards dataflow analysis. The analysis begins with all fields final at the exits of the function, then traverses the control flow graph backwards from the function exits. At each program statement, the analysis computes the set of final fields at the start of the statement from the set of final fields at the exit of the statement by removing any fields which were assigned in the current statement; which fields are assigned within the statement is determined using the inferred physical type information. At each control flow join point, the set of final fields at a control flow graph node is taken to be the intersection of the final fields at the entry to each of its successors.

The interprocedural analysis simply computes per-function final fields information and runs the intraprocedural analysis. The process is iterated to a fixed point.

Inferring uses of T-FINALIZE and T-ASSUME-FINAL Next, we use the inferred final fields information to determine where the non-syntax directed rules T-FINALIZE and T-ASSUME-FINAL may be used. Rule T-FINALIZE is invoked in two places. First, whenever a field is written and its value remains final for the remainder of the function body, T-FINALIZE is used to mark that field as final. Second, whenever a call is made from a function where a field is mutable to a function whose input heap marks the field as final, T-FINALIZE is used to wrap the call, allowing the function to be called in spite of the fact that the field is not final in the caller.

Uses of T-ASSUME-FINAL also appear in two places. First, whenever a field is written and finalized with T-FINALIZE, T-ASSUME-FINAL is used to record the relationship between the pointer that was written through and the value that was just written. Second, T-ASSUME-FINAL is used after invocations of T-UNFOLD to assert the relationship between the unfolded pointer and the values of those of its final fields which occur at singleton offsets.

3.4.4 Refinement Inference

Finally, we use liquid type inference, as described in the previous chapter, to infer refinement types thus automatically discover data structure invariants. As before, we observe

that our type checking rules encode an algorithm for type inference and so we perform type inference by attempting to produce a type derivation. At various points in the derivation, we encounter types, heaps, and function schemas which cannot be synthesized directly from the form of the expression and the current environment, but instead must be inferred. We insist that such types, blocks, heaps, and schemas be liquid, denoted $\hat{\tau}$ (respectively, $\hat{b}$, $\hat{h}$, $\hat{\sigma}$), i.e., their refinements must be liquid refinements consisting of a conjunction of logical qualifiers, as described in the previous chapter. Whenever we encounter a type which must be inferred, we create a new template type, which is the physical type inferred earlier where a fresh variable is used to represent the as-yet-unknown liquid refinement. As in the previous chapter, we generate subtyping constraints over the template types using the subtyping premises in our type rules; the subtyping rules are used to reduce these constraints to simple implication constraints between refinement predicates and unknown refinement variables. These constraints are solved to yield a liquid refinement for each refinement variable; replacing each variable with its solution yields a refinement typing for the program.

3.5 Implementation and Evaluation

We implemented our type system in CSOLVE, a prototype static verifier for C programs. In this section, we describe the architecture of CSOLVE and the results of applying CSOLVE to a variety of challenging benchmarks.

3.5.1 CSOLVE: Liquid Types for C

Below, we briefly describe the architecture and usage of CSOLVE. CSOLVE takes as input a C source file and a set of logical qualifiers, which CSOLVE uses to perform liquid type inference. CSOLVE then outputs the inferred liquid types of functions, local variables, and heap locations and reports any refinement type errors that occur.

Architecture Type inference in CSOLVE is split into four phases. In the first phase, the input C program is read by CIL [68], which generates an AST. This AST is then simplified in various ways, the most significant of which is that the code is transformed to SSA so that local variables are never mutated. The second phase generates physical types for each declared function and global variable and checks that the program code respects these types. The third phase walks the CIL AST and assigns each expression and variable in the program a refinement type with a distinct refinement predicate variable representing its as-yet-unknown refinement predicate, as well as generating subtyping constraints over these refinement types such that solving for the refinement variables within the constraints yields a valid typing for the program, in the same style as the ConsGen function of section 2.3. The fourth phase attempts to solve the subtyping

constraints using a fixpoint procedure based on predicate abstraction, using the Z3 SMT solver [28] to discharge the logical validity queries that arise in constraint solving.

Input CSOLVE takes as input a C source code file and a file specifying the set of logical qualifiers to use in refinement inference. Qualifiers are also read from a standard library of qualifiers that have proven to be useful on a large variety of programs, further easing the programmer's annotation burden.

Output If the program is type-safe, CSOLVE outputs "Safe". Otherwise, the program may be type-unsafe, according to either the physical type system or the refinement type system. In either case, for each error, CSOLVE prints the name of the file and line number where the error occurs, as well as a description of the error. In the case where the error is in refinement type inference, CSOLVE prints the subtyping constraints which cannot be solved. Whether the program type checks or not, CSOLVE produces a mapping of program identifiers to their inferred types, which can be viewed using the tag browsing facilities provided by common editors, e.g., Vim and Emacs.

Compatibility With C Infrastructure Thanks to the infrastructure provided by CIL, CSOLVE is able to work as a drop-in replacement for GCC. Hence, to check a multi-file program one need only construct or slightly modify a makefile which builds the program from source.

Modular Type Checking If the user specifies a type for a function with the `extern` keyword, CSOLVE will use the provided type when checking the current source file, allowing the user to omit the body of the external function. This allows for modular type checking and, by abstracting away the details of other source files, it permits the user to work around cases where a function may be too complex for CSOLVE to type check.

3.5.2 Memory Safety Benchmarks

We applied CSOLVE to several challenging benchmarks, drawn from [11], [60], [73], and the example of section 3.1, which illustrate common low-level coding idioms. The results are shown in Table 3.1. In each case, CSOLVE was able to precisely reason about complex invariants of in-heap data structures and memory access patterns to statically verify memory safety by proving the absence of null pointer dereferences and array bounds violations. (In the case of `ft`, we show only array bounds safety; see chapter 4.) We explain several of the benchmarks below.

String Lists Using CSOLVE, we verified the safety of a program implementing a C idiom for linked list manipulation which is particularly common in operating system code [18] and which requires precise reasoning about pointer arithmetic. Recall the example of section 3.1, which contained functions for creating and initializing strings and for creating lists of strings. We

add to that example the function `succ`, shown below, which takes a pointer to the `str` field of a `stringlist` and returns the next `string` in the list. (Explicit null checks checks have been omitted for brevity.) This function is used in `init_succ`, which creates a list of several strings and initializes the second one using `init_string`. CSOLVE precisely tracks pointer arithmetic to verify `init_succ`, by proving that that the input to `init_string` has the type from section 3.1.

```
slist *string_succ (string **s) {
1: slist *parent = (slist **) s - 1;
2: return parent->next->s;
}

void init_succ () {
  slist  *sl;
  string *succ;
  sl   = new_strings (3);
  succ = string_succ (&sl1->s);
  init_string (succ, '\0');
}
```

The `string_succ` function expects an argument `s` of type $\texttt{ref}(\widetilde{\ell^1}, 4)$ in a heap of the form

$$\widetilde{\ell^1} \mapsto 0 : \texttt{ref}(\widetilde{\ell^1}, 0),\ 4 : \texttt{ref}(\widetilde{\ell^2}, 0)$$
$$\widetilde{\ell^2} \mapsto 0 : \{\nu : \texttt{int} \mid 0 \leq \nu\},\ 4 : \{\nu : \texttt{ref}(\widetilde{\ell^3}, 0) \mid \text{BLen}(\nu, @0)\}$$
$$\widetilde{\ell^3} \mapsto 0^{+1} : \texttt{char}.$$

From section 3.1, we know that the return type of `new_strings` provides a pointer of this type, assigned to `sl`, in the appropriate heap. Thus, we begin in `succ` with the assignment to `parent` on line 1. Since `s` is cast to a `stringlist`$*$, which is 4 bytes long, and decremented, the type of the pointer assigned to `parent` is $\texttt{ref}(\widetilde{\ell^1}, 0)$. Continuing on line 2, the type of `parent->next` is the same, since the `next` pointer points to a structure of the same type. Finally, the type of `parent->next->s` is given by the type at offset 4 of $\widetilde{\ell^1}$, since `s` is the second item in the `stringlist` structure. Thus, `string_succ` returns a pointer of type $\texttt{ref}(\widetilde{\ell^2}, 0)$ — a pointer to a `string` — in a heap of the form shown above. This pointer is passed to `init_string`; as the pointer and heap meet the required invariants, CSOLVE verifies safety. Thus, CSOLVE precisely reasons about pointers and in-heap data structures and automatically verifies this example using the qualifiers Q from section 3.1.

Audio Compression Using CSOLVE, we verified the memory safety of routines for ADPCM audio encoding and decoding. The encoder, outlined below, takes as input an audio stream consisting of an array of 16-bit samples and outputs a compressed stream using 4 bits to represent each sample. The encoder relies on complex loop invariants to ensure memory safety.

Table 3.1: **Results. Lines** is the number of source lines without comments. **Qualifiers** is the number of manually-provided logical qualifiers used. **Assumes** is the number of manual assumptions inserted. **Time (s)** is the time in seconds CSOLVE requires to verify safety.

Program	Lines	Qualifiers	Assumes	Time (s)
stringlist	72	1	0	2
strcpy	77	3	0	4
adpcm	198	13	0	42
pmap	250	3	0	34
mst	309	1	0	16
power	620	7	2	111
ft	652	2	6	310
ks	742	9	7	721
Total	2,920	39	15	1,240

```
void encoder (int nsamp, short *inz, char *outz) {
  short *in          = inz;
  char  *out         = outz;
  int    bufferempty = 1;
  char   buffer;
  for (int len = nsamp; 0 < len; len--) {
    // Read an input sample
    ... *in++ ...;
    if (!bufferempty) {
      // Write to buffer elided
      *out++ = buffer;
    } else {
      // Write to buffer elided
    }
    bufferempty = !bufferempty;
  }
  if (!bufferempty) *out++ = buffer;
}
```

The encoder takes three parameters: `nsamp`, the total number of samples in the input; `inz`, a pointer to the start of the input buffer, an array of 16-bit `short` values; and `outz`, a pointer to the output buffer, an array of 8-bit `char` values. The number of elements in the input buffer is twice the number of elements in the output buffer. The pointer `in`, initially set to `inz`, is used to read data from the input buffer; the pointer `out`, initially set to `outz`, is used to write data to the output buffer. The `for` loop iterates through each element of the input buffer. At each iteration, the loop reads 16 bits (a single `short` value) from the input buffer and advances `in`. Each iteration also computes a new 4-bit value for the output; however, since `out` is a `char` pointer, the encoder must write 8 bits at a time. Thus, the encoder buffers output into a local `char` value and only writes to `out` every other iteration. The flag `bufferempty` indicates whether to write to and advance `out`. The final `if` writes to the output in case there is a value in the buffer which has not been written, i.e., if there are an odd number of samples in the input.

CSOLVE verifies the safety of dereferences of `in` and `out`, by inferring that `in` and `out` have the respective types

$$\{\nu = \texttt{inz} + \texttt{nsamp} - \texttt{len}\}$$
$$\{2 * (\nu - \texttt{outz}) = \texttt{nsamp} - \texttt{len} - (1 - \texttt{bufferempty})\}$$

which encode the crucial loop-invariants that relate the values of the respective pointers with the number of iterations and the flag. By inferring similar invariants CSOLVE verifies the decoding routine.

Virtual Memory Using CSOLVE, we verified the array safety of `pmap`, a 317-line program implementing a virtual memory subsystem of the JOS OS kernel [11] that comprises functions for allocating and freeing virtual address spaces, allocating and freeing a physical page backing a virtual page, and mapping two virtual pages onto the same physical page.

To ensure the safety of array accesses in `pmap` we must precisely reason about the values contained in the collection of environment structures that represents virtual address spaces. Each environment includes a mapping from virtual pages to physical pages, `envpgdir`, represented as an array of fixed length. Each index of `envpgdir` is mapped to either the physical page allocated to the virtual page or -1 if no physical page has been allocated. Environments are joined together in doubly-linked fashion to form a list of virtual address spaces.

The physical address space is described by the array `pages`, which contains N page entries. Operations like allocating and freeing physical pages use entries from an `envpgdir` field to index into `pages`. Thus, to prove array safety, we must verify that the items in *every* `envpgdir` in *every* environment are valid indices into `pages`. Formally, we must verify that every pointer to an environment points to a heap location $\widetilde{\ell}$ whose description is

$$\widetilde{\ell} \mapsto 0 : \texttt{ref}(\widetilde{\ell}, 0),\ 4 : \texttt{ref}(\widetilde{\ell}, 0),\ 8^{+4} : \{\nu : \texttt{int} \mid \nu < N\}$$

where the pointers at offsets 0 and 4 are pointers to the next and previous environments, respectively, and the integers at indices in 8^{+4} are the entries in `envpgdir`. Note that we cannot prove that every entry in `envpgdir` is non-negative, as -1 is used to indicate an unused virtual page. However, every item in `envpgdir` is verified to be non-negative before use as an index into `pages`.

Using CSOLVE, we were able to verify that the above heap typing holds and thus determine that every array access in `pmap` is within bounds. This is challenging because the majority of array accesses are indirect, using an entry in an `envpgdir` field to index into an array of physical page data. This requires precise reasoning about the values of all elements contained in an in-heap data structure. Further, array offsets are frequently checked for validity in a different function from the one in which they are used to access an array, requiring flow-sensitive reasoning about values across function boundaries. Nevertheless, CSOLVE is able to verify the safety of all array accesses in `pmap`.

3.5.3 Data Structure Benchmarks

Using CSOLVE, we verified several functions for manipulating sorted singly- and doubly-linked lists in `glib` [78], a widely-used open source library. We specialize the list implementations, which use void pointers to implement a sort of polymorphism, to instead operate only on integers; we leave the issue of polymorphism to future work.

Singly-Linked Sorted Lists The `glib` library contains functions for manipulating sorted singly-linked lists by inserting elements, removing an element identified either by a pointer to the element or by its contents, and finding the n-th element of the list. We added a function for checking the invariant that the list is sorted and a driver which exercises the library by constructing and manipulating a sorted list using all of the above functions; together, the program totals 182 physical (non-comment, non-blank) lines. CSOLVE is able to infer the sortedness invariant and verify memory safety using 3 user-provided qualifiers in 6 seconds.

Doubly-Linked Sorted Lists The `glib` library also contains functions for manipulating sorted doubly-linked lists; these include analogues of the mentioned functions for manipulating singly-linked lists, as well as a function for retrieving the n-th previous element from a list element. As before, we add a driver function which exercises the library; the total size of the program is 138 physical lines. Verifying the sorted insertion function required modifying two lines and inserting a single trusted annotation. With this small modification, CSOLVE is able to infer sortedness invariant and verify memory safety using 4 user-provided qualifiers in 9 seconds.

We briefly explain the nature of the annotation. The sorted list insertion function uses a loop to iterate through the list in order using a pointer `p` until it reaches the node after which the new element should be inserted. The loop has the invariant `p->prev->data` $\leq$ `data`, where `data` is the new value to be inserted. As the `prev` field of `p` is modified by the function after the loop, this loop invariant is not expressible as a refinement that explicitly dereferences `p->prev`. Instead, we explicitly assume the loop invariant where it is used.

Memory Manager We applied CSOLVE to the memory manager example of Section 3.3.1, along with a driver function which exercises the allocator and checks the invariants of the free list and allocated regions, altogether 84 physical lines of code. CSOLVE verified the memory safety of all field accesses (which are simply memory dereferences) within the program and statically guarantees that all asserts used to check invariants will pass. We used the qualifiers $\mathbb{Q}_4$ of Section 3.3.1, 4 of which are in CSOLVE's standard library, along with one additional qualifier required for proving memory safety. Thus, with 6 user-provided qualifiers, CSOLVE was able to infer the invariants of the example and verify its safety in 9 seconds.

Acknowledgements

This chapter contains material adapted from the following publications:

Patrick Rondon, Ming Kawaguchi, Ranjit Jhala. "Low-Level Liquid Types", *Proceedings of the 2010 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 131–144, 2010.

Patrick Rondon, Alexander Bakst, Ming Kawaguchi, Ranjit Jhala. "CSolve: Verifying C with Liquid Types", *Proceedings of Computer Aided Verification 2012 (CAV)*, pages 744–750, 2012.

Chapter 4

Conclusions and Future Work

In this dissertation, we have shown that the liquid types refinement type inference technique allows for the highly-automated verification of safety properties in programs written in both high-level, pure functional languages and low-level imperative languages. We implemented the liquid types technique in two tools, DSOLVE and CSOLVE, which perform refinement type inference for OCaml and C programs, respectively, and showed, through a number of benchmarks taken both from the literature and from the wild, that liquid types enables the verification of safety and correctness properties of real-world programs while imposing a very small annotation burden. We see a number of directions for future work to both increase the expressiveness of our type systems and bring liquid type inference into new domains.

4.1 Polymorphism

Presently, our low-level type system lacks several forms of polymorphism which would be useful in verifying real-world programs.

Void Pointer Polymorphism Our current low-level type system has no support for any form of type or heap polymorphism. Nonetheless, it is common for C programmers to write functions which are polymorphic in their arguments or in the contents of their heaps; in particular, large libraries like `glib` make use of polymorphism to implement generic data structures and algorithms. Since the C type system does not include any form of polymorphism, polymorphism is simulated by using pointers to the `void` type and inserting casts as appropriate. Scaling to large C programs which take advantage of the generic data structures and algorithms will require accommodating this style of polymorphism.

Structural Subtyping Some C programs take advantage of *structural subtyping*: a function expects a pointer to a data structure and is called with pointers to "subtypes" of that structure

which may contain additional fields that are not accessed by the function. Because the callee may modify some fields of the structure, it is not sound to keep the refinements on the untouched fields as they were before the call, since they may depend on the modified fields. On the other hand, eliminating these refinements could lead to unnecessary losses in precision when fields are read but not written. Future versions of our low-level type system should accommodate the combination of mutability, dependent refinements, and structural subtyping.

4.2 Flow-Sensitive Invariants

Our low-level type system allows flow-sensitive strong updates to the type of a *single* member of an in-memory data structure. However, the type of the *whole* data structure is flow-invariant: each individual element must reestablish the data structure's type before the next member of the structure is accessed.

For example, suppose that a list contains cells each of which has a data field with the value 0, and suppose that an loop iterates over the list and sets each data field to 1. Our system can only verify that at all points in time, each cell has the value 0 or 1. In particular, our system cannot determine that before the loop, the data fields have the value 0, while after the loop the data fields have the value 1. An important area of future work will be to capture and infer such flow-sensitive invariants; more example target invariants follow.

Modeling Deallocation Currently, NANOC does not model deallocation: we assume that a location, once allocated, is allocated for the duration of program execution. This assumption is rarely justified in real-world low-level programs, which manually free memory when they are no longer using it, so that our analysis does not guarantee the absence of use-after-free or double free errors. Thus, a verifier which hopes to make strong guarantees about the absence of such errors will need to track whether each region of memory is *currently* allocated.

Array Initialization Our low-level type system allows strong updates to the type of a *single* elements within a data structure, but only allows weak updates to the type of *all elements* of the data structure. In a similar vein, it only allows weak updates to the type of an array's contents. Suppose that we have an array of pointers which is allocated using `malloc`. Initially, all pointers within the array are null. The array is then initialized so that all pointers are non-null. At present, our type system can only determine that, after initialization, all pointers within the array *may* be null — but it cannot verify that they *must* be non-null. Handling such array initializations will be an important direction for future work.

4.3 Liquid Types for Dynamic Languages

The work of Chugh et al. [15] demonstrates how a refinement type system can be used to verify the type safety of sophisticated higher-order dynamic language programs which operate on dictionaries with dynamically-determined keys. While their work gives an algorithm for type checking such programs, they do not give an inference algorithm. An exciting area of future work would adapt the refinement type inference techniques shown here to the setting of dynamic languages like Javascript and Python.

Bibliography

[1] Tilak Agerwala and Jayadev Misra. Assertion graphs for verifying and synthesizing programs. Technical Report 83, University of Texas, Austin, 1978.

[2] Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^3: A linear language with locations. *Fundam. Inf.*, 77(4):397–449, December 2007.

[3] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 129–140, New York, NY, USA, 2003. ACM.

[4] Lennart Augustsson. Cayenne — a language with dependent types. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 239–250, New York, NY, USA, 1998. ACM.

[5] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 1–3, New York, NY, USA, 2002. ACM.

[6] João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. Polymorphic contracts. In *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software*, ESOP'11/ETAPS'11, pages 18–37, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, CSF '08, pages 17–32, Washington, DC, USA, 2008. IEEE Computer Society.

[8] Yves Bertot and Pierre Castéran. Interactive theorem proving and program development. coq'art: The calculus of inductive constructions, 2004.

[9] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 196–207, New York, NY, USA, 2003. ACM.

[10] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *Proceedings of the 13th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, pages 1–22, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] Josh Cates, Frans Kaashoek, and Emil Sit. The JOS operating system. http://pdos.csail.mit.edu/.

[12] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004.

[13] Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. A reachability predicate for analyzing low-level software. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'07, pages 19–33, Berlin, Heidelberg, 2007. Springer-Verlag.

[14] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 234–245, New York, NY, USA, 2011. ACM.

[15] Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested refinements: a logic for duck typing. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 231–244, New York, NY, USA, 2012. ACM.

[16] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[17] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 302–314, New York, NY, USA, 2009. ACM.

[18] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 302–314, New York, NY, USA, 2009. ACM.

[19] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 520–535, Berlin, Heidelberg, 2007. Springer-Verlag.

[20] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 232–244, New York, NY, USA, 2003. ACM.

[21] R.L. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[22] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[23] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Conference Record of the 38$^{\text{th}}$ Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 105–118, Austin, Texas, January 26-28 2011. ACM Press, New York.

[24] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[25] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[26] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[27] Rowan Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005.

[28] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[30] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Proceedings of the 12th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 287–302, Berlin, Heidelberg, 2006. Springer-Verlag.

[31] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2007.

[32] Joshua Dunfield. Refined typechecking with stardust. In *Proceedings of the 2007 workshop on Programming languages meets program verification*, PLPV '07, pages 21–32, New York, NY, USA, 2007. ACM.

[33] B. Dutertre and L. De Moura. Yices SMT solver. http://yices.csl.sri.com/.

[34] Manuel Fahndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 13–24, New York, NY, USA, 2002. ACM.

[35] Cormac Flanagan. Hybrid type checking. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 245–256, New York, NY, USA, 2006. ACM.

[36] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Inf. Process. Lett.*, 77(2-4):97–108, February 2001.

[37] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 191–202, New York, NY, USA, 2002. ACM.

[38] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.

[39] R.W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[40] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 1–12, New York, NY, USA, 2002. ACM.

[41] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM.

[42] Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 338–350, New York, NY, USA, 2005. ACM.

[43] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83. Springer-Verlag, London, UK, UK, 1997.

[44] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[45] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 235–246, New York, NY, USA, 2008. ACM.

[46] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 232–244, New York, NY, USA, 2004. ACM.

[47] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *Proceedings of the 10th international conference on Model checking software*, SPIN'03, pages 235–239, Berlin, Heidelberg, 2003. Springer-Verlag.

[48] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[49] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.

[50] Himanshu Jain, Franjo Ivančić, Aarti Gupta, Ilya Shlyakhter, and Chao Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *Proceedings of the 18th international conference on Computer Aided Verification*, CAV'06, pages 137–151, Berlin, Heidelberg, 2006. Springer-Verlag.

[51] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[52] Ming Kawaguchi, Patrick Rondon, and Ranjit Jhala. Type-based data structure verification. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 304–315, New York, NY, USA, 2009. ACM.

[53] Kenneth Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 505–519, Berlin, Heidelberg, 2007. Springer-Verlag.

[54] Kenneth Knowles and Cormac Flanagan. Compositional reasoning and decidable checking for dependent contract types. In *Proceedings of the 3rd workshop on Programming languages meets program verification*, PLPV '09, pages 27–38, New York, NY, USA, 2008. ACM.

[55] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34, February 2010.

[56] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 416–428, New York, NY, USA, 2009. ACM.

[57] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 222–233, New York, NY, USA, 2011. ACM.

[58] Shuvendu K. Lahiri and Randal E. Bryant. Predicate abstraction with indexed predicates. volume 9, New York, NY, USA, December 2007. ACM.

[59] Shuvendu K. Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 115–126, New York, NY, USA, 2006. ACM.

[60] Chunho Lee, M. Potkonjak, and W.H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. *Microarchitecture, IEEE/ACM International Symposium on*, 0:330, 1997.

[61] K. Rustan Leino, Peter Müller, and Angela Wallenburg. Flexible immutability with frozen objects. In *Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, VSTTE '08, pages 192–208, Berlin, Heidelberg, 2008. Springer-Verlag.

[62] Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In *Proceedings of the 7th International Symposium on Static Analysis*, SAS '00, pages 280–301. Springer-Verlag, London, UK, UK, 2000.

[63] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In *Proceedings of the 17th international conference on Computer Aided Verification*, CAV'05, pages 476–490, Berlin, Heidelberg, 2005. Springer-Verlag.

[64] Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, LCTES '06, pages 54–63, New York, NY, USA, 2006. ACM.

[65] Antoine Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19(1):31–100, March 2006.

[66] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 221–231, New York, NY, USA, 2001. ACM.

[67] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 229–240, New York, NY, USA, 2008. ACM.

[68] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC 02: Compiler Construction*, Lecture Notes in Computer Science 2304, pages 213–228. Springer, 2002.

[69] G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[70] Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 457–474, New York, NY, USA, 2008. ACM.

[71] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *IFIP TCS*, pages 437–450, 2004.

[72] Andreas Podelski and Thomas Wies. Boolean heaps. In *Proceedings of the 12th international conference on Static Analysis*, SAS'05, pages 268–283, Berlin, Heidelberg, 2005. Springer-Verlag.

[73] The GNU Project. GNU Coreutils. http://www.gnu.org/software/coreutils/.

[74] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[75] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.

[76] Swaroop Sridhar, Jonathan S. Shapiro, and Scott F. Smith. Sound and complete type inference for a systems programming language. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 290–306, Berlin, Heidelberg, 2008. Springer-Verlag.

[77] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 223–234, New York, NY, USA, 2009. ACM.

[78] The GTK+ Team. glib. http://www.gtk.org/.

[79] Tachio Terauchi. Dependent types from counterexamples. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 119–130, New York, NY, USA, 2010. ACM.

[80] David Walker and J. Gregory Morrisett. Alias types for recursive data structures. In *Selected papers from the Third International Workshop on Types in Compilation*, TIC '00, pages 177–206, London, UK, UK, 2001. Springer-Verlag.

[81] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 1–12, New York, NY, USA, 1995. ACM.

[82] H. Xi. DML code examples. http://www.cs.bu.edu/fac/hwxi/DML/.

[83] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 249–257, New York, NY, USA, 1998. ACM.

[84] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 214–227, New York, NY, USA, 1999. ACM.

[85] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 351–363, New York, NY, USA, 2005. ACM.

[86] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 385–398, Berlin, Heidelberg, 2008. Springer-Verlag.

[87] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 349–361, New York, NY, USA, 2008. ACM.

[88] Dengping Zhu and Hongwei Xi. Safe programming with pointers through stateful views. In *Proceedings of the 7th international conference on Practical Aspects of Declarative Languages*, PADL'05, pages 83–97, Berlin, Heidelberg, 2005. Springer-Verlag.

Appendix A

Correctness of Liquid Type Inference

In this chapter, we prove that the liquid type inference algorithm for λ_L given in section 2.3 is sound and complete. That is, we prove the following: if our algorithm infers a type for a program, then the program does truly satisfy that type according to the declarative typing rules (soundness), and if our algorithm fails to infer a type for a program with a given set of qualifiers, then there is no liquid type over that set of qualifiers that can be ascribed to the program (completeness).

As our refinement-typed λ calculus, λ_L, is relatively standard, we refer readers interested in a proof of soundness for the refinement type system to other work — for example, by Gordon et al.

We begin with some standard assumptions about the declarative refinement type system and underlying refinement logic:

Assumption 3. *(Well-Formedness) If* $\Gamma \vdash e : \hat{\sigma}$ *then* $\Gamma \vDash \hat{\sigma}$.

Assumption 4. *(Weakening) If*

$$\begin{aligned} \Gamma &= \Gamma_1; \Gamma_2 \\ \Gamma' &= \Gamma_1; x : \hat{\sigma}_x; \Gamma_2 \\ x &\notin \mathrm{FV}(\Gamma_2) \end{aligned}$$

then:

1. *if* $\Gamma \vDash e_1 \Rightarrow e_2$ *then* $\Gamma' \vDash e_1 \Rightarrow e_2$,
2. *if* $\Gamma \vdash \hat{\sigma}_1 <: \hat{\sigma}_2$ *then* $\Gamma' \vdash \hat{\sigma}_1 <: \hat{\sigma}_2$,

3. *if* $\Gamma \vDash \hat{\sigma}$ *then* $\Gamma' \vDash \hat{\sigma}$,

4. *if* $\Gamma \vdash_{\mathbb{Q}} e : \hat{\sigma}$ *then* $\Gamma' \vdash_{\mathbb{Q}} e : \hat{\sigma}$.

Assumption 5. *(Guard Weakening) If*

$$\Gamma = \Gamma_1; \Gamma_2$$
$$\Gamma' = \Gamma_1; \phi; \Gamma_2$$

then,

1. *if* $\Gamma \vDash e_1 \Rightarrow e_2$ *then* $\Gamma' \vDash e_1 \Rightarrow e_2$,

2. *if* $\Gamma \vdash \hat{\sigma}_1 <: \hat{\sigma}_2$ *then* $\Gamma' \vdash \hat{\sigma}_1 <: \hat{\sigma}_2$,

3. *if* $\Gamma \vDash \hat{\sigma}$ *then* $\Gamma' \vDash \hat{\sigma}$,

4. *if* $\Gamma \vdash_{\mathbb{Q}} e : \hat{\sigma}$ *then* $\Gamma' \vdash_{\mathbb{Q}} e : \hat{\sigma}$.

Assumption 6. *(Subtyping Reflexive Transitive)*

1. *if* $\Gamma \vDash \hat{\sigma}$ *then* $\Gamma \vdash \hat{\sigma} <: \hat{\sigma}$,

2. *if* $\Gamma \vDash e_1 \Rightarrow e_2$ *and* $\Gamma \vDash e_2 \Rightarrow e_3$ *then* $\Gamma \vDash e_1 \Rightarrow e_3$,

3. *if* $\Gamma \vdash \hat{\sigma}_1 <: \hat{\sigma}_2$ *and* $\Gamma \vdash \hat{\sigma}_2 <: \hat{\sigma}_3$ *then* $\Gamma \vdash \hat{\sigma}_1 <: \hat{\sigma}_3$.

Theorem 3. *(Soundness of Decidable Checking) If* $\Gamma \vdash_{\mathbb{Q}} e : \hat{\sigma}$ *then* $\Gamma \vdash e : \hat{\sigma}$.

Proof. By induction on the structure of the derivation of $\Gamma \vdash_{\mathbb{Q}} e : \hat{\sigma}$. The key observations are that each liquid type (schema) is also a dependent type schema and that each liquid type deriviation rule [LT-*] has a matching refinement type derivation rule. □

Lemma 1. *(Fresh) For each type schema* $\dot{\sigma}$ *and assignment* A *over* $\mathbb{Q}$, $A(\text{Fresh}(\dot{\sigma}))$ *is a liquid type over* $\mathbb{Q}$.

Proof. By induction on the structure of $\dot{\sigma}$ □

Lemma 2. *(Shape) For every liquid type assignment* A*:*

1. $\text{Shape}(T) = \text{Shape}(AT)$,

2. $\text{Shape}(\Gamma) = \text{Shape}(A\Gamma)$.

Proof. (1) follows by induction on the structure of T. (2) follows from (1). □

Lemma 3. *(Derivation Projection) If* $\Gamma \vdash_{\mathbb{Q}} e : \hat{\sigma}$ *then* $\text{Shape}(\Gamma) \vdash e : \text{Shape}(\hat{\sigma})$.

Proof. Induction on the derivation of $\Gamma \vdash_{\mathbb{Q}} e : \hat{\sigma}$, and observing that each derivation rule for $\vdash_{\mathbb{Q}}$ is a *refinement* of a matching rule for $\vdash$. □

Lemma 4. *(Constraint Substitution) For every template environment* Γ*, expression* e*, and liquid type assignment* A*, if* $\text{ConsGen}(\Gamma,\ e) = (T, \mathbb{C})$ *then* $\text{ConsGen}(A\Gamma,\ e) = (AT, A\mathbb{C})$.

Proof. By induction on the structure of e. □

Lemma 5. *(Update) For any assignment* A*, template* T *fresh with respect to* A *(if a liquid type variable* κ *appears in* T *then it appears only once and it is not in* $\text{dom}(A)$*) and liquid type* $\hat{\tau}$ *such that* $\text{Shape}(\hat{\tau}) = \text{Shape}(T)$*:*

1. $\text{SolUpd}(A, T, \hat{\tau})(T) = \hat{\tau}$
2. *if* $\text{PredVars}(T') \subseteq \text{dom}(A)$ *then* $\text{SolUpd}(A, T, \hat{\tau})(T') = AT'$.

Proof. By induction on the structure of T and $\hat{\tau}$. □

Theorem 4. *(Constraint Generation) For every type environment* Γ *and expression* e *such that*

$$\text{ConsGen}(\Gamma,\ e) = (T, \mathbb{C}),$$

$\Gamma \vdash_{\mathbb{Q}} e : \hat{\sigma}$ *iff there exists an assignment* A *over* $\mathbb{Q}$ *such that* $AT = \hat{\sigma}$ *and* $A\mathbb{C}$ *is valid.*

Proof. Only if (⇒): By induction on the derivation $\Gamma \vdash_{\mathbb{Q}} e : \hat{\sigma}$.

- case $e \equiv \mathsf{c}$ or $e \equiv x$: Here $\text{PredVars}(T) = \emptyset$, i.e., T has no liquid type variables, and $\mathbb{C} = \emptyset$ so *any* solution A suffices.
- case $e \equiv \lambda x.e_1$: Here,

$$\begin{aligned} T &= x : T_x \ \rightarrow\ T_1 \\ \mathbb{C} &= \mathbb{C}_1 \cup \{\Gamma \vDash T\} \cup \{\Gamma; x : T_x \vdash T_1' <: T_1\} \\ \hat{\sigma} &= x : \hat{\tau}_x \ \rightarrow\ \hat{\tau}_1 \\ x : T_x \ \rightarrow\ T_1 &= \text{Fresh}(\text{Shape}(x : \hat{\tau}_x \ \rightarrow\ \hat{\tau}_1)) \\ (T_1', \mathbb{C}_1) &= \text{ConsGen}(\Gamma; x : T_x,\ e_1) \end{aligned}$$

  Let $A_0 = \text{SolUpd}(\emptyset, T, \hat{\sigma})$. By Lemma 4,

$$\begin{aligned} (A_0\mathbb{C}_1, A_0T_1') &= \text{ConsGen}(\Gamma; x : A_0T_x,\ e_1) \\ &= \text{ConsGen}(\Gamma; x : \hat{\tau}_x,\ e_1) \end{aligned}$$

  By inversion, $\Gamma \vDash \hat{\tau}_x$ and $\Gamma; x : \hat{\tau}_x \vdash_{\mathbb{Q}} e_1 : \hat{\tau}_1$. Thus, by IH, there exists A_1 such that:

$$A_1(A_0\mathbb{C}_1), \text{is valid} \tag{a}$$

$$A_1(A_0T_1') = \hat{\tau}_1 \tag{b}$$

Thus, $A = A_1; A_0$ is such that:

$$\begin{aligned} AT &= A_1(A_0 T) \\ &= A_1(\hat{\sigma}) \\ &= \hat{\sigma} \quad \text{as } \mathrm{PredVars}(\hat{\sigma}) = \emptyset \end{aligned}$$

Moreover,

$$A; A_1\mathbb{C} = A_1; A_0\mathbb{C}_1 \cup \{\Gamma \vDash A_1; A_0 T\} \cup \{\Gamma; x : A_1; A_0 T_x \vdash A_1; A_0 T_1' <: T_1\}$$

as $\mathrm{dom}(A_0)$ and $\mathrm{dom}(A_1)$ are disjoint, and (b)

$$= A_1; A_0\mathbb{C}_1 \cup \{\Gamma \vDash \hat{\tau}\} \cup \{\Gamma; x : \hat{\tau}_x \vdash \hat{\tau}_1 <: \hat{\tau}_1\}$$

which, by (a), Lemma 3 and Lemma 6 respectively, is valid.

- case $e \equiv v_1\ v_2$: Here,

$$\begin{aligned} T &= T'[x \mapsto v_2] \\ \mathbb{C} &= \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\Gamma \vdash T_2' <: T_2''\} \\ (c : T_2'' \ \to\ T', \mathbb{C}_1) &= \mathrm{ConsGen}(\Gamma,\ e_1) \\ (T_2', \mathbb{C}_2) &= \mathrm{ConsGen}(\Gamma,\ e_2) \end{aligned}$$

By inversion there exist $\hat{\tau}_2$ and $\hat{\tau}$ such that:

$$\Gamma \vdash_{\mathrm{Q}} v_1 : x : \hat{\tau}_2 \ \to\ \hat{\tau} \tag{a}$$
$$\Gamma \vdash_{\mathrm{Q}} v_2 : \hat{\tau}_2 \tag{b}$$
$$\hat{\sigma} = \hat{\tau}[x \mapsto v_2] \tag{c}$$

By IH and (a), there exist A_1 such that:

$$A_1 T_2'' = \hat{\tau}_2 \quad \text{and} \quad A_1 T' = \hat{\tau} \tag{d}$$
$$A_1\mathbb{C}_1 \text{ is valid} \tag{e}$$

By IH and (b), there exist A_2 such that:

$$A_2 T_2' = \hat{\tau}_2 \tag{f}$$
$$A_2\mathbb{C}_2 \text{ is valid} \tag{g}$$

Moreover,

$$\begin{aligned}\mathrm{dom}(A_1) &= \mathrm{PredVars}(x : T_2'' \ \rightarrow \ T') \cup \mathrm{PredVars}(\mathbb{C}_1)\\ \mathrm{dom}(A_2) &= \mathrm{PredVars}(T_2') \cup \mathrm{PredVars}(\mathbb{C}_2)\\ &\quad \text{are disjoint} \qquad \text{(h)}\end{aligned}$$

as they result from generating constraints on different subexpressions and $\mathrm{PredVars}(\Gamma) = \emptyset$. Consider $A = A_1; A_2$.

$$AT = A_1; A_2 T'[x \mapsto v_2]$$

which, due to delayed substitutions

$$= A_1; A_2 T'[x \mapsto v_2]$$

which, because of disjoint domains (g)

$$= A_1 T'[x \mapsto v_2]$$

which, due to (d)

$$\begin{aligned}&= \hat{\tau}[x \mapsto v_2]\\ &= \hat{\sigma}\end{aligned}$$

Moreover,

$$A\mathbb{C} = A_1; A_2\mathbb{C}_1 \cup A_1; A_2\mathbb{C}_2 \cup \{\Gamma \vdash A_1; A_2 T_2' <: A_1; A_2 T_2''\}$$

which, due to disjoint domains (g)

$$= A_1\mathbb{C}_1 \cup A_2\mathbb{C}_2 \cup \{\Gamma \vdash A_2 T_2' <: A_1 T_2''\}$$

which, due to (f) and (d)

$$\begin{aligned}&= A_1\mathbb{C}_1 \cup A_2\mathbb{C}_2 \cup \{\Gamma \vdash \hat{\tau}_2 <: \hat{\tau}_2\}\\ &\quad \text{which, by (e),(g) and Lemma 6 is valid.}\end{aligned}$$

- case $e \equiv \textbf{if } v \textbf{ then } e_2 \textbf{ else } e_3$: Here,

$$
\begin{aligned}
T &= \text{Fresh}(\text{Shape}(\hat{\sigma})) \quad \text{(by Lemma 3)} \\
\mathbb{C} &= \mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \{\Gamma \vDash T\} \\
&\quad \cup \{\Gamma; v \vdash T_2' <: T\} \\
&\quad \cup \{\Gamma; \neg e_1 \vdash T_3' <: T\} \\
(\cdot, \mathbb{C}_1) &= \text{ConsGen}(\Gamma,\ v) \\
(T_2', \mathbb{C}_2) &= \text{ConsGen}(\Gamma; v,\ e_2) \\
(T_3', \mathbb{C}_3) &= \text{ConsGen}(\Gamma; \neg v,\ e_3)
\end{aligned}
$$

  where PredVars(C_1),PredVars(C_2),PredVars(C_3) are disjoint. By inversion, and applying the IH, there exist solutions A_1, A_2, A_3 such that:

$$A_1\mathbb{C}_1,\ A_2\mathbb{C}_2,\ A_3\mathbb{C}_3 \text{ are valid} \tag{a}$$

$$A_2 T_2' = A_3 T_3' = \hat{\sigma} \tag{b}$$

$$\Gamma \vDash \hat{\sigma} \tag{c}$$

  Consider $A = \text{SolUpd}(A_1; A_2; A_3, T, \hat{\sigma})$, By Lemma 5,

$$
\begin{aligned}
AT &= \hat{\sigma} \\
A\mathbb{C} &= A_1\mathbb{C}_1 \cup A_2\mathbb{C}_2 \cup A_3\mathbb{C}_3 \cup \{\Gamma \vDash \hat{\sigma}\} \\
&\quad \cup \{\Gamma; v \vdash A_2 T_2' <: \hat{\sigma}\} \\
&\quad \cup \{\Gamma; \neg v \vdash A_3 T_3' <: \hat{\sigma}\}
\end{aligned}
$$

  by (b) and (c)

$$
\begin{aligned}
&= A_1\mathbb{C}_1 \cup A_2\mathbb{C}_2 \cup A_3\mathbb{C}_3 \cup \{\Gamma \vDash \hat{\sigma}\} \\
&\cup \{\Gamma; v \vdash \hat{\sigma} <: \hat{\sigma}\} \\
&\cup \{\Gamma; \neg v \vdash \hat{\sigma} <: \hat{\sigma}\}
\end{aligned}
$$

  which, by (a), inversion and Lemma 6 is valid.

