UNIVERSITY OF CALIFORNIA, SAN DIEGO

High-Level Liquid Types

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

in

Computer Science

by

Ming Woo-Kawaguchi

Committee in charge:

Professor Ranjit Jhala, Chair
Professor Samuel Buss
Professor Brian Demsky
Professor Sorin Lerner
Professor Geoffrey Voelker

2016

DEDICATION

To my grandmother, Hanako Umino Kawaguchi, who taught me that the the moment you step off of the train from Minidoka, you run as fast as you can; you never look back.

To Jordan, Cezario, Nathan and all whose brilliance were muted far before their time. Knowing what you would have contributed to the world crushes me. I stand on your shoulders, and will always remember.

EPIGRAPH

”Be careful what you wish for” - Ranjit Jhala

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Above all, I must thank the amazing Ranjit Jhala, Patrick Rondon and Geoff Voelker for their support, research co-conspiracy, wisdom, and most of all, friendship, over my many years at UC San Diego.

Yet, this thesis still would not have been possible without the intervention of vast numbers of excellent people at UCSD and elsewhere I've found myself landing. I would be remiss not to thank my labmates (in no particular order) Zach Tatlock, Ross Tate, Jan Voung, Alexander Bakst, Ravi Chugh, Jean Yang (honorary via cloneship) and so on and so on..

Further, it would be near criminal if I did not thank the inimitable Margo Seltzer for forcing me to finish and defend, and most of all, for giving me the inspiration and support to re-enter academia and rediscover my love of teaching and research.

And yet! This thesis would not be possible without the entirety of the CSE department at UCSD. In the twelve years that I spent in San Diego, I met and worked with a panoply of brilliant, contagiously excited researchers, teachers, staff and most of all, lifetime friends.

The cohort of UCSD CSE is and will always be family to me, and no aspect of my career, this dissertation among countless other papers, documents, presentations, lectures *etc.*, has been untouched by their wisdom and friendship.

Published Works Adapted in This Dissertation

Chapter 2 contains material adapted from the publication: Ming Kawaguchi, Patrick Rondon, Ranjit Jhala. "Type-Based Data Structure Verification", *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 159–169, 2009. The dissertation author was principal researcher and author on this

publication. This chapter also contains material adapted from the publication: 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. The dissertation author was a co-author on this publication.

Chapter 3 contains material adapted from the publication: Ming Kawaguchi, Patrick Rondon, Ranjit Jhala. “Type-Based Data Structure Verification”, *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 159–169, 2009. The dissertation author was principal researcher and author on this publication. This chapter also contains material adapted from the publication: 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 an author on this publication.

Chapter 4 contains material adapted from the publication: Ming Kawaguchi, Patrick Rondon, Ranjit Jhala. “Deterministic Parallelism via Liquid Effects”, *Proceedings of the 2012 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 45–54, 2012. The dissertation author was principal researcher and author on this publication. This chapter also contains material adapted from the publication: 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 a co-author on this publication.

VITA

2016	PhD., Computer Science
2011	PhD. Candidate, Computer Science
2010	MS, Computer Science
2005	BS, Computer Science

PUBLICATIONS

Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, Henrique Rebelo, “Towards Modularly Comparing Programs Using Automated Theorem Provers.”, CADE 2013, Lake Placid, New York

Ming Kawaguchi, Patrick Maxim Rondon, Alexander Bakst, Ranjit Jhala, “Deterministic Parallelism via Liquid Effects.”, PLDI 2012, Beijing, China

Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, Henrique Rebelo, “SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs.”, CAV 2012, Berkeley, California

Patrick Maxim Rondon, Alexander Bakst, Ming Kawaguchi, Ranjit Jhala, “CSolve: Verifying C with Liquid Types.”, CAV 2012, Berkeley, California

Patrick Maxim Rondon, Ming Kawaguchi, Ranjit Jhala, “Low-Level Liquid Types.”, POPL 2010, Madrid, Spain