- case $e \equiv \textbf{let } x = e_1 \textbf{ in } e_2$: Here,

$$
\begin{aligned}
T &= \text{Fresh}(\text{Shape}(\hat{\sigma})) \quad \text{(by Lemma 3)} \\
\mathbb{C} &= \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\Gamma \vDash T\} \cup \{\Gamma; x : T_1' \vdash T_2' <: T\} \qquad \text{(a0)} \\
(T_1', \mathbb{C}_1) &= \text{ConsGen}(\Gamma,\ e_1) \\
(T_2', \mathbb{C}_2) &= \text{ConsGen}(\Gamma; x : T_1',\ e_2) \qquad \text{(a1)}
\end{aligned}
$$

By inversion there exists $\hat{\sigma}_1$ such that:

$$\Gamma \vdash_{\mathrm{Q}} e_1 : \hat{\sigma}_1 \tag{a}$$
$$\Gamma; x : \hat{\sigma}_1 \vdash_{\mathrm{Q}} e_2 : \hat{\sigma} \tag{b}$$
$$\Gamma \vDash \hat{\sigma} \tag{c}$$

By (a) and IH there exists A_1 such that:

$$A_1\mathbb{C}_1 \text{ is valid} \tag{d}$$
$$A_1 T_1' = \hat{\sigma}_1 \tag{e}$$

By Lemma 4 and (a1),

$$\begin{aligned}(A_1 T_2', A_1\mathbb{C}_2) &= \mathrm{ConsGen}(\Gamma; x : A_1 T_1',\ expr_2) \\ &= \mathrm{ConsGen}(\Gamma; x : \hat{\sigma}_1',\ e_2) \quad \text{by (e)}\end{aligned}$$

By (b) and IH, there exists A_2 such that:

$$\mathrm{dom}(A_2) = \mathrm{PredVars}(C_2) \text{ which is disjoint from } \mathrm{dom}(A_1) \tag{f}$$
$$A_2(A_1\mathbb{C}_2) \text{ is valid} \tag{g}$$
$$A_2(A_1 T_2') = \hat{\sigma} \tag{h}$$

Consider $A = \mathrm{SolUpd}(A_2; A_1, T, \hat{\sigma})$. By Lemma 5,

$$AT = \hat{\sigma}$$

By (a0), (f), and (i):

$$A\mathbb{C} = A_1\mathbb{C}_1 \cup A_2(A_1\mathbb{C}_2) \cup \{\Gamma \vDash AT\} \cup \{\Gamma; x : A_1 T_1' \vdash A_2(A_1 T_2') <: AT\}$$

by (i), (e), (h)

$$= A_1\mathbb{C}_1 \cup A_2(A_1\mathbb{C}_2) \cup \{\Gamma \vDash \hat{\sigma}\} \cup \{\Gamma; x : \hat{\sigma}_1 \vdash \hat{\sigma} <: \hat{\sigma}\}$$

which, by (d), (g), (c), and Lemma 6 is valid.

- case $e \equiv \Lambda\alpha.e_1$: Here,

$$(T, \mathbb{C}) = (\Lambda\alpha.T_1', \mathbb{C}_1)$$
$$(T_1', \mathbb{C}_1) = \mathrm{ConsGen}(\Gamma,\ e_1)$$

By inversion, exists $\hat{\sigma}_1$ such that $\Gamma \vdash_Q e_1 : \hat{\sigma}_1$. Thus, by IH, exists a A_1 such that:

$$A_1\mathbb{C}_1 \text{ is valid} \tag{a}$$
$$A_1 T_1' = \hat{\sigma}_1 \tag{b}$$

Consider $A = A_1$.

$$\begin{aligned} AT &= A_1 T \\ &= A_1(\Lambda\alpha.T_1') \\ &= \Lambda\alpha.A_1 T_1' \end{aligned}$$

which, by (b),

$$\begin{aligned} &= \Lambda\alpha.\hat{\sigma}_1 \\ &= \hat{\sigma} \end{aligned}$$

Finally,

$$A\mathbb{C} = A_1\mathbb{C}_1 \text{ which, by (a), is valid.}$$

- case $e \equiv e_1[\dot{\tau}]$: Here,

$$\begin{aligned} T &= T_1'[\alpha \mapsto T_\alpha] \\ \mathbb{C} &= \mathbb{C}_1 \cup \{\Gamma \vDash T_\alpha\}) \\ (T_1', C_1) &= \text{ConsGen}(\Gamma,\ e_1) \\ T_\alpha &= \text{Fresh}(\dot{\tau}) \end{aligned}$$

By inversion, there exists $\hat{\tau}$, $\hat{\sigma}_1$ such that:

$$\Gamma \vDash \hat{\tau} \tag{a}$$
$$\text{Shape}(\hat{\tau}) = \dot{\tau} \tag{b}$$
$$\Gamma \vdash_Q e_1 : \Lambda\alpha.\hat{\sigma}_1 \tag{c}$$

By IH, there exists A_1 such that:

$$A_1\mathbb{C}_1 \text{ is valid} \tag{d}$$
$$A_1 T_1' = \Lambda\alpha.\hat{\sigma}_1 \tag{e}$$

Consider $A = \text{SolUpd}(A_1, T_\alpha, \hat{\tau})$:

$$\begin{aligned} AT &= A(T_1'[\alpha \mapsto T_\alpha]) \\ &= AT_1'[\alpha \mapsto AT_\alpha] \end{aligned}$$

by Lemma 5

$$\begin{aligned} &= A_1 T_1'[\alpha \mapsto \hat{\tau}] \\ &= \Lambda\alpha.\hat{\sigma}_1[\alpha \mapsto \hat{\tau}] \\ &= \hat{\sigma} \end{aligned}$$

If (⇐): By induction on the structure of e.

- case $e \equiv \mathsf{c}$ or $e \equiv x$ Trivial as $\mathbb{C} = \emptyset$ and T such that $\text{PredVars}(T) = \emptyset$ and $\Gamma \vdash_{\mathbb{Q}} e : T$.
- case $e \equiv \lambda x.e_1$: Here

$$\begin{aligned} T &= x : T_x \ \rightarrow \ T_1 \\ \mathbb{C} &= \mathbb{C}_1 \cup \{\Gamma \vDash T\} \cup \{\Gamma; x : T_x \vdash T_1' <: T_1\} \\ \hat{\sigma} &= x : \hat{\tau}_x \ \rightarrow \ \hat{\tau}_1 \\ x : T_x \ \rightarrow \ T_1 &= \text{Fresh}(\text{Shape}(x : \hat{\tau}_x \ \rightarrow \ \hat{\tau}_1)) \\ (T_1', \mathbb{C}_1) &= \text{ConsGen}(\Gamma; x : T_x, \ e_1) \end{aligned}$$

As $A\mathbb{C}$ is valid,

$$A\mathbb{C}_1 \text{ is valid} \tag{a}$$

$$\Gamma \vDash x : AT_x \ \rightarrow \ AT_1 \tag{b}$$

$$\Gamma; x : AT_x \vdash AT_1' <: AT_1 \tag{c}$$

By Lemma 4,

$$(AT_1', A\mathbb{C}_1) = \text{ConsGen}(\Gamma; x : AT_x, \ e_1)$$

By (a) and the IH,

$$\Gamma; x : AT_x \vdash_{\mathbb{Q}} e_1 : AT_1' \tag{d}$$

By (d),(b),(c) and rule LT-SUB,

$$\Gamma; x : AT_x \vdash_{\mathbb{Q}} e_1 : AT_1 \tag{e}$$

By Lemma 1, and rule LT-FUN,

$$\begin{aligned} &\Gamma \vdash_{\mathbb{Q}} \lambda x.e_1 : x : AT_x \ \rightarrow \ AT_1 \\ \text{implies } &\Gamma \vdash_{\mathbb{Q}} \lambda x.e_1 : A(x : T_x \ \rightarrow \ T_1) \\ \text{implies } &\Gamma \vdash_{\mathbb{Q}} \lambda x.e_1 : AT \end{aligned}$$

- case $e \equiv v_1\ v_2$: Here,

$$
\begin{aligned}
T &= T'[x \mapsto v_2] \\
\mathbb{C} &= \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\Gamma \vdash T_2' <: T_2''\} \\
(c : T_2'' \ \rightarrow\ T', \mathbb{C}_1) &= \text{ConsGen}(\Gamma,\ v_1) \\
(T_2', \mathbb{C}_2) &= \text{ConsGen}(\Gamma,\ v_2) \\
A\mathbb{C} &\text{ is valid} \\
A\mathbb{C}_1 \cup A\mathbb{C}_2 &\text{ is valid} \\
\Gamma &\vdash AT_2' <: AT_2'' \qquad \text{(a)}
\end{aligned}
$$

  Hence, by the IH,

$$
\begin{aligned}
\Gamma &\vdash_{\mathrm{Q}} v_1 : x : AT_2''\ \rightarrow\ AT' \qquad \text{(b)} \\
\Gamma &\vdash_{\mathrm{Q}} v_2 : AT_2' \qquad \text{(c)}
\end{aligned}
$$

  From (b), and Assumption 3,

$$
\Gamma \vDash AT_2'' \qquad \text{(d)}
$$

  Thus, from (a),(c), (d) and rule LT-SUB,

$$
\Gamma \vdash_{\mathrm{Q}} v_2 : AT_2''
$$

  From (b) and rule LT-APP,

$$
\Gamma \vdash_{\mathrm{Q}} v_1\ v_2 : AT'[x \mapsto v_2] \qquad \text{(e)}
$$

  As substitutions are delayed,

$$
AT'[x \mapsto v_2] = AT'[x \mapsto v_2] = AT
$$

  and so, from (e),

$$
\Gamma \vdash_{\mathrm{Q}} v_1\ v_2 : AT
$$

- case $e \equiv \textbf{if}\ v_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3$: Here,

$$
\begin{aligned}
T &= \text{Fresh}(\text{Shape}(\hat{\sigma})) \quad \text{(by Lemma 3)} \\
\mathbb{C} &= \mathbb{C}_1 \cup \mathbb{C}_2 \cup \mathbb{C}_3 \cup \{\Gamma \vDash T\} \cup \{\Gamma; v_1 \vdash T_2' <: T\}\{\Gamma; \neg v_1 \vdash T_3' <: T\} \\
(\cdot, \mathbb{C}_1) &= \text{ConsGen}(\Gamma,\ v_1) \\
(T_2', \mathbb{C}_2) &= \text{ConsGen}(\Gamma; v_1,\ e_2) \\
(T_3', \mathbb{C}_3) &= \text{ConsGen}(\Gamma; \neg v_1,\ e_3)
\end{aligned}
$$

As $A\mathbb{C}$ is valid,

$$A\mathbb{C}_1, A\mathbb{C}_2, A\mathbb{C}_3 \text{ are valid} \quad \text{(a)}$$
$$\Gamma \vDash AT \quad \text{(b)}$$
$$\Gamma; v_1 \vdash AT_2' <: AT \quad \text{(ct)}$$
$$\Gamma; \neg v_1 \vdash AT_3' <: AT \quad \text{(cf)}$$

By (a) and IH,

$$\Gamma; v_1 \vdash_{\mathbb{Q}} e_2 : AT_2'$$
$$\Gamma; \neg v_1 \vdash_{\mathbb{Q}} e_3 : AT_3'$$

By (b), (ct), (cf), Lemma 5,

$$\Gamma; v_1 \vdash_{\mathbb{Q}} e_2 : AT$$
$$\Gamma; \neg v_1 \vdash_{\mathbb{Q}} e_3 : AT$$

By (a), Lemma 1 and rule LT-IF,

$$\Gamma \vdash_{\mathbb{Q}} \textbf{if } v_1 \textbf{ then } e_2 \textbf{ else } e_3 : AT$$

- case $e \equiv \textbf{let } x = e_1 \textbf{ in } e_2$: Here,

$$T = \text{Fresh}(\text{Shape}(\hat{\sigma})) \quad \text{(by Lemma 3)}$$
$$\mathbb{C} = \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\Gamma \vDash T\} \cup \{\Gamma; x : T_1' \vdash T_2' <: T\}$$
$$(T_1', \mathbb{C}_1) = \text{ConsGen}(\Gamma,\ e_1)$$
$$(T_2', \mathbb{C}_2) = \text{ConsGen}(\Gamma; x : T_1',\ e_2)$$

As $A\mathbb{C}$ is valid

$$A\mathbb{C}_1, A\mathbb{C}_2 \text{ are valid} \quad \text{(a)}$$
$$\Gamma \vDash AT \quad \text{(b)}$$
$$\Gamma; x : AT_1' \vdash AT_2' <: T \quad \text{(c)}$$

By (a) and IH,

$$\Gamma \vdash_{\mathbb{Q}} e_1 : AT_1' \quad \text{(d1)}$$
$$\Gamma; x : AT_1' \vdash_{\mathbb{Q}} e_2 : AT_2' \quad \text{(d2)}$$

By (b), (c), (d1), Assumption 4, and rule LT-SUB,

$$\Gamma; x : AT_1' \vdash_{\mathbb{Q}} e_2 : AT \tag{e}$$

Thus, by (b), (c), (d1), (e) and rule LT-LET,

$$\Gamma \vdash_{\mathbb{Q}} \textbf{let } x = e_1 \textbf{ in } e_2 : AT$$

- case $e \equiv \Lambda\alpha.e_1$: Here,

$$(T, \mathbb{C}) = (\Lambda\alpha.T_1', \mathbb{C}_1)$$
$$(T_1', \mathbb{C}_1) = \text{ConsGen}(\Gamma,\ e_1)$$

As $A\mathbb{C}$ is valid, $A\mathbb{C}_1$ is valid, and so,

$$\Gamma \vdash_{\mathbb{Q}} e : AT_1' \tag{a}$$

As $\text{Shape}(\Gamma) \vdash e : \dot{\sigma}$,

$$\alpha \notin \Gamma \tag{b}$$

Thus, by rule LT-GEN,

$$\Gamma \vdash_{\mathbb{Q}} \Lambda\alpha.e_1 : \Lambda\alpha.AT_1'$$

as $\alpha \notin A$

$$\text{implies } \Gamma \vdash_{\mathbb{Q}} \Lambda\alpha.e_1 : A\Lambda\alpha.T_1'$$
$$\text{implies } \Gamma \vdash_{\mathbb{Q}} \Lambda\alpha.e_1 : AT$$

- case $e \equiv e_1[\dot{\tau}]$: Here,

$$T = T_1'[\alpha \mapsto T_\alpha]$$
$$\mathbb{C} = \mathbb{C}_1 \cup \{\Gamma \vDash T_\alpha\})$$
$$(\Lambda\alpha.T_1', \mathbb{C}_1)\ \text{ConsGen}(\Gamma,\ e_1)$$
$$T_\alpha = \text{Fresh}(\dot{\tau})$$

As $A\mathbb{C}$ is valid,

$$A\mathbb{C}_1 \text{ is valid} \tag{a}$$
$$\Gamma \vDash AT_\alpha \tag{b}$$

By (a) and IH,

$$\Gamma \vDash A\Lambda\alpha.T_1'$$

As $\alpha \notin A$, $A\Lambda\alpha.T_1' = \Lambda\alpha.AT_1'$. Thus, by (a) and IH,

$$\Gamma \vDash A\Lambda\alpha.T_1' \tag{c}$$

Thus, by (b), (c), Lemma 2 and rule LT-INST,

$$\Gamma \vdash_{\mathbb{Q}} e_1[\dot{\tau}] : AT_1'[\alpha \mapsto AT_\alpha]$$

as $\alpha \notin A$, $T_\alpha \notin \text{rng}(A)$

$$\Rightarrow \Gamma \vdash_{\mathbb{Q}} e_1[\dot{\tau}] : AT_1'[\alpha \mapsto T_\alpha]$$
$$\Rightarrow \Gamma \vdash_{\mathbb{Q}} e_1[\dot{\tau}] : AT$$

□

Definition 1. *(Simple Constraints) A* simple constraint *is of the form:*

- $\Gamma \vDash \{\nu : t \mid p\}$
- $\Gamma \vdash \{\nu : t \mid p_1\} <: \{\nu : t \mid p_2\}$

where p_1 and p_2 are either refinement predicates or predicate variables with pending substitutions.

Lemma 6. *(Constraint Splitting) For every set of constraints $\mathbb{C}$,*

1. $\text{Split}(\mathbb{C})$ *is a set of simple constraints,*
2. *For every assignment A, $A\mathbb{C}$ is valid iff $A(\text{Split}(\mathbb{C}))$ is valid.*

Proof. For both cases, we prove the lemma when $\mathbb{C}$ is a singleton set and then lift to arbitrary sets. For singleton sets $\{C\}$, both cases follow by induction on the structure of C. □

Definition 2. *(Minimum Solution) For two assignments A and A' over $\mathbb{Q}$, we say $A \leq A'$ if for all κ, $A(\kappa) \supseteq A'(\kappa)$. For $\mathbb{C}$, a set of constraints, A^* is the* minimum solution *over $\mathbb{Q}$ if*

1. *$A^*\mathbb{C}$ is valid and,*
2. *For each A over $\mathbb{Q}$, if $A\mathbb{C}$ is valid then $A^* \leq A$.*

Lemma 7. *(Embedding) If $A \leq A'$ are two assignments over $\mathbb{Q}$ then:*

1. *$A\kappa \Rightarrow A\kappa$,*

2. $A(\theta\kappa) \Rightarrow A(\theta\kappa)$,

3. $A\Gamma \Rightarrow A'\Gamma$.

Proof. Immediate from definition of solution ordering ($\leq$). □

Theorem 5. *(Minimum Solution) If* $\mathbb{C}$ *has a solution over* $\mathbb{Q}$ *then* $\mathbb{C}$ *has a minimum solution over* $\mathbb{Q}$.

Proof. By Lemma 6 it suffices to prove the theorem for sets of simple constraints $\mathbb{C}$. As the number of liquid type variables and logical qualifiers $\mathbb{Q}$ is finite, any $\mathbb{C}$ can only have a finite number of solutions over $\mathbb{Q}$. Suppose that $A_1, \ldots, A_n$ are the solutions for $\mathbb{C}$, i.e., for each $1 \leq i \leq n$, we have $A_i\mathbb{C}$ is valid. Then we shall show that:

$$A^* = \lambda\kappa. \cap_i A_i(\kappa)$$

is a minimum solution for $\mathbb{C}$ over $\mathbb{Q}$. Trivially, for each i, we have $A^* \leq A_i$. Next, we shall prove for each simple constraint $c \in \mathbb{C}$, that as each $A_i c$ is valid, $A^* c$ is also valid.

- case $C \equiv \Gamma \vDash \{\nu : t \mid p\}$: where p is either a predicate ϕ or a variable with pending substitutions $\theta\kappa$.

  In the first case $p \equiv \phi$,

  $$A_i(\Gamma \vDash \{\nu : t \mid \phi\})$$
  $$\text{i.e.,} \quad \phi \text{ well-sorted in } \text{Shape}(A_i\Gamma); \nu : t$$

  By Lemma 2, $\text{Shape}(A^*\Gamma) = \text{Shape}(A_i\Gamma) = \text{Shape}(\Gamma)$, thus,

  $$\phi \text{ well-sorted in } \text{Shape}(A^*\Gamma); \nu : t$$
  $$\text{i.e.,} \quad A^*(\Gamma \vDash \{\nu : t \mid \phi\})$$

  In the second case $p \equiv \theta\kappa$,

  $$A_i(\Gamma \vDash \{\nu : t \mid \theta\kappa\})$$
  $$\text{i.e.,} \quad \theta A_i(\kappa) \text{ well-sorted in } \text{Shape}(A_i\Gamma); \nu : t$$

  i.e., for each $\phi \in A_i(\kappa)$,

  $$A_i\Gamma \vDash \{\nu : t \mid \theta\phi\}$$
  $$\text{i.e.,} \quad \theta\phi \text{ well-sorted in } \text{Shape}(A_i\Gamma); \nu : t$$

  as $A^*(\kappa) \subseteq A_i(\kappa)$ and by Lemma 2, $\text{Shape}(A^*\Gamma) = \text{Shape}(A_i\Gamma) = \text{Shape}(\Gamma)$, for each $\phi \in A^*(\kappa)$,

  $$\theta\phi \text{ well-sorted in } \text{Shape}(A^*\Gamma); \nu : t$$
  $$\text{i.e.,} \quad A^*(\Gamma \vDash \{\nu : t \mid \theta\kappa\})$$

- case $C \equiv \Gamma \vdash \{\nu : t \mid p_1\} <: \{\nu : t \mid p_2\}$: where each of p_1 and p_2 is either a refinement predicate or a predicate variable with pending substitution.

$$\text{For each } i, \quad A_i(\Gamma \vdash \{\nu : t \mid p_1\} <: \{\nu : t \mid p_2\})$$
$$\text{i.e.,} \quad A_i\Gamma \vDash A_i p_1 \Rightarrow A_i p_2$$

by the properties of implication,

$$\wedge_i A_i\Gamma \vDash \wedge_i A_i p_1 \Rightarrow \wedge_i A_i p_2$$

as $[\![A^*\Gamma]\!] = [\![\wedge_i A_i\Gamma]\!]$ and $[\![A^* p_1]\!] = [\![\wedge_i A_i p_1]\!]$ and $[\![A^* p_2]\!] = [\![\wedge_i A_i p_2]\!]$, we have:

$$A^*\Gamma \vDash A^* p_1 \Rightarrow A^* p_2$$
$$\text{i.e.,} \quad A^*(\Gamma \vdash \{\nu : t \mid p_1\} <: \{\nu : t \mid p_2\})$$

□

Lemma 8. *(Refinement) If* $A' = \text{Refine}(A,\ C)$ *then:*

1. $A \leq A'$,
2. *if* AC *is not valid, then* $A \neq A'$,
3. *if* $A''C$ *is valid and* $A \leq A''$ *then* $A' \leq A''$.

Proof. 1. From the definition of Refine, we have:

$$A' \equiv \text{SolUpd}(A, \kappa_c, A(\kappa) \cap Q')$$

for some κ_c and Q'. We shall show that for any κ, we have $A'(\kappa) \subseteq A(\kappa)$.

Consider $\kappa \neq \kappa_c$. Here $A'(\kappa) = A(\kappa) \subseteq A(\kappa)$.

Consider $\kappa = \kappa_c$. Here $A'(\kappa) = A(\kappa) \cap Q' \subseteq A(\kappa)$.

2. We split cases on the type of constraint used for refinement, and in each case, show that if $A' = A$ then Ac must be valid.

   - case $C \equiv \Gamma \vDash \{\nu : t \mid p\}$: where p is a refinement predicate or a predicate variable with pending substitutions. From the definition of Refine, we have:

$$A'p \text{ well-sorted in } \text{Shape}(A\Gamma); \nu : \texttt{bool}$$

   If $A = A'$ then,

$$Ap \text{ well-sorted in } \text{Shape}(A\Gamma); \nu : \texttt{bool}$$

   i.e., AC is valid.

- case $C \equiv \Gamma \vDash p \Rightarrow \theta\kappa_c$: From the definition of Refine, we have:

$$A\Gamma \vDash Ap \Rightarrow \theta A'(\kappa_c)$$

If $A = A'$ then,

$$A\Gamma \vDash Ap \Rightarrow \theta A(\kappa_c)$$

i.e., AC is valid.

3. We split cases on the type of constraint used for refinement.

- case $C \equiv \Gamma \vDash \{\nu : t \mid p\}$: where p is an expression or a refinement predicate variable with pending substitutions. We have,

$$A''C \text{ is valid} \tag{a}$$
$$\forall\kappa.A''(\kappa) \subseteq A(\kappa) \tag{b}$$

as $A \leq A''$. From the definition of Refine, we have:

$$A' \equiv \text{SolUpd}(A, \kappa_c, A(\kappa) \cap Q') \tag{c}$$
$$Q' \equiv \{\phi \mid \phi \in \mathbb{Q}, \phi \text{ well-sorted in } \text{Shape}(\Gamma); \nu : t\} \tag{d}$$

for some κ_c. We shall show that for any κ, we have $A''(\kappa) \subseteq A'(\kappa)$.

Consider $\kappa \neq \kappa_c$. From (b)

$$A''(\kappa) \subseteq A(\kappa) = A'(\kappa)$$

Consider $\kappa = \kappa_c$. Now (a) implies that

$$\text{for each} \quad \phi \in A''(\kappa_c), \phi \in \mathbb{Q}, \text{and} \phi \text{ well-sorted in } \text{Shape}(\Gamma); \nu : t$$
$$\text{i.e., for each} \quad \phi \in A''(\kappa_c), \phi \in Q'$$

From (b),

$$A''(\kappa_c) \subseteq A(\kappa_c) \cap Q' = A'(\kappa_c)$$

- case $C \equiv \Gamma \vDash p \Rightarrow \theta\kappa_c$: We have,

$$A''C \text{ is valid} \tag{a}$$
$$\forall\kappa.A''(\kappa) \subseteq A(\kappa) \tag{b}$$

as $A \leq A''$. From the definition of Refine, we have:

$$A' \equiv \text{SolUpd}(A, \kappa_c, A(\kappa) \cap Q') \tag{c}$$
$$Q' \equiv \{\phi \mid \phi \in \mathbb{Q} \text{ and } A\Gamma \vDash Ap \Rightarrow \theta\phi\} \tag{d}$$

We shall show that for any κ, we have $A''(\kappa) \subseteq A'(\kappa)$.

Consider $\kappa \neq \kappa_c$. From (b)

$$A''(\kappa) \subseteq A(\kappa) = A'(\kappa)$$

Consider $\kappa = \kappa_c$. Now (a) implies that

$$\text{for each} \quad \phi \in A''(\kappa_c), \phi \in \mathbb{Q}, \text{and} A''\Gamma \vDash A''p \Rightarrow \theta\phi$$

As $A \leq A''$, from Lemma 7 we have

$$[\![A\Gamma]\!] \Rightarrow [\![A''\Gamma]\!]$$
$$[\![Ap]\!] \Rightarrow [\![A''p]\!]$$
$$\text{i.e.,} \quad [\![A\Gamma]\!] \wedge [\![Ap]\!] \Rightarrow [\![A''\Gamma]\!] \wedge [\![A''p]\!] \Rightarrow$$

and so,

$$\text{for each} \quad \phi \in A''(\kappa_c), \phi \in \mathbb{Q}, \text{and} A''\Gamma \vDash A''p \Rightarrow \theta e$$
$$\text{i.e., for each} \quad \phi \in A''(\kappa_c), \phi \in Q'$$
$$\text{i.e.,} A''(\kappa_c) \subseteq Q'$$

From (b),

$$A''(\kappa_c) \subseteq A(\kappa_c) \cap Q' = A'(\kappa_c)$$

□

Theorem 6. *(Constraint Solving) For every set of constraints* $\mathbb{C}$ *and qualifiers* $\mathbb{Q}$*,*

1. Solve($\mathbb{C}$, $\lambda\kappa.\mathbb{Q}$) *terminates,*

2. *if* $\text{Solve}(C,\ \lambda\kappa.\mathbb{Q})$ *returns* A *then* A *is the minimum solution for* $\mathbb{C}$ *over* $\mathbb{Q}$,

3. *if* $\text{Solve}(\mathbb{C},\ \lambda\kappa.\mathbb{Q})$ *returns* $\bot$ *then* $\mathbb{C}$ *has no solution over* $\mathbb{Q}$.

Proof. 1. To prove that Solve terminates, we associate a well-founded measure with solutions and show that in each iteration of the refinement loop, the potential of the solution A strictly decreases. Let,

$$MA \equiv \sum_{\kappa} \|A(\kappa)\|$$

where $\|A(\kappa)\|$ is the cardinality of $A(\kappa)$. Consider any loop iteration. From the definition of Solve, we know that the constraint C chosen for refinement is such that AC is not valid. Let A' be the refined solution returned by calling $\text{Solve}(A,\ C)$. By Lemma 8,

$$A \leq A'$$
$$A \neq A'$$

Thus, by the definition of $\leq$, we have $MA' < MA$, i.e., the potential of the solution strictly decreases in the iteration. As the potential is non-negative, Refine must terminate.

2. Assume that Solve returns a solution A. Then $A\mathbb{C}$ is valid (as otherwise the loop would finish), and so by Theorem 5, $\mathbb{C}$ has a minimum solution A^* over $\mathbb{Q}$. To prove that the returned solution is the same as A^*, we show by induction over n that after n iterations of the loop in Solve, the solution $A \leq A^*$. In the base case, A has the initial assignment mapping each liquid type variable to $\mathbb{Q}$ and thus A is less than every solution over $\mathbb{Q}$, including A^*. Let us assume the induction hypothesis, that after n iterations, $A \leq A^*$. The value of A after $n+1$ iterations is $\text{Refine}(A,\ C)$ where A is the solution after n iterations. As A^*C is valid (A^* is the minimum solution for $\mathbb{C}$ over $\mathbb{Q}$, and $A \leq A^*$ (by the induction hypothesis), from Lemma 8, we deduce that $\text{Refine}(A,\ C) \leq A^*$, and so after $n+1$ iterations, $A \leq A^*$. Thus, if A is the solution returned by Solve, then $A \leq A^*$. As A^* is the minimum solution over $\mathbb{Q}$, and A is a valid solution, $A = A^*$.

3. Suppose that Solve fails,but that there is a valid solution for $\mathbb{C}$ over $\mathbb{Q}$. Then, there exists a minimum solution A^* over $\mathbb{Q}$, and by the reasoning above, at each iteration, $A \leq A^*$. By the definition of Refine, the outcome $\bot$ only happens when A and the constraint C are such that $C \equiv \Gamma \vDash p \Rightarrow \phi$, where p is either a predicate or a predicate variable with pending substitutions, and:

$$A\Gamma \vDash Ap \not\Rightarrow \phi$$

Now, as $A \leq A^*$, by Lemma 7, we have

$$[\![A\Gamma]\!] \Rightarrow [\![A^*\Gamma]\!]$$
$$[\![Ap]\!] \Rightarrow [\![A^*p]\!]$$

and therefore,

$$A^*\Gamma \vDash A^*p \not\Rightarrow \phi$$

i.e., A^*C is not valid, which is a contradiction, and so there is no valid solution for $\mathbb{C}$ over $\mathbb{Q}$.

□

Proof. (of Theorem 2) Immediate corollary of Theorems 4 and 6. □

Appendix B

Dynamic Semantics of NANOC

In this chapter, we give a standard small-step, call-by value semantics for our core low-level language, NANOC.

B.1 Reference Values

The program syntax of section 3.2 is meant to describe source-level programs. However, as a program executes, it produces values of reference type, which are not accounted for in our syntax of values. Thus, we add reference values to the syntax of values:

v	::=		**Values**
	|	$\ldots$	
	|	$\text{ref}(r, n)$	constant pointer

A reference value $\text{ref}(r, n)$ represents a pointer to run-time location r at offset n from the start of the location r. We assume locations r are drawn from a countable, totally-ordered set of run-time heap locations, *RLoc*.

B.2 Semantics

For clarity, we assume that the global environment G is threaded through the evaluation rules, and thus omit the global environment from the rules. The rules for evaluating programs use a mapping D from function names to their body expressions.

Definition 3 (Run-Time Block)**.** *A* run-time block *c is a partial function from natural number offsets to values:*

$$c : \mathbb{N} \rightharpoonup v.$$

$$n_{|w|} \circ m_{|w|} \hookrightarrow (n \circ m)_{|w|} \qquad \text{E-ARITH}$$

$$\text{ref}(r, n) +_p m_{|W|} \hookrightarrow \text{ref}(r, n + m) \qquad \text{E-PTR-PLUS}$$

$$0_{|W|} +_p m_{|W|} \hookrightarrow 0_{|W|} \qquad \text{E-NULL-PLUS}$$

$$v_1 \bowtie v_2 \hookrightarrow 1_{|W|} \text{ if Comparable}(v_1, v_2), v_1 \bowtie v_2 \qquad \text{E-REL-TRUE}$$

$$v_1 \bowtie v_2 \hookrightarrow 0_{|W|} \text{ if Comparable}(v_1, v_2), \neg(v_1 \bowtie v_2) \qquad \text{E-REL-FALSE}$$

Figure B.1: Small-step semantics of pure NANOC expressions

Definition 4 (Run-Time Store)**.** *A* run-time store *s is a map from run-time location names r to run-time blocks c:*

$$s : r \rightharpoonup c.$$

Definition 5 (Comparable Values)**.** *Values* v_1, v_2 *are* comparable*, written* Comparable(v_1, v_2)*, if and only if one of the following holds:*

- $v_1 = n_{1|w|}$, $v_2 = n_{2|w|}$
- $v_1 = \text{ref}(r_1, n_1)$, $v_2 = \text{ref}(r_2, n_2)$
- $v_1 = 0_{|W|}$, $v_2 = \text{ref}(r, n)$
- $v_1 = \text{ref}(r, n)$, $v_2 = 0_{|W|}$

Definition 6 (Value Comparisons)**.** *We define the behavior of the comparison operators as follows:*

- $w_{|n|} \bowtie w_{|m|}$ *iff* $n \bowtie m$
- $W_{|0|} < \text{ref}(r, n)$ *for all r, n*
- $\text{ref}(r_1, n) \bowtie \text{ref}(r_2, m)$ *iff* $(r_1, n) \bowtie (r_2, m)$ *lexicographically*
- *All relations between comparable values (as defined by Definition 5) that are not explicitly specified satisfy the appropriate laws (e.g.,* $=$ *is always an equivalence relation on comparable values). Relations between non-comparable values are left unspecified.*

Definition 7 (Type Sizes)**.** *We obtain the size of a type* τ*, written* SizeOf(τ)*, as*

$$\text{SizeOf}(\texttt{int}(w, i)) = w$$
$$\text{SizeOf}(\texttt{ref}(\ell, i)) = W$$

$$\text{SizeOf}(\{v : t \mid \phi\}) = \text{SizeOf}(t).$$

Definition 8 (Value Sizes)**.** *We define the size of a value, written* $\text{SizeOf}(v)$*, as*

$$\text{SizeOf}(n_{|w|}) = w$$
$$\text{SizeOf}(\text{ref}(r, n)) = W$$

Definition 9 (Fitting Data Into Blocks)**.** *We define a predicate,* Fits*, which determines when a value of size m fits in a run-time block c when stored at offset n:*

$$\text{Fits}(n, m, c) = \{n \ldots n + m - 1\} \subseteq \text{dom}(c)$$

Definition 10 (Block Value Updates)**.** *We define a function,* Write*, which computes the block resulting from updating the block c to contain the value v at offset n:*

$$\text{Write}(c, n, v) = c[n \mapsto v][(n + 1 \ldots \text{SizeOf}(v) - 1) \mapsto 0_{|1|}]$$

$$\text{E-PURE}\quad \frac{a \hookrightarrow a'}{a/s \hookrightarrow a'/s}$$

$$\text{E-SEQ}\quad \frac{e_1/s \hookrightarrow e_1'/s' \qquad e_1 \text{ not a value}}{\textbf{let } x = e_1 \textbf{ in } e_2/s \hookrightarrow \textbf{let } x = e_1' \textbf{ in } e_2/s'}$$

$$\text{E-LET}\quad \frac{}{\textbf{let } x = v \textbf{ in } e/s \hookrightarrow e[x \mapsto v]/s}$$

$$\text{E-IF-TRUE}\quad \frac{n \neq 0}{\textbf{if } n_{|W|} \textbf{ then } e_1 \textbf{ else } e_2/s \hookrightarrow e_1/s} \qquad \text{E-IF-FALSE}\quad \frac{}{\textbf{if } 0_{|W|} \textbf{ then } e_1 \textbf{ else } e_2/s \hookrightarrow e_2/s}$$

$$\text{E-READ}\quad \frac{v = s(r)(n) \qquad \text{SizeOf}(v) = m \qquad \text{Fits}(n, m, s(r))}{*_m\text{ref}(r, n)/s \hookrightarrow v/s}$$

$$\text{E-WRITE}\quad \frac{\text{Fits}(n, \text{SizeOf}(v), s(r))}{*\text{ref}(r, n) := v/s \hookrightarrow 0_{|0|}/s[r \mapsto \text{Write}(s(r), n, v)]}$$

$$\text{E-CALL}\quad \frac{G(f) = f\ (\overline{x})\ \{\ e\ \}}{f(\overline{v})[\overline{\ell_f} \mapsto \overline{\ell}]/s \hookrightarrow e[\overline{\ell_f} \mapsto \overline{\ell}][\overline{x} \mapsto \overline{v}]/s}$$

$$\text{E-MALLOC}\quad \frac{n \geq 0 \qquad r \notin \text{dom}(s)}{\textbf{malloc}(n_{|W|})/s \hookrightarrow \text{ref}(r, 0)/s[r \mapsto ([0, n) \mapsto 0_{|0|})]}$$

$$\text{E-UNFOLD}\quad \frac{}{\textbf{letu } x = \textbf{unfold } \text{ref}(r, n) \textbf{ in } e/s \hookrightarrow e[x \mapsto \text{ref}(r, n)]/s}$$

$$\text{E-FOLD}\quad \frac{}{\textbf{fold } \ell/s \hookrightarrow 0_{|0|}/s}$$

Figure B.2: Small-step semantics of effectful NANOC expressions

$$\text{E-FUN}\quad \frac{P/s \hookrightarrow_{D[\mathtt{f}\mapsto e]} e'/s}{\mathtt{f}\ (\overline{\mathtt{x}_j})\ \{\ e\ \}\ P/s \hookrightarrow_D e'/s'}$$

$$\text{E-PROGRAM}\quad \frac{e/s \hookrightarrow_D e'/s'}{e/s \hookrightarrow_D e'/s'}$$

Figure B.3: Small-step semantics of NANOC programs

Appendix C

Soundness of NANOC Type Checking

In this chapter, we give a proof of type soundness for the NANOC language and associated refinement type system. We begin with an overview of our approach to proving soundness, paying special attention to those aspects of the proof which are unconventional.

C.1 Proof Overview

At a high level, the proof follows the standard format: we prove the usual progress and preservation lemmas, thus ensuring that every well-typed, closed expression either evaluates to a value or does not terminate. However, the presence of a mutable heap, along with the unfold and fold expressions which facilitate strong updates to the types of heap locations, requires us to add extra machinery to relate the run-time store arising from evaluating an expression to the static heap type used to type the expression.

The key element of the soundness proof is the heap modeling relation, $s \vDash_m h$ (Definition 23), read "run-time store s *models* the heap h, where m is a correspondence between run-time store locations r and concrete location names ℓ_j". Intuitively, this relation says that the contents of each block in the run-time store s satisfy a type given by a block type in the heap type h. Which block type that is is specified by the *location map* m (Definition 18), which maps concrete location names to their corresponding run-time location names. (Recall that our concrete locations are meant to correspond to exactly one run-time location.) If a run-time location has a corresponding concrete location ℓ_j in the heap type h — that is, the location is currently unfolded — the run-time block corresponding to that location must satisfy the block type bound to ℓ_j in the heap type h. Otherwise, the location is not unfolded, and the run-time block corresponding to that location

must have the type specified by the corresponding abstract location in h. The key to showing type preservation is demonstrating that this heap modeling relation is preserved through location unfolds and folds (T-UNFOLD and T-FOLD), heap reads and writes (T-READ, T-SUPD, T-WUPD), and memory allocations (T-MALLOC).

In addition to establishing a correspondence between run-time stores and heap types, location maps m are used to assign types to pointer-valued constants, which do not appear in user code but arise as a result of evaluation: they are created by calls to **malloc** and pointer arithmetic operations. Thus, we add the current location map m as an additional parameter of the typing judgment in our updated, proof-oriented typing rules, shown in Figure C.2. A pointer value that points to run-time location r can be typed as a reference to any concrete location ℓ_j such that the location map m maps ℓ_j to r. As evaluation proceeds and new concrete locations are either allocated with **malloc** or unfolded with **unfold**, we introduce new mappings into the location map m. In order to prove preservation, we require that m grows monotonically, i.e., mappings from concrete location names to run-time location names are only ever *added* after a step of evaluation; this ensures that every pointer value can be given the same type both before and after a step of evaluation.

The type checking rules of Section 3.2.3 require that certain concrete location names used in derivations are "fresh"; to put our proof on firm footing, we must specify exactly what it means for a location name to be fresh. It is important that concrete location names assigned to freshly-allocated or unfolded locations are fresh so that every concrete location name corresponds to exactly one run-time location throughout the course of evaluation, which is the crucial requirement for our strong update reasoning to be sound. This restriction is reflected in the proof by our requirement that the location map m is a function, and thus a concrete location name can only be assigned a single run-time location at a given point in evaluation, and that, after a step of evaluation, the location map with which we type the result must *include* m, so that, in the course of evaluating a program, a single concrete location name is never assigned to two different run-time locations.

We can satisfy these requirements if, when we require a fresh concrete location name, we choose a name which is not in the domain of the current location map, m; this ensures that, after a step of evaluation, we can grow the location map, instead of having to replace a binding for an existing location name. However, this restriction alone is not sufficient. Suppose we run expressions e_1 and e_2 in sequence, and our current location map is m. Suppose that we evaluate e_1 to e_1', and evaluating e_1 unfolds a new concrete location ℓ_j. In order to type the resulting sequence expression $e_1'; e_2$, we must expand our location map m to include a binding for ℓ_j. However, we know only that e_2 was typable in m, which did not include a binding for ℓ_j; if we do not make any further restrictions, it may be the case that our initial type derivation for e_2 used the "fresh" location ℓ_j, so that if we add ℓ_j to the location map, that location is no longer fresh and so e_2 is not typable in the expanded location map.

To resolve this problem, we update our typing rules to ensure that two expressions which are evaluated in sequence must use *disjoint* sets of location names (Figure C.2). We parameterize the typing relation by a countably infinite set of concrete location names, I. Our rule for sequence expressions, T-LET, enforces the restriction that sequenced expressions e_1 and e_2 use disjoint concrete location names by typing e_1 with name set I_1 and e_2 with name set I_2, where $I_1 \cap I_2 = \emptyset$. Our preservation lemma ensures that, after a step of evaluation, any concrete locations which must be added to the location map in order to type the resulting expression are be included in the name set used to type the original expression, so that the situation described in the preceding paragraph cannot arise: if a concrete location is unfolded in e_1, it cannot have been used in the type derivation of e_2, since the expressions must have been typed with disjoint sets of location names.

Our proof rests on a complement of substitution lemmas concerning well-formedness, subtyping, and typing. We use standard value substitution lemmas to eliminate free variables from environments; these lemmas are used in the proofs of preservation for **let**, **unfold**, and function call expressions. Because our function types are polymorphic in the names of the heap locations over which they operate, proving preservation in the T-CALL case requires us to prove a series of location name substitution lemmas. Note that we cannot prove that heap well-formedness is preserved by arbitrary location name substitutions: if a substitution maps two location names to a single location name, then a well-formed heap may become ill-formed, as it may now contain two bindings for the same location. Instead, we show that location name substitutions preserve heap well-formedness if they are injective on the location names bound in the heap, i.e., they do not map two distinct location names to the same name. Finally, because the sets of fresh names used to type check a function and the sets of fresh names available in the calling context of a function may differ, we must use substitution to prove type preservation after a function call in two steps. First, we prove that, since both sets of concrete location names involved are countably infinite, we may consistently substitute location names in the type derivation of the callee to obtain a substituted version of the function's type which uses the set of fresh names available in the calling context. Second, we use the fact that function types may not contain concrete locations to eliminate the concrete name substitutions, thus showing type preservation after a function call.

Our type preservation lemma (Lemma 94) incorporates all of the above concerns. We assume a closed, well-typed expression e, which is typable using location map m and a well-formed initial heap h, and suppose that s is a run-time store that models h using location map m. We show that if we take a step of evaluation, yielding expression e' and store s', then there is a heap h_s and location map m' such that the s' models h_s under m', h_s is well-formed, and m' includes m, such that e' is typable in then empty environment and heap h_s under location map m'. We further stipulate that any newly-bound concrete location names added to m' must have come from the set of location names initially used to type e, which, as we have explained above,

is useful in proving preservation in the T-LET case.

To make the proof of preservation for certain rules easier, we make a (results-preserving) simplification to the dynamic semantics of NANOC (Section C.2.1): we assume that every run-time block contains a binding for *all* offsets, rather than some finite region determined by the number of bytes that were requested in a call to **malloc**. This change means that the dynamic semantics can no longer perform bounds checking by checking the domains of run-time blocks; instead, our syntax of constant pointer values uses fat pointers, which carry their bounds information as part of the pointer, and our revised evaluation rules for memory access expressions (Figure C.1) check these bounds at each access. These changes permit three simplifications in the proof and associated definitions. First, we note that an allocated location may not be large enough to hold all the items specified in its block type — for example, a block type may specify a type for the value at offset 10, yet the program may allocate only 8 bytes to a location which has this type. This would complicate the block modeling relation (Definition 22), which determines when a run-time block satisfies a block type; because the field types within a block type may depend on the values defined at non-sequence offsets within the same block, the block modeling relationship is much simpler if we can assume that these values are always defined. Second, the proof of preservation for the T-UNFOLD rule crucially relies on being able to substitute in concrete values for each of the fresh variables introduced to represent the values at non-sequence offsets within the unfolded block; the theory is again much simpler if we can always assume that such values exist. Finally, using fat pointers allows us to give a simple, straightforward definition of the block boundary functions BBegin and BEnd (Assumption 14), which can now be defined as functions on pointer values, without any reference to heap types or stores. We prove that these simplifications do not alter the behavior of programs in any significant way (Lemma 13): terminating expressions yield exactly the same values in either semantics, modulo the difference between fat and thin pointers, and the resulting stores contain the same values, modulo the contents of the out-of-bounds portions of the run-time store.

The progress lemma (Lemma 96) is more standard: We show that, if an expression e is typable with initial heap h and location map m, and if s models h under m, then the expresion is either a value or may take a step of evaluation. The refinement validity lemma (Lemma 27) guarantees that, whenever a pointer is dereferenced, it is non-null and within bounds. Heap modeling guarantees that the location that the pointer refers to actually exists in the run-time store.

Together, the above lemmas imply that closed, well-typed expressions cannot get stuck, but must evaluate to a value or loop forever (Theorem 7). That no well-typed, closed expression can get stuck has three important consequences in our system. First, all memory accesses are to non-null pointers that are within an allocated region of memory. Second, all primitive operators (arithmetic, pointer arithmetic, and relations) are provided with valid operands. Finally, the physical type system ensures that there are never any partial field reads or writes or "overlaps"

between fields in the heap: every read or write to the heap accesses an entire contiguous field, and the fields at each offset within a block are disjoint.

C.2 Changes to the Base Language

To begin, we make several small alterations to NANOC to make the soundness proof easier, and show that these changes do not affect program execution in any significant way.

C.2.1 Dynamic Semantics

In the dynamic semantics of NANOC given in Appendix B, each allocated run-time block has the domain $[0, n)$, where n is the number of bytes allocated to the block by a call to **malloc**. This accurately reflects the fact that a program may not access a run-time block with an out-of-bounds index. Further, reference values are of the form $\text{ref}(r, n)$, for some run-time location r and offset into the location n. This accurately reflects the conventional run-time representation of "thin" pointers that do not carry bounds information.

For the purposes of the soundness proof, however, we make two simplifying changes to the semantics of programs and run-time pointer representation. First, we assume that run-time block has the domain $\mathbb{Z}$, i.e., each block is defined at every possible offset. This simplifies the block modeling relation, which establishes a correspondence between run-time stores and heap types, as well as the cases for T-UNFOLD and T-MALLOC within the preservation lemma, which may freely assume that every run-time block is defined at every possible offset. Second, we change the representation of run-time reference values so that each reference contains the length of the run-time block where it points. This has two effects. First, it simplifies the definitions of the BBegin and BEnd predicates, which can now be defined on reference values directly, without reference to a particular run-time store. Second, it allows us to perform bounds checks in the simplified semantics by using the bounds information contained within the reference itself; this will allow us to show that the simplified and original semantics permit exactly the same evaluations, proving that our simplifications are sound.

We formalize these simplifications in two steps. First, we update the evaluation relation $\hookrightarrow$ to operate on reference values which carry their bounds; note that, while the updated $\hookrightarrow$ propagates these bounds, it does not use them to enforce memory safety, and they may safely be erased. We then create a new evaluation relation, $\rightsquigarrow$, whose evaluation rules for heap reads, writes, and allocations incorporate our assumption that every run-time block is infinitely large and use the bounds information contained in reference values to enforce bounds safety. The updated rules and values are shown in in Figure C.1. The remaining rules for the evaluation relation $\rightsquigarrow$ are unchanged from $\hookrightarrow$, modulo the substitution of $\rightsquigarrow$ for $\hookrightarrow$. We give a bisimulation establishing that expressions have the same behavior under both $\hookrightarrow$ and $\rightsquigarrow$, modulo

v	$::=$		**Values**
		x	variable
	$\mid$	$n_{\lvert w \rvert}$	integer
	$\mid$	$\text{bref}(r, n, z)$	constant pointer with bound

$$\text{bref}(r, n, z) +_p m_{\lvert W \rvert} \hookrightarrow \text{bref}(r, n + m, z) \qquad \text{PE-PTR-PLUS}$$

$$\text{PE-READ}\ \frac{0 \leq n \qquad n + m - 1 < z \qquad v = s(r)(n)}{*_m \text{bref}(r, n, z)/s \hookrightarrow v/s}$$

$$\text{PE-WRITE}\ \frac{0 \leq n \qquad n + \text{SizeOf}(v) - 1 < z}{*\text{bref}(r, n, z) := v/s \hookrightarrow 0_{\lvert 0 \rvert}/s[r \mapsto \text{Write}(s(r), n, v)]}$$

$$\text{PE-MALLOC}\ \frac{z > 0 \qquad r \notin \text{dom}(s)}{\textbf{malloc}(z_{\lvert W \rvert})/s \hookrightarrow \text{bref}(r, 0, z)/s[r \mapsto (\mathbb{Z} \mapsto 0_{\lvert 0 \rvert})]}$$

Figure C.1: Updated reference values and semantics for NANOC

the (inaccessible) contents of the out-of-bounds portions of the run-time store. (We note, without proof, that bounds information does not figure in to evaluation in the original semantics, and may be safely erased once safety has been established.)

Definition 11 (Store Similarity)**.** *Stores s_1 and s_2 are* similar*, written $s_1 \sim s_2$, if*

1. $\text{dom}(s_1) = \text{dom}(s_2)$
2. $\forall r \in \text{dom}(s_1), n \in \text{dom}(s_1(r)).\ s_1(r)(n) = s_2(r)(n)$.

Definition 12 (Reference-Store Consistency)**.** *We say a reference value $v = \text{bref}(r, n, z)$ is* consistent with *store s, written $s \vDash v$, if* $\text{dom}(s(r)) = [0, z)$.

We say pure and impure expressions a and e are consistent with store s, written $s \vDash a$ and $s \vDash e$, respectively, if every value v contained in the expression is consistent with s.

Lemma 9 (Store-Consistent Substitution)**.** *For any store s and reference value v,*

1. *If $s \vDash a$ and $s \vDash v$, then $s \vDash a[x \mapsto v]$.*
2. *If $s \vDash e$ and $s \vDash v$, then $s \vDash e[x \mapsto v]$.*

Proof. Each case proceeds by straightforward induction on the structure of the expression. □

Lemma 10 (Store Consistency Preservation)**.** *For all stores s, pure expressions a, and impure expressions e,*

1. *If $s \vDash a$ and $a \hookrightarrow a'$ then $s \vDash a'$.*
2. *If $s \vDash e$ and $e/s \hookrightarrow e'/s'$ then $s' \vDash e'$.*

Proof. The first case proceeds by cases on the evaluation rule used.

The second case proceeds by straightforward induction on the derivation of $e/s \hookrightarrow e'/s'$, using the first case and Lemma 9. Additionally, the T-CALL case uses the assumption that source programs do not contain reference constants, and hence that $s \vDash e_f$ trivially holds for every function body e_f. □

Lemma 11 (Offsets Fit If And Only If In Reference Bounds)**.**

$$\begin{aligned} &\textit{If } v = \mathrm{bref}(r, n, z) \\ &\textit{and } s \vDash v \\ &\textit{then } \mathrm{Fits}(n, m, s(r)), \\ \textit{if and only if } &0 \leq n \\ &\textit{and } n + m - 1 < z. \end{aligned} \tag{11.1}$$

Proof. By Fact 11.1 and Definition 12,

$$\mathrm{dom}(s(r)) = [0, z).$$

Both directions follow immediately from Definition 9. □

Lemma 12 (Consistent Update Preserves Store Similarity)**.** *If $s_1 \sim s_2$, then*

$$s_1[r \mapsto \mathrm{Write}(s_1(r), n, v)] \sim s_2[r \mapsto \mathrm{Write}(s_2(r), n, v)].$$

Proof. The proof is straightforward from Definition 11. □

Lemma 13 (Bisimulation Between Proof and Ordinary Semantics)**.**

$$\textit{If } s_1 \sim s_2, \tag{13.1}$$

$$\textit{and } s_1 \vDash e, \tag{13.2}$$

$$\begin{aligned} &\textit{then } e/s_1 \hookrightarrow e'/s_1' \\ &\textit{if and only if } e/s_1 \rightsquigarrow e'/s_2', \\ &\textit{with } s_1' \sim s_2'. \end{aligned}$$

Proof. Both directions proceed by straightforward induction on the derivation of the hypothesized evaluation step, splitting cases on the final rule used. We show the forward direction first.

Case $e/s_1 \hookrightarrow e'/s_1'$

The only interesting cases are E-READ, E-WRITE, and E-MALLOC.

Case E-READ

By the form of the rule,

$$e \equiv *_m \text{bref}(r, n, z) \tag{13.3}$$

$$v = s_1(r)(n) \tag{13.4}$$

$$\text{Fits}(n, m, s_1(r)) \tag{13.5}$$

$$e' \equiv v \tag{13.6}$$

$$s_1' = s_1 \tag{13.7}$$

By Fact 13.2 and Definition 12,

$$\text{dom}(s(r)) = [0, z) \tag{13.8}$$

By Fact 13.2, Fact 13.5, and Lemma 11,

$$0 \leq n \tag{13.9}$$

$$n + m - 1 < z \tag{13.10}$$

By Fact 13.1, Fact 13.4, and Definition 11,

$$s_2(r)(n) = v \tag{13.11}$$

By Fact 13.3, Fact 13.9, Fact 13.10, Fact 13.11, and PE-READ,

$$e/s \rightsquigarrow e'/s_2'$$

with

$$s_2' = s_2$$

so that, by Fact 13.7 and Fact 13.1,

$$s_1' \sim s_2'$$

as required.

Case E-WRITE

By the form of the rule,

$$e \equiv *\text{bref}(r, n, z) := v \tag{13.12}$$

$$\text{Fits}(n, \text{SizeOf}(v), s(r)) \tag{13.13}$$

$$e' \equiv 0_{|0|} \tag{13.14}$$

$$s_1' = s_1[r \mapsto \text{Write}(s_1(r), n, v)] \tag{13.15}$$

By Fact 13.2, Fact 13.13, and Lemma 11,

$$0 \leq n \tag{13.16}$$

$$n + \text{SizeOf}(v) - 1 < z \tag{13.17}$$

By Fact 13.16, Fact 13.17, and PE-WRITE,

$$e/s_2 \rightsquigarrow e'/s_2'$$

where

$$s_2' = s_2[r \mapsto \text{Write}(s_2(r), n, v)] \tag{13.18}$$

By Fact 13.14, Fact 13.18, and Lemma 12,

$$s_1' \sim s_2'$$

as required.

Case E-MALLOC

By the form of the rule,

$$e \equiv \textbf{malloc}(n_{|W|}) \tag{13.19}$$

$$n > 0 \tag{13.20}$$

$$r \notin \text{dom}(s_1) \tag{13.21}$$

$$e' \equiv \text{bref}(r, 0, n) \tag{13.22}$$

$$s_1' = s_1[r \mapsto ([0, n) \mapsto 0_{|0|})] \tag{13.23}$$

By Fact 13.1, Fact 13.21, and Definition 11,

$$r \notin \text{dom}(s_2) \tag{13.24}$$

By Fact 13.19, Fact 13.20, Fact 13.24, and E-MALLOC,

$$e/s_2 \rightsquigarrow e'/s_2'$$

where

$$s_2' = s_2[r \mapsto ([0,n) \mapsto 0_{|0|})] \tag{13.25}$$

By Fact 13.1, Fact 13.23, Fact 13.25, and Definition 11,

$$s_1' \sim s_2'$$

as required.

Case $e/s_2 \rightsquigarrow e'/s_2'$

This direction is nearly identical to the other, but uses the reverse direction of Lemma 11.

□

Lemma 14 (Similar Executions)**.** *For any expression e such that* $\emptyset \vDash e$*,*

$$e/\emptyset \hookrightarrow^* e'/s_1$$
$$\textit{if and only if } e/\emptyset \rightsquigarrow^* e'/s_2$$
$$\textit{with } s_1 \sim s_2.$$

Proof. The proof proceeds by straightforward induction on the hypothesized derivation, using Lemma 10 and Lemma 13. □

C.2.2 Static Semantics

The typing rules of section 3.2 are sufficient to type check source-level programs. However, to prove the standard progress and preservation lemmas, we must enhance the language's typing judgments to account for reference-valued constants that appear in the course of evaluation as the result of allocating memory. We update the rules of Section 3.2.3 in two ways. First, to formalize the notion of what it means for a name to be "fresh", we thread a countable set of fresh names, I, through the rules. Note that each variable I is meant to represent an infinite set of names. Second, we add another parameter to the typing judgment, m, which is described in section C.5; its purpose is to allow us to assign types to constant reference values $\text{bref}(r, n, z)$. We show the new and non-trivially-updated rules in Figure C.2; all rules not shown are the same as in Section 3.2.3, except that m and I have been threaded through the derivation.

Updated Pure Typing Rules $\boxed{\Gamma \vdash_m a : \tau}$

$$\frac{\ell \in \text{Clocs}(r, m)}{\Gamma \vdash_m \text{bref}(r, n, z) : \{\nu : \texttt{ref}(\ell, n) \mid \nu = \text{bref}(r, n, z)\}} \text{ T-REF}$$

Updated Impure Typing Rules $\boxed{G,\ \Gamma,\ h \vdash_{m,\ I} e : \tau / h_2}$

$$\frac{\Gamma \vdash_m v : \texttt{int}(W, i) \qquad G,\ \Gamma; v \neq 0,\ h \vdash_{m,\ I} e_1 : \tau / h' \qquad G,\ \Gamma; v = 0,\ h \vdash_{m,\ I} e_2 : \tau / h'}{G,\ \Gamma,\ h \vdash_{m,\ I} \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2 : \tau / h'} \text{ T-IF}$$

$$\frac{\begin{array}{c} G,\ \Gamma,\ h \vdash_{m,\ I_1} e_1 : \tau_1 / h_1 \\ G,\ \Gamma; x : \tau_1,\ h_1 \vdash_{m,\ I_2} e_2 : \tau_2 / h_2 \qquad \Gamma \vDash \tau_2 / h_2 \qquad I_1 \cap I_2 = \emptyset \end{array}}{G,\ \Gamma,\ h \vdash_{m,\ I_1 \cup I_2} \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2 / h_2} \text{ T-LET}$$

$$\frac{\begin{array}{c} \Gamma \vdash_m v : \{\nu : \texttt{ref}(\widetilde{\ell}, i_y) \mid \nu \neq 0\} \qquad h = h_0 * \widetilde{\ell} \mapsto \overline{n_k} : \overline{\tau_k}, \overline{i^+} : \overline{\tau^+} \\ \overline{x_k} \text{ disjoint} \qquad \overline{x_k} \notin \Gamma,\ e,\ \text{FV}(h) \qquad \theta = [\overline{@n_k} \mapsto \overline{x_k}] \\ \Gamma_1 = \Gamma; \overline{x_k} : \overline{\theta \tau_k} \qquad \ell_j \notin \Gamma, h, m \qquad h_1 = h * \ell_j \mapsto \overline{n_k} : \overline{\{\nu = x_k\}}, \overline{i^+} : \overline{\theta \tau^+} \\ G,\ \Gamma_1; x : \{\nu : \texttt{ref}(\ell_j, i_y) \mid \nu = v\},\ h_1 \vdash_{m,\ I} e : \tau_2 / h_2 \qquad \Gamma_1 \vDash h_1 \qquad \Gamma \vDash \tau_2 / h_2 \end{array}}{G,\ \Gamma,\ h \vdash_{m,\ I \cup \{\ell_j\}} \textbf{letu } x = \textbf{unfold } v \textbf{ in } e : \tau_2 / h_2} \text{ T-UNFOLD}$$

$$\frac{\begin{array}{c} \ell_j \notin \Gamma, h, m \qquad h = h_0 * \widetilde{\ell} \mapsto b \qquad \Gamma \vDash h * \ell_j \mapsto b \\ \Gamma \vdash_m v : \{\nu : \texttt{int}(W, i) \mid \nu > 0\} \qquad \tau = \{\nu : \texttt{ref}(\ell_j, 0) \mid \text{Allocated}(\nu, v)\} \end{array}}{G,\ \Gamma,\ h \vdash_{m,\ I \cup \{\ell_j\}} \textbf{malloc}(v) : \tau / h * \ell_j \mapsto b^0} \text{ T-MALLOC}$$

Figure C.2: Updated typing rules for NANOC expressions

C.3 Logical Embedding

We assume a multi-sorted logic, with the sorts $\texttt{int}(w)$ and $\texttt{ref}$, corresponding to w-byte integer-valued data and pointer-valued data, respectively. We use the metavariable s to stand in for sorts.

Definition 13 (Embedding Environments as Predicates)**.** *We embed environments Γ into our refinement logic as*

$$[\![\Gamma]\!] \equiv \bigwedge \{\phi \mid \phi \in \Gamma\} \wedge \bigwedge \{\phi[\nu \mapsto x] \mid x : \{\nu : t \mid \phi\} \in \Gamma\}.$$

Definition 14 (Sorts from Types)**.** *The corresponding sort for a refinement type is given by*

$$\text{Sort}(\{\nu : \texttt{int}(w, i) \mid \phi\}) = \texttt{int}(w)$$
$$\text{Sort}(\{\nu : \texttt{ref}(\ell, i) \mid \phi\}) = \texttt{ref}$$

Definition 15 (Sort Environments from Type Environments)**.** *We create a sort environment from a type environment using the operation* SortEnv*:*

$$\begin{aligned}\text{SortEnv}(\emptyset) &= \emptyset \\ \text{SortEnv}(x : \tau; \Gamma) &= x : \text{Sort}(\tau); \text{SortEnv}(\Gamma) \\ \text{SortEnv}(a; \Gamma) &= \text{SortEnv}(\Gamma)\end{aligned}$$

Definition 16 (Well-Sorted Refinement Predicates)**.** *We say that ϕ well-sorted in Γ if ϕ is a well-sorted predicate in the refinement logic under the sort environment* $\text{SortEnv}(\Gamma)$*. A well-sorted predicate may not contain any offset expressions @n.*

C.4 Concrete Name Sets

In this section, we give lemmas and definitions concerning the substitution of concrete location names.

Assumption 7 (Disjoint Concrete Location Numberings)**.** *We assume that, for any concrete location set I, if $\ell_j, \ell'_k \in I$, then $j \neq k$. In particular, this means that, for any location name substitution ρ, $\rho(\ell_j) \neq \rho(\ell'_k)$.*

Definition 17 (Concrete Name Substitution)**.** *A* concrete name substitution *ω is a injective function on concrete location names,*

$$\omega : CLoc \to CLoc,$$

where $CLoc$ is the set of all concrete location names. We require that concrete name substitutions preserve location names, i.e., that $\omega(\ell_j) = \ell_k$ for some k.

Lemma 15 (Concrete Name Substitution: Well-Formedness)**.** *For any concrete name substitution ω,*

1. *If $\Gamma \vDash \tau$, then $\omega\Gamma \vDash \omega\tau$.*
2. *If $\Gamma \vDash b$, then $\omega\Gamma \vDash \omega b$.*
3. *If $\Gamma \vDash_{@} b$, then $\omega\Gamma \vDash_{@} \omega b$.*
4. *If $\Gamma \vDash h$, then $\omega\Gamma \vDash \omega h$.*
5. *If $\Gamma \vDash \tau / h$, then $\omega\Gamma \vDash \omega\tau / \omega h$.*

Proof. The first case proceeds by cases on the rule used to prove $\Gamma \vDash \tau$, while the next three cases proceed by straightforward induction on the hypothesized derivation, splitting cases on the final rule used. The final case is immediate from the previous cases. □

Lemma 16 (Concrete Name Substitution: Subtyping)**.** *For any concrete name substitution ω,*

1. *If* $\Gamma \vdash \tau_1 <: \tau_2$, *then* $\omega\Gamma \vdash \omega\tau_1 <: \omega\tau_2$.
2. *If* $\Gamma \vdash b_1 <: b_2$, *then* $\omega\Gamma \vdash \omega b_1 <: \omega b_2$.
3. *If* $\Gamma \vdash h_1 <: h_2$, *then* $\omega\Gamma \vdash \omega h_1 <: \omega h_2$.
4. *If* $\Gamma \vdash \tau_1/h_1 <: \tau_2/h_2$, *then* $\omega\Gamma \vdash \omega\tau_1/\omega h_1 <: \omega\tau_2/\omega h_2$.

Proof. Each proof proceeds by induction on the structure of the hypothesized derivation, splitting cases on the final rule used and using the previous cases of the lemma. □

Lemma 17 (Concrete Name Substitution: Pure Typing)**.**

$$\begin{gathered} \textit{If } \Gamma \vdash_m a : \tau \\ \textit{then } \omega\Gamma \vdash_{\omega m} a : \omega\tau \end{gathered}$$

Proof. The proof proceeds by straightforward induction on the derivation of $\Gamma \vdash_m a : \tau$, splitting cases on the final rule used, and using Lemma 15 and Lemma 16. □

Lemma 18 (Concrete Name Substitution: Typing)**.**

$$\begin{gathered} \textit{If } G,\ \Gamma,\ h \vdash_{m,\ I} e : \tau/h \\ \textit{then } G,\ \omega\Gamma,\ \omega h \vdash_{\omega m,\ \omega I} e : \omega\tau/\omega h \end{gathered}$$

Proof. The proof proceeds by straightforward induction on the derivation of

$$G,\ \Gamma,\ h \vdash_{m,\ I} e : \tau/h,$$

splitting cases on the final rule used and using Lemma 15, Lemma 16, and Lemma 17. We also use the fact that ω is injective to guarantee that concrete name sets which were distinct in the original derivation remain so in the updated derivation. □

Lemma 19 (Concrete Name Set Weakening)**.**

$$\begin{gathered} \textit{If } G,\ \Gamma,\ h \vdash_{m,\ I_1} e : \tau/h' \\ \textit{and } I_1 \subseteq I_2, \\ \textit{then } G,\ \Gamma,\ h \vdash_{m,\ I_2} e : \tau/h' \end{gathered}$$

Proof. The proof proceeds by straightforward induction on the derivation of

$$G,\ \Gamma,\ h \vdash_{m,\ I_1} e : \tau/h',$$

splitting cases on the final rule used. □

C.5 Relating Run-Time Stores and Heap Types

Definition 18 (Location Map)**.** *A location map is a function $m : CLoc \to RLoc$ from concrete location names to run-time location names.*

The initial location map is the empty function $\emptyset$.

Definition 19 (Concrete Locations of a Run-Time Location)**.** *Given a run-time location r and a location map m, we define the set of concrete locations corresponding to r,* Clocs*, as the inverse image of r under m:*

$$\text{Clocs}(r, m) = m^{-1}(r)$$

Definition 20 (Location Map Well-Formedness)**.** *A location map m is* well-formed*, written $\vDash m$, iff for all $r \in \text{rng}(m)$, if $\ell_j, \ell'_k \in \text{Clocs}(r, m)$, then $\ell = \ell'$.*

Definition 21 (Location Order)**.** *We define an ordering on location names, $\ell_1 \sqsubseteq \ell_2$, as the smallest reflexive, transitive relation satisfying $\ell_j \sqsubseteq \widetilde{\ell}$.*

Definition 22 (Block Modeling)**.** *We define when a run-time block c* models *block type b under location map m, written $c \vDash_m b$, according to the following inference rules:*

$$\frac{\emptyset \vdash_m c(n) : \tau \qquad c \vDash_m b[@n \mapsto c(n)]}{c \vDash_m n : \tau,\ b}\ \text{BM-SINGLE}$$

$$\frac{\forall n \in \text{dom}(c) \cap [\![i^+]\!].\ \emptyset \vdash_m c(n) : \tau \qquad c \vDash_m b}{c \vDash_m i^+ : \tau,\ b}\ \text{BM-SEQUENCE}$$

Definition 23 (Heap Modeling)**.** *Run-time store s* models *heap type h under location map m, written $s \vDash_m h$, iff*

1. $\forall \ell_j \in \text{dom}(h).\ \ell_j \in \text{dom}(m)$
2. $\text{rng}(m) \subseteq \text{dom}(s)$
3. *For all $r \mapsto c \in s$, exists $\ell_j \in \text{Clocs}(r, m)$, and either*
   (a) $h = h_1 * \ell_j \mapsto b * h_2$*, and $c \vDash_m b$, or*
   (b) $\text{Clocs}(r, m) \cap \text{dom}(h) = \emptyset$, $h = h_1 * \widetilde{\ell} \mapsto b * h_2$*, and $c \vDash_m b$.*

We use the following lemma to justify reordering a heap's bindings as needed to make the proof's notation simpler:

Lemma 20 (Location Order is Irrelevant in Heap Modeling)**.** *$s \vDash_m h_1 * h_2$ if and only if $s \vDash_m h_2 * h_1$.*

Proof. Immediate from Definition 23. □

Lemma 21 (Location Map Weakening: Pure Typing)**.** *If $\Gamma \vdash_{m_1} a : \tau$ and $m_1 \subseteq m_2$, then $\Gamma \vdash_{m_2} a : \tau$.*

Proof. The proof proceeds by induction on the derivation of $\Gamma \vdash_{m_1} a : \tau$, splitting cases on the final rule used.

The only interesting case is T-REF. By the form of the rule,

$$\begin{aligned} a &\equiv \text{bref}(r, n, z) \\ \tau &= \{\nu : \texttt{ref}(\ell, n) \mid \nu = \text{bref}(r, n, z)\} \\ \ell &\in \text{Clocs}(r, m_1) \end{aligned} \tag{21.1}$$

Since $m_1 \subseteq m_2$,

$$\text{Clocs}(r, m_1) \subseteq \text{Clocs}(r, m_2) \tag{21.2}$$

So by Fact 21.1, Fact 21.2, and T-REF,

$$\Gamma \vdash_{m_2} \text{bref}(r, n, l) : \{\nu : \texttt{ref}(\ell, n) \mid \nu = \text{bref}(r, n, z)\}$$

as required. □

Lemma 22 (Location Map Weakening)**.**

$$\begin{aligned} &\textit{If } G,\ \Gamma,\ h \vdash_{m_1,\ I} e : \tau / h', \\ &\quad m_1 \subseteq m_2, \\ &\textit{and } \text{dom}(m_2 \setminus m_1) \cap I = \emptyset, \\ &\textit{then } G,\ \Gamma,\ h \vdash_{m_2,\ I} e : \tau / h'. \end{aligned}$$

Proof. The proof proceeds by induction on the derivation of $G,\ \Gamma,\ h \vdash_{m_1,\ I} e : \tau / h'$, splitting cases on the final rule used. The requirement that $\text{dom}(m_2 \setminus m_1) \cap I = \emptyset$ is used in the cases for T-UNFOLD and T-MALLOC to ensure that we can use the same concrete locations in both derivations. □

Lemma 23 (Abstract Types and Location Map Weakening)**.**

$$\textit{If } G,\ \Gamma,\ h \vdash_{\emptyset,\ I} e : \tau / h' \tag{23.1}$$

$$\textit{and } \Gamma, h, \tau, h' \textit{ abstract}, \tag{23.2}$$

$$\textit{then } G,\ \Gamma,\ h \vdash_{m,\ I} e : \tau / h'.$$

Proof. Let ω be such that

$$I \setminus \text{dom}(m) = \omega I. \tag{23.3}$$

We know such a substitution exists because I is countable. By Fact 23.1 and Lemma 18,

$$G,\ \omega\Gamma,\ \omega h \vdash_{\emptyset,\ \omega I} e : \omega\tau / \omega h.$$

By Fact 23.2, this is equivalent to

$$G,\ \Gamma,\ h \vdash_{\emptyset,\ \omega I} e : \tau / h. \tag{23.4}$$

By Fact 23.3, Fact 23.4, and Lemma 22,

$$G,\ \Gamma,\ h \vdash_{m,\ \omega I} e : \tau / h. \tag{23.5}$$

By Fact 23.3,

$$\omega I \subseteq I. \tag{23.6}$$

By Fact 23.5, Fact 23.6, and Lemma 19,

$$G,\ \Gamma,\ h \vdash_{m,\ I} e : \tau / h,$$

as required. □

Lemma 24 (Location Map Weakening Preserves Block Modeling)**.**

$$\begin{gathered} \textit{If } c \vDash_{m_1} b \\ \textit{and } m_1 \subseteq m_2 \\ \textit{then } c \vDash_{m_2} b. \end{gathered}$$

Proof. Straightforward induction on the derivation of $c \vDash_{m_1} b$, using Lemma 21. □

Lemma 25 (Typable References)**.** *If* $\Gamma \vdash_m \text{bref}(r, n, z) : \tau$*, then* $r \in \text{rng}(m)$*.*

Proof. The proof proceeds by straightforward induction on the derivation of $\Gamma \vdash_m \text{bref}(r, n, z) : \tau$. The only interesting case is T-REF, where the conclusion follows immediately from the premise that there exists some $\ell \in \text{Clocs}(r, m)$ and Definition 19. □

C.6 Index Properties

Proposition 1 (Reflexivity and Transitivity of $\widetilde{\subseteq}$)**.** *The operator* $\widetilde{\subseteq}$ *is reflexive and transitive.*

Proof. Immediate from the definition of $\widetilde{\subseteq}$. □

Proposition 2 (Singleton Subindexing is Inclusion)**.** $n \in [\![i]\!]$ *if and only if* $n \mathrel{\widetilde{\subseteq}} i$*.*

Proof. Immediate from the definition of $\widetilde{\subseteq}$. □

Proposition 3 (Soundness of Abstract Arithmetic)**.** *If* $n_1 \mathrel{\widetilde{\subseteq}} i_1$ *and* $n_2 \mathrel{\widetilde{\subseteq}} i_2$*, and* $n_1 \circ n_2$ *is defined, then* $n_1 \circ n_2 \mathrel{\widetilde{\subseteq}} i_1 \mathrel{\widetilde{\circ}} i_2$*.*

Proof. We prove the case for each operator separately, splitting cases on the forms of i_1 and i_2.

Case $\widetilde{+}$

We split cases on the forms of i_1 and i_2.

Case $i_1 = n, i_2 = m$

Immediate.

Case $i_1 = n, i_2 = [l,\ u]_m^c$

Then $i_1 \mathbin{\widetilde{+}} i_2 = [l + n,\ u + n]_m^{c+n \bmod m}$. Let $n_1 \mathbin{\widetilde{\subseteq}} i_1$ and $n_2 \mathbin{\widetilde{\subseteq}} i_2$. Then by Proposition 2 and Figure 3.10,

$$n_1 = n$$

$$l \leq n_2 \tag{25.1}$$

$$n_2 \leq u \tag{25.2}$$

$$n_2 \equiv c \bmod m \tag{25.3}$$

By Fact 25.1,

$$l + n_1 \leq n_1 + n_2 \tag{25.4}$$

By Fact 25.2,

$$n_1 + n_2 \leq u + n_1 \tag{25.5}$$

By Fact 25.3,

$$n_1 + n_2 \equiv c + n_1 \bmod m \tag{25.6}$$

By Fact 25.4, Fact 25.5, Fact 25.6, and the definition of $[\![i]\!]$,

$$n_1 + n_2 \in [\![[l + n_1,\ u + n_1]_m^{c+n_1 \bmod m}]\!]$$

By Proposition 2, this is equivalent to

$$n_1 + n_2 \mathbin{\widetilde{\subseteq}} [l + n_1,\ u + n_1]_m^{c+n_1 \bmod m}$$

as required.

Case $i_1 = [l_1,\ u_1]_{m_1}^{c_1}, i_2 = [l_2,\ u_2]_{m_2}^{c_2}$

Then $i_1 \mathbin{\widetilde{+}} i_2 = [l_1 + l_2,\ u_1 + u_2]_{\gcd(m_1,m_2,c_1,c_2)}^0$. Let $n_1 \mathbin{\widetilde{\subseteq}} i_1$ and $n_2 \mathbin{\widetilde{\subseteq}} i_2$. Then by

Proposition 2 and Figure 3.10,

$$l_1 \leq n_1 \tag{25.7}$$
$$n_1 \leq u_1 \tag{25.8}$$
$$n_1 \equiv c_1 \bmod m_1 \tag{25.9}$$
$$l_2 \leq n_2 \tag{25.10}$$
$$n_2 \leq u_2 \tag{25.11}$$
$$n_2 \equiv c_2 \bmod m_2 \tag{25.12}$$

By Fact 25.7 and Fact 25.10,

$$l_1 + l_2 \leq n_1 + n_2 \tag{25.13}$$

By Fact 25.8 and Fact 25.11,

$$n_1 + n_2 \leq u_1 + u_2 \tag{25.14}$$

By Fact 25.9,

$$n_1 - c_1 = xm_1 \text{ for some } x \tag{25.15}$$

By Fact 25.12,

$$n_2 - c_2 = ym_2 \text{ for some } y \tag{25.16}$$

By Fact 25.15 and Fact 25.16,

$$n_1 + n_2 = xm_1 + ym_2 + c_1 + c_2 \tag{25.17}$$

Let

$$d = \gcd(m_1, m_2, c_1, c_2)$$

Then, by Fact 25.17,

$$\begin{aligned} n_1 + n_2 &= xdm_1' + ydm_2' + dc_1' + dc_2' \\ &= d(xm_1' + ym_2' + c_1' + c_2') \end{aligned}$$

for some m_1', m_2', c_1', and c_2'. Equivalently,

$$n_1 + n_2 \equiv 0 \bmod d \tag{25.18}$$

By Fact 25.13, Fact 25.14, Fact 25.18, and the definition of $[\![i]\!]$,

$$n_1 + n_2 \in [\![[l_1 + l_2,\ u_1 + u_2]^0_{\gcd(m_1,m_2,c_1,c_2)}]\!]$$

By Proposition 2, this is equivalent to

$$n_1 + n_2 \mathrel{\widetilde{\subseteq}} [l_1 + l_2,\ u_1 + u_2]^0_{\gcd(m_1,m_2,c_1,c_2)}$$

as required.

Case $\mathrel{\widetilde{\cdot}}$

We split cases on the forms of i_1 and i_2.

Case $i_1 = n$, $i_2 = m$

Immediate.

Case $i_1 = n$, $i_2 = [l,\ u]^c_m$

Then $i_1 \mathrel{\widetilde{\cdot}} i_2 = [nl,\ nu]^{nc}_{nm}$. Let $n_1 \mathrel{\widetilde{\subseteq}} i_1$ and $n_2 \mathrel{\widetilde{\subseteq}} i_2$. Then by Proposition 2 and Figure 3.10,

$$n_1 = n$$
$$l \leq n_2 \tag{25.19}$$
$$n_2 \leq u \tag{25.20}$$
$$n_2 \equiv c \bmod m \tag{25.21}$$

By Fact 25.19,

$$n_1 l \leq n_1 n_2 \tag{25.22}$$

By Fact 25.20,

$$n_1 n_2 \leq n_1 u_2 \tag{25.23}$$

By Fact 25.21,

$$n_2 - c = xm \text{ for some } x$$

Then

$$\begin{aligned} n_1(n_2 - c) &= n_1 n_2 - n_1 c \\ &= n_1 xm \\ &= x(n_1 m) \end{aligned}$$

Equivalently,

$$n_1 n_2 \equiv n_1 c \bmod n_1 m \tag{25.24}$$

By Fact 25.22, Fact 25.23, Fact 25.24, and the definition of $[\![i]\!]$,

$$n_1 n_2 \in [\![[n_1 l,\ n_1 u]^{n_1 c}_{n_1 m}]\!]$$

By Proposition 2, this is equivalent to

$$n_1 n_2 \mathrel{\tilde{\subseteq}} [n_1 l,\ n_1 u]^{n_1 c}_{n_1 m}$$

as required.

Case $i_1 = [l_1,\ u_1]^{c_1}_{m_1}, i_2 = [l_2,\ u_2]^{c_2}_{m_2}$
Immediate.

Case $\tilde{/}$
Immediate. □

C.7 Logical Assumptions

We assume a many-sorted refinement logic that subsumes quantifier-free first-order logic with equality and theories of integer and pointer arithmetic. We detail our assumptions with respect to well-sortedness and the theories of integer and pointer arithmetic below.

Assumption 8 (Well-Sorted Logical Substitution)**.** *If ϕ is well-sorted in sort environment Γ and x and t have the same sort in Γ, then $\phi[x \mapsto t]$ is well-sorted in Γ.*

Assumption 9 (Free Variables in Well-Sorted Predicates)**.** *If ϕ well-sorted in Γ, then* $\mathrm{FV}(\phi) \subseteq \mathrm{dom}(\Gamma)$.

Assumption 10 (Values' Logical Sorts)**.** *For every value v, if $\emptyset \vdash_m v : \tau$, then v has sort* $\mathrm{Sort}(\tau)$.
In particular, zero-valued integers $0_{|w|}$ can take on both $\texttt{int}(w)$ *and* `ref` *sorts.*

Assumption 11 (Weakening Well-Sortedness)**.**

$$\begin{gathered}\textit{If } \phi \textit{ well-sorted in } \Gamma_1; \Gamma_2 \\ \textit{and } \mathrm{dom}(\Gamma_1; \Gamma_2) \cap \mathrm{dom}(\Gamma) = \emptyset, \\ \textit{then } \phi \textit{ well-sorted in } \Gamma_1; \Gamma; \Gamma_2.\end{gathered}$$

Assumption 12 (Strengthening Well-Sortedness)**.** *If ϕ well-sorted in $\Gamma; x : s$ and $x \notin \mathrm{FV}(\phi)$, then ϕ well-sorted in Γ.*

Assumption 13 (Theories of Arithmetic)**.** *The refinement logic includes a complete theory of arithmetic for each integer width. That is, for all w, n_1, n_2, and operations $\circ$,*

$$n_{1|w|} \circ n_{2|w|} = (n_1 \circ n_2)_{|w|}$$

is valid.

We also assume a theory of pointer arithmetic such that the following two are valid:

$$\text{bref}(r, n_1, z) +_p n_{2|W|} = \text{bref}(r, n_1 + n_2, z)$$
$$0_{|W|} +_p n_{2|W|} = 0_{|W|}$$

For all values v_1, v_2 of the same sort, we assume that:

1. *$v_1 \bowtie v_2$ is valid iff $v_1 \bowtie v_2$.*
2. *$v_1 \bowtie v_2$ is valid iff $\neg(v_1 \bowtie v_2)$*

Assumption 14 (Block Boundary Functions)**.** *We define the* BBegin *and* BEnd *functions as follows:*

$$\text{BBegin}(\text{bref}(r, n, z)) = \text{bref}(r, 0, z)$$
$$\text{BEnd}(\text{bref}(r, n, z)) = \text{bref}(r, z, z)$$

$$\text{BBegin}(0_{|W|}) = 0_{|W|}$$
$$\text{BEnd}(0_{|W|}) = 0_{|W|}$$

C.8 Relating Logic and Typing

Lemma 26 (Subtyping Implication)**.** *If $\emptyset \vdash \{\nu : t_1 \mid \phi_1\} <: \{\nu : t_2 \mid \phi_2\}$, then $\emptyset \vDash \phi_1 \Rightarrow \phi_2$.*

Proof. The proof proceeds by straightforward induction on the derivation of $\emptyset \vdash \{\nu : t_1 \mid \phi_1\} <: \{\nu : t_2 \mid \phi_2\}$, splitting cases on the final rule used.

□

Lemma 27 (Refinement Validity)**.** *For any value v, if $\emptyset \vdash_m v : \{\nu : t \mid \phi\}$, then $\phi[\nu \mapsto v]$ is valid.*

Proof. The proof proceeds by straightforward induction on the derivation of $\emptyset \vdash_m v : \{\nu : t \mid \phi\}$, splitting cases on the final rule used and using Lemma 26 in the T-PURESUB case. □

C.9 Environments, Free Variables, and Free Locations

Definition 24 (Well-Formed Environment)**.** *The following judgments define what it means for an environment to be well-formed:*

$$\frac{}{\vDash \varnothing}\ \text{WE-EMPTY}$$

$$\frac{\vDash \Gamma \qquad x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vDash \tau}{\vDash \Gamma; x : \tau}\ \text{WE-VAR}$$

$$\frac{\vDash \Gamma \qquad \phi \textit{ well-sorted in } \Gamma}{\vDash \Gamma; \phi}\ \text{WE-PRED}$$

Definition 25 (Well-Formed Global Environments)**.** *A global environment G is well-formed, written $\vDash G$, iff, for every binding*

$$f : (\overline{x_i} : \overline{\tau_i})/h_1 \ \rightarrow\ \tau/h_2$$

in G, the following hold:

1. $\vDash (\overline{x_i} : \overline{\tau_i})/h_1 \ \rightarrow\ \tau/h_2$
2. $\mathrm{Locs}(h_1) = \mathrm{Locs}(h_2)$

Definition 26 (Global Environment Modeling)**.** *A mapping from function names to their bodies D* models *a global envirnonment G, written $D \vDash G$, if*

$$G(f) = (\overline{x_i} : \overline{\tau_i})/h_1 \ \rightarrow\ \tau/h_2$$
$$\textit{and } G,\ \overline{x_i} : \overline{\tau_i},\ h_1 \vdash_{\varnothing,\, I_f} D(f) : \tau/h_2$$

for some countable set of concrete location names I_f.

Definition 27 (Types' Free Variables)**.** *We define the free variables of types as follows:*

$$\mathrm{FV}(\{\nu :\ t \mid \phi\}) \triangleq \mathrm{FV}(\phi) \setminus \nu$$

$$\mathrm{FV}(\overline{i_j} : \overline{\tau_j}) \triangleq \bigcup_j \mathrm{FV}(\tau_j)$$

$$\mathrm{FV}(\varnothing) \triangleq \varnothing$$
$$\mathrm{FV}(h * \ell \mapsto b) \triangleq \mathrm{FV}(h) \cup \mathrm{FV}(b)$$

Definition 28 (Types' Free Locations)**.** *We define the free locations of a type* $\mathrm{FL}(\{\nu :\ t \mid \phi\})$ *as the set* $\{@n \mid @n \textit{ appears in } \phi\}$.

Lemma 28 (Free Variables from Environments)**.** *For any environment* Γ*,*

1. *If* $\Gamma \vDash \tau$*, then* $\mathrm{FV}(\tau) \subseteq \mathrm{dom}(\Gamma)$*.*
2. *If* $\Gamma \vDash b$*, then* $\mathrm{FV}(b) \subseteq \mathrm{dom}(\Gamma)$*.*
3. *If* $\Gamma \vDash_@ b$*, then* $\mathrm{FV}(b) \subseteq \mathrm{dom}(\Gamma)$*.*
4. *If* $\Gamma \vDash h$*, then* $\mathrm{FV}(h) \subseteq \mathrm{dom}(\Gamma)$*.*
5. *If* $\Gamma \vDash \tau / h$*, then* $\mathrm{FV}(\tau) \subseteq \mathrm{dom}(\Gamma)$ *and* $\mathrm{FV}(h) \subseteq \mathrm{dom}(\Gamma)$*.*

Proof. We consider each case separately.

1. By WF-TYPE, we have:

$$\begin{aligned} &\tau = \{\nu : t \mid \phi\}, \\ &\phi \text{ well-sorted in } \Gamma. \end{aligned}$$

   By Assumption 9,

$$\mathrm{FV}(\phi) \subseteq \mathrm{dom}(\Gamma).$$

   By Definition 27,

$$\mathrm{FV}(\{\nu : t \mid \phi\}) \triangleq \mathrm{FV}(\phi).$$

   so

$$\mathrm{FV}(\{\nu : t \mid \phi\}) \subseteq \mathrm{dom}(\Gamma).$$

2. Immediate by (1), the hypotheses of WF-NDBLOCK, and Definition 27.

3. The proof proceeds by induction on the derivation of $\Gamma \vDash_@ b$. We split cases on the final rule used.

   Case WF-DBLOCK-SEQUENCE
   Immediate by the rule's premise, (2), and the inductive hypothesis.

   Case WF-DBLOCK-SINGLE
   By the form of the rule,

$$\begin{aligned} &b = n : \tau,\ \overline{i_j} : \overline{\tau_j} \\ &x \notin \Gamma,\ \mathrm{FV}(\overline{\tau_j}) && (28.1) \\ &\Gamma \vDash \tau && (28.2) \\ &\Gamma, x : \tau \vDash_@ \overline{i_j} : \overline{\tau_j[@n \mapsto x]} && (28.3) \end{aligned}$$

By (1) and Fact 28.2,

$$\mathrm{FV}(\tau) \subseteq \mathrm{dom}(\Gamma) \tag{28.4}$$

By the inductive hypothesis and Fact 28.3,

$$\mathrm{FV}(\overline{i_j} : \overline{\tau_j[@n \mapsto x]}) \subseteq \mathrm{dom}(\Gamma) \cup \{x\}$$

By Fact 28.1, $x \notin \mathrm{FV}(\overline{\tau_j})$, and the substitution shown can only add free variable x, so this implies

$$\mathrm{FV}(\overline{i_j} : \overline{\tau_j}) \subseteq \mathrm{dom}(\Gamma) \tag{28.5}$$

By Fact 28.4, Fact 28.5, and Definition 27,

$$\mathrm{FV}(b) \subseteq \mathrm{dom}(\Gamma)$$

as required.

4. Straightforward induction on the derivation of $\Gamma \vDash h$, splitting cases on the final rule used and invoking (2) and (3) as appropriate.

5. Straightforward from the form of WF-WORLD, (1), and (4).

□

Lemma 29 (Well-Formed Non-Dependent Blocks Have No Free Locations)**.**

$$\textit{If } \Gamma \vDash \overline{i} : \overline{\tau}$$
$$\textit{then } \mathrm{FL}(\overline{\tau}) = \emptyset.$$

Proof. Immediate by the form of WF-NDBLOCK and the fact that a well-formed type cannot have free locations by Definition 16. □

C.9.1 Weakening

Lemma 30 (Well-Formedness Weakening)**.** *If $\Gamma_1; \Gamma; \Gamma_2$ is a well-formed environment, then:*

1. *If $\Gamma_1; \Gamma_2 \vDash \tau$, then $\Gamma_1; \Gamma; \Gamma_2 \vDash \tau$.*

2. *If $\Gamma_1; \Gamma_2 \vDash b$, then $\Gamma_1; \Gamma; \Gamma_2 \vDash b$.*

3. *If $\Gamma_1; \Gamma_2 \vDash_@ b$, then $\Gamma_1; \Gamma; \Gamma_2 \vDash_@ b$.*

Proof. We consider each case separately.

1. The only rule that applies is WF-TYPE, from which we have

$$\tau = \{\nu : t \mid \phi\}$$
$$\phi \text{ well-sorted in } \Gamma_1; \Gamma_2$$

   By the assumption that $\Gamma_1; \Gamma; \Gamma_2$ is well-formed, $\text{dom}(\Gamma_1; \Gamma_2) \cap \text{dom}(\Gamma) = \emptyset$, so by Assumption 11,

$$\phi \text{ well-sorted in } \Gamma_1; \Gamma; \Gamma_2$$

   By WF-TYPE,

$$\Gamma_1; \Gamma; \Gamma_2 \vDash \tau$$

   as required.

2. Follows immediately using (1).

3. The proof proceeds by straightforward induction on the derivation of $\Gamma \vDash_@ b$, using (1) and (2).

□

Lemma 31 (Subtyping Weakening)**.** *If* $\Gamma \vdash \tau_1 <: \tau_2$*, then* $\Gamma; \Gamma' \vdash \tau_1 <: \tau_2$.

Proof. The proof proceeds by induction on the derivaton of $\Gamma \vdash \tau_1 <: \tau_2$. We split cases on the final rule used.

Case <:-INT

By the form of the rule, we have

$$i_1 \overset{\sim}{\subseteq} i_2 \tag{31.1}$$
$$\Gamma \vDash \phi_1 \Rightarrow \phi_2 \tag{31.2}$$

By logical monotonicity,

$$\Gamma; \Gamma' \vDash \phi_1 \Rightarrow \phi_2 \tag{31.3}$$

By <:-INT, Fact 31.1, and Fact 31.3,

$$\Gamma; \Gamma' \vdash \tau_1 <: \tau_2 \tag{31.4}$$

as required.

Case <:-REF

Similar to the case for <:-INT.

Case <:-ABSTRACT, <:-NULL

Immediate.

Case <:-TRANS

Immediate by the inductive hypothesis and a use of <:-TRANS. □

Lemma 32 (Pure Type Weakening)**.**

$$\begin{aligned} &\textit{If } \Gamma; \Gamma' \textit{ is a well-formed environment} \\ &\textit{and } \Gamma \vdash_m a : \tau \\ &\textit{then } \Gamma; \Gamma' \vdash_m a : \tau. \end{aligned} \tag{32.1}$$

Proof. By induction on the derivation of $\Gamma \vdash_m a : \tau$. We split cases on the final rule used.

Case T-VAR

By the form of the rule, we have

$$\begin{aligned} a &\equiv x \\ \Gamma(x) &= \{\nu : t \mid \phi\} \\ \tau &= \{\nu : t \mid \nu = x\} \end{aligned}$$

Since $\Gamma; \Gamma'$ is well-formed, $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$. So

$$(\Gamma; \Gamma')(x) = \tau$$

By T-VAR, then,

$$\Gamma; \Gamma' \vdash_m x : \{\nu : t \mid \nu = x\}$$

Case T-INT, T-REF

Immediate.

Case T-ARITH

By the form of the rule,

$$\begin{aligned} &a \equiv a_1 \circ a_2 \\ &\tau = \{\nu : \mathtt{int}(n, i_1 \mathbin{\tilde{\circ}} i_2) \mid \nu = a_1 \circ a_2\} \\ &\Gamma \vdash_m a_1 : \mathtt{int}(n, i_1) \qquad (32.2) \\ &\Gamma \vdash_m a_2 : \mathtt{int}(n, i_2) \qquad (32.3) \end{aligned}$$

By Fact 32.2, Fact 32.3, and the inductive hypothesis,

$$\Gamma; \Gamma' \vdash_m a_1 : \texttt{int}(n, i_1) \tag{32.4}$$

$$\Gamma; \Gamma' \vdash_m a_2 : \texttt{int}(n, i_2) \tag{32.5}$$

By Fact 32.4, Fact 32.5, and T-ARITH,

$$\Gamma; \Gamma' \vdash_m a_1 \circ a_2 : \{\nu : \texttt{int}(n, i_1 \mathbin{\widetilde{\circ}} i_2) \mid \nu = a_1 \circ a_2\}$$

Case T-PTRARITH, T-RELATION

Similar to the case for T-ARITH.

Case T-PURESUB

By the form of the rule,

$$\Gamma \vdash_m a : \tau_1 \tag{32.6}$$

$$\Gamma \vdash \tau_1 <: \tau \tag{32.7}$$

$$\Gamma \vDash \tau \tag{32.8}$$

By Fact 32.6 and the inductive hypothesis,

$$\Gamma; \Gamma' \vdash_m a : \tau_1$$

By Fact 32.7 and Lemma 31,

$$\Gamma; \Gamma' \vdash \tau_1 <: \tau$$

By Fact 32.8, Fact 32.1, and Lemma 30,

$$\Gamma; \Gamma' \vDash \tau$$

By the above and T-PURESUB,

$$\Gamma; \Gamma' \vdash_m a : \tau$$

□

C.10 Subtyping

Lemma 33 (Narrowing To Base Subtyping). *If* $\Gamma \vdash \{\nu : t_1 \mid \phi_1\} <: \{\nu : t_2 \mid \phi_2\}$ *then* $\Gamma \vdash \{\nu : t_1 \mid \phi_1\} <: \{\nu : t_2 \mid \phi_1\}$

Proof. Straightforward induction on the derivation of $\Gamma \vdash \{\nu : t_1 \mid \phi_1\} <: \{\nu : t_2 \mid \phi_2\}$. □

Lemma 34 (Subtyping Inversion)**.** *If* $\Gamma \vdash \tau_1 <: \tau_2$ *then one of the following holds:*

1.

$$\tau_1 = \{\nu : \texttt{int}(w, i_1) \mid \phi_1\}$$
$$\tau_2 = \{\nu : \texttt{int}(w, i_2) \mid \phi_2\}$$
$$i_1 \mathrel{\tilde{\subseteq}} i_2$$
$$\Gamma \vDash \phi_1 \Rightarrow \phi_2$$

2.

$$\tau_1 = \{\nu : \texttt{ref}(\ell_1, i_1) \mid \phi_1\}$$
$$\tau_2 = \{\nu : \texttt{ref}(\ell_2, i_2) \mid \phi_2\}$$
$$\ell_1 \sqsubseteq \ell_2$$
$$i_1 \mathrel{\tilde{\subseteq}} i_2$$
$$\Gamma \vDash \phi_1 \Rightarrow \phi_2$$

3.

$$\tau_1 = \{\nu : \texttt{int}(W, 0) \mid \phi_1\}$$
$$\tau_2 = \{\nu : \texttt{ref}(\tilde{\ell}, i_2) \mid \phi_2\}$$
$$\Gamma \vDash \phi_1 \Rightarrow \nu = 0$$
$$\Gamma \vDash \nu = 0 \Rightarrow \phi_2$$

Proof. The proof proceeds by induction on the derivation of $\Gamma \vdash \tau_1 <: \tau_2$. We split cases on the final rule used.

Case <:-**INT**

By the form of the rule, case (1) is immediately satisfied.

Case <:-**REF**

By the form of the rule, case (2) is immediately satisfied.

Case <:-**ABSTRACT**

By the form of the rule, case (2) is immediately satisfied.

Case <:-**NULL**

By the form of the rule, case (3) is immediately satisfied.

Case <:-**TRANS**

By the form of the rule,

$$\Gamma \vdash \tau_1 <: \tau_3 \tag{34.1}$$

$$\Gamma \vdash \tau_3 <: \tau_2 \tag{34.2}$$

By the inductive hypothesis and Fact 34.2, there are three cases to consider, corresponding to the three cases of the lemma.

Case (1)

Then

$$\tau_3 = \{\nu : \texttt{int}(w, i_3) \mid \phi_3\} \tag{34.3}$$

$$\tau_2 = \{\nu : \texttt{int}(w, i_2) \mid \phi_2\} \tag{34.4}$$

$$i_3 \mathrel{\widetilde{\subseteq}} i_2 \tag{34.5}$$

$$\Gamma \vDash \phi_3 \Rightarrow \phi_2 \tag{34.6}$$

By Fact 34.3, Fact 34.1, and the inductive hypothesis,

$$\tau_1 = \{\nu : \texttt{int}(w, i_1) \mid \phi_1\} \tag{34.7}$$

$$i_1 \mathrel{\widetilde{\subseteq}} i_3 \tag{34.8}$$

$$\Gamma \vDash \phi_1 \Rightarrow \phi_3 \tag{34.9}$$

By Fact 34.5, Fact 34.8, and Proposition 1,

$$i_1 \mathrel{\widetilde{\subseteq}} i_2 \tag{34.10}$$

By Fact 34.6, Fact 34.9, and the transitivity of implication,

$$\Gamma \vDash \phi_1 \Rightarrow \phi_2 \tag{34.11}$$

By Fact 34.4, Fact 34.7, Fact 34.10, and Fact 34.11, case (1) is satisfied.

Case (2)

Then

$$\tau_3 = \{\nu : \texttt{ref}(\ell_3, i_3) \mid \phi_3\} \tag{34.12}$$

$$\tau_2 = \{\nu : \texttt{ref}(\ell_2, i_2) \mid \phi_2\} \tag{34.13}$$

$$\ell_3 \sqsubseteq \ell_2 \tag{34.14}$$

$$i_3 \mathrel{\widetilde{\subseteq}} i_2 \tag{34.15}$$

$$\Gamma \vDash \phi_3 \Rightarrow \phi_2 \tag{34.16}$$

By Fact 34.12 and the inductive hypothesis, there are two more cases to consider.

First, consider the case where

$$\tau_1 = \{\nu : \texttt{ref}(\ell_1, i_1) \mid \phi_1\} \tag{34.17}$$

$$\ell_1 \sqsubseteq \ell_3 \tag{34.18}$$

$$i_1 \mathrel{\widetilde{\subseteq}} i_3 \tag{34.19}$$

$$\Gamma \vDash \phi_1 \Rightarrow \phi_3 \tag{34.20}$$

By Fact 34.14, Fact 34.18, and Definition 21,

$$\ell_1 \sqsubseteq \ell_2 \tag{34.21}$$

By Fact 34.15, Fact 34.19, and Proposition 1,

$$i_1 \mathrel{\widetilde{\subseteq}} i_2 \tag{34.22}$$

By Fact 34.16, Fact 34.20, and the transitivity of implication,

$$\Gamma \vDash \phi_1 \Rightarrow \phi_2 \tag{34.23}$$

By Fact 34.13, Fact 34.17, Fact 34.21, Fact 34.22, and Fact 34.23, case (2) of the lemma is satisfied.

Next, consider the case where

$$\tau_1 = \{\nu : \texttt{int}(W, 0) \mid \phi_1\} \tag{34.24}$$

$$\ell_3 = \widetilde{\ell} \tag{34.25}$$

$$\Gamma \vDash \phi_1 \Rightarrow \nu = 0 \tag{34.26}$$

$$\Gamma \vDash \nu = 0 \Rightarrow \phi_3 \tag{34.27}$$

By Fact 34.25 and Definition 21,

$$\ell_2 = \widetilde{\ell} \tag{34.28}$$

By Fact 34.16, Fact 34.27, and the transitivity of implication,

$$\Gamma \vDash \nu = 0 \Rightarrow \phi_2 \tag{34.29}$$

By Fact 34.24, Fact 34.26, Fact 34.28, and Fact 34.29, case (3) of the lemma is satisfied.

Case (3)

Then

$$\tau_3 = \{\nu : \texttt{int}(W, 0) \mid \phi_3\} \tag{34.30}$$

$$\tau_2 = \{\nu : \texttt{ref}(\widetilde{\ell}, i_2) \mid \phi_2\} \tag{34.31}$$

$$\Gamma \vDash \phi_3 \Rightarrow \nu = 0 \tag{34.32}$$

$$\Gamma \vDash \nu = 0 \Rightarrow \phi_2 \tag{34.33}$$

By Fact 34.1, Fact 34.30, and the inductive hypothesis,

$$\tau_1 = \{\nu : \texttt{int}(W, 0) \mid \phi_1\} \tag{34.34}$$

$$\Gamma \vDash \phi_1 \Rightarrow \phi_3 \tag{34.35}$$

By Fact 34.32, Fact 34.35, and the transitivity of implication,

$$\Gamma \vDash \phi_1 \Rightarrow \nu = 0 \tag{34.36}$$

By Fact 34.31, Fact 34.34, Fact 34.33, and Fact 34.36, case (3) of the lemma is satisfied. □

Lemma 35 (Subtyping is Transitive)**.** *For any environment* Γ*,*

1. *If* $\Gamma \vdash \tau_1 <: \tau_2$ *and* $\Gamma \vdash \tau_2 <: \tau_3$*, then* $\Gamma \vdash \tau_1 <: \tau_3$*.*
2. *If* $\Gamma \vdash b_1 <: b_2$ *and* $\Gamma \vdash b_2 <: b_3$*, then* $\Gamma \vdash b_1 <: b_3$*.*
3. *If* $\Gamma \vdash h_1 <: h_2$ *and* $\Gamma \vdash h_2 <: h_3$*, then* $\Gamma \vdash h_1 <: h_3$*.*

Proof. We consider each case separately.

1. Follows immediately from the assumptions and an application of <:-TRANS.
2. By induction on the derivation of $\Gamma \vdash b_1 <: b_2$, using (2).
3. By induction on the derivation of $\Gamma \vdash h_1 <: h_2$, using (3).

□

Lemma 36 (Subtyping is Reflexive)**.** *For any environment* Γ*,*

1. $\Gamma \vdash \tau <: \tau$
2. $\Gamma \vdash b <: b$
3. $\Gamma \vdash h <: h$

Proof. We consider each case separately.

1. Straightforward by cases on the form of τ, using Proposition 1.
2. Straightforward induction on the derivation of $\Gamma \vdash b <: b$, using (1).
3. Straightforward induction on the derivation of $\Gamma \vdash h <: h$, using (2).

□

Lemma 37 (Related Heaps Share Bound Locations)**.** *If $\Gamma \vdash h_1 <: h_2$, then h_1 and h_2 have the same set of bound locations.*

Proof. The proof proceeds by straightforward induction on the derivation of $\Gamma \vdash h_1 <: h_2$. □

Lemma 38 (Partial Heap Subtyping)**.**

$$\text{If } \Gamma \vdash h_1 <: h_2$$
$$\text{then } \Gamma \vdash h * h_1 <: h * h_2$$

Proof. The proof proceeds by straightforward induction on the derivation of $\Gamma \vdash h_1 <: h_2$, using Lemma 36. □

C.11 Well-Formedness

Lemma 39 (Pure Values Have Well-Formed Types)**.**

$$\text{If } \Gamma \vdash_m v : \tau$$
$$\text{then } \Gamma \vDash \tau.$$

Proof. Straightforward induction on the derivation of $\Gamma \vdash_m v : \tau$. □

Lemma 40 (Base Subtyping Preserves Well-Formedness)**.** *If $\Gamma \vDash \{\nu : t_1 \mid \phi\}$ and $\Gamma \vdash \{\nu : t_1 \mid \phi\} <: \{\nu : t_2 \mid \phi\}$ then $\Gamma \vDash \{\nu : t_2 \mid \phi\}$.*

Proof. Straightforward induction on the derivation of $\Gamma \vdash \{\nu : t_1 \mid \phi\} <: \{\nu : t_2 \mid \phi\}$. □

Lemma 41 (Refinement Irrelevance in Well-Formedness)**.** *For any Γ, t, ϕ, ϕ',*

1. *If $\Gamma; \{\nu : t \mid \phi\} \vDash \tau$, then $\Gamma; \{\nu : t \mid \phi'\} \vDash \tau$.*
2. *If $\Gamma; \{\nu : t \mid \phi\} \vDash b$, then $\Gamma; \{\nu : t \mid \phi'\} \vDash b$.*
3. *If $\Gamma; \{\nu : t \mid \phi\} \vDash_@ b$, then $\Gamma; \{\nu : t \mid \phi'\} \vDash_@ b$.*

Proof. Straightforward induction on the derivation of the assumed well-formedness fact, using Definition 14 and Definition 15. □

Lemma 42 (Fresh Self Type Well-Formedness)**.**

$$\begin{aligned} &\textit{If } \Gamma \vDash \{\nu : t \mid \phi\} \\ &\textit{and } x \notin \Gamma, \\ &\textit{then } \Gamma; x : \{\nu : t \mid \phi\} \vDash \{\nu : t \mid \nu = x\}. \end{aligned}$$

Proof. Immediate by cases on the rule used to derive $\Gamma \vDash \{\nu : t \mid \phi\}$ and Definition 15. □

Lemma 43 (Well-Formedness Preserved By Same-Sorted Substitutions)**.** *If $x, y \in \mathrm{dom}(\Gamma)$, and x and y have the same sort in* $\mathrm{SortEnv}(\Gamma)$*, then:*

1. *If $\Gamma \vDash \phi$, then $\Gamma \vDash \phi[x \mapsto y]$.*
2. *If $\Gamma \vDash b$, then $\Gamma \vDash b[x \mapsto y]$.*
3. *If $\Gamma \vDash_{@} b$, then $\Gamma \vDash_{@} b[x \mapsto y]$.*

Proof. We consider each case separately.

1. Follows immediately by Assumption 8.
2. Follows immediately by (1).
3. Straightforward induction on the derivation of $\Gamma \vDash_{@} b$, using (1) and (2).

□

Lemma 44 (Well-Formedness Strenghtening)**.** *For any Γ,*

1. *If $\Gamma; x : \tau_x \vDash \tau$ and $x \notin \mathrm{FV}(\tau)$, then $\Gamma \vDash \tau$.*
2. *If $\Gamma; x : \tau_x \vDash b$ and $x \notin \mathrm{FV}(b)$, then $\Gamma \vDash b$.*
3. *If $\Gamma; x : \tau_x \vDash_{@} b$ and $x \notin \mathrm{FV}(b)$, then $\Gamma \vDash_{@} b$.*

Proof. The proof of (1) is by cases on the rule used to derive $\Gamma; x : \tau \vDash \tau$, using Assumption 12.

The proof of (2) is immediate from (1).

The proof of (3) is a straightforward induction on the derivation of $\Gamma; x : \tau \vDash_{@} b$, using (1) and (2). □

Lemma 45 (Eliminating Free Locations From Abstract Blocks)**.**

$$\begin{aligned} &\textit{If } \Gamma \vDash_{@} n : \{\nu : t \mid \phi\}, \overline{i_j} : \overline{\tau_j}, \\ &\quad x \notin \Gamma, \mathrm{FV}(\overline{\tau_j}), \\ &\textit{and } \theta = [@n \mapsto x], \\ &\textit{then } \Gamma; x : \{\nu : t \mid \phi\} \vDash_{@} n : \{\nu : t \mid \nu = x\}, \overline{i_j} : \overline{\theta\tau_j} \\ &\textit{and } @n \notin \mathrm{FL}(\overline{\theta\tau_j}). \end{aligned}$$

Proof. We split cases on the final rule used to show $\Gamma \vDash_@ n : \{\nu : t \mid \phi\}, \overline{i_j} : \overline{\tau_j}$.

Case WF-DBLOCK-SEQUENCE

Impossible.

Case WF-DBLOCK-SINGLE

By the form of the rule,

$$\text{DisjointOffsets}(n : \{\nu : t \mid \phi\}, \overline{i_j} : \overline{\tau_j}) \tag{45.1}$$

$$y \notin \Gamma,\ \text{FV}(\overline{\tau_j}) \tag{45.2}$$

$$\Gamma \vDash \{\nu : t \mid \phi\} \tag{45.3}$$

$$\Gamma, y : \{\nu : t \mid \phi\} \vDash_@ \overline{i_j} : \overline{\tau_j[@n \mapsto y]} \tag{45.4}$$

By Fact 45.3 and Lemma 42,

$$\Gamma; x : \{\nu : t \mid \phi\} \vDash \{\nu : t \mid \nu = x\} \tag{45.5}$$

Assume, without loss of generality, that $x \neq y$, so that

$$\Gamma; x : \{\nu : t \mid \phi\}; y : \{\nu : t \mid \phi\} \tag{45.6}$$

is well-formed. By Fact 45.4, Fact 45.6, and Lemma 30,

$$\Gamma; x : \{\nu : t \mid \phi\}; y : \{\nu : t \mid \phi\} \vDash_@ \overline{i_j} : \overline{\tau_j[@n \mapsto y]}$$

By Lemma 41, we have

$$\Gamma; x : \{\nu : t \mid \phi\}; y : \{\nu : t \mid \nu = x\} \vDash_@ \overline{i_j} : \overline{\tau_j[@n \mapsto y]}$$

By Lemma 43,

$$\Gamma; x : \{\nu : t \mid \phi\}; y : \{\nu : t \mid \nu = x\} \vDash_@ \overline{i_j} : \overline{\tau_j[@n \mapsto y][y \mapsto x]}$$

Equivalently,

$$\Gamma; x : \{\nu : t \mid \phi\}; y : \{\nu : t \mid \nu = x\} \vDash_@ \overline{i_j} : \overline{\theta\tau_j}$$

Note that $@n \notin \text{FL}(\overline{\theta\tau_j})$, as required, and so

$$\Gamma; x : \{\nu : t \mid \phi\}; y : \{\nu : t \mid \nu = x\} \vDash_@ \overline{i_j} : \overline{\theta\tau_j[@n \mapsto y]} \tag{45.7}$$

By Fact 45.1, Fact 45.2, $x \neq y$, Fact 45.5, Fact 45.7, and WF-DBLOCK-SINGLE,

$$\Gamma; x : \{\nu : t \mid \phi\} \vDash_@ n : \{\nu : t \mid \nu = x\}, \overline{i_j} : \overline{\theta\tau_j}$$

as required. □

Lemma 46 (Dependent Blocks Without Free Locations)**.** *If* $\Gamma \vDash_@ b$ *and* $\mathrm{FL}(b) = \emptyset$*, then* $\Gamma \vDash b$*.*

Proof. The proof proceeds by induction on the derivation of $\Gamma \vDash_@ b$. We split cases on the final rule used.

Case WF-DBLOCK-SEQUENCE

The desired conclusion follows immediately.

Case WF-DBLOCK-SINGLE

By the form of the rule,

$$b = n : \tau,\ \overline{i_j : \tau_j} \tag{46.1}$$
$$\mathrm{DisjointOffsets}(n : \tau,\ \overline{i_j : \tau_j}) \tag{46.2}$$
$$x \notin \Gamma,\ \mathrm{FV}(\overline{\tau_j}) \tag{46.3}$$
$$\Gamma \vDash \tau \tag{46.4}$$
$$\Gamma, x : \tau \vDash_@ \overline{i_j} : \overline{\tau_j[@n \mapsto x]} \tag{46.5}$$

Since $\mathrm{FL}(b) = \emptyset$, Fact 46.4 gives

$$\Gamma, x : \tau \vDash_@ \overline{i_j} : \overline{\tau_j}$$

By Fact 46.2 and Lemma 44,

$$\Gamma \vDash_@ \overline{i_j} : \overline{\tau_j} \tag{46.6}$$

By the inductive hypothesis,

$$\Gamma \vDash \overline{i_j} : \overline{\tau_j}$$

The only rule by which this can be derived is WF-NDBLOCK, by which we have

$$\forall j. \Gamma \vDash \tau_j \tag{46.7}$$

By Fact 46.1, Fact 46.3, Fact 46.6, and WF-NDBLOCK,

$$\Gamma \vDash n : \tau,\ \overline{i_j} : \overline{\tau_j}$$

as required. □

Lemma 47 (One Binding per Location in Well-Formed Heaps)**.** *If* $\Gamma \vDash h$*, then any location is bound at most once in* h*.*

Proof. The proof proceeds by straightforward induction on the derivation of $\Gamma \vDash h$. □

Lemma 48 (Partial Heap Subtyping Preserves Well-Formedness)**.**

$$\begin{gathered}\textit{If } \Gamma \vDash h_1 * h_2, \\ \Gamma \vDash h_3, \\ \textit{and } \Gamma \vdash h_2 <: h_3, \\ \textit{then } \Gamma \vDash h_1 * h_3\end{gathered}$$

Proof. The proof proceeds by straightforward induction on the derivation of $\Gamma \vDash h_3$. □

Lemma 49 (Unfolded Block Well-Formedness)**.**

$$\begin{aligned}&\textit{If } \Gamma \vDash_{@} \overline{n_k} : \overline{\{\nu : t_k \mid \phi_k\}}, \overline{i_m^+} : \overline{\tau_m}, \\ &\quad \theta = [\overline{@n_k} \mapsto \overline{x_k}], \\ &\quad \overline{x_k} \textit{ disjoint}, \\ &\textit{and } \overline{x_k} \notin \Gamma, \mathrm{FV}(\overline{n_k} : \overline{\{\nu : t_k \mid \phi_k\}}, \overline{i_m^+} : \overline{\tau_m}), \\ &\textit{then } \Gamma; \overline{x_k} : \overline{\{\nu : t_k \mid \theta\phi_k\}} \vDash \overline{n_k} : \overline{\{\nu : t_k \mid \nu = x_k\}}, \overline{i_m^+} : \overline{\theta\tau_m}.\end{aligned}$$

Proof. By repeated application of Lemma 45, ending with an application of Lemma 46. □

C.12 Values

Lemma 50 (Value Self-Typing)**.**

$$\begin{gathered}\textit{If } \Gamma \vdash_m v : \{\nu : t \mid \phi\} \\ \textit{then } \Gamma \vdash_m v : \{\nu : t \mid \nu = v\}.\end{gathered}$$

Proof. By induction on the derivation of $\Gamma \vdash_m v : \{\nu : t \mid \phi\}$. We split cases on the final rule used.

Case T-VAR, T-INT, T-REF

Immediate.

Case T-ARITH, T-PTRARITH, T-RELATION

Impossible, since these rules do not apply to values.

Case T-PURESUB

By the form of the rule,

$$\Gamma \vdash_m v : \{\nu : t_1 \mid \phi_1\} \tag{50.1}$$

$$\Gamma \vdash \{\nu : t_1 \mid \phi_1\} <: \{\nu : t \mid \phi\} \tag{50.2}$$

By the inductive hypothesis and Fact 50.1,

$$\Gamma \vdash_m v : \{\nu : t_1 \mid \nu = v\} \tag{50.3}$$

By Fact 50.3 and Lemma 39,

$$\Gamma \vDash \{\nu : t_1 \mid \nu = v\} \tag{50.4}$$

By Fact 50.2 and Lemma 33,

$$\Gamma \vdash \{\nu : t_1 \mid \nu = v\} <: \{\nu : t \mid \nu = v\} \tag{50.5}$$

By Fact 50.4, Fact 50.5, and Lemma 40,

$$\Gamma \vDash \{\nu : t \mid \nu = v\} \tag{50.6}$$

By Fact 50.3, Fact 50.5, Fact 50.6, and T-PureSub,

$$\Gamma \vdash_m v : \{\nu : t \mid \nu = v\}$$

as required. □

C.13 Substitutions

Definition 29 (Substitution Combination)**.** *Let* $\theta_1 = [\overline{x_j} \mapsto \overline{v_j}]$*. Define the* combination $\theta_2[\theta_1]$ *as*

$$\theta_2[\theta_1] = [\overline{x_j} \mapsto \overline{\theta_2 v_j}].$$

Lemma 51 (Substitution Composition, Combination, and Free Variables)**.** *Let* θ_1*,* θ_2 *be substitutions.*

1. *If* $\mathrm{FV}(\phi) \subseteq \mathrm{dom}(\theta_1)$*, then* $\theta_2(\theta_1\phi) = \theta_2[\theta_1]\phi$*.*
2. *If* $\mathrm{FV}(\tau) \subseteq \mathrm{dom}(\theta_1)$*, then* $\theta_2(\theta_1\tau) = \theta_2[\theta_1]\tau$*.*
3. *If* $\mathrm{FV}(b) \subseteq \mathrm{dom}(\theta_1)$*, then* $\theta_2(\theta_1 b) = \theta_2[\theta_1]b$*.*
4. *If* $\mathrm{FV}(h) \subseteq \mathrm{dom}(\theta_1)$*, then* $\theta_2(\theta_1 h) = \theta_2[\theta_1]h$*.*

Proof. We prove only the first case, for refinement predicates, from which the other cases follow easily.

Let x be a variable in $\mathrm{FV}(\phi)$. Since $\mathrm{FV}(\phi) \subseteq \mathrm{dom}(\theta_1)$, $\theta_1(x) = v$ for some v. So $\theta_2(\theta_1(x)) = \theta_2(v)$. But this is the same as $\theta_2[\theta_1](x)$. □

Definition 30 (Well-Formed Substitutions)**.** *A substitution θ from variables to values is well-formed in environment Γ with respect to location map m, written $\Gamma \vDash_m \theta$, according to the following rules:*

$$\frac{}{\emptyset \vDash_m \emptyset} \text{ WFSUBST-EMPTY}$$

$$\frac{\emptyset \vdash_m v : \tau \qquad \Gamma[x \mapsto v] \vDash_m \theta}{x : \tau; \Gamma \vDash_m [x \mapsto v]\theta} \text{ WFSUBST-VAR}$$

$$\frac{\emptyset \vDash \texttt{true} \Rightarrow \phi \qquad \Gamma \vDash_m \theta}{\phi; \Gamma \vDash_m \theta} \text{ WFSUBST-PRED}$$

Lemma 52 (Substitution Domains)**.** *If $\Gamma \vDash_m \theta$, then $\mathrm{dom}(\theta) = \mathrm{dom}(\Gamma)$.*

Proof. The proof proceeds by straightforward induction on the derivation of $\Gamma \vDash_m \theta$. □

Lemma 53 (Value Substitution: Well-Formedness)**.** *If $\Gamma \vDash_m \theta$, then*

1. *If ϕ well-sorted in $\Gamma; \Gamma'$, then $\theta\phi$ well-sorted in $\theta\Gamma'$.*
2. *If $\Gamma; \Gamma' \vDash \tau$, then $\theta\Gamma' \vDash \theta\tau$.*
3. *If $\Gamma; \Gamma' \vDash b$, then $\theta\Gamma' \vDash \theta b$.*
4. *If $\Gamma; \Gamma' \vDash_@ b$, then $\theta\Gamma' \vDash_@ \theta b$.*
5. *If $\Gamma; \Gamma' \vDash h$, then $\theta\Gamma' \vDash \theta h$.*
6. *If $\Gamma; \Gamma' \vDash \tau/h$, then $\theta\Gamma' \vDash \theta\tau/\theta h$.*

Proof. We consider each case separately.

1. We proceed by induction on the derivation of $\Gamma \vDash_m \theta$, splitting cases on the final rule used.

   Case WFSUBST-EMPTY
   Then $\theta = \cdot$ and $\Gamma = \emptyset$, so by assumption we have ϕ well-sorted in $\emptyset; \Gamma'$. Since θ is empty, this is equivalent to $\theta\phi$ well-sorted in $\theta\Gamma'$.

   Case WFSUBST-PRED
   By the form of WFSUBST-PRED, we have

   $$\begin{aligned} \Gamma &= a; \Gamma'', \\ \Gamma'' &\vDash_m \theta. \end{aligned} \tag{53.1}$$

   Note that, by Definition 15, $\mathrm{SortEnv}(a; \Gamma'') = \mathrm{SortEnv}(\Gamma'')$, so

   $$\phi \text{ well-sorted in } \Gamma; \Gamma' \Rightarrow \phi \text{ well-sorted in } \Gamma''; \Gamma'.$$

By the inductive hypothesis and Fact 53.1, then, we have

$$\theta\phi \text{ well-sorted in } \theta\Gamma'.$$

Case WFSUBST-VAR

By the form of WFSUBST-VAR, we have

$$\Gamma = x : \tau; \Gamma'',$$
$$\theta = [x \mapsto v]\theta'$$
$$\emptyset \vdash_m v : \tau \tag{53.2}$$
$$\Gamma''[x \mapsto v] \models_m \theta'. \tag{53.3}$$

By assumption, we have

$$\phi \text{ well-sorted in } x : \tau; \Gamma''; \Gamma'. \tag{53.4}$$

By Assumption 10 and Fact 53.2, v has sort $\text{Sort}(\tau)$. By Assumption 8 and Fact 53.4, and using the fact that $\text{SortEnv}(\Gamma) = \text{SortEnv}(\theta\Gamma)$ for all Γ, θ,

$$\phi[x \mapsto v] \text{ well-sorted in } x : \tau; \Gamma''[x \mapsto v]; \Gamma'[x \mapsto v].$$

Since $x \notin \text{FV}(\phi[x \mapsto v])$, by Assumption 12 and the above,

$$\phi[x \mapsto v] \text{ well-sorted in } \Gamma''[x \mapsto v]; \Gamma'[x \mapsto v].$$

By the above, the inductive hypothesis, and Fact 53.3,

$$\theta'(\phi[x \mapsto v]) \text{ well-sorted in } \theta'(\Gamma'[x \mapsto v]).$$

That is,

$$\theta\phi \text{ well-sorted in } \theta\Gamma'.$$

as required.

2. The proof is straightforward by cases on the rule used to derive $\Gamma; \Gamma' \vDash \phi$ and (1).

3. The proof follows immediately from the premises of WF-NDBLOCK, which is the only rule that may be used to show $\Gamma; \Gamma' \vDash b$, and (2).

4. The proof proceeds by straightforward induction on the derivation of $\Gamma; \Gamma' \vDash_@ b$, splitting cases on the final rule used and using (2).

5. The proof proceeds by straightforward induction on the derivation of $\Gamma; \Gamma' \vDash h$, splitting cases on the final rule used and using (3) and (4).

6. Follows immediately from (2) and (5).

□

Lemma 54 (Value Substitution: Implication)**.** *If* $\Gamma_1 \vDash_m \theta$ *and* $\Gamma_1; \Gamma_2 \vDash \phi_1 \Rightarrow \phi_2$*, then* $\theta\Gamma_2 \vDash \theta\phi_1 \Rightarrow \theta\phi_2$*.*

Proof. The proof proceeds by induction on the derivation of $\Gamma_1 \vDash_m \theta$. We split cases on the final rule used.

Case WFSUBST-EMPTY

Immediate.

Case WFSUBST-VAR

By the form of WFSUBST-VAR, we have

$$\Gamma_1 = x : \{\nu : t \mid \phi_x\}; \Gamma'$$
$$\theta = [x \mapsto v]\theta'$$
$$\emptyset \vdash_m v : \{\nu : t \mid \phi_x\} \tag{54.1}$$
$$\Gamma'[x \mapsto v] \vDash_m \theta' \tag{54.2}$$

By assumption and Definition 13, we have

$$\phi_x[\nu \mapsto x] \wedge [\![\Gamma']\!] \wedge [\![\Gamma_2]\!] \wedge \phi_1 \Rightarrow \phi_2.$$

This is equivalent to

$$\phi_x[\nu \mapsto x] \Rightarrow ([\![\Gamma']\!] \wedge [\![\Gamma_2]\!] \wedge \phi_1 \Rightarrow \phi_2).$$

The above is universally closed, so we may replace x by v to obtain

$$\phi_x[\nu \mapsto v] \Rightarrow ([\![\Gamma']\!] \wedge [\![\Gamma_2]\!] \wedge \phi_1 \Rightarrow \phi_2)[x \mapsto v].$$

By Lemma 27 and Fact 54.1,

$$\texttt{true} \Rightarrow \phi_x[\nu \mapsto v].$$

So

$$([\![\Gamma']\!] \wedge [\![\Gamma_2]\!] \wedge \phi_1 \Rightarrow \phi_2)[x \mapsto v].$$

By pushing substitutions inward, this is

$$[\![\Gamma'[x \mapsto v]]\!] \wedge [\![\Gamma_2[x \mapsto v]]\!] \wedge \phi_1[x \mapsto v] \Rightarrow \phi_2[x \mapsto v].$$

Which, by Definition 13, is equivalent to

$$\Gamma'[x \mapsto v]; \Gamma_2[x \mapsto v] \vDash \phi_1[x \mapsto v] \Rightarrow \phi_2[x \mapsto v].$$

By the inductive hypothesis and Fact 54.2,

$$\theta'\Gamma_2[x \mapsto v] \vDash \theta'\phi_1[x \mapsto v] \Rightarrow \theta'\phi_2[x \mapsto v].$$

Equivalently,

$$\theta\Gamma_2 \vDash \theta\phi_1 \Rightarrow \theta\phi_2.$$

Case WFSUBST-PRED

By the form of WFSUBST-PRED, we have

$$\Gamma_1 = \phi; \Gamma'$$

$$\emptyset \vDash \texttt{true} \Rightarrow \phi \tag{54.3}$$

$$\Gamma' \vDash_m \theta \tag{54.4}$$

By assumption and Definition 13,

$$\phi \wedge [\![\Gamma']\!] \wedge [\![\Gamma_2]\!] \wedge \phi_1 \Rightarrow \phi_2.$$

This is equivalent to

$$\phi \Rightarrow ([\![\Gamma']\!] \wedge [\![\Gamma_2]\!] \wedge \phi_1 \Rightarrow \phi_2).$$

Which, by Fact 54.3, becomes

$$[\![\Gamma']\!] \wedge [\![\Gamma_2]\!] \wedge \phi_1 \Rightarrow \phi_2.$$

Which, by Definition 13, is equivalent to

$$\Gamma'; \Gamma_2 \vDash \phi_1 \Rightarrow \phi_2.$$

By the inductive hypothesis and Fact 54.4

$$\theta\Gamma_2 \vDash \theta\phi_1 \Rightarrow \theta\phi_2.$$

□

Lemma 55 (Value Substitution: Subtyping)**.** *If* $\Gamma \vDash_m \theta$*, then*

1. *If* $\Gamma; \Gamma' \vdash \tau_1 <: \tau_2$*, then* $\theta\Gamma' \vdash \theta\tau_1 <: \theta\tau_2$.
2. *If* $\Gamma; \Gamma' \vdash b_1 <: b_2$*, then* $\theta\Gamma' \vdash \theta b_1 <: \theta b_2$.
3. *If* $\Gamma; \Gamma' \vdash h_1 <: h_2$*, then* $\theta\Gamma' \vdash \theta h_1 <: \theta h_2$.

4. *If* $\Gamma; \Gamma' \vdash \tau_1 / h_1 <: \tau_2 / h_2$, *then* $\theta\Gamma' \vdash \theta\tau_1 / \theta h_1 <: \theta\tau_2 / \theta h_2$.

Proof. We consider each case separately.

1. The proof proceeds by induction on the derivation of $\Gamma; \Gamma' \vdash \tau_1 <: \tau_2$. We split cases on the final rule used.

   Case <:**-INT**
   By the form of the rule, we have

   $$\tau_1 = \{\nu : \texttt{int}(n, i_1) \mid \phi_1\}$$
   $$\tau_2 = \{\nu : \texttt{int}(n, i_2) \mid \phi_2\}$$
   $$i_1 \mathrel{\tilde{\subseteq}} i_2 \tag{55.1}$$
   $$\Gamma; \Gamma' \vDash \phi_1 \Rightarrow \phi_2 \tag{55.2}$$

   By Fact 55.2 and Lemma 54,

   $$\theta\Gamma' \vDash \theta\phi_1 \Rightarrow \theta\phi_2 \tag{55.3}$$

   By Fact 55.1, Fact 55.3, and <:-INT,

   $$\theta\Gamma' \vdash \theta\tau_1 <: \theta\tau_2$$

   Case <:**-REF**
   Similar to <:-INT.

   Case <:**-ABSTRACT,** <:**-NULL**
   Immediate.

   Case <:**-TRANS**
   Follows immediately from the inductive hypothesis and another application of <:-TRANS.

2. The proof proceeds by induction on the derivation of $\Gamma; \Gamma' \vdash b_1 <: b_2$. We split cases on the final rule used.

   Case <:**-SEQUENCE**
   By the form of the rule, we have

$$b_1 = i^+ : \tau_1,\ b_1'$$
$$b_2 = i^+ : \tau_2,\ b_2'$$
$$\Gamma; \Gamma' \vdash \tau_1 <: \tau_2 \tag{55.4}$$
$$\Gamma; \Gamma' \vdash b_1' <: b_2' \tag{55.5}$$

By Fact 55.4 and (1),

$$\theta\Gamma' \vdash \theta\tau_1 <: \theta\tau_2 \tag{55.6}$$

By Fact 55.5 and the inductive hypothesis,

$$\theta\Gamma' \vdash \theta b_1' <: \theta b_2' \tag{55.7}$$

By Fact 55.6, Fact 55.7, and <:-SEQUENCE,

$$\theta\Gamma' \vdash i^+ : \theta\tau_1,\ \theta b_1' <: i^+ : \theta\tau_2,\ \theta b_2'$$

as required.

Case <:-SINGLE

By the form of the rule,

$$b_1 = i^+ : \tau_1,\ b_1'$$
$$b_2 = i^+ : \tau_2,\ b_2'$$
$$\Gamma; \Gamma' \vdash \tau_1 <: \tau_2 \tag{55.8}$$
$$x \notin \Gamma; \Gamma' \tag{55.9}$$
$$\Gamma; \Gamma'; x : \tau_1 \vdash b_1[@n \mapsto x] <: b_2[@n \mapsto x] \tag{55.10}$$

By Fact 55.8 and (1),

$$\theta\Gamma' \vdash \theta\tau_1 <: \theta\tau_2 \tag{55.11}$$

By Fact 55.10 and the inductive hypothesis,

$$\theta\Gamma'; x : \theta\tau_1 \vdash \theta(b_1[@n \mapsto x]) <: \theta(b_2[@n \mapsto x])$$

Since $x \notin \Gamma$, $x \notin \mathrm{dom}(\theta)$ by Lemma 52, so

$$\theta\Gamma'; x : \theta\tau_1 \vdash \theta b_1[@n \mapsto x] <: \theta b_2[@n \mapsto x] \tag{55.12}$$

By Fact 55.11, Fact 55.12, Fact 55.9, and <:-SINGLE,

$$\theta\Gamma' \vdash i^+ : \theta\tau_1,\ \theta b_1' <: i^+ : \theta\tau_2,\ \theta b_2'$$

as required.

3. The proof proceeds by induction on the derivation of $\Gamma; \Gamma' \vdash h_1 <: h_2$. We split cases on the final rule used.

   Case $<:$**-EMPTY-HEAP**
   Immediate.

   Case $<:$**-HEAP**
   By the form of the rule,

$$h_1 = h_1' * \ell \mapsto b_1$$
$$h_2 = h_2' * \ell \mapsto b_2$$
$$\Gamma; \Gamma' \vdash b_1 <: b_2 \tag{55.13}$$
$$\Gamma; \Gamma' \vdash h_1' <: h_2' \tag{55.14}$$

   By Fact 55.13 and (2),

$$\theta\Gamma' \vdash \theta b_1 <: \theta b_2 \tag{55.15}$$

   By Fact 55.14 and the inductive hypothesis,

$$\theta\Gamma' \vdash \theta h_1' <: \theta h_2' \tag{55.16}$$

   By Fact 55.15, Fact 55.16, and $<:$-HEAP,

$$\theta\Gamma' \vdash \theta h_1 <: \theta h_2$$

4. The only rule that applies is $<:$-WORLD. By the form of the rule:

$$\Gamma; \Gamma' \vdash \tau_1 <: \tau_2 \tag{55.17}$$
$$\Gamma; \Gamma' \vdash h_1 <: h_2 \tag{55.18}$$

   By Fact 55.17 and (1),

$$\theta\Gamma' \vdash \theta\tau_1 <: \theta\tau_2 \tag{55.19}$$

   By Fact 55.18 and (3),

$$\theta\Gamma' \vdash \theta h_1 <: \theta h_2 \tag{55.20}$$

   By Fact 55.19, Fact 55.20, and $<:$-WORLD,

$$\theta\Gamma' \vdash \theta\tau_1 / \theta h_1 <: \theta\tau_2 / \theta h_2$$

□

Lemma 56 (Variable Substitution)**.**

$$\begin{aligned} &\textit{If } \Gamma \vDash_m \theta \\ &\textit{and } \Gamma(x) = \{\nu : t \mid \phi\}, \\ &\textit{then } \varnothing \vdash_m \theta x : \{\nu : t \mid \nu = \theta x\}. \end{aligned}$$

Proof. The proof proceeds by induction on the derivation of $\Gamma \vDash_m \theta$. We split cases on the final rule used.

Case WFSUBST-EMPTY

Impossible.

Case WFSUBST-VAR

By the form of the rule, we have

$$\Gamma = y : \tau; \Gamma'$$
$$\theta = [y \mapsto v]\theta'$$
$$\varnothing \vdash_m v : \tau \tag{56.1}$$
$$\Gamma'[y \mapsto v] \vDash_m \theta' \tag{56.2}$$

We split cases on whether $x = y$.

Case $x = y$

Then $\theta x = \theta y = v$. By Fact 56.1 and Lemma 50,

$$\varnothing \vdash_m \theta x : \{\nu : t \mid \nu = \theta x\}$$

as required.

Case $x \neq y$

Then $x \in \mathrm{dom}(\Gamma')$, so that

$$\Gamma'(x) = \{\nu : t \mid \phi\}$$

from which we have

$$(\Gamma'[y \mapsto v])(x) = \{\nu : t \mid \phi[y \mapsto v]\}$$

By the above, Fact 56.2, and the inductive hypothesis,

$$\varnothing \vdash_m \theta x : \{\nu : t \mid \nu = \theta x\}$$

as required.

Case WFSUBST-PRED

By the form of the rule,

$$\Gamma = \phi; \Gamma'$$
$$\Gamma' \vDash_m \theta \tag{56.3}$$

Then $x \in \mathrm{dom}(\Gamma')$, so that

$$\Gamma'(x) = \{\nu : t \mid \phi\} \tag{56.4}$$
$$\tag{56.5}$$

By Fact 56.3, Fact 56.5, and the inductive hypothesis,

$$\emptyset \vdash_m \theta x : \{\nu : t \mid \nu = \theta x\}$$

as required. □

Lemma 57 (Pure Typing Value Substitution)**.** *If* $\Gamma \vDash_m \theta$ *and* $\Gamma; \Gamma' \vdash_m a : \tau$*, then* $\theta\Gamma' \vdash_m \theta a : \theta\tau$*.*

Proof. The proof proceeds by induction on the derivation of $\Gamma; \Gamma' \vdash_m a : \tau$. We split cases on the final rule used.

Case T-VAR

Then

$$a \equiv x$$
$$\tau = \{\nu : t \mid \nu = x\}$$

By the form of the rule,

$$(\Gamma; \Gamma')(x) = \{\nu : t \mid \phi\}. \tag{57.1}$$

We split cases on whether $x \in \mathrm{dom}(\Gamma)$.

Case $x \in \mathrm{dom}(\Gamma)$

By Lemma 56,

$$\emptyset \vdash_m \theta x : \{\nu : t \mid \nu = \theta x\}$$

By Lemma 32,

$$\theta\Gamma' \vdash_m \theta x : \{\nu : t \mid \nu = \theta x\}$$

as required.

Case $x \notin \mathrm{dom}(\Gamma)$

Then $x \in \mathrm{dom}(\Gamma')$. By Fact 57.1,

$$\Gamma'(x) = \{\nu : t \mid \phi\}$$

from which we have

$$(\theta\Gamma')(x) = \{\nu : t \mid \theta\phi\}$$

By T-VAR,

$$\theta\Gamma' \vdash_m x : \{\nu : t \mid \nu = x\}$$

Since $x \notin \mathrm{dom}(\Gamma)$, $x \notin \mathrm{dom}(\theta)$ by Lemma 52, so this is equivalent to

$$\theta\Gamma' \vdash_m \theta x : \{\nu : t \mid \nu = \theta x\}$$

as required.

Case T-INT, T-REF

Immediate.

Case T-ARITH

By the form of the rule,

$$a \equiv v_1 \circ v_2$$

$$\tau = \{\nu : \mathtt{int}(n, i_1 \mathbin{\tilde{\circ}} i_2) \mid \nu = v_1 \circ v_2\}$$

$$\Gamma; \Gamma' \vdash_m v_1 : \mathtt{int}(n, i_1) \tag{57.2}$$

$$\Gamma; \Gamma' \vdash_m v_2 : \mathtt{int}(n, i_2) \tag{57.3}$$

By the inductive hypothesis and Fact 57.2,

$$\theta\Gamma' \vdash_m \theta v_1 : \mathtt{int}(n, i_1) \tag{57.4}$$

By the inductive hypothesis and Fact 57.3,

$$\theta\Gamma' \vdash_m \theta v_2 : \mathtt{int}(n, i_2) \tag{57.5}$$

By Fact 57.4, Fact 57.5, and T-ARITH,

$$\theta\Gamma' \vdash_m \theta v_1 \circ \theta v_2 : \{\nu : \mathtt{int}(n, i_1 \mathbin{\tilde{\circ}} i_2) \mid \nu = \theta v_1 \circ \theta v_2\}$$

as required.

Case T-PTRARITH, T-RELATION

Similar to T-ARITH.

Case T-PURESUB

By the form of the rule, we have

$$\Gamma; \Gamma' \vdash_m a : \tau_1 \tag{57.6}$$

$$\Gamma; \Gamma' \vdash \tau_1 <: \tau \tag{57.7}$$

$$\Gamma; \Gamma' \vDash \tau \tag{57.8}$$

By the inductive hypothesis and Fact 57.6,

$$\theta\Gamma' \vdash_m \theta a : \theta\tau_1 \tag{57.9}$$

By Lemma 55 and Fact 57.7,

$$\theta\Gamma' \vdash \theta\tau_1 <: \theta\tau \tag{57.10}$$

By Lemma 53 and Fact 57.8,

$$\theta\Gamma' \vDash \theta\tau \tag{57.11}$$

By Fact 57.9, Fact 57.10, Fact 57.11, and T-PURESUB,

$$\theta\Gamma' \vdash_m \theta a : \theta\tau$$

as required. □

Lemma 58 (Value Substitution)**.**

$$\begin{aligned} &\Gamma \vDash_m \theta, \\ &G,\ \Gamma; \Gamma',\ h \vdash_{m,\ I} e : \tau / h', \\ &\textit{and}\ \vDash G, \\ &\textit{then}\ G,\ \theta\Gamma',\ \theta h \vdash_{m,\ I} \theta e : \theta\tau / \theta h'. \end{aligned}$$

Proof. By induction on the derivation of $G,\ \Gamma; \Gamma',\ h \vdash_{m,\ I} e : \tau / h'$. We split cases on the final rule used.

Case T-PURE

By the form of the rule,

$$h = h'$$
$$\Gamma; \Gamma' \vdash_m e : \tau$$

By Lemma 57,

$$\theta\Gamma' \vdash_m \theta e : \theta\tau$$

By T-PURE,

$$G,\ \theta\Gamma',\ \theta h \vdash_{m,\ I} \theta e : \theta\tau/\theta h$$

as required.

Case T-SUB

By the form of the rule,

$$G,\ \Gamma; \Gamma',\ h \vdash_{m,\ I} e : \tau_1/h_1 \tag{58.1}$$
$$\Gamma; \Gamma' \vdash \tau_1/h_1 <: \tau/h' \tag{58.2}$$
$$\Gamma; \Gamma' \vDash \tau/h' \tag{58.3}$$

By the inductive hypothesis and Fact 58.1,

$$G,\ \theta\Gamma',\ \theta h \vdash_{m,\ I} \theta e : \theta\tau_1/\theta h_1 \tag{58.4}$$

By Lemma 55 and Fact 58.2,

$$\theta\Gamma' \vdash \theta\tau_1/\theta h_1 <: \theta\tau/\theta h' \tag{58.5}$$

By Lemma 53 and Fact 58.3,

$$\theta\Gamma' \vDash \theta\tau/\theta h' \tag{58.6}$$

By Fact 58.4, Fact 58.5, Fact 58.6, and T-SUB,

$$G,\ \theta\Gamma',\ \theta h \vdash_{m,\ I} \theta e : \theta\tau/\theta h'$$

as required.

Case T-IF

By the form of the rule,

$$e \equiv \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2$$

$$\Gamma; \Gamma' \vdash_m v : \texttt{int}(W, i) \tag{58.7}$$

$$G,\ \Gamma; \Gamma'; v \neq 0,\ h \vdash_{m,\ I} e_1 : \tau / h' \tag{58.8}$$

$$G,\ \Gamma; \Gamma'; v = 0,\ h \vdash_{m,\ I} e_2 : \tau / h' \tag{58.9}$$

By Lemma 57 and Fact 58.7,

$$\theta\Gamma' \vdash_m \theta v : \texttt{int}(W, i) \tag{58.10}$$

By the inductive hypothesis and Fact 58.8,

$$G,\ \theta\Gamma'; \theta v \neq 0,\ \theta h \vdash_{m,\ I} \theta e_1 : \theta\tau / \theta h' \tag{58.11}$$

By the inductive hypothesis and Fact 58.9,

$$G,\ \theta\Gamma'; \theta v = 0,\ \theta h \vdash_{m,\ I} \theta e_2 : \theta\tau / \theta h' \tag{58.12}$$

By Fact 58.10, Fact 58.11, Fact 58.12, and T-IF,

$$G,\ \theta\Gamma',\ \theta h \vdash_{m,\ I} \textbf{if } \theta v \textbf{ then } \theta e_1 \textbf{ else } \theta e_2 : \theta\tau / \theta h'$$

as required.

Case T-LET

By the form of the rule,

$$e \equiv \textbf{let } x = e_1 \textbf{ in } e_2$$

$$I = I_1 \cup I_2$$

$$I_1 \cap I_2 = \emptyset$$

$$G,\ \Gamma; \Gamma',\ h \vdash_{m,\ I_1} e_1 : \tau_1 / h_1 \tag{58.13}$$

$$G,\ \Gamma; \Gamma'; x : \tau_1,\ h_1 \vdash_{m,\ I_2} e_2 : \tau / h' \tag{58.14}$$

$$\Gamma; \Gamma' \vDash \tau / h' \tag{58.15}$$

By the inductive hypothesis and Fact 58.13,

$$G,\ \theta\Gamma',\ \theta h \vdash_{m,\ I_1} \theta e_1 : \theta\tau_1 / \theta h_1 \tag{58.16}$$

By the inductive hypothesis and Fact 58.14,

$$G,\ \theta\Gamma'; x : \theta\tau_1,\ \theta h_1 \vdash_{m,\ I_2} \theta e_2 : \theta\tau / \theta h' \tag{58.17}$$

By Lemma 53 and Fact 58.15,

$$\theta\Gamma' \vDash \theta\tau/\theta h' \tag{58.18}$$

By Fact 58.16, Fact 58.17, Fact 58.18, and T-LET,

$$G,\ \theta\Gamma',\ \theta h \vdash_{m,\ I} \textbf{let}\ x = \theta e_1\ \textbf{in}\ \theta e_2 : \theta\tau/\theta h'$$

as required.

Case T-READ

By the form of the rule,

$$e \equiv *_n v$$
$$h' = h$$
$$\Gamma; \Gamma' \vdash_m v : \{\nu : \texttt{ref}(\ell_j, i) \mid \text{Safe}(\nu, n)\} \tag{58.19}$$
$$h = h_1 * \ell_j \mapsto \ldots, i : \tau, \ldots \tag{58.20}$$
$$\text{SizeOf}(\tau) = n \tag{58.21}$$

By Lemma 57 and Fact 58.19,

$$\theta\Gamma' \vdash_m \theta v : \{\nu : \texttt{ref}(\ell_j, i) \mid \text{Safe}(\theta\nu, n)\} \tag{58.22}$$

By Fact 58.20,

$$\theta h = \theta h_1 * \ell_j \mapsto \ldots, i : \theta\tau, \ldots \tag{58.23}$$

By Fact 58.21 and Definition 7,

$$\text{SizeOf}(\theta\tau) = n \tag{58.24}$$

By Fact 58.22, Fact 58.23, Fact 58.24, and T-READ,

$$G,\ \theta\Gamma',\ \theta h \vdash_{m,\ I} *_n \theta v : \theta\tau/\theta h$$

as required.

Case T-SUPD

By the form of the rule,

$$e \equiv *v_1 := v_2$$

$$h = h_1 * \ell_j \mapsto \ldots, n : \tau_1, \ldots \tag{58.25}$$

$$\Gamma; \Gamma' \vdash_m v_1 : \{\nu : \texttt{ref}(\ell_j, n) \mid \text{Safe}(\nu, \text{SizeOf}(\tau_2))\} \tag{58.26}$$

$$\Gamma; \Gamma' \vdash_m v_2 : \tau_2 \tag{58.27}$$

$$\text{SizeOf}(\tau_2) = \text{SizeOf}(\tau_1) \tag{58.28}$$

$$h' = h_1 * \ell_j \mapsto \ldots, n : \tau_2, \ldots \tag{58.29}$$

$$\tau = \texttt{void}$$

By Fact 58.25,

$$\theta h = \theta h_1 * \ell_j \mapsto \ldots, n : \theta\tau_1, \ldots \tag{58.30}$$

By Lemma 57 and Fact 58.26,

$$\theta\Gamma' \vdash_m \theta v_1 : \{\nu : \texttt{ref}(\ell_j, n) \mid \text{Safe}(\theta\nu, \text{SizeOf}(\tau_2))\}$$

By Definition 7,

$$\text{SizeOf}(\tau_2) = \text{SizeOf}(\theta\tau_2)$$

so

$$\theta\Gamma' \vdash_m \theta v_1 : \{\nu : \texttt{ref}(\ell_j, n) \mid \text{Safe}(\theta\nu, \text{SizeOf}(\theta\tau_2))\} \tag{58.31}$$

By Lemma 57 and Fact 58.27,

$$\theta\Gamma' \vdash_m \theta v_2 : \theta\tau_2 \tag{58.32}$$

By Fact 58.29,

$$\theta h' = \theta h_1 * \ell_j \mapsto \ldots, n : \theta\tau_2, \ldots \tag{58.33}$$

By Fact 58.28 and Definition 7,

$$\text{SizeOf}(\theta\tau_2) = \text{SizeOf}(\theta\tau_1) \tag{58.34}$$

By Fact 58.30, Fact 58.31, Fact 58.32, Fact 58.33, Fact 58.34, and T-SUPD,

$$G,\ \theta\Gamma',\ \theta h \vdash_{m,\ I} *\theta v_1 := \theta v_2 : \texttt{void}/\theta h'$$

as required.

Case T-WUPD

By the form of the rule,

$$e \equiv *v_1 := v_2$$
$$h' = h$$
$$\Gamma; \Gamma' \vdash_m v_1 : \{\nu : \texttt{ref}(\ell_j, n^{+m}) \mid \text{Safe}(\nu, \text{SizeOf}(\tau))\} \tag{58.35}$$
$$\Gamma; \Gamma' \vdash_m v_2 : \tau \tag{58.36}$$
$$h = h_1 * \ell_j \mapsto \ldots, n^{+m} : \tau, \ldots \tag{58.37}$$

By Lemma 57 and Fact 58.35,

$$\theta\Gamma' \vdash_m \theta v_1 : \{\nu : \texttt{ref}(\ell_j, n^{+m}) \mid \text{Safe}(\theta\nu, \text{SizeOf}(\tau))\}$$

By Definition 7,

$$\text{SizeOf}(\tau) = \text{SizeOf}(\theta\tau)$$

so

$$\theta\Gamma' \vdash_m \theta v_1 : \{\nu : \texttt{ref}(\ell_j, n^{+m}) \mid \text{Safe}(\theta\nu, \text{SizeOf}(\theta\tau))\} \tag{58.38}$$

By Lemma 57 and Fact 58.36,

$$\theta\Gamma' \vdash_m \theta v_2 : \theta\tau \tag{58.39}$$

By Fact 58.37,

$$\theta h = \theta h_1 * \ell_j \mapsto \ldots, n^{+m} : \theta\tau, \ldots \tag{58.40}$$

By Fact 58.38, Fact 58.39, Fact 58.40, and T-WUPD,

$$G,\ \theta\Gamma',\ \theta h \vdash_{m,\ I} *\theta v_1 := \theta v_2 : \texttt{void}/\theta h$$

as required.

Case T-UNFOLD

By the form of the rule,

$$e \equiv \textbf{letu } x = \textbf{unfold } v \textbf{ in } e'$$

$$\Gamma_1; \Gamma_2 \vdash_m v : \{\nu : \texttt{ref}(\widetilde{\ell}, i_y) \mid \nu \neq 0\} \tag{58.41}$$

$$h = h_0 * \widetilde{\ell} \mapsto \overline{n_k} : \overline{\tau_k}, \overline{i^+} : \overline{\tau^+} \tag{58.42}$$

$$\overline{x_k} \text{ disjoint} \tag{58.43}$$

$$\overline{x_k} \notin \Gamma_1; \Gamma_2,\ e',\ \mathrm{FV}(h) \tag{58.44}$$

$$\theta' = [\overline{@n_k} \mapsto \overline{x_k}] \tag{58.45}$$

$$\Gamma' = \Gamma_1; \Gamma_2; \overline{x_k} : \overline{\theta' \tau_k} \tag{58.46}$$

$$\ell_j \notin \Gamma_1; \Gamma_2, h, m \tag{58.47}$$

$$h_1 = h * \ell_j \mapsto \overline{n_k} : \overline{\{\nu = x_k\}}, \overline{i^+} : \overline{\theta' \tau^+} \tag{58.48}$$

$$G,\ \Gamma'; x : \{\nu : \texttt{ref}(\ell_j, i_y) \mid \nu = v\},\ h_1 \vdash_{m,\ I} e' : \tau' / h' \tag{58.49}$$

$$\Gamma' \vDash h_1 \tag{58.50}$$

$$\Gamma_1; \Gamma_2 \vDash \tau' / h' \tag{58.51}$$

By Fact 58.41 and Lemma 57,

$$\theta\Gamma_2 \vdash_m \theta v : \{\nu : \texttt{ref}(\widetilde{\ell}, i_y) \mid \nu \neq 0\} \tag{58.52}$$

By Fact 58.42,

$$\theta h = \theta h_0 * \widetilde{\ell} \mapsto \overline{n_k} : \overline{\theta \tau_k}, \overline{i^+} : \overline{\theta \tau^+} \tag{58.53}$$

By Fact 58.44, and noting that applying value substitution θ can only remove free variables,

$$\overline{x_k} \notin \theta\Gamma_2,\ \theta e',\ \mathrm{FV}(\theta h) \tag{58.54}$$

Let

$$\Gamma'' = \theta\Gamma_2; \overline{x_k} : \overline{\theta\theta' \tau_k} \tag{58.55}$$

By Fact 58.47,

$$\ell_j \notin \theta\Gamma_2, \theta h, m \tag{58.56}$$

By Fact 58.48,

$$\theta h_1 = \theta h * \ell_j \mapsto \overline{n_k} : \overline{\{\nu = \theta x_k\}}, \overline{i^+} : \overline{\theta\theta' \tau^+}$$

By Fact 58.44 and Lemma 52, this is equivalent to

$$\theta h_1 = \theta h * \ell_j \mapsto \overline{n_k} : \overline{\{\nu = x_k\}}, \overline{i^+} : \overline{\theta\theta' \tau^+} \tag{58.57}$$

By Fact 58.49, Fact 58.46, Fact 58.55, and the inductive hypothesis,

$$G,\ \Gamma'';x : \{\nu :\ \texttt{ref}(\ell_j, i_y)\ \mid\ \nu = \theta v\},\ \theta h_1 \vdash_{m,\ I} \theta e' : \theta\tau'/\theta h' \tag{58.58}$$

By Fact 58.50, Fact 58.46, Fact 58.55, and Lemma 53,

$$\Gamma'' \vDash \theta h_1 \tag{58.59}$$

By Fact 58.51 and Lemma 53,

$$\theta\Gamma_2 \vDash \theta\tau'/\theta h' \tag{58.60}$$

By Fact 58.52, Fact 58.53, Fact 58.54, Fact 58.55, Fact 58.56, Fact 58.57, Fact 58.58, Fact 58.59, Fact 58.60, and T-UNFOLD,

$$G,\ \theta\Gamma_2,\ \theta h_1 \vdash_{m,\ I} \textbf{letu}\ x = \textbf{unfold}\ \theta v\ \textbf{in}\ \theta e' : \theta\tau'/\theta h'$$

as required.

Case T-CALL

By the form of the rule,

$$e \equiv f(\overline{v_j})$$
$$G(f) = (\overline{x_j} : \overline{\tau_j})/h_f\ \rightarrow\ \tau'/h'_f \tag{58.61}$$
$$h = h_m * h_u$$
$$\Gamma;\Gamma' \vDash h_m \tag{58.62}$$
$$\Gamma;\Gamma' \vDash h_u \tag{58.63}$$
$$\theta' = [\overline{x_j} \mapsto \overline{v_j}][\overline{\ell_f} \mapsto \overline{\ell}] \tag{58.64}$$
$$\Gamma;\Gamma' \vDash h_u * \theta' h'_f \tag{58.65}$$
$$\Gamma;\Gamma' \vdash_m \overline{v_j} : \overline{\theta'\tau_j} \tag{58.66}$$
$$\Gamma;\Gamma' \vdash h_m <: \theta' h_f \tag{58.67}$$
$$h' = h_u * \theta' h'_f$$
$$\tau = \theta'\tau'$$

By Lemma 53 and Fact 58.62,

$$\theta\Gamma' \vDash \theta h_m \tag{58.68}$$

By Lemma 53 and Fact 58.63,

$$\theta\Gamma' \vDash \theta h_u \tag{58.69}$$

Using Fact 58.64, define

$$\begin{aligned}\theta_2 &= [\overline{x_j} \mapsto \overline{\theta v_j}][\overline{\ell_f} \mapsto \overline{\ell}] \\ &= \theta[\theta'] \end{aligned} \tag{58.70}$$

By the assumption that $\vDash G$ and Definition 25,

$$\vDash (\overline{x_j} : \overline{\tau_j})/h_f \ \rightarrow \ \tau'/h'_f \tag{58.71}$$

By the form of WF-FunScheme and Fact 58.71,

$$\begin{aligned}&\Gamma_f = \overline{x_j} : \overline{\tau_j} \\ &\Gamma_f \vDash \overline{\tau_j} \\ &\Gamma_f \vDash h_f \\ &\Gamma_f \vDash \tau'/h'_f\end{aligned} \tag{58.72}$$

By the preceding, Fact 58.72, and Lemma 28,

$$\mathrm{FV}(\overline{\tau_j}) \subseteq \{\overline{x_j}\} \tag{58.73}$$

$$\mathrm{FV}(h_f) \subseteq \{\overline{x_j}\} \tag{58.74}$$

$$\mathrm{FV}(\tau') \subseteq \{\overline{x_j}\} \tag{58.75}$$

$$\mathrm{FV}(h'_f) \subseteq \{\overline{x_j}\} \tag{58.76}$$

By Lemma 53 and Fact 58.65,

$$\theta\Gamma' \vDash \theta h_u * \theta(\theta' h'_f) \tag{58.77}$$

By Fact 58.74, Fact 58.77, and Lemma 51,

$$\theta\Gamma' \vDash \theta h_u * \theta_2 h'_f \tag{58.78}$$

By Lemma 57 and Fact 58.66,

$$\theta\Gamma' \vdash_m \overline{\theta v_j} : \overline{\theta(\theta' \tau_j)} \tag{58.79}$$

By Fact 58.73, Fact 58.79, and Lemma 51,

$$\theta\Gamma' \vdash_m \overline{\theta v_j} : \overline{\theta_2 \tau_j} \tag{58.80}$$

By Lemma 55 and Fact 58.67,

$$\theta\Gamma' \vdash \theta h_m <: \theta(\theta' h_f) \tag{58.81}$$

By Fact 58.74, Fact 58.81, and Lemma 51,

$$\theta\Gamma' \vdash \theta h_m <: \theta_2 h_f \tag{58.82}$$

By Fact 58.68, Fact 58.69, Fact 58.61, Fact 58.70, Fact 58.78, Fact 58.80, Fact 58.82, and T-CALL,

$$G,\ \theta\Gamma',\ \theta h \vdash_{m,\ I} f(\overline{\theta v_j}) : \theta_2\tau' / \theta h_u * \theta_2 h'_f$$

By Fact 58.70, Fact 58.75, Fact 58.76, and Lemma 51, this is equivalent to

$$G,\ \theta\Gamma',\ \theta h \vdash_{m,\ I} f(\overline{\theta v_j}) : \theta(\theta'\tau') / \theta h_u * \theta(\theta' h'_f)$$

as required.

Case T-MALLOC

By the form of the rule

$$e \equiv \mathbf{malloc}(v)$$

$$\ell_j \notin \Gamma; \Gamma', h, m \tag{58.83}$$

$$h = h_0 * \widetilde{\ell} \mapsto b \tag{58.84}$$

$$h' = h * \ell_j \mapsto b^0 \tag{58.85}$$

$$\Gamma; \Gamma' \vDash h * \ell_j \mapsto b \tag{58.86}$$

$$\Gamma; \Gamma' \vdash_m v : \{\nu : \texttt{int}(W, i) \mid \nu \geq 0\} \tag{58.87}$$

$$\tau = \{\nu : \texttt{ref}(\ell_j, 0) \mid \text{Allocated}(\nu, v)\} \tag{58.88}$$

By Fact 58.83,

$$\ell_j \notin \theta\Gamma', \theta h, m \tag{58.89}$$

By Fact 58.84,

$$\theta h = \theta h_0 * \widetilde{\ell} \mapsto \theta b \tag{58.90}$$

By Fact 58.85,

$$\theta h' = \theta h * \ell_j \mapsto (\theta b)^0$$

As b^0 contains no free variables, we have

$$\theta h' = \theta h * \ell_j \mapsto \theta b^0 \tag{58.91}$$

By Fact 58.86 and Lemma 53,

$$\theta\Gamma' \vDash \theta h * \ell_j \mapsto \theta b \tag{58.92}$$

By Fact 58.87 and Lemma 57,

$$\theta\Gamma' \vdash_m \theta v : \{\nu : \texttt{int}(W, i) \mid \theta\nu \geq 0\} \tag{58.93}$$

By Fact 58.88,

$$\theta\tau = \{\nu : \texttt{ref}(\ell_j, 0) \mid \text{Allocated}(\nu, \theta v)\} \tag{58.94}$$

By Fact 58.89, Fact 58.90, Fact 58.91, Fact 58.92, Fact 58.93, Fact 58.94, and T-MALLOC,

$$G,\ \theta\Gamma',\ \theta h \vdash_{m,\ I} \mathbf{malloc}(v) : \theta\tau / \theta h'$$

as required.

□

C.14 Location Name Sets

Definition 31 (Location Name Sets)**.** *The set of location names bound in a heap h,* $\text{Locs}(h)$*, is defined as*

$$\{\ell \mid \widetilde{\ell} \in \text{dom}(h)\} \cup \{\ell \mid \ell_j \in \text{dom}(h)\}.$$

Lemma 59 (Same Location Names in Related Heaps)**.** *If* $\Gamma \vdash h_1 <: h_2$*, then* $\text{Locs}(h_1) = \text{Locs}(h_2)$*.*

Proof. Straightforward induction on the derivation of $\Gamma \vdash h_1 <: h_2$, using Definition 31. □

Lemma 60 (Same Location Names In Input and Output Heaps)**.**

$$\begin{aligned} &\textit{If } G,\ \Gamma,\ h \vdash_{m,\ I} e : \tau / h' \\ &\textit{and } \vDash G \\ &\textit{then } \text{Locs}(h) = \text{Locs}(h'). \end{aligned} \tag{60.1}$$

Proof. The proof proceeds by induction on the derivation of $G,\ \Gamma,\ h \vdash_{m,\ I} e : \tau / h'$, splitting cases on the final rule used. The only interesting case is T-CALL.

Case T-CALL

By the form of the rule,

$$h = h_u * h_m$$

$$\Gamma \vdash h_m <: \theta h_f \tag{60.2}$$

$$G(f) = (\overline{x_j} : \overline{\tau_j})/h_f \ \rightarrow \ \tau'/h'_f \tag{60.3}$$

$$h' = h_u * \theta h'_f \tag{60.4}$$

By Fact 60.2 and Lemma 59,

$$\mathrm{Locs}(h_m) = \mathrm{Locs}(\theta h_f) \tag{60.5}$$

By Fact 60.3, Fact 60.1, and Definition 25,

$$\mathrm{Locs}(h_f) = \mathrm{Locs}(h'_f) \tag{60.6}$$

By Fact 60.5 and Fact 60.6,

$$\mathrm{Locs}(\theta h'_f) = \mathrm{Locs}(h_m)$$

from which it follows by Definition 31 that

$$\mathrm{Locs}(h_u * h_m) = \mathrm{Locs}(h_u * \theta h'_f)$$

as required. □

C.15 Location Name Substitution

Definition 32. *A* location name substitution *is a function ρ : LocName $\rightarrow$ LocName, where LocName is the set of location names (i.e., names ℓ from which we form abstract location names $\widetilde{\ell}$ and concrete location names ℓ_j).*

We define the application of location name substitutions as follows:

$$\rho\ \widetilde{\ell} = \widetilde{\rho\ell}$$
$$\rho\ell_j = (\rho\ell)_j$$

$$\rho(\texttt{int}(w,i)) = \texttt{int}(w,i)$$
$$\rho(\texttt{ref}(\ell,i) = \texttt{ref}(\rho(\ell),i)$$

$$\rho(\{\nu : t \mid \phi\}) = \{\nu : \rho t \mid \phi\}$$

$$\rho(\overline{i} : \overline{\tau}) = \overline{i} : \overline{\rho\tau}$$

$$\rho(h * \ell \mapsto b) = \rho(h) * \rho\ell \mapsto \rho b$$

$$\rho(m[\ell \mapsto r]) = \rho m[\rho\ell \mapsto r]$$

$$\rho(\Gamma; x : \tau) = \rho\Gamma; x : \rho\tau$$
$$\rho(\Gamma; \phi) = \rho\Gamma; \phi$$

Lemma 61 (Location Name Substitution and Well-Formed Subtypes)**.**

$$\textit{If } \Gamma \vDash h_1 \tag{61.1}$$
$$\textit{and } \Gamma \vdash h_1 <: \rho h_2 \tag{61.2}$$
$$\textit{then } |\text{Locs}(\rho h_2)| = |\text{Locs}(h_2)|$$

Proof. By Fact 61.1 and Lemma 47, each location is bound at most once in h_1. By Fact 61.2 and Lemma 37, h_1 and ρh_2 bind the same locations. Thus, every location is bound at most once in ρh_2. Note that the cardinalities of $\text{Locs}(h_2)$ and $\text{Locs}(\rho h_2)$ can only differ if ρ has the effect of mapping two disjoint location names to a single location name, which would mean that ρh_2 binds a single location name twice. □

Lemma 62 (Location Name Substitution: Subtyping)**.** *For any location name substitution ρ,*

1. *If $\Gamma \vdash \tau_1 <: \tau_2$, then $\rho\Gamma \vdash \rho\tau_1 <: \rho\tau_2$.*
2. *If $\Gamma \vdash b_1 <: b_2$, then $\rho\Gamma \vdash \rho b_1 <: \rho b_2$.*
3. *If $\Gamma \vdash h_1 <: h_2$, then $\rho\Gamma \vdash \rho h_1 <: \rho h_2$.*
4. *If $\Gamma \vdash \tau_1/h_1 <: \tau_2/h_2$, then $\rho\Gamma \vdash \rho\tau_1/\rho h_1 <: \rho\tau_2/\rho h_2$.*

Proof. Each case proceeds by straightforward induction on the assumed derivation, using the previous cases, except for (4), which follows directly by (1) and (3). □

Lemma 63 (Location Name Substitution: Well-Formedness)**.** *For any location name substitution ρ,*

1. *If $\Gamma \vDash \tau$, then $\rho\Gamma \vDash \rho\tau$.*
2. *If $\Gamma \vDash b$, then $\rho\Gamma \vDash \rho b$.*
3. *If $|\text{Locs}(h)| = |\text{Locs}(\rho h)|$ and $\Gamma \vDash h$, then $\rho\Gamma \vDash \rho h$.*
4. *If $|\text{Locs}(h)| = |\text{Locs}(\rho h)|$ and $\Gamma \vDash \tau/h$, then $\rho\Gamma \vDash \rho\tau/\rho h$.*

Proof. We prove each item separately.

1. Straightforward by cases on the rule used to prove $\Gamma \vDash \tau$ and Definition 14, which is oblivious to location names.
2. Straightforward induction on the structure of b, using (1).
3. The proof proceeds by induction on the derivation of $\Gamma \vDash h$. We split cases on the final rule used.

   Case WF-HAbstract
   By the form of the rule,

   $$h = h_0 * \widetilde{\ell} \mapsto b$$
   $$\Gamma \vDash h_0 \tag{63.1}$$
   $$\widetilde{\ell} \notin \text{dom}(h_0) \tag{63.2}$$
   $$\Gamma \vDash_{@} b \tag{63.3}$$

   By Fact 63.1 and the inductive hypothesis,

   $$\rho\Gamma \vDash \rho h_0 \tag{63.4}$$

   By Fact 63.2 and $|\text{Locs}(h)| = |\text{Locs}(\rho h)|$,

   $$\rho(\widetilde{\ell}) \notin \text{dom}(\rho h_0) \tag{63.5}$$

   By Fact 63.3 and (2),

   $$\rho\Gamma \vDash_{@} \rho b \tag{63.6}$$

By Fact 63.4, Fact 63.5, Fact 63.6, and WF-HABSTRACT,

$$\rho\Gamma \vDash h_0 * \rho \widetilde{\ell} \mapsto \rho b$$

as required.

Case WF-HCONCRETE

By the form of the rule,

$$h = h_0 * \ell_j \mapsto b$$

$$\Gamma \vDash h_0 \tag{63.7}$$

$$\Gamma \vDash b \tag{63.8}$$

$$\widetilde{\ell} \in \mathrm{dom}(h_0) \tag{63.9}$$

$$\ell_k \notin \mathrm{dom}(h_0) \tag{63.10}$$

By Fact 63.7 and the inductive hypothesis,

$$\rho\Gamma \vDash \rho h_0 \tag{63.11}$$

By Fact 63.8 and (2),

$$\rho\Gamma \vDash \rho b \tag{63.12}$$

By Fact 63.9,

$$\rho(\widetilde{\ell}) \in \mathrm{dom}(\rho h_0) \tag{63.13}$$

By Fact 63.10 and $|\mathrm{Locs}(h)| = |\mathrm{Locs}(\rho h)|$,

$$\rho(\ell_k) \notin \mathrm{dom}(\rho h_0) \tag{63.14}$$

By Fact 63.11, Fact 63.12, Fact 63.13, Fact 63.14, and WF-HCONCRETE,

$$\rho\Gamma \vDash \rho h_0 * \rho(\ell_j) \mapsto \rho b$$

4. Follows immediately from (1) and (3).

□

Lemma 64 (Location Name Substitution: Pure Typing)**.**

$$\textit{If } \Gamma \vdash_m a : \tau$$
$$\textit{then } \rho\Gamma \vdash_{\rho m} a : \rho\tau.$$

Proof. The proof proceeds by induction on the derivation of $\Gamma \vdash_m a : \tau$. We split cases on the final rule used.

Case T-VAR

By the form of the rule,

$$\begin{aligned} a &\equiv x \\ \Gamma(x) &= \{\nu : t \mid \phi\} \\ \tau &= \{\nu : t \mid \nu = x\} \end{aligned}$$

By Definition 32,

$$(\rho\Gamma)(x) = \{\nu : \rho t \mid \phi\}$$

By T-VAR,

$$\rho\Gamma \vdash_{\rho m} x : \{\nu : \rho t \mid \nu = x\}$$

as required.

Case T-INT

Immediate.

Case T-REF

By the form of the rule,

$$\begin{aligned} a &\equiv \text{bref}(r, n, z) \\ \ell &\in \text{Clocs}(r, m) \\ \tau &= \{\nu : \texttt{ref}(\ell, n) \mid \nu = \text{bref}(r, n, z)\} \end{aligned} \tag{64.1}$$

By Definition 18, Definition 32, and Fact 64.1,

$$\rho\ell \in \text{Clocs}(r, \rho m) \tag{64.2}$$

By Fact 64.2 and T-REF,

$$\rho(\Gamma) \vdash_{\rho m} \text{bref}(r, n, z) : \{\nu : \texttt{ref}(\rho\ell, n) \mid \nu = \text{bref}(r, n, z)\}$$

as required.

Case T-ARITH, T-PTRARITH, T-RELATION

Immediate from the form of the rule, the inductive hypothesis, and another application

of the rule originally used.

Case T-PURESUB

By the form of the rule,

$$\Gamma \vdash_m a : \tau_1 \tag{64.3}$$

$$\Gamma \vdash \tau_1 <: \tau \tag{64.4}$$

$$\Gamma \vDash \tau \tag{64.5}$$

By Fact 64.3 and the inductive hypothesis,

$$\rho\Gamma \vdash_{\rho m} a : \rho\tau_1 \tag{64.6}$$

By Fact 64.4 and Lemma 62,

$$\rho\Gamma \vdash \rho\tau_1 <: \rho\tau \tag{64.7}$$

By Fact 64.5 and Lemma 63,

$$\rho\Gamma \vDash \rho\tau \tag{64.8}$$

By Fact 64.6, Fact 64.7, Fact 64.8, and T-PURESUB,

$$\rho\Gamma \vdash_{\rho m} a : \rho\tau$$

as required. □

Lemma 65 (Location Name Substitution Has No Effect on Pure Expressions)**.** *For any pure expression a, $\rho a = a$.*

Proof. Immediate by the fact that no locations appear in pure expressions. □

Lemma 66 (Location Name Substitution: Typing)**.**

$$\begin{aligned} &\textit{If } G,\ \Gamma,\ h \vdash_{m,\ I} e : \tau / h', \\ &\quad \vDash G, && (66.1) \\ &\textit{and } \left|\mathrm{Locs}(h)\right| = \left|\mathrm{Locs}(\rho h)\right|, && (66.2) \\ &\textit{then } G,\ \rho\Gamma,\ \rho h \vdash_{\rho m,\ \rho I} \rho e : \rho\tau / \rho h' \end{aligned}$$

Proof. The proof proceeds by induction on the derivation of $G,\ \Gamma,\ h \vdash_{m,\ I} e : \tau / h'$. We split cases on the final rule used.

Case T-PURE

Immediate by Lemma 65 and Lemma 64.

Case T-SUB

By the form of the rule,

$$G,\ \Gamma,\ h \vdash_{m,\ I} e : \tau_1 / h_1 \tag{66.3}$$

$$\Gamma \vdash \tau_1 / h_1 <: \tau / h' \tag{66.4}$$

$$\Gamma \vDash \tau / h' \tag{66.5}$$

By Fact 66.3, Fact 66.2, and the inductive hypothesis,

$$G,\ \rho\Gamma,\ \rho h \vdash_{\rho m,\ \rho I} \rho e : \rho\tau_1 / \rho h_1 \tag{66.6}$$

By Fact 66.4 and Lemma 62,

$$\rho\Gamma \vdash \rho\tau_1 / \rho h_1 <: \rho\tau / \rho h' \tag{66.7}$$

By Fact 66.4 and Lemma 59,

$$\mathrm{Locs}(h_1) = \mathrm{Locs}(h') \tag{66.8}$$

By Fact 66.2 and Fact 66.8,

$$\left|\mathrm{Locs}(h')\right| = \left|\mathrm{Locs}(\rho h')\right| \tag{66.9}$$

By Fact 66.5, Fact 66.9, and Lemma 63,

$$\rho\Gamma \vDash \rho\tau / \rho h' \tag{66.10}$$

By Fact 66.6, Fact 66.7, Fact 66.10, and T-SUB,

$$G,\ \rho\Gamma,\ \rho h \vdash_{\rho m,\ \rho I} \rho e : \rho\tau / \rho h'$$

as required.

Case T-IF

By the form of the rule,

$$e \equiv \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2$$

$$\Gamma \vdash_m v : \mathtt{int}(W, i) \tag{66.11}$$

$$G,\ \Gamma; v \neq 0,\ h \vdash_{m,\ I} e_1 : \tau / h' \tag{66.12}$$

$$G,\ \Gamma; v = 0,\ h \vdash_{m,\ I} e_2 : \tau / h' \tag{66.13}$$

By Fact 66.11, Lemma 65, and Lemma 64,

$$\rho\Gamma \vdash_{\rho m} \rho v : \texttt{int}(W, i) \tag{66.14}$$

By Fact 66.12, Fact 66.2, and the inductive hypothesis,

$$G,\ \rho\Gamma; \rho v \neq 0,\ \rho h \vdash_{\rho m,\ \rho I} \rho e_1 : \rho\tau / \rho h' \tag{66.15}$$

By Fact 66.13, Fact 66.2, and the inductive hypothesis,

$$G,\ \rho\Gamma; \rho v = 0,\ \rho h \vdash_{\rho m,\ \rho I} \rho e_2 : \rho\tau / \rho h' \tag{66.16}$$

By Fact 66.14, Fact 66.15, Fact 66.16, and T-If,

$$G,\ \rho\Gamma,\ \rho h \vdash_{\rho m,\ \rho I} \textbf{if } \rho v \textbf{ then } \rho e_1 \textbf{ else } \rho e_2 : \rho\tau / \rho h'$$

as required.

Case T-Let

By the form of the rule,

$$e \equiv \textbf{let } x = e_1 \textbf{ in } e_2$$

$$I = I_1 \cup I_2 \tag{66.17}$$

$$I_1 \cap I_2 = \emptyset \tag{66.18}$$

$$G,\ \Gamma,\ h \vdash_{m,\ I_1} e_1 : \tau_1 / h_1 \tag{66.19}$$

$$G,\ \Gamma; x : \tau_1,\ h_1 \vdash_{m,\ I_2} e_2 : \tau_2 / h_2 \tag{66.20}$$

$$\Gamma \vDash \tau_2 / h_2 \tag{66.21}$$

By Fact 66.18 and Assumption 7,

$$\rho I_1 \cap \rho I_2 = \emptyset \tag{66.22}$$

By Fact 66.19, Fact 66.2, and the inductive hypothesis,

$$G,\ \rho\Gamma,\ \rho h \vdash_{\rho m,\ \rho I_1} \rho e_1 : \rho\tau_1 / \rho h_1 \tag{66.23}$$

By Fact 66.19 and Lemma 60,

$$\mathrm{Locs}(h) = \mathrm{Locs}(h_1) \tag{66.24}$$

By Fact 66.24 and Fact 66.2,

$$|\mathrm{Locs}(h_1)| = |\mathrm{Locs}(\rho h_1)| \tag{66.25}$$

By Fact 66.20, Fact 66.25, and the inductive hypothesis,

$$G,\ \rho\Gamma; x : \rho\tau_1,\ \rho h_1 \vdash_{\rho m,\ \rho I_2} \rho e_2 : \rho\tau_2 / \rho h_2 \tag{66.26}$$

By Fact 66.19 and Lemma 60,

$$\text{Locs}(h_2) = \text{Locs}(h_1)$$

By the above and Fact 66.24,

$$\text{Locs}(h_2) = \text{Locs}(h) \tag{66.27}$$

By Fact 66.27, and Fact 66.2,

$$|\text{Locs}(h_2)| = |\text{Locs}(\rho h_2)| \tag{66.28}$$

By Fact 66.21, Fact 66.28, and Lemma 63,

$$\rho\Gamma \vDash \rho\tau_2 / \rho h_2 \tag{66.29}$$

By Fact 66.17, Fact 66.22, Fact 66.23, Fact 66.26, Fact 66.29, and T-LET,

$$G,\ \rho\Gamma,\ \rho h \vdash_{\rho m,\ \rho I} \rho e : \rho\tau_2 / \rho h_2$$

as required.

Case T-READ

By the form of the rule,

$$e \equiv *_n v$$

$$\Gamma \vdash_m v : \{\nu : \texttt{ref}(\ell_j, i) \mid \text{Safe}(\nu, n)\} \tag{66.30}$$

$$h = h_1 * \ell_j \mapsto \ldots, i : \tau, \ldots \tag{66.31}$$

$$h' = h$$

$$\text{SizeOf}(\tau) = n \tag{66.32}$$

By Fact 66.30, Lemma 65, and Lemma 64,

$$\rho\Gamma \vdash_{\rho m} \rho v : \{\nu : \texttt{ref}(\rho(\ell_j), i) \mid \text{Safe}(\nu, n)\} \tag{66.33}$$

By Fact 66.31 and Definition 32,

$$\rho h = \rho h_1 * \rho(\ell_j) \mapsto \ldots, i : \rho\tau, \ldots \tag{66.34}$$

By Fact 66.32 and Definition 7,

$$\text{SizeOf}(\rho\tau) = n \tag{66.35}$$

By Fact 66.33, Fact 66.34, Fact 66.35, and T-READ,

$$G,\ \rho\Gamma,\ \rho h \vdash_{\rho m,\ \rho I} *_n \rho v : \rho\tau / \rho h'$$

as required.

Case T-SUPD

By the form of the rule,

$$e \equiv *v_1 := v_2$$

$$h = h_1 * \ell_j \mapsto \ldots, n : \tau_1, \ldots \tag{66.36}$$

$$\Gamma \vdash_m v_1 : \{\nu : \texttt{ref}(\ell_j, n) \mid \text{Safe}(\nu, \text{SizeOf}(\tau_2))\} \tag{66.37}$$

$$\Gamma \vdash_m v_2 : \tau_2 \tag{66.38}$$

$$\text{SizeOf}(\tau_2) = \text{SizeOf}(\tau_1) \tag{66.39}$$

$$h' = h_1 * \ell_j \mapsto \ldots, n : \tau_2, \ldots \tag{66.40}$$

By Fact 66.36 and Definition 32,

$$\rho h = \rho h_1 * \rho(\ell_j) \mapsto \ldots, n : \rho\tau_1, \ldots \tag{66.41}$$

By Fact 66.37, Lemma 65, and Lemma 64,

$$\rho\Gamma \vdash_{\rho m} \rho v_1 : \{\nu : \texttt{ref}(\rho(\ell_j), n) \mid \text{Safe}(\nu, \text{SizeOf}(\tau_2))\}$$

By Definition 7,

$$\text{SizeOf}(\tau_2) = \text{SizeOf}(\rho\tau_2)$$

so

$$\rho\Gamma \vdash_{\rho m} \rho v_1 : \{\nu : \texttt{ref}(\rho(\ell_j), n) \mid \text{Safe}(\nu, \text{SizeOf}(\rho\tau_2))\} \tag{66.42}$$

By Fact 66.38, Lemma 65, and Lemma 64,

$$\rho\Gamma \vdash_{\rho m} \rho v_2 : \rho\tau_2 \tag{66.43}$$

By Fact 66.40 and Definition 32,

$$\rho h' = \rho h_1 * \rho(\ell_j) \mapsto \ldots, n : \rho\tau_2, \ldots \tag{66.44}$$

By Fact 66.39 and Definition 7,

$$\text{SizeOf}(\rho\tau_2) = \text{SizeOf}(\rho\tau_1) \tag{66.45}$$

By Fact 66.41, Fact 66.42, Fact 66.43, Fact 66.44, and T-SUPD,

$$G,\ \rho\Gamma,\ \rho h \vdash_{\rho m,\ \rho I} *\rho v_1 := \rho v_2 : \texttt{void} / \rho h'$$

as required.

Case T-WUPD

By the form of the rule,

$$\begin{aligned}
&e \equiv *v_1 := v_2 \\
&\Gamma \vdash_m v_1 : \{\nu : \texttt{ref}(\ell_j, n^{+m}) \mid \text{Safe}(\nu, \text{SizeOf}(\tau))\} && (66.46)\\
&\Gamma \vdash_m v_2 : \tau && (66.47)\\
&h = h_1 * \ell_j \mapsto \ldots, n^{+m} : \tau, \ldots && (66.48)\\
&h' = h
\end{aligned}$$

By Fact 66.46, Lemma 65, and Lemma 64,

$$\rho\Gamma \vdash_{\rho m} \rho v_1 : \{\nu : \texttt{ref}(\rho(\ell_j), n^{+m}) \mid \text{Safe}(\nu, \text{SizeOf}(\tau))\}$$

By Definition 7,

$$\text{SizeOf}(\tau) = \text{SizeOf}(\rho\tau)$$

so

$$\rho\Gamma \vdash_{\rho m} \rho v_1 : \{\nu : \texttt{ref}(\rho(\ell_j), n^{+m}) \mid \text{Safe}(\nu, \text{SizeOf}(\rho\tau))\} \tag{66.49}$$

By Fact 66.47, Lemma 65, and Lemma 64,

$$\rho\Gamma \vdash_{\rho m} \rho v_2 : \rho\tau \tag{66.50}$$

By Fact 66.48 and Definition 32,

$$\rho h = \rho h_1 * \rho(\ell_j) \mapsto \ldots, n^{+m} : \rho\tau, \ldots \tag{66.51}$$

By Fact 66.49, Fact 66.50, Fact 66.51, T-WUPD,

$$G,\ \rho\Gamma,\ \rho h \vdash_{\rho m,\ \rho I} *\rho v_1 := \rho v_2 : \texttt{void}/\rho h$$

as required.

Case T-UNFOLD

By the form of the rule,

$$e \equiv \textbf{letu}\ x = \textbf{unfold}\ v\ \textbf{in}\ e'$$
$$I = I_1 \cup \{\ell_j\} \tag{66.52}$$
$$h' = h_2$$
$$\Gamma \vdash_m v : \{\nu : \texttt{ref}(\widetilde{\ell}, i_y) \mid \nu \neq 0\} \tag{66.53}$$
$$h = h_0 * \widetilde{\ell} \mapsto \overline{n_k} : \overline{\tau_k}, \overline{i^+} : \overline{\tau^+} \tag{66.54}$$
$$\overline{x_k} \text{ disjoint} \tag{66.55}$$
$$\overline{x_k} \notin \Gamma,\ e,\ \mathrm{FV}(h) \tag{66.56}$$
$$\theta = [\overline{@n_k} \mapsto \overline{x_k}] \tag{66.57}$$
$$\Gamma_1 = \Gamma; \overline{x_k} : \overline{\theta\tau_k} \tag{66.58}$$
$$\ell_j \notin \Gamma, h, m \tag{66.59}$$
$$h_1 = h * \ell_j \mapsto \overline{n_k} : \overline{\{\nu = x_k\}}, \overline{i^+} : \overline{\theta\tau^+} \tag{66.60}$$
$$G,\ \Gamma_1; x : \{\nu : \texttt{ref}(\ell_j, i_y) \mid \nu = v\},\ h_1 \vdash_{m,\ I} e : \tau_2/h_2 \tag{66.61}$$
$$\Gamma_1 \vDash h_1 \tag{66.62}$$
$$\Gamma \vDash \tau_2/h_2 \tag{66.63}$$

By Fact 66.52 and Assumption 7,

$$\rho I = \rho I_1 \cup \{\rho(\ell_j)\} \tag{66.64}$$

By Fact 66.53, Lemma 65, and Lemma 64,

$$\rho\Gamma \vdash_{\rho m} \rho v : \{\nu : \texttt{ref}(\rho\ \widetilde{\ell}, i_y) \mid \nu \neq 0\} \tag{66.65}$$

By Fact 66.54 and Definition 32,

$$\rho h = \rho h_0 * \rho\ \widetilde{\ell} \mapsto \overline{n_k} : \overline{\rho\tau_k}, \overline{i^+} : \overline{\rho\tau^+} \tag{66.66}$$

By Fact 66.56, Fact 66.66, and Definition 32,

$$\overline{x_k} \notin \rho\Gamma,\ \rho e,\ \mathrm{FV}(\rho h) \tag{66.67}$$

By Fact 66.58 and Definition 32,

$$\rho\Gamma_1 = \rho\Gamma; \overline{x_k} : \overline{\rho\theta\tau_k} \tag{66.68}$$

By Fact 66.59 and Definition 32,

$$\rho(\ell_j) \notin \rho\Gamma, \rho h, \rho m \tag{66.69}$$

By Fact 66.60 and Definition 32,

$$\rho h_1 = \rho h * \rho(\ell_j) \mapsto \overline{n_k} : \overline{\rho(\{\nu = x_k\})}, \overline{i^+} : \overline{\rho\theta\tau^+} \tag{66.70}$$

By Fact 66.2, Fact 66.70, and Definition 31,

$$|\text{Locs}(h_1)| = |\text{Locs}(\rho h_1)| \tag{66.71}$$

By Fact 66.71, Fact 66.61, and the inductive hypothesis,

$$G,\ \rho\Gamma_1; x : \{\nu :\ \texttt{ref}(\rho(\ell_j), i_y)\ \mid\ \nu = v\},\ \rho h_1 \vdash_{\rho m,\ \rho I_1} \rho e' : \rho\tau_2/\rho h_2 \tag{66.72}$$

By Fact 66.62, Fact 66.71, and Lemma 63,

$$\rho\Gamma_1 \vDash \rho h_1 \tag{66.73}$$

By Fact 66.1, Fact 66.72, and Lemma 60,

$$\text{Locs}(h_1) = \text{Locs}(h_2)$$

so

$$|\text{Locs}(h_2)| = |\text{Locs}(\rho h_2)| \tag{66.74}$$

By Fact 66.74, Fact 66.63, and Lemma 63,

$$\rho\Gamma \vDash \rho\tau_2/\rho h_2 \tag{66.75}$$

By Fact 66.64, Fact 66.65, Fact 66.66, Fact 66.55, Fact 66.67, Fact 66.57, Fact 66.68, Fact 66.69, Fact 66.70, Fact 66.72, Fact 66.73, Fact 66.75, and T-UNFOLD,

$$G,\ \rho\Gamma,\ \rho h \vdash_{\rho m,\ \rho I} \textbf{letu}\ x = \textbf{unfold}\ \rho v\ \textbf{in}\ \rho e' : \rho\tau_2/\rho h_2$$

as required.

Case T-FOLD

By the form of the rule,

$$e \equiv \textbf{fold}\ \ell$$
$$h = h_0 * \widetilde{\ell} \mapsto b_1 * \ell_j \mapsto b_2 \tag{66.76}$$
$$h' = h_0 * \widetilde{\ell} \mapsto b_1 \tag{66.77}$$
$$\Gamma \vdash b_2 <: b_1 \tag{66.78}$$

By Fact 66.76 and Definition 32,

$$\rho h = \rho h_0 * \rho\ \widetilde{\ell} \mapsto \rho b_1 * \rho(\ell_j) \mapsto \rho b_2 \tag{66.79}$$

By Fact 66.78 and Lemma 62,

$$\rho\Gamma \vdash \rho b_2 <: \rho b_1 \tag{66.80}$$

By Fact 66.77 and Definition 32,

$$\rho h = \rho h_0 * \rho\ \widetilde{\ell} \mapsto \rho b_1 \tag{66.81}$$

By Fact 66.79, Fact 66.80, Fact 66.81, and T-FOLD,

$$G,\ \rho\Gamma,\ \rho h \vdash_{\rho m,\, \rho I} \textbf{fold}\ \rho\ell : \texttt{void}/\rho h'$$

as required.

Case T-CALL

By the form of the rule,

$$e \equiv f(\overline{v_j})[\overline{\ell_f} \mapsto \overline{\ell}]$$
$$h = h_u * h_m$$
$$h' = h_u * \theta h'_f$$
$$\tau = \tau'$$
$$\Gamma \vDash h_m \tag{66.82}$$
$$\Gamma \vDash h_u \tag{66.83}$$
$$G(f) = (\overline{x_j} : \overline{\tau_j})/h_f\ \rightarrow\ \tau'/h'_f \tag{66.84}$$
$$\theta = [\overline{x_j} \mapsto \overline{v_j}][\overline{\ell_f} \mapsto \overline{\ell}] \tag{66.85}$$
$$\Gamma \vDash h_u * \theta h'_f \tag{66.86}$$
$$\Gamma \vdash_m \overline{v_j} : \overline{\theta\tau_j} \tag{66.87}$$
$$\Gamma \vdash h_m <: \theta h_f \tag{66.88}$$

By Fact 66.82, Fact 66.2, and Lemma 63,

$$\rho\Gamma \vDash \rho h_m \tag{66.89}$$

By Fact 66.83, Fact 66.2, and Lemma 63,

$$\rho\Gamma \vDash \rho h_u \tag{66.90}$$

By Fact 66.88 and Lemma 59,

$$\text{Locs}(h_m) = \text{Locs}(\theta h_f) \tag{66.91}$$

By Fact 66.84, Definition 25, and Fact 66.1,

$$\text{Locs}(h'_f) = \text{Locs}(h_f) \tag{66.92}$$

By Fact 66.91 and Fact 66.92,

$$\text{Locs}(\theta h'_f) = \text{Locs}(h_m) \tag{66.93}$$

By Fact 66.93 and Fact 66.2,

$$\left|\text{Locs}(h_u * \theta h'_f)\right| = \left|\text{Locs}(\rho h_u * \rho\theta h'_f)\right| \tag{66.94}$$

By Fact 66.86, Fact 66.94, and Lemma 63,

$$\rho\Gamma \vDash \rho h_u * \rho\theta h'_f \tag{66.95}$$

By Fact 66.87, Lemma 65, and Lemma 64,

$$\rho\Gamma \vdash_{\rho m} \overline{\rho v_j} : \overline{\rho\theta\tau_j} \tag{66.96}$$

By Fact 66.88 and Lemma 64,

$$\rho\Gamma \vdash \rho h_m <: \rho\theta h_f \tag{66.97}$$

By Fact 66.89, Fact 66.90, Fact 66.84, Fact 66.95, Fact 66.96, Fact 66.97, and T-CALL,

$$G,\ \rho\Gamma,\ \rho h \vdash_{\rho m,\ \rho I} f(\overline{\rho v_j}) : \rho\tau' / \rho h'$$

as required.

Case T-MALLOC

By the form of the rule,

$$e \equiv \mathbf{malloc}(v)$$
$$h' = h * \ell_j \mapsto b$$
$$I = I_1 \cup \{\ell_j\} \tag{66.98}$$
$$\tau = \{\nu : \texttt{ref}(\ell_j, 0) \mid \text{Allocated}(\nu, v)\}$$
$$\ell_j \notin \Gamma, h, m \tag{66.99}$$
$$h = h_0 * \widetilde{\ell} \mapsto b \tag{66.100}$$
$$\Gamma \vDash h * \ell_j \mapsto b \tag{66.101}$$
$$\Gamma \vdash_m v : \{\nu : \texttt{int}(W, i) \mid \nu \geq 0\} \tag{66.102}$$

By Fact 66.98 and Assumption 7,

$$\rho I = \rho I_1 \cup \{\rho(\ell_j)\} \tag{66.103}$$

By Fact 66.99 and Definition 32,

$$\rho(\ell_j) \notin \rho\Gamma, \rho h, \rho m \tag{66.104}$$

By Fact 66.100 and Definition 32,

$$\rho h = \rho h_0 * \rho\, \widetilde{\ell} \mapsto \rho b \tag{66.105}$$

By Fact 66.101, Fact 66.2, and Lemma 63,

$$\rho\Gamma \vDash \rho h * \rho(\ell_j) \mapsto \rho b \tag{66.106}$$

By Fact 66.102, Lemma 65, and Lemma 64,

$$\rho\Gamma \vdash_{\rho m} \rho v : \{\nu : \texttt{int}(W, i) \mid \nu \geq 0\} \tag{66.107}$$

By Fact 66.103, Fact 66.104, Fact 66.105, Fact 66.106, Fact 66.107, and T-MALLOC,

$$G,\ \rho\Gamma,\ \rho h \vdash_{\rho m,\, \rho I} \mathbf{malloc}(\rho v) : \rho\tau / \rho h'$$

as required. □

C.16 Heap Weakening

Lemma 67 (Heap Weakening: Well-Formedness)**.**

$$\begin{aligned} &\textit{If } \Gamma \vDash h_1, \\ &\quad \Gamma \vDash h_2, \\ &\textit{and } \operatorname{dom}(h_1) \cap \operatorname{dom}(h_2) = \emptyset, \\ &\textit{then } \Gamma \vDash h_1 * h_2. \end{aligned}$$

Proof. The proof proceeds by straightforward induction on the derivation of $\Gamma \vDash h_2$. □

Lemma 68 (Heap Weakening: Subtyping)**.**

$$\begin{aligned} &\textit{If } \Gamma \vdash h_1 <: h_2 \\ &\textit{then } \Gamma \vdash h * h_1 <: h * h_2. \end{aligned}$$

Proof. The proof proceeds by straightforward induction on the derivation of $\Gamma \vdash h_1 <: h_2$, using Lemma 36. □

Lemma 69 (Heap Weakening: Typing)**.**

$$\begin{aligned} &\textit{If } G,\ \Gamma,\ h \vdash_{m,\ I} e : \tau / h', \\ &\quad \Gamma \vDash h, \\ &\quad \Gamma \vDash h_2, \\ &\quad \mathrm{dom}(h_1) \cap \mathrm{dom}(\mathit{heap}_2) = \emptyset, \\ &\textit{and } \mathrm{Clocs}(h_2) \cap I = \emptyset, \\ &\textit{then } G,\ \Gamma,\ h_2 * h \vdash_{m,\ I} e : \tau / h_2 * h'. \end{aligned}$$

Proof. The proof proceeds by straightforward induction on the derivation of $G,\ \Gamma,\ h \vdash_{m,\ I} e : \tau / h'$, using Lemma 67 and Lemma 68. □

C.17 Canonical Forms

Lemma 70 (Subtyping Preserves Sizes)**.** *If* $\Gamma \vdash \tau_1 <: \tau_2$, *then* $\mathrm{SizeOf}(\tau_1) = \mathrm{SizeOf}(\tau_2)$.

Proof. The proof proceeds by straightforward induction on the derivation of $\Gamma \vdash \tau_1 <: \tau_2$. □

Lemma 71 (Value Sizes)**.** *If* $\emptyset \vdash_m v : \tau$, *then* $\mathrm{SizeOf}(v) = \mathrm{SizeOf}(\tau)$.

Proof. The proof proceeds by straightforward induction on the derivation of $\emptyset \vdash_m v : \tau$, using Lemma 70. □

Lemma 72 (Canonical Forms)**.** *For any value v:*

1. *If* $\emptyset \vdash_m v : \mathtt{int}(w, i)$, *then* $v = n_{|w|}$ *for some* $n \in [\![i]\!]$.
2. *If* $\emptyset \vdash_m v : \mathtt{ref}(\ell, i)$, *then either* $v = 0_{|W|}$ *or* $v = \mathrm{bref}(r, n, z)$ *for some* $n \in [\![i]\!]$ *and there exists* $\ell_j \sqsubseteq \ell$ *such that* $\ell_j \in \mathrm{Clocs}(r, m)$.

Proof. The proofs proceed by induction on the given typing derivation, splitting cases on the final rule used.

1. The proof proceeds by induction on the derivation of $\emptyset \vdash_m v : \texttt{int}(w, i)$. We split cases on the final rule used.

   Case T-INT
   Immediate.

   Case T-VAR, T-REF, T-ARITH, T-PTRARITH, T-RELATION
   Impossible.

   Case T-PURESUB
   By the form of the rule,

   $$\emptyset \vdash_m v : \tau \tag{72.1}$$
   $$\emptyset \vdash \tau <: \texttt{int}(w, i) \tag{72.2}$$

   By Fact 72.2 and Lemma 34,

   $$\tau = \texttt{int}(w, i_1) \tag{72.3}$$

   By the Fact 72.1, Fact 72.3, and the inductive hypothesis,

   $$v = n_{|w|}$$
   $$n \in [\![i_1]\!]$$

   for some n. By Fact 72.2 and Lemma 34,

   $$i_1 \mathrel{\widetilde{\subseteq}} i$$

   By Proposition 1,

   $$n \mathrel{\widetilde{\subseteq}} i$$

   By Proposition 2,

   $$n \in [\![i]\!]$$

   as required.

2. The proof proceeds by induction on the derivation of $\emptyset \vdash_m v : \texttt{ref}(\ell, i)$. We split cases on the final rule used.

Case T-VAR

Impossible.

Case T-REF

Immediate, using Proposition 1, Proposition 2, and $\sqsubseteq$ is reflexive by Definition 21.

Case T-INT, T-ARITH, T-PTRARITH, T-RELATION

Impossible.

Case T-PURESUB

By the form of the rule,

$$\emptyset \vdash_m v : \tau \tag{72.4}$$

$$\emptyset \vdash \tau <: \texttt{ref}(\ell, i) \tag{72.5}$$

By Fact 72.5 and Lemma 34, there are two cases:

Case 1

$$\tau = \{\nu : \texttt{ref}(\ell_1, i_1) \mid \phi_1\} \tag{72.6}$$

$$\ell_1 \sqsubseteq \ell \tag{72.7}$$

$$i_1 \stackrel{\sim}{\subseteq} i \tag{72.8}$$

$$\Gamma \vDash \phi_1 \Rightarrow \phi$$

By the Fact 72.4, Fact 72.6, and the inductive hypothesis, either

$$v = 0_{|W|}$$

in which case we are done, or

$$v = \text{bref}(r, n, z)$$

$$n \in [\![i_1]\!]$$

$$\exists \ell_j \sqsubseteq \ell . \ell_j \in \text{Clocs}(r, m)$$

for some n. Note that, by Definition 21, this final condition implies

$$\exists \ell_j \sqsubseteq \ell_1 . \ell_j \in \text{Clocs}(r, m)$$

By Fact 72.8 and Proposition 1,

$$n \mathrel{\widetilde{\subseteq}} i$$

By Proposition 2,

$$n \in [\![i]\!]$$

as required.

Case 2

$$\tau = \{\nu : \texttt{int}(W, 0) \mid \phi_1\} \tag{72.9}$$

By Fact 72.4, (1), and Fact 72.9,

$$v = 0_{|W|}$$

as required.

□

C.18 Unfolding, Folding, Heap Modeling, Well-Formedness

This section presents the key lemmas that relate the location unfold and fold operations to our concepts of heap modeling and well-formedness.

Lemma 73 (Modeling of Freshly-Allocated Locations)**.** *For any block b such that $\emptyset \vDash b$,*

$$\mathbb{Z} \mapsto 0_{|1|} \vDash_m b^0.$$

Proof. We assume that the run-time block $\mathbb{Z} \mapsto 0_{|1|}$ can be treated as the run-time block containing the bindings

$$\{n \mapsto 0_{|\text{SizeOf}(\tau)|} \mid n \in [\![i]\!],\ i : \tau \in b^0\} \cup \{n \mapsto 0_{|1|} \mid n \notin \text{dom}(b^0)\}.$$

This assumption is justified by the facts that the run-time blocks have the same physical representation in memory and that all bindings in the block are disjoint.

The remainder of the proof proceeds by straightforward induction on the structure of b, using BM-SINGLE, BM-SEQUENCE, and the definition of b^0. □

Lemma 74 (Fresh Concrete Location Preserves Heap Modeling)**.**

$$\begin{aligned} &\textit{If } s \vDash_m h, && (74.1)\\ &\quad c \vDash_m b, && (74.2)\\ &\quad \ell_j \notin \mathrm{dom}(m) && (74.3)\\ &\quad m' = m[\ell_j \mapsto r], && (74.4)\\ &\quad h' = h * \ell_j \mapsto b && (74.5)\\ &\quad s' = s[r \mapsto c] && (74.6)\\ &\textit{then } s' \vDash_{m'} h'. \end{aligned}$$

Proof. By Fact 74.1 and Definition 23,

$$\forall \ell'_k \in \mathrm{dom}(h).\ \ell'_k \in \mathrm{dom}(m) \quad (74.7)$$

$$\mathrm{rng}(m) \subseteq \mathrm{dom}(s) \quad (74.8)$$

And, for all $r' \mapsto c' \in s$, either

$$\begin{aligned} &\text{exists } \ell'_k \in \mathrm{Clocs}(r', m),\\ &\quad h = h_0 * \ell'_k \mapsto b',\\ &\quad c' \vDash_m b', \end{aligned}$$

or

$$\begin{aligned} &\mathrm{Clocs}(r', m) \cap \mathrm{dom}(h) = \emptyset,\\ &\quad h = h_0 * \widetilde{\ell}' \mapsto b',\\ &\quad c' \vDash_m b' \end{aligned}$$

By Fact 74.3 and Fact 74.4,

$$m \subseteq m' \quad (74.9)$$

By Fact 74.4, Fact 74.5, Fact 74.7, and Fact 74.9,

$$\forall \ell'_k \in \mathrm{dom}(h').\ \ell'_k \in \mathrm{dom}(m') \quad (74.10)$$

By Fact 74.4, Fact 74.8, and Fact 74.6,

$$\mathrm{rng}(m') \subseteq \mathrm{dom}(s') \quad (74.11)$$

By Fact 74.10 and Fact 74.11, we have satisifed the first two conditions of Definition 23. To show the final condition, we choose an arbitrary $r' \mapsto c' \in s'$ and split cases on whether $r' = r$.

Case $r' \neq r$

We again split cases, this time on how the location was modeled according to Fact 74.1 and Definition 23.

Case exists $\ell'_k \in \text{Clocs}(r', m),\ h = h_0 * \ell'_k \mapsto b',\ c' \vDash_m b'$

Then by Fact 74.9 and Definition 19,

$$\ell'_k \in \text{Clocs}(r', m')$$

By Fact 74.5,

$$h' = h'_0 * \ell'_k \mapsto b'$$

And, by Fact 74.9 and Lemma 24,

$$c' \vDash_{m'} b'$$

Thus, part (3a) of Definition 23 remains satisfied.

Case exists $\ell'_k \in \text{Clocs}(r', m),\ \text{Clocs}(r', m) \cap \text{dom}(h) = \emptyset,\ h = h_0 * \overset{\sim}{\ell}' \mapsto b',\ c' \vDash_m b'$

Then by Fact 74.9 and Definition 19,

$$\ell'_k \in \text{Clocs}(r', m')$$

Note that

$$\text{Clocs}(r', m) \cap \text{dom}(h) = \emptyset$$

implies, by, $r' \neq r$, Fact 74.4, Fact 74.3, Definition 19, that

$$\text{Clocs}(r', m') \cap \text{dom}(h') = \emptyset$$

By Fact 74.5,

$$h' = h'_0 * \overset{\sim}{\ell}' \mapsto b'$$

Finally, by Fact 74.9 and Lemma 24,

$$c' \vDash_{m'} b'$$

Thus, part (3b) of Definition 23 remains satisfied.

Case $r' = r$

Then

$$c' = c$$

By Fact 74.4 and Definition 19,

$$\ell_j \in \text{Clocs}(r, m')$$

By Fact 74.5,

$$h' = h * \ell_j \mapsto b$$

By Fact 74.2 and Lemma 24,

$$c \vDash_{m'} b$$

Thus, part (3a) of Definition 23 is satisfied. □

Lemma 75 (Unfolding Preserves Block Modeling)**.**

$$\begin{aligned} &\textit{If } c \vDash_m \overline{n_k} : \overline{\{\nu : t_k \mid \phi_k\}},\ \overline{i_j^+} : \overline{\tau_j} \\ &\textit{then } c \vDash_m \overline{n_k} : \overline{\{\nu : t_k \mid \nu = c(n_k)\}},\ \overline{i_j^+} : \overline{\theta\tau_j} \\ &\textit{where } \theta = [\overline{@n_k} \mapsto \overline{c(n_k)}] \end{aligned}$$

Proof. The proof proceeds by induction on the derivation of $c \vDash_m \overline{n_k} : \overline{\{\nu : t_k \mid \phi_k\}},\ \overline{i_j^+} : \overline{\tau_j}$. We split cases on the final rule used.

Case BM-SEQUENCE

Trivial, as the substitution is empty.

Case BM-SINGLE

By the form of the rule,

$$\overline{n_k} : \overline{\{\nu : t_k \mid \phi_k\}} = n : \{\nu : t_n \mid \phi_n\},\ \overline{n_m} : \overline{\{\nu : t_m \mid \phi_m\}}$$

$$\emptyset \vdash_m c(n) : \{\nu : t_n \mid \phi_n\} \tag{75.1}$$

$$c \vDash_m \overline{n_m} : \overline{\{\nu : t_m \mid \phi_m[@n \mapsto c(n)]\}},\ \overline{i_j^+} : \overline{\tau_j[@n \mapsto c(n)]} \tag{75.2}$$

By the inductive hypothesis and Fact 75.2,

$$c \vDash_m \overline{n_m} : \overline{\{\nu : t_m \mid \nu = c(n_m)\}},\ \overline{i_j^+} : \overline{\theta'(\tau_j[@n \mapsto c(n)])}$$

where $\theta' = [\overline{n_m} \mapsto \overline{c(n_m)}]$

Equivalently,

$$c \vDash_m \overline{n_m} : \overline{\{\nu : t_m \mid \nu = c(n_m)\}},\ \overline{i_j^+} : \overline{\theta\tau_j}$$

Since $@n \in \mathrm{dom}(\theta)$ and $@n \notin \mathrm{rng}(\theta)$, this is equivalent to

$$c \vDash_m \overline{n_m} : \overline{\{\nu : \ t_m \ \mid \ (\nu = c(n_m))[@n \mapsto c(n)]\}}, \ \overline{i_j^+} : \overline{(\theta\tau_j)[@n \mapsto c(n)]} \tag{75.3}$$

By Fact 75.1 and Lemma 50,

$$\emptyset \vdash_m c(n) : \{\nu : \ t_n \ \mid \ \nu = c(n)\} \tag{75.4}$$

By Fact 75.3, Fact 75.4, and BM-SINGLE,

$$c \vDash_m n : \{\nu : \ t_n \ \mid \ \nu = c(n)\}, \ \overline{n_m} : \overline{\{\nu : \ t_m \ \mid \ \nu = c(n_m)\}}, \ \overline{i_j^+} : \overline{\theta\tau_j}$$

as required. □

Lemma 76 (Unfolding Preserves Heap Modeling)**.**

$$\begin{aligned}
&\textit{If } s \vDash_m h, \\
&\quad h = h_0 * \widetilde{\ell} \mapsto b, \\
&\quad b = \overline{n_k} : \overline{\{\nu : \ t_k \ \mid \ \phi_k\}}, \ \overline{i^+} : \overline{\tau^+}, \\
&\quad c = s(r), \\
&\quad c \vDash_m b, && (76.1) \\
&\quad \ell_j \notin \mathrm{dom}(m), \\
&\quad m' = m[\ell_j \mapsto r], \\
&\quad \theta = [\overline{@n_k} \mapsto \overline{c(n_k)}], && (76.2) \\
&\quad h' = h_0 * \widetilde{\ell} \mapsto b * \ell_j \mapsto \overline{n_k} : \overline{\{\nu : \ t_k \ \mid \ \nu = c(n_k)\}}, \ \overline{i^+} : \overline{\theta\tau^+}, \\
&\textit{then } s \vDash_{m'} h'.
\end{aligned}$$

Proof. By Fact 76.1, Fact 76.2, and Lemma 75,

$$c \vDash_m \overline{n_k} : \overline{\{\nu : \ t_k \ \mid \ \nu = c(n_k)\}}, \ \overline{i^+} : \overline{\theta\tau^+}.$$

The rest follows by the above, the given premises, and Lemma 74, with $s' = s[r \mapsto s(r)] = s$. □

Lemma 77 (Subtyping Preserves Block Modeling)**.**

$$\begin{aligned}
&\textit{If } c \vDash_m b_1 \\
&\textit{and } \emptyset \vdash b_1 <: b_2, \\
&\textit{then } c \vDash_m b_2.
\end{aligned}$$

Proof. The proof proceeds by induction on the derivation of $\emptyset \vdash b_1 <: b_2$. We split cases on the final rule used.

Case <:-**SINGLE**

By the form of the rule,

$$b_1 = n : \tau_1,\ b_1'$$
$$b_2 = n : \tau_2,\ b_2'$$
$$\emptyset \vdash \tau_1 <: \tau_2 \tag{77.1}$$
$$x : \tau_1 \vdash b_1'[@n \mapsto x] <: b_2'[@n \mapsto x] \tag{77.2}$$

The only rule that could have been used to prove $c \vDash_m b_1$ is BM-SINGLE, from which we have

$$\emptyset \vdash_m c(n) : \tau_1 \tag{77.3}$$
$$c \vDash_m b_1'[@n \mapsto c(n)] \tag{77.4}$$

By Fact 77.3 and WFSUBST-VAR,

$$x : \tau_1 \vDash_m [x \mapsto c(n)] \tag{77.5}$$

By Fact 77.2, Fact 77.5, and Lemma 55,

$$\emptyset \vdash b_1'[@n \mapsto c(n)] <: b_2'[@n \mapsto c(n)] \tag{77.6}$$

By Fact 77.4, Fact 77.6, and the inductive hypothesis,

$$c \vDash_m b_2'[@n \mapsto c(n)] \tag{77.7}$$

By Fact 77.1, Fact 77.3, and T-PURESUB,

$$\emptyset \vdash_m c(n) : \tau_2 \tag{77.8}$$

By Fact 77.8, Fact 77.7, and BM-SINGLE,

$$c \vDash_m n : \tau_2,\ b_2'$$

as required.

Case <:-**SEQUENCE**

By the form of the rule,

$$b_1 = i^+ : \tau_1,\ b_1'$$
$$b_2 = i^+ : \tau_2,\ b_2'$$
$$\emptyset \vdash \tau_1 <: \tau_2 \tag{77.9}$$
$$\emptyset \vdash b_1' <: b_2' \tag{77.10}$$

The only rule that could have been used to prove $c \vDash_m b_1$ is BM-SEQUENCE, from which we have

$$\forall n \in \mathrm{dom}(c) \cap [\![i^+]\!]. \emptyset \vdash_m c(n) : \tau_1 \tag{77.11}$$

$$c \vDash_m b_1' \tag{77.12}$$

By Fact 77.9, Fact 77.11, and T-PURESUB,

$$\forall n \in \mathrm{dom}(c) \cap [\![i^+]\!]. \emptyset \vdash_m c(n) : \tau_2 \tag{77.13}$$

By Fact 77.10, Fact 77.12, and the inductive hypothesis,

$$c \vDash_m b_2' \tag{77.14}$$

By Fact 77.13, Fact 77.14, and BM-SEQUENCE,

$$c \vDash_m i^+ : \tau_2,\ b_2'$$

as required. □

Lemma 78 (Subtyping Preserves Heap Modeling)**.**

$$\begin{aligned} &\textit{If } s \vDash_m h_1 \\ &\textit{and } \emptyset \vdash h_1 <: h_2 \\ &\textit{then } s \vDash_m h_2. \end{aligned}$$

Proof. The proof follows straightforwardly by induction on the derivation of $\emptyset \vdash h_1 <: h_2$, using $s \vDash_m h_1$, Definition 23, and Lemma 77. □

Lemma 79 (Well-Formed Substitutions from Block Contents)**.**

$$\begin{aligned} &\textit{If } c \vDash_m \overline{n_k} : \overline{\tau_k},\ \overline{i^+} : \overline{\tau^+}, \\ &\quad \overline{x_k} \textit{ disjoint}, \\ &\textit{and } \theta = [\overline{x_k} \mapsto \overline{c(n_k)}], \\ &\textit{then } \overline{x_k} : \overline{\theta\tau_k} \vDash_. \theta \end{aligned}$$

Proof. The proof proceeds by induction on the derivation of $c \vDash_m \overline{n_k} : \overline{\tau_k},\ \overline{i^+} : \overline{\tau^+}$, using WFSUBST-VAR. □

Lemma 80 (One Location Name Mapped Per Run-Time Location)**.**

$$\textit{If } \emptyset \vdash_m \mathrm{bref}(r, n, z) : \{\nu : \mathtt{ref}(\widetilde{\ell}, i) \mid \phi\}, \tag{80.1}$$

$$\vDash m, \tag{80.2}$$

$$\textit{and } \ell_j' \in \mathrm{Clocs}(r, m), \tag{80.3}$$

$$\textit{then } \ell' = \ell.$$

Proof. By Fact 80.1 and Lemma 72, there exists

$$\ell_k'' \in \text{Clocs}(r, m) \tag{80.4}$$

$$\ell_k'' \sqsubseteq \widetilde{\ell} \tag{80.5}$$

By Fact 80.5 and Definition 21,

$$\ell'' = \ell \tag{80.6}$$

By Fact 80.3, Fact 80.4, Fact 80.2, and Definition 20,

$$\ell'' = \ell' \tag{80.7}$$

By Fact 80.6 and Fact 80.7,

$$\ell' = \ell$$

as required. □

Lemma 81 (Folding Preserves Heap Modeling)**.**

$$\begin{aligned}
&\textit{If } s \vDash_m h_1, && (81.1)\\
&\quad h_1 = h_2 * \ell_j \mapsto b_1,\\
&\quad h_2 = h_0 * \widetilde{\ell} \mapsto b_2,\\
&\quad \varnothing \vdash b_1 <: b_2, && (81.2)\\
&\quad \varnothing \vDash h_1, && (81.3)\\
&\textit{and } \vDash m, && (81.4)\\
&\textit{then } s \vDash_m h_2.
\end{aligned}$$

Proof. We note that, by assumption, parts (1) and (2) of Definition 23 are satisfied. It only remains to show that part (3) is satisfied.

Let $r \mapsto c \in s$. By Fact 81.1 and part (3) of Definition 23, there are two cases:

Case exists $\ell_k' \in \text{Clocs}(r, m)$**,** $h_1 = h_0' * \ell_k' \mapsto b$**, and** $c \vDash_m b$

Then there are two cases:

Case $\ell' \neq \ell$

Then $h_2 = h_0'' * \ell_k' \mapsto b$ for some h_0''. By assumption, $\ell_k' \in \text{Clocs}(r, m)$ and $c \vDash_m b$. Thus, part (3a) of Definition 23 is satisfied.

Case $\ell' = \ell$

The only rule which could have been used to prove Fact 81.3 is WF-HConcrete. By

the form of the rule,

$$\ell_q \notin \mathrm{dom}(h_2) \tag{81.5}$$

for any q, so we have $\ell_k = \ell_j$ and $b = b_1$. By assumption, Fact 81.2, and Lemma 77,

$$c \vDash_m b_2 \tag{81.6}$$

By Fact 81.4 and Definition 20,

$$\text{if } \ell''_z \in \mathrm{Clocs}(r, m) \text{ then } \ell'' = \ell$$

So by Fact 81.5,

$$\mathrm{Clocs}(r, m) \cap \mathrm{dom}(h_2) = \emptyset \tag{81.7}$$

So by the definition of h_2, Fact 81.6, and Fact 81.7, part (3b) of Definition 23 is satisfied.

Case exists $\ell'_k \in \mathrm{Clocs}(r, m)$**,** $\mathrm{Clocs}(r, m) \cap \mathrm{dom}(h_1) = \emptyset$**,** $h_1 = h'_0 * \overset{\sim}{\ell}' \mapsto b$**, and** $c \vDash_m b$

Note that

$$\mathrm{dom}(h_2) \subsetneq \mathrm{dom}(h_1)$$

Note also that h_2 contains all the same abstract location bindings as h_1, so

$$h_2 = h''_0 * \overset{\sim}{\ell}' \mapsto b$$

And by assumption we have

$$c \vDash_m b$$

Thus, part (3b) of Definition 23 is satisfied. □

Lemma 82 (Just-Unfolded Location Models Corresponding Abstract Location)**.**

$$\begin{aligned}
&\textit{If } \emptyset \vDash h * \ell_k \mapsto b_2, && (82.1)\\
&\quad h = h_0 * \overset{\sim}{\ell} \mapsto b,\\
&\quad \ell_j \in \mathrm{Clocs}(r, m), && (82.2)\\
&\quad \vDash m, && (82.3)\\
&\quad s \vDash_m h, && (82.4)\\
&\textit{and } s(r) = c,\\
&\textit{then } c \vDash_m b.
\end{aligned}$$

Proof. The only rule which could have been used to prove Fact 82.1 is WF-HConcrete, by which we have

$$\ell_u \notin \mathrm{dom}(h) \text{ for any } u \tag{82.5}$$

$$\emptyset \vDash h \tag{82.6}$$

Suppose

$$\ell''_v \in \mathrm{Clocs}(r, m) \tag{82.7}$$

By Fact 82.2, Fact 82.3, and Definition 20,

$$\ell'' = \ell \tag{82.8}$$

By Fact 82.5 and Fact 82.8,

$$\mathrm{Clocs}(r, m) \cap \mathrm{dom}(h) = \emptyset \tag{82.9}$$

By Fact 82.4, Fact 82.9, and Definition 23, there exists b' such that

$$h = h'_0 * \widetilde{\ell} \mapsto b' \tag{82.10}$$

$$c \vDash_m b' \tag{82.11}$$

By Fact 82.10, Fact 82.6, and WF-Abstract,

$$\widetilde{\ell} \notin \mathrm{dom}(h'_0)$$

So

$$b' = b$$

and by Fact 82.11,

$$c \vDash_m b$$

as required. □

Lemma 83 (Uniqueness of Modeled Concrete Locations)**.**

$$\begin{aligned} &\textit{If } s \vDash_m h, && (83.1)\\ &\quad h = h_0 * \ell_j \mapsto b, \\ &\quad r \in \mathrm{dom}(s), && (83.2)\\ &\quad \ell_j \in \mathrm{Clocs}(r, m), && (83.3)\\ &\quad \emptyset \vDash h, && (83.4)\\ &\textit{and } \vDash m, && (83.5)\\ &\textit{then } s(r) \vDash_m b. \end{aligned}$$

Proof. By Definition 23, either part (3a) or (3b) of Definition 23 must apply to $s(r)$. By Fact 83.3,

$$\text{Clocs}(r, m) \cap \text{dom}(h) \neq \emptyset$$

So (3b) does not apply; instead, by (3a), there must exist some $\ell'_k \in \text{Clocs}(r, m)$ such that

$$h = h'_0 * \ell'_k \mapsto b_k$$
$$s(r) \vDash_m b_k$$

By Fact 83.3, Fact 83.5, and Definition 20,

$$\ell' = \ell$$

So

$$h = h'_0 * \ell_k \mapsto b_k$$

The only rule that could have been used to prove Fact 83.4 is WF-HCONCRETE, by which we have

$$\ell_k = \ell_j$$
$$b_k = b$$

So

$$s(r) \vDash_m b$$

as required. □

Lemma 84 (Types of Modeled Values)**.**

$$\begin{aligned} &\textit{If } \emptyset \vDash b_1,\ i : \tau,\ b_2, && (84.1)\\ &\quad c \vDash_m b_1,\ i : \tau,\ b_2,\\ &\textit{and } n \in [\![i]\!],\\ &\textit{then } \emptyset \vdash_m c(n) : \tau. \end{aligned}$$

Proof. By induction on the derivation of $c \vDash_m b_1,\ i : \tau,\ b_2$. We split cases on the final rule used.

Case BM-SINGLE

By the form of the rule,

$$b_1,\ i : \tau,\ b_2 = n : \tau',\ b'$$
$$\emptyset \vdash_m c(n) : \tau' \quad (84.2)$$
$$c \vDash_m b'[@n \mapsto c(n)] \quad (84.3)$$

If b_1 is empty, then $i = n$, $\tau = \tau'$, and the desired conclusion follows by Fact 84.2. Otherwise, by Fact 84.1 and Lemma 29, no type in the block contains free locations, so

$$b'[@n \mapsto c(n)] = b'$$

so by Fact 84.3,

$$c \vDash_m b' \tag{84.4}$$

By Fact 84.1 and WF-NDBLOCK,

$$\emptyset \vDash b' \tag{84.5}$$

The desired conclusion then follows by Fact 84.4, Fact 84.5, and the inductive hypothesis.

Case BM-SEQUENCE

By the form of the rule,

$$b_1,\ i : \tau,\ b_2 = i^+ : \tau',\ b'$$

$$\forall m \in \mathrm{dom}(c) \cap [\![i^+]\!].\ \emptyset \vdash_m c(m) : \tau' \tag{84.6}$$

$$c \vDash_m b' \tag{84.7}$$

If b_1 is empty, then $i = i^+$, $\tau = \tau'$, and the desired conclusion follows by Fact 84.6. Otherwise, the rest of the case proceeds as in the case for BM-SINGLE. □

Lemma 85 (Types of Read Values)**.**

$$\begin{aligned}
&\textit{If } \emptyset \vdash_m \mathrm{bref}(r, n, z) : \mathtt{ref}(\ell_j, i), && (85.1)\\
&\quad h = h_0 * \ell_j \mapsto \ldots,\ i : \tau,\ \ldots, && (85.2)\\
&\quad \emptyset \vDash h, && (85.3)\\
&\quad s \vDash_m h, && (85.4)\\
&\textit{and } \vDash m, && (85.5)\\
&\textit{then } \emptyset \vdash_m s(r)(n) : \tau.
\end{aligned}$$

Proof. By Fact 85.1 and Lemma 72,

$$n \in [\![i]\!] \tag{85.6}$$

$$\ell_j \in \mathrm{Clocs}(r, m) \tag{85.7}$$

By Fact 85.7, Fact 85.4, Definition 19, and Definition 23,

$$r \in \mathrm{dom}(s) \tag{85.8}$$

By Fact 85.3, Fact 85.4, Fact 85.5, Fact 85.2, Fact 85.8, Fact 85.7, and Lemma 83,

$$s(r) \vDash_m \ldots, \; i : \tau, \; \ldots \tag{85.9}$$

The only rule that could have been used to prove Fact 85.3 is WF-HCONCRETE, by which we have

$$\emptyset \vDash \ldots, \; i : \tau, \; \ldots \tag{85.10}$$

By Fact 85.9, Fact 85.10, Fact 85.6, and Lemma 84,

$$\emptyset \vdash_m s(r)(n) : \tau$$

as required. □

Lemma 86 (Irrelevant Offsets in Modeling)**.**

$$\begin{aligned} &\textit{If } c \vDash_m \bar{i} : \overline{\tau} \\ &\textit{and } n \notin [\![\bar{i}]\!], \\ &\textit{then } c[n \mapsto v] \vDash_m \bar{i} : \overline{\tau}. \end{aligned}$$

Proof. Straightforward induction on the derivation of $c \vDash_m \bar{i} : \overline{\tau}$. □

Lemma 87 (Weak Updates Preserve Block Modeling)**.**

$$\begin{aligned} &\textit{If } c \vDash_m b, \\ &\quad b = b_1, \; i^+ : \tau, \; b_2, \\ &\quad \emptyset \vDash b, && (87.1) \\ &\quad n \in [\![i^+]\!], && (87.2) \\ &\textit{and } \emptyset \vdash_m v : \tau, && (87.3) \\ &\textit{then } \; \mathrm{Write}(c, n, v) \vDash_m b. \end{aligned}$$

Proof. The proof proceeds by induction on the derivation of $c \vDash_m b$. We split cases on the final rule used.

Case BM-SINGLE

By the form of the rule,

$$\begin{aligned} &b_1 = p : \tau_p, \; b_1' \\ &\emptyset \vdash_m c(p) : \tau_p && (87.4) \\ &c \vDash_m (b_1', \; i^+ : \tau, \; b_2)[@p \mapsto c(p)] \end{aligned}$$

By Fact 87.1 and Lemma 29, no type in the above block can contain a location, so this is equivalent to

$$c \vDash_m b_1',\ i^+ : \tau,\ b_2 \tag{87.5}$$

The only rule that could have been used to prove Fact 87.1 is WF-NDBLOCK, from which we have

$$\emptyset \vDash b_1',\ i^+ : \tau,\ b_2 \tag{87.6}$$

By Fact 87.5, Fact 87.6, Fact 87.2, Fact 87.3, and the inductive hypothesis,

$$\text{Write}(c, n, v) \vDash_m b_1',\ i^+ : \tau,\ b_2$$

Again, this is equivalent to

$$\text{Write}(c, n, v) \vDash_m (b_1',\ i^+ : \tau,\ b_2)[@p \mapsto c(p)] \tag{87.7}$$

By Fact 87.1 and WF-NDBLOCK,

$$p \cap \bigcup \text{OffsetsOf}(i^+, \tau) = \emptyset$$

So by Fact 87.4 and Definition 10,

$$\emptyset \vdash_m (\text{Write}(c, n, v))(p) : \tau_p \tag{87.8}$$

By Fact 87.8, Fact 87.7, and BM-SINGLE,

$$\text{Write}(c, n, v) \vDash_m b$$

as required.

Case BM-SEQUENCE

We split cases on whether b_1 is empty.

Case b_1 is empty

By the form of the rule,

$$\forall k \in \text{dom}(c) \cap [\![i^+]\!].\ \emptyset \vdash_m c(k) : \tau \tag{87.9}$$

$$c \vDash_m b_2 \tag{87.10}$$

The only rule that could have been used to prove Fact 87.1 is WF-NDBLOCK, from which we have

$$\text{DisjointOffsets}(i^+ : \tau)$$

That is, for any $p, q \in [\![i^+]\!]$ such that $p \neq q$,

$$(\bigcup \text{OffsetsOf}(p, \tau)) \cap (\bigcup \text{OffsetsOf}(q, \tau)) = \emptyset \tag{87.11}$$

By Fact 87.3, Fact 87.2, Fact 87.9, Fact 87.11, and Definition 10,

$$\forall k \in \text{dom}(\text{Write}(c, n, v)) \cap [\![i^+]\!]. \emptyset \vdash_m (\text{Write}(c, n, v))(k) : \tau \tag{87.12}$$

By Fact 87.1 and WF-NDBLOCK, all $\text{OffsetsOf}(i^+, \tau)$ are disjoint from all $\text{OffsetsOf}(b_2)$. Thus, by Fact 87.10 and Lemma 86,

$$\text{Write}(c, n, v) \vDash_m b_2 \tag{87.13}$$

By Fact 87.12, Fact 87.13, and BM-SEQUENCE,

$$\text{Write}(c, n, v) \vDash_m i^+ : \tau,\ b_2$$

as required.

Case $b_1 = i_1^+ : \tau_1,\ b_1'$

By the form of the rule,

$$\forall k \in \text{dom}(c) \cap [\![i_1^+]\!]. \emptyset \vdash_m c(k) : \tau_1 \tag{87.14}$$

$$c \vDash_m b_1',\ i^+ : \tau,\ b_2 \tag{87.15}$$

By Fact 87.1 and WF-NDBLOCK,

$$\emptyset \vDash b_1',\ i^+ : \tau,\ b_2 \tag{87.16}$$

$$(\bigcup \text{OffsetsOf}(i_1^+, \tau_1)) \cap (\bigcup \text{OffsetsOf}(i^+, \tau)) = \emptyset \tag{87.17}$$

By Fact 87.15, Fact 87.16, Fact 87.2, Fact 87.3, and the inductive hypothesis,

$$\text{Write}(c, n, v) \vDash_m b_1',\ i^+ : \tau,\ b_2 \tag{87.18}$$

By Fact 87.14 Fact 87.17, and Definition 10,

$$\forall k \in \text{dom}(\text{Write}(c, n, v)) \cap [\![i_1^+]\!]. \emptyset \vdash_m c(k) : \tau_1 \tag{87.19}$$

By Fact 87.18 and Fact 87.19 and BM-SEQUENCE,

$$\text{Write}(c, n, v) \vDash_m i_1^+ : \tau_1,\ b_1',\ i^+ : \tau,\ b_2$$

as required. □

Lemma 88 (Non-Dependent Partial Block Well-Formedness)**.** $\Gamma \vDash b_1,\ b_2$ *iff* $\Gamma \vDash b_1$ *and* $\Gamma \vDash b_2$.

Proof. Immediate by the form of WF-NDBLOCK. □

Lemma 89 (Strong Update Preserves Block Modeling)**.**

$$
\begin{aligned}
&\textit{If } b = b_1,\ n : \tau_1,\ b_2, \\
&\quad c \vDash_m b, && (89.1)\\
&\quad \emptyset \vDash b, && (89.2)\\
&\quad \emptyset \vdash_m v : \tau_2, && (89.3)\\
&\quad \text{SizeOf}(\tau_2) = \text{SizeOf}(\tau_1), && (89.4)\\
&\textit{and } \emptyset \vDash \tau_2, && (89.5)\\
&\textit{then } \text{Write}(c, n, v) \vDash_m b_1,\ n : \tau_2,\ b_2.
\end{aligned}
$$

Proof. First, we note that the only rule that could have been used to prove Fact 89.2 is WF-NDBLOCK. By the form of the rule,

$$
\begin{aligned}
&b = \overline{i_j} : \overline{\tau_j} \\
&\quad \text{DisjointOffsets}(\overline{i_j} : \overline{\tau_j}) && (89.6)\\
&\forall j. \emptyset \vDash \tau_j && (89.7)
\end{aligned}
$$

Also, by Definition 7, Fact 89.3, Fact 89.4, and Lemma 71,

$$\text{SizeOf}(v) = \text{SizeOf}(\tau_1)$$

so that

$$[n, n + \text{SizeOf}(v)) = \text{OffsetsOf}(n, \tau_1) \tag{89.8}$$

The rest of the proof proceeds by induction on the derivation of $c \vDash_m b_1,\ n : \tau_1,\ b_2$. We split cases on the final rule used.

Case BM-SINGLE

By the form of the rule, we have

$$
\begin{aligned}
&b = k : \tau,\ b_1' \\
&\emptyset \vdash_m c(k) : \tau && (89.9)\\
&c \vDash_m b_2[@k \mapsto c(k)] && (89.10)
\end{aligned}
$$

By Fact 89.7, the types of b_2 cannot contain locations, so Fact 89.10 is equivalent to

$$c \vDash_m b_2 \tag{89.11}$$

We further split cases on whether b_1 is empty.

Case b_1 is empty

Then $k = n$ and $b_1' = b_2$. By Fact 89.6 and Fact 89.8,

$$[n, n + \text{SizeOf}(v)) \cap \text{OffsetsOf}(b_2) = \emptyset \tag{89.12}$$

By Fact 89.12, Definition 10, and Lemma 86,

$$\text{Write}(c, n, v) \vDash_m b_2$$

Since b_2 cannot contain offsets, this is equivalent to

$$\text{Write}(c, n, v) \vDash_m b_2[@n \mapsto v] \tag{89.13}$$

By Fact 89.3, Fact 89.13, and BM-SINGLE,

$$\text{Write}(c, n, v) \vDash_m n : \tau_2,\ b_2 \tag{89.14}$$

as required.

Case $b_1 = k : \tau,\ b_1'$

By Fact 89.6,

$$n \neq k$$

Then by Fact 89.9,

$$\emptyset \vdash_m (\text{Write}(c, n, v))(k) : \tau \tag{89.15}$$

By Fact 89.2 and Lemma 88,

$$\emptyset \vDash b_1',\ n : \tau_1,\ b_2$$

By Fact 89.7, this is equivalent to

$$\emptyset \vDash (b_1',\ n : \tau_1,\ b_2)[@k \mapsto c(k)] \tag{89.16}$$

The only rule which could have been used to prove Fact 89.1 is BM-SINGLE, from which we have

$$c \vDash_m (b_1',\ n : \tau_1,\ b_2)[@k \mapsto c(k)] \tag{89.17}$$

By Fact 89.17, Fact 89.16, Fact 89.3, Fact 89.5, Fact 89.4, and the inductive hypothesis,

$$\text{Write}(c, n, v) \vDash_m b_1',\ n : \tau_2,\ b_2$$

By Fact 89.7, this is equivalent to

$$\text{Write}(c, n, v) \vDash_m (b_1',\ n : \tau_2,\ b_2)[@k \mapsto c[n \mapsto v](k)] \tag{89.18}$$

By Fact 89.15, Fact 89.18, and BM-SINGLE,

$$\text{Write}(c, n, v) \vDash_m b_1,\ n : \tau_2,\ b_2$$

as required.

Case BM-SEQUENCE

By the form of the rule, we have

$$b_1 = i^+ : \tau,\ b_1'$$

$$\forall k \in \text{dom}(c) \cap [\![i^+]\!].\ \emptyset \vdash_m c(k) : \tau \tag{89.19}$$

$$c \vDash_m b_1',\ n : \tau_1,\ b_2 \tag{89.20}$$

By Fact 89.6 and Fact 89.8,

$$[n, n + \text{SizeOf}(v)) \cap [\![i^+]\!] = \emptyset \tag{89.21}$$

By Fact 89.21, Fact 89.19, and Definition 10,

$$\forall k \in \text{dom}(\text{Write}(c, n, v)) \cap [\![i^+]\!].\ \emptyset \vdash_m (\text{Write}(c, n, v))(k) : \tau \tag{89.22}$$

By Fact 89.2 and Lemma 88,

$$\emptyset \vDash b_1',\ n : \tau_2,\ b_2 \tag{89.23}$$

By Fact 89.20, Fact 89.23, Fact 89.3, Fact 89.5, Fact 89.4, and the inductive hypothesis,

$$\text{Write}(c, n, v) \vDash_m b_1',\ n : \tau_2,\ b_2 \tag{89.24}$$

By Fact 89.22, Fact 89.24, and BM-SEQUENCE,

$$\text{Write}(c, n, v) \vDash_m b_1,\ n : \tau_2,\ b_2$$

as required. □

Lemma 90 (Strong Update Preserves Block Well-Formedness)**.**

$$\begin{gathered} \textit{If } \emptyset \vDash b_1,\ n : \tau_1,\ b_2, \\ \emptyset \vDash \tau_2, \\ \textit{and } \text{SizeOf}(\tau_1) = \text{SizeOf}(\tau_2), \\ \textit{then } \emptyset \vDash b_1,\ n : \tau_2,\ b_2 \end{gathered}$$

Proof. Immediate from the form of WF-NDBLOCK and the given assumptions. □

Lemma 91 (Consistent Block Updating)**.**

$$\begin{aligned}
&\textit{If } \emptyset \vDash h, && (91.1)\\
&\quad s \vDash_m h, && (91.2)\\
&\quad h = h_0 * \ell_j \mapsto b_1, &\\
&\quad \ell_j \in \text{Clocs}(r, m), && (91.3)\\
&\quad c \vDash_m b_2, && (91.4)\\
&\quad s_2 = s[r \mapsto c], &\\
&\textit{and } h_2 = h_0 * \ell_j \mapsto b_2, && (91.5)\\
&\textit{then } s_2 \vDash_m h_2.
\end{aligned}$$

Proof. By Definition 23, we must show

1. $\forall \ell_j \in \text{dom}(h).\ \ell_j \in \text{dom}(m)$
2. $\text{rng}(m) \subseteq \text{dom}(s_2)$
3. For all $r' \in \text{dom}(s_2)$, either
   (a) exists $\ell'_k \in \text{Clocs}(r', m)$, $h = h'_0 * \ell'_k \mapsto b$, and $s_2(r') \vDash_m b$, or
   (b) $\text{Clocs}(r', m) \cap \text{dom}(h) = \emptyset$, $h = h'_0 * \widetilde{\ell} \mapsto b$, and $s_2(r') \vDash_m b$

Note (1) and (2) follow immediately from the assumptions.

We choose an arbitrary $r' \in \text{dom}(s')$ and show that one of the conditions of (3) is satisfied. We split cases on whether $r' = r$.

Case $r' = r$

Then

$$s_2(r') = c$$

And, by Fact 91.3,

$$\ell_j \in \text{Clocs}(r', m)$$

Note also that, by Fact 91.5,

$$h_2 = h_0 * \ell_j \mapsto b_2$$

while, by Fact 91.4,

$$c \vDash_m b_2$$

so (3a) is satisfied.

Case $r' \neq r$

By Definition 23 and Fact 91.2, either

1. exists $\ell'_k \in \text{Clocs}(r', m)$, $h = h'_0 * \ell'_k \mapsto b$, and $s(r') \vDash_m b$, or
2. exists $\ell'_k \in \text{Clocs}(r', m)$, $\text{Clocs}(r', m) \cap \text{dom}(h_2) = \emptyset$, $h = h'_0 * \widetilde{\ell} \mapsto b$, and $s(r') \vDash_m b$

Suppose (1) holds. The only rule that could have been used to prove Fact 91.1 is WF-HCONCRETE, by which we have

$$\ell' \neq \ell$$

so by Fact 91.5,

$$h_2 = h''_0 * \ell'_k \mapsto b$$

Since $r' \neq r$,

$$s_2(r') = s(r')$$

so

$$s_2(r') \vDash_m b$$

and (3a) is satisfied.

In the case where (2) initially holds, it is easy to show that (3b) holds after the store is updated. □

Lemma 92 (Relating Typed Values' Input and Output Heaps)**.**

$$\begin{aligned} &\textit{If } G,\ \emptyset,\ h \vdash_{m,\ I} v : \tau / h' \\ &\textit{and } \emptyset \vDash h, \\ &\textit{then } \emptyset \vdash h <: h' \\ &\textit{and } \emptyset \vDash h'. \end{aligned} \tag{92.1}$$

Proof. The proof proceeds by induction on the derivation of $G,\ \emptyset,\ h \vdash_{m,\ I} v : \tau / h'$. We split cases on the final rule used.

Case T-PURE

By the form of the rule, $h' = h$, so the desired conclusions follow from Fact 92.1 and Lemma 36.

Case T-SUB

By the form of the rule,

$$G, \emptyset, h \vdash_{m,\, I} v : \tau_1 / h_1 \tag{92.2}$$

$$\emptyset \vdash \tau_1 / h_1 <: \tau / h' \tag{92.3}$$

$$\emptyset \vDash \tau / h' \tag{92.4}$$

By the inductive hypothesis and Fact 92.2,

$$\emptyset \vdash h <: h_1 \tag{92.5}$$

Fact 92.3 can only be proved by <:-WORLD, from which we have

$$\emptyset \vdash h_1 <: h' \tag{92.6}$$

By Fact 92.5, Fact 92.6, and Lemma 35,

$$\emptyset \vdash h <: h'$$

The only rule that can be used to prove Fact 92.4 is WF-WORLD, from which we have

$$\Gamma \vDash h'$$

as required.

Case T-IF, T-LET, T-READ, T-SUPD, T-WUPD, T-UNFOLD, T-FOLD, T-CALL, T-MALLOC

Impossible, as the expression considered in each case is not a value. □

C.19 Preservation

Lemma 93 (Pure Expression Preservation)**.** *If* $\emptyset \vdash_m a : \tau$ *and* $a \hookrightarrow a'$*, then* $\emptyset \vdash_m a' : \tau$*.*

Proof. The proof proceeds by induction on the derivation of $\emptyset \vdash_m a : \tau$. We split cases on the final rule used.

Case T-VAR

Impossible.

Case T-INT, T-REF

Since a is a value, there is no a' such that $a \hookrightarrow a'$.

Case T-ARITH

By the form of the rule,

$$a \equiv v_1 \circ v_2$$
$$\emptyset \vdash_m v_1 : \texttt{int}(w, i_1) \tag{93.1}$$
$$\emptyset \vdash_m v_2 : \texttt{int}(w, i_2) \tag{93.2}$$
$$\tau = \{\nu : \texttt{int}(w, i_1 \mathbin{\widetilde{\circ}} i_2) \mid \nu = v_1 \circ v_2\} \tag{93.3}$$

By Fact 93.1 and Lemma 72,

$$v_1 = n_{1|w|}$$
$$n_1 \mathrel{\widetilde{\subseteq}} i_1 \tag{93.4}$$

By Fact 93.2 and Lemma 72,

$$v_2 = n_{2|w|}$$
$$n_2 \mathrel{\widetilde{\subseteq}} i_2 \tag{93.5}$$

The only evaluation rule that applies is E-ARITH, from which we have

$$a' \equiv (n_1 \circ n_2)_{|w|}$$

By T-INT,

$$\emptyset \vdash_m (n_1 \circ n_2)_{|w|} : \{\nu : \texttt{int}(w, n_1 \circ n_2) \mid \nu = (n_1 \circ n_2)_{|w|}\} \tag{93.6}$$

By Fact 93.4, Fact 93.5, and Proposition 3,

$$n_1 \circ n_2 \mathrel{\widetilde{\subseteq}} i_1 \mathbin{\widetilde{\circ}} i_2 \tag{93.7}$$

By Assumption 13,

$$\emptyset \vDash \nu = (n_1 \circ n_2)_{|w|} \Rightarrow \nu = n_{1|w|} \circ n_{2|w|} \tag{93.8}$$

By Fact 93.8, Fact 93.7, and <:-INT,

$$\emptyset \vdash \{\nu : \texttt{int}(w, n_1 \circ n_2) \mid \nu = (n_1 \circ n_2)_{|w|}\} <: \{\nu : \texttt{int}(w, i_1 \mathbin{\widetilde{\circ}} i_2) \mid \nu = n_{1|w|} \circ n_{2|w|}\} \tag{93.9}$$

By Fact 93.6, Fact 93.9, and T-PURESUB,

$$\emptyset \vdash_m (n_1 \circ n_2)_{|w|} : \{\nu : \texttt{int}(w, i_1 \mathbin{\widetilde{\circ}} i_2) \mid \nu = n_{1|w|} \circ n_{2|w|}\}$$

as required.

Case T-PTRARITH

By the form of the rule,

$$a \equiv v_1 +_p v_2$$
$$\emptyset \vdash_m v_1 : \texttt{ref}(\ell, i_1) \tag{93.10}$$
$$\emptyset \vdash_m v_2 : \texttt{int}(w, i_2) \tag{93.11}$$
$$\tau = \{\nu : \texttt{ref}(\ell, i_1 \mathbin{\widetilde{+}} i_2) \mid \nu = v_1 +_p v_2\} \tag{93.12}$$

By Fact 93.10 and Lemma 72,

$$v = 0_{|W|}$$

or

$$v = \text{bref}(r, n, z)$$
$$n \in [\![i_1]\!]$$
$$\exists \ell_j \sqsubseteq \ell.\ell_j \in \text{Clocs}(r, m)$$

We now split cases.

Case $v = 0_{|W|}$

Then the evaluation rule which applied must have been E-NULL-PLUS. Then

$$a' \equiv 0_{|W|}$$
$$v_2 = m_{|W|}$$
$$\ell = \widetilde{\ell}$$

By <:-NULL,

$$\emptyset \vdash \{\nu : \texttt{int}(W, 0) \mid \nu = 0_{|W|}\} <: \{\nu : \texttt{ref}(\widetilde{\ell}, i_1 \mathbin{\widetilde{\circ}} i_2) \mid \nu = 0_{|W|}\}$$

By Assumption 13,

$$\emptyset \vDash \nu = 0_{|W|} \Rightarrow \nu = 0_{|W|} +_p m_{|W|}$$

By the above and <:-REF,

$$\emptyset \vdash \{\nu : \texttt{ref}(\widetilde{\ell}, i_1 \mathbin{\widetilde{\circ}} i_2) \mid \nu = 0_{|W|}\} <: \{\nu : \texttt{ref}(\widetilde{\ell}, i_1 \mathbin{\widetilde{\circ}} i_2) \mid \nu = 0_{|W|} +_p m_{|W|}\}$$

We can tie these subtyping relations together with <:-TRANS:

$$\emptyset \vdash \{\nu : \texttt{int}(W, 0) \mid \nu = 0_{|W|}\} <: \{\nu : \texttt{ref}(\widetilde{\ell}, i_1 \mathbin{\widetilde{\circ}} i_2) \mid \nu = 0_{|W|} +_p m_{|W|}\}$$

By T-INT,

$$\emptyset \vdash_m 0_{|W|} : \{\nu : \texttt{int}(W, 0) \mid \nu = 0_{|W|}\}$$

By the above and T-PURESUB,

$$\emptyset \vdash_m 0_{|W|} : \{\nu : \texttt{ref}(\widetilde{\ell}, i_1 \mathbin{\widetilde{\circ}} i_2) \mid \nu = 0_{|W|} +_p m_{|W|}\}$$

as required.

Case $v = \text{bref}(r, n, z)$

We assume

$$\begin{gathered} n \in [\![i_1]\!] \\ \exists \ell_j \sqsubseteq \ell.\ell_j \in \text{Clocs}(r, m) \end{gathered} \tag{93.13}$$

By Fact 93.11 and Lemma 72,

$$\begin{aligned} v_2 &= m_{|w|} \\ m &\mathbin{\widetilde{\subseteq}} i_2 \end{aligned} \tag{93.14}$$

The evaluation rule which applied must have been E-INT-PLUS. From the form of the rule,

$$a' \equiv \text{bref}(r, n + m, z)$$

By T-REF and Fact 93.13,

$$\emptyset \vdash_m \text{bref}(r, n + m, z) : \{\nu : \texttt{ref}(\ell_j, n + m) \mid \nu = \text{bref}(r, n + m, z)\} \tag{93.15}$$

By $n \mathbin{\widetilde{\subseteq}} i_1$, Fact 93.14, and Proposition 3,

$$n + m \mathbin{\widetilde{\subseteq}} i_1 \mathbin{\widetilde{+}} i_2 \tag{93.16}$$

By Assumption 13,

$$\emptyset \vDash \nu = \text{bref}(r, n + m, z) \Rightarrow \nu = \text{bref}(r, m, z) + \text{bref}(r, m, z) \tag{93.17}$$

By Fact 93.17, Fact 93.16, and <:-REF,

$$\begin{aligned}\emptyset \vdash \{\nu : \texttt{ref}(\ell_j, n+m) \mid \nu = \text{bref}(r, n+m, z)\} <: \\ \{\nu : \texttt{ref}(\ell_j, i_1 \mathbin{\widetilde{+}} i_2) \mid \nu = \text{bref}(r,n,z) +_p \text{bref}(r,m,z)\}\end{aligned} \tag{93.18}$$

By Fact 93.15, Fact 93.18, and T-PURESUB,

$$\emptyset \vdash_m \text{bref}(r, n+m, z) : \{\nu : \texttt{ref}(\ell_j, i_1 \mathbin{\widetilde{+}} i_2) \mid \nu = \text{bref}(r,n,z) +_p \text{bref}(r,m,z)\} \tag{93.19}$$

If $\ell = \ell_j$, this completes the proof of this case. Otherwise, by Fact 93.13 and Definition 21, it must be that $\ell = \widetilde{\ell}$. By <:-ABSTRACT,

$$\begin{aligned}\emptyset \vdash \{\nu : \texttt{ref}(\ell_j, i_1 \mathbin{\widetilde{+}} i_2) \mid \nu = \text{bref}(r,n,z) +_p \text{bref}(r,m,z)\} <: \\ \{\nu : \texttt{ref}(\widetilde{\ell}, i_1 \mathbin{\widetilde{+}} i_2) \mid \nu = \text{bref}(r,n,z) +_p \text{bref}(r,m,z)\}\end{aligned} \tag{93.20}$$

By Fact 93.19, Fact 93.20, and T-PURESUB,

$$\emptyset \vdash_m \text{bref}(r, n+m, z) : \{\nu : \texttt{ref}(\widetilde{\ell}, i_1 \mathbin{\widetilde{+}} i_2) \mid \nu = \text{bref}(r,n,z) +_p \text{bref}(r,m,z)\}$$

as required.

Case T-RELATION

By the form of the rule,

$$\begin{aligned}a &\equiv v_1 \bowtie v_2 \\ \tau &= \{\nu : \texttt{int}(W, [0,\ 1]_1^0) \mid \text{if } v_1 \bowtie v_2 \text{ then } \nu = 1_{|W|} \text{ else } \nu = 0_{|W|}\}\end{aligned}$$

By the form of the expression, one of the evaluation rules E-REL-TRUE or E-REL-FALSE must have been used. We consider only the E-REL-TRUE case; the other is similar. Then

$$\begin{aligned}a' &\equiv 1_{|W|} \\ v_1 &\bowtie v_2\end{aligned} \tag{93.21}$$

By T-INT,

$$\emptyset \vdash_m a' : \{\nu : \texttt{int}(W, 1) \mid \nu = 1_{|W|}\} \tag{93.22}$$

By Fact 93.21 and Assumption 13, $v_1 \bowtie v_2$ is valid, so, by the laws of propositional logic,

$$\emptyset \vDash \nu = 1_{|W|} \Rightarrow \text{if } v_1 \bowtie v_2 \text{ then } \nu = 1_{|W|} \text{ else } \nu = 0_{|W|} \tag{93.23}$$

By Proposition 2,

$$1 \stackrel{\sim}{\subseteq} [0,\ 1]_1^0 \tag{93.24}$$

By Fact 93.23, Fact 93.24, and <:-INT,

$$\begin{aligned} &\emptyset \vdash \{\nu :\ \mathtt{int}(W, 1)\ \mid\ \nu = 1_{|W|}\} \\ &\quad <: \{\nu :\ \mathtt{int}(W, [0,\ 1]_1^0)\ \mid\ \text{if } v_1 \bowtie v_2 \text{ then } \nu = 1_{|W|} \text{ else } \nu = 0_{|W|}\} \end{aligned} \tag{93.25}$$

By Fact 93.22, Fact 93.25, and T-PURESUB,

$$\emptyset \vdash_m a' : \{\nu :\ \mathtt{int}(W, [0,\ 1]_1^0)\ \mid\ \text{if } v_1 \bowtie v_2 \text{ then } \nu = 1_{|W|} \text{ else } \nu = 0_{|W|}\} \tag{93.26}$$

as required.

Case T-PURESUB

By the form of the rule,

$$\emptyset \vdash_m a : \tau_1 \tag{93.27}$$

$$\emptyset \vdash \tau_1 <: \tau \tag{93.28}$$

$$\emptyset \vDash \tau \tag{93.29}$$

By the inductive hypothesis and Fact 93.27,

$$\emptyset \vdash_m a' : \tau_1 \tag{93.30}$$

By Fact 93.28, Fact 93.29, Fact 93.30, and T-PURESUB,

$$\emptyset \vdash_m a' : \tau$$

as required. □

Lemma 94 (Type Preservation)**.**

$$\begin{aligned}
&\textit{If } G,\ \emptyset,\ h \vdash_{m,\ I} e : \tau / h', \\
&\quad \emptyset \vDash h, && (94.1)\\
&\quad s \vDash_m h, && (94.2)\\
&\quad \vDash m, && (94.3)\\
&\quad \vDash G, && (94.4)\\
&\quad D \vDash G, && (94.5)\\
&\textit{and } e / s \hookrightarrow e' / s', \\
&\textit{then there exist } h_s,\ m' \textit{ such that} \\
&\quad G,\ \emptyset,\ h_s \vdash_{m',\ I} e' : \tau / h', \\
&\quad \emptyset \vDash h_s, \\
&\quad s' \vDash_{m'} h_s, \\
&\quad \vDash m', \\
&\quad m \subseteq m', \\
&\textit{and } \ \mathrm{dom}(m' \setminus m) \subseteq I.
\end{aligned}$$

Proof. The proof proceeds by induction on the derivation of $G,\ \emptyset,\ h \vdash_{m,\ I} e : \tau / h'$. We split cases on the final rule used. Unless noted otherwise, we let $m' = m$ and $h_s = h$.

Case T-PURE

Immediate from Lemma 93.

Case T-SUB

By the form of the rule,

$$\begin{aligned}
G,\ \emptyset,\ h \vdash_{m,\ I} e : \tau_1 / h_1 && (94.6)\\
\emptyset \vdash \tau_1 / h_1 <: \tau / h && (94.7)\\
\emptyset \vDash \tau / h && (94.8)
\end{aligned}$$

By the initial assumptions, Fact 94.6, and the inductive hypothesis, there exist h_s and m' such that

$$\begin{aligned}
G,\ \emptyset,\ h_s \vdash_{m',\ I} e' : \tau_1 / h_1 && (94.9)\\
\emptyset \vDash h_s \\
s' \vDash_{m'} h_s \\
\vDash m' \\
m \subseteq m' \\
\mathrm{dom}(m' \setminus m) \subseteq I
\end{aligned}$$

Most obligations follow immediately. By Fact 94.9, Fact 94.7, Fact 94.8, and T-SUB,

$$G,\ \emptyset,\ h_s \vdash_{m',\ I} e' : \tau / h,$$

as required.

Case T-IF

By the form of the rule,

$$e \equiv \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2$$

$$G,\ v \neq 0,\ h \vdash_{m,\ I_1} e_1 : \tau / h' \tag{94.10}$$

$$G,\ v = 0,\ h \vdash_{m,\ I_2} e_2 : \tau / h' \tag{94.11}$$

The only evaluation rules that could have applied are E-IF-TRUE and E-IF-FALSE. We show only the E-IF-TRUE case; the case for E-IF-FALSE is similar.

By the form of E-IF-TRUE, we have

$$v = 1_{|W|}$$
$$e' \equiv e_1$$
$$s' = s$$

By Assumption 13,

$$1_{|W|} \neq 0$$

is valid, so, by WFSUBST-PRED,

$$v \neq 0 \vDash_m \cdot$$

By the above, Fact 94.10, and Lemma 58, then,

$$G,\ \emptyset,\ h \vdash_{m,\ I} e_1 : \tau / h'$$

as required.

Case T-LET

By the form of the rule,

$$e \equiv \textbf{let } x = e_1 \textbf{ in } e_2$$

$$I = I_1 \cup I_2 \tag{94.12}$$

$$I_1 \cap I_2 = \emptyset \tag{94.13}$$

$$G,\ \emptyset,\ h \vdash_{m,\ I_1} e_1 : \tau_1 / h_1 \tag{94.14}$$

$$G,\ x : \tau_1,\ h_1 \vdash_{m,\ I_2} e_2 : \tau / h' \tag{94.15}$$

$$\emptyset \vDash \tau / h' \tag{94.16}$$

We split cases on whether e_1 is a value.

Case e_1 is not a value

Then the only evaluation rule that applies is E-SEQ, so that we have:

$$\begin{aligned} e_1/s &\hookrightarrow e_1'/s' \\ e' &\equiv \textbf{let } x = e_1' \textbf{ in } e_2 \end{aligned}$$

By the inductive hypothesis and Fact 94.14,

$$G,\ \emptyset,\ h_s \vdash_{m',\ I_1} e_1' : \tau_1/h_1 \tag{94.17}$$

for some h_s and m' such that

$$\begin{aligned} \emptyset &\vDash h_s \\ s' &\vDash_{m'} h_s \\ m &\subseteq m' \\ &\vDash m' \\ \mathrm{dom}(m' \setminus m) &\subseteq I_1. \end{aligned} \tag{94.18}$$

Most obligations follow immediately. By Fact 94.13, Fact 94.15, Fact 94.18, and Lemma 22,

$$G,\ x : \tau_1,\ h_1 \vdash_{m',\ I_2} e_2 : \tau/h' \tag{94.19}$$

By Fact 94.12, Fact 94.17, Fact 94.19, Fact 94.16, and T-LET,

$$G,\ \emptyset,\ h_s \vdash_{m',\ I} \textbf{let } x = e_1' \textbf{ in } e_2 : \tau/h$$

as required.

Case $e_1 \equiv v$ for some value v

The only evaluation rule that applies is E-LET. From the form of the rule, we have:

$$\begin{aligned} e' &\equiv e_2[x \mapsto v] \\ s' &= s \end{aligned}$$

The only rule that could have been used to prove Fact 94.14 is T-PURE, from which we have

$$\emptyset \vdash v : \tau_1$$

By WF-VAR, then,

$$\emptyset \vDash_m [x \mapsto v] \tag{94.20}$$

By Fact 94.20, Fact 94.15, and Lemma 58,

$$G,\ \emptyset,\ h_1[x \mapsto v] \vdash_{m,\ I_2} e_2[x \mapsto v] : \tau[x \mapsto v] / h'[x \mapsto v]$$

By Fact 94.16 and Lemma 28,

$$\mathrm{FV}(\tau) = \emptyset$$
$$\mathrm{FV}(h') = \emptyset$$

So the above is equivalent to

$$G,\ \emptyset,\ h_1[x \mapsto v] \vdash_{m,\ I_2} e_2[x \mapsto v] : \tau / h' \tag{94.21}$$

By Fact 94.14 and Lemma 92,

$$\emptyset \vdash h <: h_1 \tag{94.22}$$
$$\emptyset \vDash h_1 \tag{94.23}$$

By Fact 94.23 and Lemma 28,

$$\mathrm{FV}(h_1) = \emptyset \tag{94.24}$$

so

$$h_1[x \mapsto v] = h_1 \tag{94.25}$$

so we have

$$G,\ \emptyset,\ h_1 \vdash_{m,\ I_2} e_2[x \mapsto v] : \tau / h' \tag{94.26}$$

By Fact 94.12 and Lemma 19,

$$G,\ \emptyset,\ h_1 \vdash_{m,\ I} e_2[x \mapsto v] : \tau / h'$$

which settles type preservation. By Fact 94.22, Fact 94.2, and Lemma 78,

$$s \vDash_m h_1$$

Thus, we let

$$h_s = h_1,$$

which satisfies the heap well-formedness and modeling requirements.

Case T-READ

By the form of the rule,

$$
\begin{aligned}
e &\equiv *_n v \\
h &= h_1 * \ell_j \mapsto \ldots,\ i : \tau,\ \ldots && (94.27)\\
h' &= h \\
\varnothing &\vdash_m v : \{\nu : \texttt{ref}(\ell_j, i) \mid \mathrm{Safe}(\nu, n)\} && (94.28)
\end{aligned}
$$

The only evaluation rule that applies is E-READ, from which we have

$$
\begin{aligned}
v &= \mathrm{bref}(r, n, z) \\
e' &\equiv s(r)(n) \\
s' &= s
\end{aligned}
$$

for some r, n, and z. By Fact 94.28, Fact 94.27, Fact 94.1, Fact 94.2, and Fact 94.3,

$$\varnothing \vdash_m s(r)(n) : \tau$$

By the above and T-PURE,

$$G,\ \varnothing,\ h \vdash_{m,}\ s(r)(n) : \tau / h$$

as required.

Case T-SUPD

By the form of the rule,

$$
\begin{aligned}
e &\equiv *v_1 := v_2 \\
\tau &= \texttt{void} \\
\varnothing &\vdash_m v_1 : \{\nu : \texttt{ref}(\ell_j, n) \mid \mathrm{Safe}(\nu, \mathrm{SizeOf}(\tau_2))\} && (94.29)\\
\varnothing &\vdash_m v_2 : \tau_2 && (94.30)\\
\mathrm{SizeOf}(\tau_2) &= \mathrm{SizeOf}(\tau_1) && (94.31)\\
h &= h_1 * \ell_j \mapsto b_1,\ n : \tau_1,\ b_2 && (94.32)\\
h' &= h_1 * \ell_j \mapsto b_1,\ n : \tau_2,\ b_2
\end{aligned}
$$

The only evaluation rule that applies is E-WRITE, from which we have

$$\begin{aligned} v_1 &= \text{bref}(r, k, z) \\ e' &\equiv 0_{|0|} \\ r &\in \text{dom}(s) \\ s(r) &= c \\ s' &= s[r \mapsto \text{Write}(c, k, v_2)] \end{aligned} \tag{94.33}$$

for some r, k, and z. By Fact 94.29 and Lemma 72,

$$\begin{aligned} k &= n \\ \ell_j &\in \text{Clocs}(r, m) \end{aligned} \tag{94.34}$$

By Fact 94.30 and Lemma 39,

$$\emptyset \vDash \tau_2 \tag{94.35}$$

By Fact 94.1, Fact 94.2, Fact 94.3, Fact 94.32, Fact 94.33, Fact 94.34, and Lemma 83,

$$c \vDash_m b_1,\ n : \tau_1,\ b_2 \tag{94.36}$$

The only rule that could have been used to prove Fact 94.1 is WF-HCONCRETE, by which we have

$$\emptyset \vDash b_1,\ n : \tau_1,\ b_2 \tag{94.37}$$

$$\emptyset \vDash h_1 \tag{94.38}$$

By Fact 94.36, Fact 94.37, Fact 94.30, Fact 94.35, Fact 94.31, and Lemma 89,

$$\text{Write}(c, n, v_2) \vDash_m b_1,\ n : \tau_2,\ b_2 \tag{94.39}$$

We let

$$h_s = h' \tag{94.40}$$

By Fact 94.1, Fact 94.2, Fact 94.34, Fact 94.39, Fact 94.40, and Lemma 91,

$$s' \vDash_m h_s$$

as required. By Fact 94.37, Fact 94.35, Fact 94.31, and Lemma 90,

$$\emptyset \vDash b_1,\ n : \tau_2,\ b_2 \tag{94.41}$$

By Fact 94.38, Fact 94.41, Fact 94.40, and WF-HConcrete,

$$\emptyset \vDash h_s$$

as required. By T-Int, <:-Int, T-PureSub, and T-Pure,

$$G,\ \emptyset,\ h_s \vdash_{I,\,m} 0_{|0|} : \texttt{void}/h_s$$

as required.

Case T-WUpd

By the form of the rule,

$$\begin{aligned}
&e \equiv *v_1 := v_2 \\
&\tau = \texttt{void} \\
&\emptyset \vdash_m v_1 : \{\nu : \texttt{ref}(\ell_j, n^{+k}) \mid \text{Safe}(\nu, \text{SizeOf}(\tau))\} && (94.42)\\
&\emptyset \vdash_m v_2 : \tau && (94.43)\\
&h = h_1 * \ell_j \mapsto \ldots, n^{+k} : \tau, \ldots && (94.44)\\
&h' = h
\end{aligned}$$

The only evaluation rule that applies is E-Write, from which we have

$$\begin{aligned}
&v_1 = \text{bref}(r, q, z) \\
&e' \equiv 0_{|0|} \\
&s' = s[r \mapsto \text{Write}(s(r), q, v_2)] \\
&r \in \text{dom}(s) && (94.45)
\end{aligned}$$

for some r, q, z. By Fact 94.42 and Lemma 72,

$$q \in [\![n^{+k}]\!] \tag{94.46}$$

$$\ell_j \in \text{Clocs}(r, m) \tag{94.47}$$

By Fact 94.1, Fact 94.2, Fact 94.3, Fact 94.44, Fact 94.45, Fact 94.47, and Lemma 83,

$$s(r) \vDash_m \ldots,\ n^{+k} : \tau,\ \ldots \tag{94.48}$$

The only rule that could have been used to prove Fact 94.1 is WF-HConcrete, by which we have

$$\emptyset \vDash \ldots,\ i : \tau,\ \ldots \tag{94.49}$$

By Fact 94.48, Fact 94.49, Fact 94.46, Fact 94.43, and Lemma 87,

$$\text{Write}(s(r), q, v_2) \vDash_m \ldots,\ i : \tau,\ \ldots \tag{94.50}$$

We let

$$h_s = h \tag{94.51}$$

By Fact 94.1, Fact 94.2, Fact 94.47, Fact 94.50, Fact 94.51, and Lemma 91,

$$s' \vDash_m h_s$$

as required. By Fact 94.1 and Fact 94.51,

$$\emptyset \vDash h_s$$

as required. Finally, by T-INT, <:-INT, T-PURESUB, and T-PURE,

$$G,\ \emptyset,\ h_s \vdash_{I,\ m} 0_{|0|} : \texttt{void}/h_s$$

as required.

Case T-UNFOLD

By the form of the rule,

$$e \equiv \textbf{letu}\ x = \textbf{unfold}\ v\ \textbf{in}\ e$$

$$I = I_1 \cup \{\ell_j\} \tag{94.52}$$

$$\emptyset \vdash_m v : \{\nu : \texttt{ref}(\widetilde{\ell}, i_y) \mid \nu \neq 0\} \tag{94.53}$$

$$h = h_0 * \widetilde{\ell} \mapsto \overline{n_k} : \overline{\tau_k}, \overline{i^+} : \overline{\tau^+} \tag{94.54}$$

$$\overline{x_k} \text{ disjoint} \tag{94.55}$$

$$\overline{x_k} \notin e,\ \text{FV}(h) \tag{94.56}$$

$$\theta = [\overline{@n_k} \mapsto \overline{x_k}] \tag{94.57}$$

$$\Gamma_1 = \overline{x_k} : \overline{\theta\tau_k} \tag{94.58}$$

$$\ell_j \notin h, m \tag{94.59}$$

$$h_1 = h * \ell_j \mapsto \overline{n_k} : \overline{\{\nu = x_k\}}, \overline{i^+} : \overline{\theta\tau^+} \tag{94.60}$$

$$G,\ \Gamma_1; x : \{\nu : \texttt{ref}(\ell_j, i_y) \mid \nu = v\},\ h_1 \vdash_{m,\ I_1} e : \tau_2/h_2 \tag{94.61}$$

$$\Gamma_1 \vDash h_1 \tag{94.62}$$

$$\emptyset \vDash \tau_2/h_2 \tag{94.63}$$

The only evaluation rule which applies is E-UNFOLD, from which we have:

$$e' \equiv e[x \mapsto v]$$
$$s' = s$$

By Fact 94.53 and Lemma 72, either

$$v = 0_{|W|} \tag{94.64}$$

or

$$v = \text{bref}(r, n, z) \tag{94.65}$$
$$n \in [\![i_y]\!] \tag{94.66}$$
$$\text{exists } \ell_k \sqsubseteq \widetilde{\ell} \in \text{Clocs}(r, m) \tag{94.67}$$

By Fact 94.53 and Lemma 27,

$$v \neq 0_{|W|}$$

Thus, Fact 94.64 does not hold, and Fact 94.65, Fact 94.66, and Fact 94.67 do. By Fact 94.53 and Lemma 25,

$$r \in \text{rng}(m)$$

By Definition 23, this implies

$$r \in \text{dom}(s)$$

So there exists a run-time block c such that

$$s(r) = c$$

By Fact 94.62, Fact 94.2, Fact 94.3, Fact 94.67, Fact 94.54 and Lemma 82,

$$c \vDash_m \overline{n_k} : \overline{\tau_k}, \overline{i^+} : \overline{\tau^+} \tag{94.68}$$

By Fact 94.68, Fact 94.55, and Lemma 79,

$$\Gamma_1 \vDash_m \theta_h$$
$$\text{where } \theta_h = [\overline{x_k} \mapsto \overline{c(n_k)}] \tag{94.69}$$

By Fact 94.69, Fact 94.61, and Lemma 58,

$$G,\ x : \{\nu :\ \texttt{ref}(\ell_j, i_y)\ \mid\ \nu = v\},\ \theta_h h_1 \vdash_{m,\ I_1} \theta_h e : \theta_h \tau_2 / \theta_h h_2$$

By Fact 94.56, this is equivalent to

$$G,\ x : \{\nu :\ \texttt{ref}(\ell_j, i_y)\ \mid\ \nu = v\},\ \theta_h h_1 \vdash_{m,\ I_1} e : \theta_h \tau_2 / \theta_h h_2 \tag{94.70}$$

By Fact 94.63 and Lemma 28,

$$\mathrm{FV}(\tau_2 / h_2) = \emptyset \tag{94.71}$$

By Fact 94.70 and Fact 94.71,

$$G,\ x : \{\nu :\ \texttt{ref}(\ell_j, i_y)\ \mid\ \nu = v\},\ \theta_h h_1 \vdash_{m,\ I_1} e : \tau_2 / h_2 \tag{94.72}$$

We now show location map inclusion, then conclude type preservation. We let

$$m' = m[\ell_j \mapsto r] \tag{94.73}$$

Note that, by Fact 94.52,

$$\mathrm{dom}(m' \setminus m) \subseteq I$$

as required. By Fact 94.73 and Fact 94.59,

$$m \subseteq m' \tag{94.74}$$

which settles location map inclusion. By Fact 94.74, Fact 94.72, Fact 94.52, and Lemma 22,

$$G,\ x : \{\nu :\ \texttt{ref}(\ell_j, i_y)\ \mid\ \nu = v\},\ \theta_h h_1 \vdash_{m',\ I_1} e : \tau_2 / h_2 \tag{94.75}$$

By Fact 94.73, Fact 94.66, Proposition 2, Definition 19, T-REF, and <:-REF,

$$\emptyset \vdash_{m'} \mathrm{bref}(r, n, z) : \{\nu :\ \texttt{ref}(\ell_j, i_y)\ \mid\ \nu = \mathrm{bref}(r, n, z)\} \tag{94.76}$$

Define θ_x by

$$\theta_x = [x \mapsto \mathrm{bref}(r, n, z)]$$

By Fact 94.76 and WFSUBST-VAR,

$$x : \{\nu :\ \texttt{ref}(\ell_j, i_y)\ \mid\ \nu = \mathrm{bref}(r, n, z)\} \vDash_{m'} \theta_x \tag{94.77}$$

By Fact 94.75, Fact 94.77, and Lemma 58,

$$G,\ \emptyset,\ \theta_x\theta_h h_1 \vdash_{m',\ I_1} \theta_x e : \theta_x\tau_2 / \theta_x h_2$$

By Fact 94.71, this implies

$$G,\ \emptyset,\ \theta_x\theta_h h_1 \vdash_{m',\ I_1} \theta_x e : \tau_2 / h_2$$

We assume that

$$x \notin \mathrm{FV}(h_1)$$

So we have

$$G,\ \emptyset,\ \theta_h h_1 \vdash_{m',\ I_1} \theta_x e : \tau_2 / h_2 \tag{94.78}$$

By Fact 94.78, Fact 94.52, and Lemma 19,

$$G,\ \emptyset,\ \theta_h h_1 \vdash_{m',\ I} \theta_x e : \tau_2 / h_2$$

which settles type preservation. We now show that the new input heap

$$\begin{aligned} h_s &= \theta_h h_1 \\ &= \theta_h h * \ell_j \mapsto \overline{n_k} : \overline{\{\nu = \theta_h x_k\}}, \overline{i^+} : \overline{\theta_h \theta \tau^+} \end{aligned}$$

is well-formed. By Fact 94.56,

$$h_s = h * \ell_j \mapsto \overline{n_k} : \overline{\{\nu = \theta_h x_k\}}, \overline{i^+} : \overline{\theta_h \theta \tau^+} \tag{94.79}$$

By WF-HCONCRETE, we must show

$$\emptyset \vDash h \tag{94.80}$$

$$\widetilde{\ell} \in \mathrm{dom}(h) \tag{94.81}$$

$$\ell_k \notin \mathrm{dom}(h) \tag{94.82}$$

$$\emptyset \vDash \overline{n_k} : \overline{\{\nu = \theta_h x_k\}},\ \overline{i^+} : \overline{\theta_h \theta \tau^+} \tag{94.83}$$

Note that Fact 94.80 was assumed, while Fact 94.81 follows from Fact 94.54. The only rule that could be used to prove Fact 94.62 is WF-HCONCRETE, from which we have

$$\ell_k \notin \mathrm{dom}(h) \text{ for any } k$$

We now show Fact 94.83. By Fact 94.54, the only rule that could have been used to show Fact 94.80 is WF-HABSTRACT, from which we have

$$\emptyset \vDash_{@} \overline{n_k} : \overline{\tau_k}, \overline{i^+} : \overline{\tau^+} \tag{94.84}$$

By Fact 94.58, Fact 94.84, Fact 94.55, Fact 94.56, Fact 94.57, and Lemma 49,

$$\Gamma_1 \vDash \overline{n_k} : \overline{\{\nu = x_k\}},\ \overline{i^+} : \overline{\theta\tau^+}$$

By the above, Fact 94.69, and Lemma 53,

$$\emptyset \vDash \overline{n_k} : \overline{\{\nu = \theta_h x_k\}},\ \overline{i^+} : \overline{\theta_h\theta\tau^+}$$

as required. By Fact 94.2, Fact 94.68, Fact 94.59, Fact 94.73, Fact 94.79, and Lemma 76,

$$s \vDash_{m'} h_s$$

as required. Finally, we show that m' is well-formed. Suppose

$$\ell'_k \in \text{Clocs}(r, m)$$

By Fact 94.53 and Lemma 80,

$$\ell' = \ell$$

So by Fact 94.73 and Definition 20,

$$\vDash m'$$

as required.

Case T-FOLD

By the form of the rule,

$$\begin{aligned} h &= h_0 * \widetilde{\ell} \mapsto b_1 * \ell_j \mapsto b_2 \\ \emptyset &\vdash b_2 <: b_1 \\ \tau &= \texttt{void} \\ h' &= h_0 * \widetilde{\ell} \mapsto b_1 \end{aligned} \tag{94.85}$$

The only evaluation rule that applies is E-FOLD, from which we have

$$\begin{aligned} e' &\equiv 0_{|0|} \\ s' &= s \end{aligned}$$

We define the output heap h_s as

$$\begin{aligned} h_s &= h' \\ &= h_0 * \widetilde{\ell} \mapsto b_1 \end{aligned}$$

By Fact 94.1, Fact 94.2, Fact 94.3, Fact 94.85, and Lemma 81,

$$s \vDash_m h_s$$

The only rule that could have been used to show Fact 94.1 is WF-HCONCRETE, from which we have

$$\emptyset \vDash h_s$$

By T-INT, T-PURESUB, and T-PURE,

$$G,\ \emptyset,\ h_s \vdash_m,\ 0_{|0|} : \texttt{void}/h'$$

Thus, all the obligations are satisfied.

Case T-CALL

By the form of the rule

$$\begin{aligned} e &\equiv f(\overline{v_j})[\overline{\ell_f} \mapsto \overline{\ell}] \\ h &= h_u * h_m && (94.86) \\ \emptyset &\vDash h_m && (94.87) \\ \emptyset &\vDash h_u && (94.88) \\ G(f) &= (\overline{x_j} : \overline{\tau_j})/h_f \ \rightarrow\ \tau'/h'_f && (94.89) \\ \theta &= [\overline{x_j} \mapsto \overline{v_j}] && (94.90) \\ \rho &= [\overline{\ell_f} \mapsto \overline{\ell}] && (94.91) \\ \emptyset &\vDash h_u * \theta\rho h_f && (94.92) \\ \emptyset &\vdash_m \overline{v_j} : \overline{\theta\rho\tau_j} && (94.93) \\ \emptyset &\vdash h_m <: \theta\rho h_f && (94.94) \\ \tau &= \theta\rho\tau' \\ h &= h_u * \theta\rho h'_f \end{aligned}$$

By Fact 94.5 and Definition 26,

$$D(f) = e_f \qquad (94.95)$$

The only evaluation rule which applies is E-CALL, from which we have

$$e' \equiv \theta\rho e_f \tag{94.96}$$

$$s' = s \tag{94.97}$$

By Fact 94.87, Fact 94.94, and Lemma 61,

$$\left|\text{Locs}(\rho h_f)\right| = \left|\text{Locs}(h_f)\right| \tag{94.98}$$

By Fact 94.89, Fact 94.5, Fact 94.95, and Definition 26,

$$G,\ \overline{x_j} : \overline{\tau_j},\ h_f \vdash_{\emptyset,\ I_f} e_f : \tau' / h'_f \tag{94.99}$$

By Fact 94.99, Fact 94.98, Fact 94.4, and Lemma 66,

$$G,\ \overline{x_j} : \overline{\rho\tau_j},\ \rho h_f \vdash_{\emptyset,\ \rho I_f} \rho e_f : \rho\tau' / \rho h'_f \tag{94.100}$$

By Fact 94.93 and repeated applications of WFSUBST-VAR,

$$\overline{x_j} : \overline{\theta\rho\tau_j} \vDash_m \theta \tag{94.101}$$

By Fact 94.101, Fact 94.99, Fact 94.4, and Lemma 58,

$$G,\ \emptyset,\ \theta\rho h_f \vdash_{\emptyset,\ \rho I_f} \theta\rho e_f : \theta\rho\tau' / \theta\rho h'_f \tag{94.102}$$

Since I and I_f are both countable, there exists a substitution ω such that

$$\omega\rho I_f = I$$

By Fact 94.107, and Lemma 18,

$$G,\ \emptyset,\ \omega\rho\theta h_f \vdash_{\emptyset,\ I} \theta\rho e_f : \omega\theta\rho\tau' / \omega\theta\rho h'_f \tag{94.103}$$

By Fact 94.4 and Definition 25,

$$\vDash (\overline{x_j} : \overline{\tau_j}) / h_f \ \rightarrow\ \tau' / h'_f \tag{94.104}$$

The only rule that could be used to prove Fact 94.104 is WF-FUNSCHEME, from which we have

$$\overline{\tau_j}, h_f, \tau', h'_f \text{ abstract} \tag{94.105}$$

$$\overline{x_j} : \overline{\tau_j} \vDash h_f \tag{94.106}$$

By Fact 94.103 and Fact 94.105,

$$G,\ \emptyset,\ \rho\theta h_f \vdash_{\emptyset,\ I} \theta\rho e_f : \theta\rho\tau' / \theta\rho h'_f \tag{94.107}$$

Let

$$I \setminus \{\ell_j \mid \ell_j \in h_u\} = \omega_u I \tag{94.108}$$

Since I is countable, such a substitution exists. By Fact 94.107, and Lemma 18,

$$G,\ \emptyset,\ \omega_u\theta\rho h_f \vdash_{\emptyset,\ \omega_u I} \theta\rho e_f : \omega_u\theta\rho\tau' / \omega_u\theta\rho h'_f \tag{94.109}$$

By Fact 94.92 and Lemma 47,

$$\mathrm{dom}(h_u) \cap \mathrm{dom}(\theta\rho h_f) = \emptyset \tag{94.110}$$

By Fact 94.109, Fact 94.88, Fact 94.117, Fact 94.108, Fact 94.110, and Lemma 69,

$$G,\ \emptyset,\ h_u * \omega_u\theta\rho h_f \vdash_{\emptyset,\ \omega_u I} \theta\rho e_f : \omega_u\theta\rho\tau' / h_u * \omega_u\theta\rho h'_f \tag{94.111}$$

By Fact 94.111 and Fact 94.105,

$$G,\ \emptyset,\ h_u * \theta\rho h_f \vdash_{\emptyset,\ \omega_u I} \theta\rho e_f : \theta\rho\tau' / h_u * \theta\rho h'_f \tag{94.112}$$

By Fact 94.112, Fact 94.108, and Lemma 19,

$$G,\ \emptyset,\ h_u * \theta\rho h_f \vdash_{\emptyset,\ I} \theta\rho e_f : \theta\rho\tau' / h_u * \theta\rho h'_f \tag{94.113}$$

By Fact 94.113, Fact 94.105, and Lemma 23,

$$G,\ \emptyset,\ h_u * \theta\rho h_f \vdash_{m,\ I} \theta\rho e_f : \theta\rho\tau' / h_u * \theta\rho h'_f \tag{94.114}$$

which settles type preservation. Let

$$h_s = h_u * \rho\theta h_f \tag{94.115}$$

By Fact 94.98, Fact 94.106, and Lemma 63,

$$\overline{x_j} : \overline{\rho\tau_j} \vDash \rho h_f \tag{94.116}$$

By Fact 94.101, Fact 94.116, and Lemma 53,

$$\emptyset \vDash \theta\rho h_f \tag{94.117}$$

By Fact 94.1, Fact 94.86, Fact 94.117, Fact 94.94, and Lemma 48,

$$\emptyset \vDash h_u * \theta\rho h_f$$

which settles heap well-formedness. By Fact 94.94 and Lemma 38,

$$\emptyset \vdash h_u * h_m <: h_u * \theta\rho h_f \tag{94.118}$$

By Fact 94.2, Fact 94.118, and Lemma 78,

$$s \vDash_m h_u * \theta\rho h_f \tag{94.119}$$

as required.

Case T-MALLOC

By the form of the rule,

$$h = h_0 * \widetilde{\ell} \mapsto b$$
$$h' = h * \ell_j \mapsto b^0 \tag{94.120}$$
$$\tau = \{v : \texttt{ref}(\ell_j, 0) \mid \text{Allocated}(v, v)\}$$
$$I = I_1 \cup \{\ell_j\} \tag{94.121}$$
$$\ell_j \notin h, m \tag{94.122}$$
$$\emptyset \vDash h * \ell_j \mapsto b \tag{94.123}$$
$$\emptyset \vdash_m v : \{v : \texttt{int}(W, i) \mid v \geq 0\} \tag{94.124}$$

The only evaluation rule that applies is E-MALLOC, from which we have

$$v = z_{|W|}$$
$$e' \equiv \text{bref}(r, 0, z) \tag{94.125}$$
$$s' = s[r \mapsto (\mathbb{Z} \mapsto 0_{|1|})] \tag{94.126}$$
$$r \notin \text{dom}(s) \tag{94.127}$$

Let

$$m' = m[\ell_j \mapsto r] \tag{94.128}$$

By Fact 94.122 and the above,

$$m \subseteq m',$$

as required, while, by Fact 94.121,

$$\mathrm{dom}(m' \setminus m) \subseteq I,$$

as required. Let

$$h_s = h' \tag{94.129}$$

The well-formedness of h_s follows by Fact 94.123. By Fact 94.128, Definition 19, and T-REF,

$$\emptyset \vdash_{m'} \mathrm{bref}(r, 0, z) : \{\nu : \mathtt{ref}(\ell_j, 0) \mid \nu = \mathrm{bref}(r, 0, z)\} \tag{94.130}$$

By Fact 94.124 and Lemma 27,

$$z \geq 0 \tag{94.131}$$

By Fact 94.131, Assumption 13 and Assumption 14,

$$\emptyset \vDash \nu = \mathrm{bref}(r, 0, z) \Rightarrow \mathrm{Allocated}(\nu, z) \tag{94.132}$$

From Definition 16 and Definition 15,

$$\mathrm{Allocated}(\nu, z) \text{ well-sorted in } \nu : \mathtt{ref}(\ell_j, 0) \tag{94.133}$$

From Fact 94.133 and WF-TYPE,

$$\emptyset \vDash \{\nu : \mathtt{ref}(\ell_j, 0) \mid \mathrm{Allocated}(\nu, z)\} \tag{94.134}$$

By Fact 94.130, Fact 94.132, Fact 94.134, and T-PURESUB,

$$\emptyset \vdash_{m'} \mathrm{bref}(r, 0, z) : \{\nu : \mathtt{ref}(\ell_j, 0) \mid \mathrm{Allocated}(\nu, z)\} \tag{94.135}$$

By Fact 94.135, Fact 94.129, and T-PURE,

$$G,\ \emptyset,\ h_s \vdash_{m',}\ \mathrm{bref}(r, 0, z) : \{\nu : \mathtt{ref}(\ell_j, 0) \mid \mathrm{Allocated}(\nu, z)\} / h'$$

which settles type preservation. By Fact 94.126 and Lemma 73,

$$s'(r) \vDash_m b^0 \tag{94.136}$$

By Fact 94.2, Fact 94.136, Fact 94.122, Fact 94.128, Fact 94.120, Fact 94.126, and Lemma 74,

$$s' \vDash_{m'} h_s$$

By Fact 94.2 and Definition 23,

$$\mathrm{rng}(m) \subseteq \mathrm{dom}(s) \tag{94.137}$$

By Fact 94.127 and Fact 94.137,

$$r \notin \mathrm{rng}(m) \tag{94.138}$$

By Fact 94.138, Fact 94.128, and Definition 20,

$$\vDash m'$$

as required. □

C.20 Progress

Lemma 95 (Pure Expression Progress)**.** *If $\emptyset \vdash_m a : \tau$ and a is not a value, then there exists a' such that $a \hookrightarrow a'$.*

Proof. The proof proceeds by induction on the derivation of $\emptyset \vdash_m a : \tau$. We split cases on the final rule used.

Case T-VAR

Impossible.

Case T-INT, T-REF

Impossible, as a is a value.

Case T-ARITH

By the form of the rule,

$$a \equiv v_1 \circ v_2$$
$$\emptyset \vdash_m v_1 : \mathtt{int}(w, i_1) \tag{95.1}$$
$$\emptyset \vdash_m v_2 : \mathtt{int}(w, i_2) \tag{95.2}$$

By Lemma 72 and Fact 95.1,

$$v_1 = n_{1_{|w|}} \tag{95.3}$$

for some n_1. By Lemma 72 and Fact 95.2,

$$v_2 = n_{2_{|w|}} \tag{95.4}$$

for some n_2. Rule E-ARITH applies:

$$n_{1|w|} \circ n_{2|w|} \hookrightarrow (n_1 \circ n_2)_{|w|}$$

as required.

Case T-PTRARITH

By the form of the rule,

$$a \equiv v_1 +_p v_2$$
$$\emptyset \vdash_m v_1 : \texttt{ref}(\ell, i_1) \tag{95.5}$$
$$\emptyset \vdash_m v_2 : \texttt{int}(W, i_2) \tag{95.6}$$

By Lemma 72 and Fact 95.6,

$$v_2 = n_{2|W|} \tag{95.7}$$

for some n_2. By Lemma 72 and Fact 95.5, either

$$v_1 = 0_{|W|} \tag{95.8}$$

or

$$v_1 = \text{bref}(r, n_1, z_1) \tag{95.9}$$

for some n_1 and z_1. In the former case, by Fact 95.7, Fact 95.8, and E-NULL-PLUS, we have

$$v_1 +_p v_2 \hookrightarrow 0_{|W|}$$

In the latter case, by Fact 95.7 and Fact 95.9, and E-PTR-PLUS, we have

$$v_1 +_p v_2 \hookrightarrow \text{bref}(r, n_1 + n_2, z_1)$$

as required.

Case T-RELATION

By the form of the rule,

$$a \equiv v_1 \bowtie v_2$$
$$\emptyset \vdash_m v_1 : \tau_1 \tag{95.10}$$
$$\emptyset \vdash_m v_2 : \tau_2 \tag{95.11}$$

By Lemma 72, Fact 95.10, and Fact 95.11, there are four cases:

$$v_1 = n_{1_{|w|}},$$
$$v_2 = n_{2_{|w|}}$$

$$v_1 = \text{bref}(r, n_1, z_1),$$
$$v_2 = 0_{|W|}$$

$$v_1 = 0_{|W|},$$
$$v_2 = \text{bref}(r, n_2, z_2)$$

$$v_1 = \text{bref}(r, n_1, z_1),$$
$$v_2 = \text{bref}(r, n_2, z_2)$$

In each of the above cases, it follows from Definition 5 that

$$\text{Comparable}(v_1, v_2) \tag{95.12}$$

Then either rule E-REL-TRUE or E-REL-FALSE applies, i.e.,

$$v_1 \bowtie v_2 \hookrightarrow 1_{|W|}$$

or

$$v_1 \bowtie v_2 \hookrightarrow 0_{|W|}$$

as required.

Case T-PURESUB

Follows immediately from the inductive hypothesis. □

Lemma 96 (Progress)**.**

$$\textit{If } G,\ \emptyset,\ h \vdash_{m,\ I} e : \tau / h',$$
$$s \vDash_m h, \tag{96.1}$$
$$\emptyset \vDash h, \tag{96.2}$$
$$\vDash m, \tag{96.3}$$
$$\textit{and } D \vDash G \tag{96.4}$$

then e is a value or there exists e'/s' such that $e/s \hookrightarrow e'/s'$.

Proof. The proof proceeds by induction on the derivation of $G,\ \emptyset,\ h \vdash_{m,\ I} e : \tau / h'$. We split cases on the final rule used.

Case T-PURE

Follows from Lemma 95.

Case T-SUB

Follows from the inductive hypothesis.

Case T-IF

By the form of the rule,

$$\begin{aligned} e &\equiv \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2 \\ \emptyset &\vdash_m v : \texttt{int}(W, i) \end{aligned} \tag{96.5}$$

By Fact 96.5 and Lemma 72,

$$v = n_{|W|}$$

for some n. Then either $n \neq 0$, in which case E-IF-TRUE applies and

$$e / s \hookrightarrow e_1 / s$$

or $n = 0$ and E-IF-FALSE applies, so

$$e / s \hookrightarrow e_2 / s$$

as required.

Case T-LET

By the form of the rule,

$$\begin{aligned} e &\equiv \textbf{let } x = e_1 \textbf{ in } e_2 \\ G,\ \emptyset,\ h &\vdash_{m,\ I} e_1 : \tau_1 / h_1 \end{aligned} \tag{96.6}$$

We split cases on whether e_1 is a value.

Case e_1 is not a value

Then by Fact 96.6 and the inductive hypothesis, there exist e_1', s'' such that

$$e_1 / s \hookrightarrow e_1' / s''$$

Thus, rule E-SEQ applies, so that

$$e/s \hookrightarrow \textbf{let } x = e_1' \textbf{ in } e_2/s''$$

as required.

Case $e_1 = v$ for some v

Then E-LET applies, and we have

$$e/s \hookrightarrow e_2[x \mapsto v]/s$$

as required.

Case T-READ

By the form of the rule,

$$e \equiv *_n v \tag{96.7}$$

$$\emptyset \vdash_m v : \{\nu : \texttt{ref}(\ell_j, i) \mid \text{Safe}(\nu, n)\} \tag{96.8}$$

$$h = h_1 * \ell_j \mapsto \ldots, i : \tau, \ldots \tag{96.9}$$

$$\text{SizeOf}(\tau) = n \tag{96.10}$$

where

$$\text{Safe}(\nu, n) \triangleq \nu \neq 0_{|W|} \wedge \text{BBegin}(\nu) \leq \nu \wedge \nu + n < \text{BEnd}(\nu) \tag{96.11}$$

By Fact 96.7 and Lemma 72, either

$$v = 0_{|W|} \tag{96.12}$$

or

$$v = \text{bref}(r, p, z) \tag{96.13}$$

$$p \in [\![i]\!] \tag{96.14}$$

$$\ell_j \in \text{Clocs}(r, m) \tag{96.15}$$

By Fact 96.7, Fact 96.10, and Lemma 27,

$$v \neq 0_{|W|} \tag{96.16}$$

$$\text{BBegin}(v) \leq v \tag{96.17}$$

$$v + n < \text{BEnd}(v) \tag{96.18}$$

By Fact 96.15, Fact 96.11 cannot hold, so Fact 96.12, Fact 96.13, and Fact 96.14 must hold. By Fact 96.16, Fact 96.17, and Assumption 14,

$$\begin{aligned} \text{bref}(r, 0, l) &\leq \text{bref}(r, p, z) \\ \text{bref}(r, p + n, z) &< \text{bref}(r, z, z) \end{aligned}$$

Equivalently,

$$0 \leq p \tag{96.18}$$

$$p + n < z \tag{96.19}$$

By Fact 96.14, Fact 96.1, and Definition 23,

$$r \in \text{dom}(s) \tag{96.20}$$

By Fact 96.18 and the assumption that run-time blocks bind all offsets,

$$p \in \text{dom}(s(r)) \tag{96.21}$$

By Fact 96.7, Fact 96.8, Fact 96.2, Fact 96.1, Fact 96.3,

$$\emptyset \vdash_m s(r)(p) : \tau \tag{96.22}$$

By Fact 96.22 and Lemma 71,

$$\text{SizeOf}(s(r)(p)) = n \tag{96.23}$$

By Fact 96.12, Fact 96.18, Fact 96.19, Fact 96.20, Fact 96.20, Fact 96.21, and Fact 96.23, E-READ applies, so that

$$*_n v/s \hookrightarrow s(r)(p)/s$$

as required.

Case T-SUPD, T-WUPD

Similar to T-READ.

Case T-UNFOLD

By the form of the rule,

$$e \equiv \textbf{letu } x = \textbf{unfold } v \textbf{ in } e$$

Rule E-UNFOLD applies, so that

$$\textbf{letu } x = \textbf{unfold } v \textbf{ in } e/s \hookrightarrow e[x \mapsto v]/s$$

as required.

Case T-FOLD

By the form of the rule,

$$e \equiv \textbf{fold } \ell$$

Rule E-FOLD applies, so that

$$e/s \hookrightarrow 0_{|0|}/s$$

as required.

Case T-CALL

By the form of the rule,

$$\begin{aligned} e &\equiv f(\overline{v_j}) \\ G(f) &= (\overline{x_j} : \overline{\tau_j})/h_f \ \rightarrow \ \tau'/h'_f \end{aligned}$$

By Fact 96.4 and Definition 26,

$$D(f) = e_f$$

for some e_f. By E-CALL,

$$f(\overline{v})/s \hookrightarrow e_f[\overline{x_j} \mapsto \overline{v_j}]/s$$

as required.

Case T-MALLOC

By the form of the rule,

$$\begin{aligned} e &\equiv \textbf{malloc}(v) \\ \emptyset &\vdash v : \{\nu : \texttt{int}(W, i) \mid \nu \geq 0\} \end{aligned} \tag{96.24}$$

By Lemma 72,

$$v = n_{|W|} \tag{96.25}$$

By Fact 96.24, Fact 96.25, and Lemma 27,

$$n \geq 0 \tag{96.26}$$

By Fact 96.25 and Fact 96.26, E-MALLOC applies, so that

$$\mathbf{malloc}(v)/s \hookrightarrow 0_{|1|}/s[r \mapsto (\mathbb{Z} \mapsto 0_{|0|})]$$

for some $r \notin \mathrm{dom}(s)$, as required. □

Lemma 97 (Iterated Preservation)**.**

$$\begin{aligned}
&\textit{If } G,\ \emptyset,\ h \vdash_{m,\ I} e : \tau/h',\\
&\quad \emptyset \vDash h,\\
&\quad s \vDash_m h,\\
&\quad \vDash m,\\
&\quad \vDash G,\\
&\quad D \vDash G,\\
&\textit{and } e/s \hookrightarrow^n e'/s',\\
&\textit{then there exist } h_s, m' \textit{ such that } G,\ \emptyset,\ h_s \vdash_{m',\ I} e' : \tau/h',\\
&\quad \emptyset \vDash h_s,\\
&\quad s' \vDash_{m'} h_s,\\
&\textit{and } \vDash m'.
\end{aligned}$$

Proof. The proof proceeds by straightforward induction on n, using Lemma 94. □

C.21 Type Soundness

Finally, we prove the type soundness theorem, which states that any terminating well-typed expression evaluates to a value.

Theorem 7 (NANOC Type Soundness)**.**

$$\textit{If } G,\ \emptyset,\ h \vdash_{\emptyset,\ I} e : \tau / h', \tag{97.1}$$

$$h \textit{ is abstract}, \tag{97.2}$$

$$\emptyset \vDash h, \tag{97.3}$$

$$\vDash G, \tag{97.4}$$

$$D \vDash G, \tag{97.5}$$

$$e / \emptyset \hookrightarrow^{n} e' / s', \tag{97.6}$$

$$\textit{and there is no } e'' / s'' \textit{ such that } e' / s' \hookrightarrow e'' / s'', \tag{97.7}$$

$$\textit{then } e' \textit{ is a value.}$$

Proof. By Fact 97.2 and Definition 23,

$$\emptyset \vDash_{\emptyset} h \tag{97.8}$$

By Definition 20,

$$\vDash \emptyset \tag{97.9}$$

By Fact 97.1, Fact 97.3, Fact 97.8, Fact 97.9, Fact 97.4, Fact 97.5, Fact 97.6, and Lemma 97,

$$G,\ \emptyset,\ h_s \vdash_{m,\ I} e' : \tau / h', \tag{97.10}$$

for some h_s and m such that

$$s' \vDash_{m} h_s, \tag{97.11}$$

$$\emptyset \vDash h_s, \tag{97.12}$$

$$\vDash m. \tag{97.13}$$

By Fact 97.10, Fact 97.11, Fact 97.12, Fact 97.13, Fact 97.5, Fact 97.7, and Lemma 96, e' is a value. □