Ming Kawaguchi, Patrick Maxim Rondon, Ranjit Jhala, “Dsolve: Safety Verification via Liquid Types.”, CAV 2010, Edinburgh, UK

Ming Kawaguchi, Patrick Maxim Rondon, Ranjit Jhala, “Type-Based Data Structure Verification.”, PLDI 2009, Dublin, Ireland

Patrick Maxim Rondon, Ming Kawaguchi, Ranjit Jhala, “Liquid Types.”, PLDI 2008, Tuscon, Arizona

Chris Hawblitzel, Ming Kawaguchi, Shuvendu Lahiri, Henrique Rebelo “Mutual Summaries: Unifying Program Comparison Techniques.”, MSR-TR-2011-78

Ming Kawaguchi, Shuvendu K. Lahiri, Henrique Rebelo, “Conditional Equivalence.”, MSR-TR-2010-119

ABSTRACT OF THE DISSERTATION

High-Level Liquid Types

by

Ming Woo-Kawaguchi

Doctor of Philosophy in Computer Science

University of California, San Diego, 2016

Professor Ranjit Jhala, Chair

Because every aspect of our lives is now inexorably dependent on software, it is crucial that the software systems that we depend on are reliable, safe, and correct. However, verifying that software systems are safe and correct in the real world is extraordinarily difficult. In this dissertation, we posit that this difficulty is often due to a fundamental impedance mismatch between a programmer's "high-level" intent for the operation of their program versus the "low-level" source code that they write. In particular, we claim that the impedance exists in the large because of the lack of mechanisms available for encoding and verifying complex properties of unbounded data structures. Then, we propose several augmentations of the Liquid Types method of automatic program verification for uniformly describing high-level specifications and for verifying that source code is correct with respect to such specifications. First, we describe a means of verifying the correctness of programs that perform subtle manipulations over complex recursive and linked data structures such as sorted lists, balanced trees and acyclic graphs. Second, we describe a means of verifying the correctness of concurrent, shared memory programs that perform subtle manipulations on the heap. In particular we automatically verify that concurrent, heap manipulating programs intended to be deterministic over all possible schedules do always produce the same heap. Finally, we show empirically that these techniques are practical and efficient by building two tools: DSOLVE, for verifying that real programs that manipulate complex data structures do so correctly, and CSOLVE for verifying that real, subtly deterministic heap manipulating programs do, in fact, always produce the same heap.

Chapter 1

Introduction

It is well known that computer programs often do not work. Computer programs contain bugs and vulnerabilities that violate the safety of both the computer systems on which they execute, and the systems that rely on them. In an ideal world, we would simply examine every program at compile time using a computer program and decide whether that program would always execute safely, or whether some execution of that program violates a safety property. This is a problem that is generally referred to as the *static* software verification problem. Further, it is a classic and trivial result that static verification reduces to the halting problem and hence is *undecidable*. That is, no such program safety verifying computer program exists.

Yet, it is also well-known that there exist important classes of programs for which static software verification can either be decided or efficiently recognized. Further, this literature on verification frameworks dates back to the inception of programming languages. Yet, static program verification is still considered to be impractical and inefficient for most use cases.

1.1 Verifying High-Level Invariants

In this dissertation, we address *high-level* specifications, complex properties that capture *programmer intent* over potentially unbounded data and control structures. In particular, we claim that one significant reason programs do not behave as the programmer intended is because of an impedance mismatch between programmer and program.

That is, we claim that it is vastly easier for a programmer to design algorithms that utilize high-level properties for efficient, practical programming than it is to correctly implement those high-level properties in source code programs. We recall, then, the automatic software verification problem and consider whether we can use tools that encode properties and verify that source code maintains them to develop better language-level constructs and development tools for such programmers.

However, in practice and theory, programmer intent is amorphous and difficult to infer from code. We define intent as properties intended to be true of a program by a programmer that may or may not be explicitly reflected in source code. For example, intent may be embedded in conditionals that lead nowhere or to a dynamic error. Programmer intent may even be embedded only in non-executable comments that are impossible to automatically reason about to any reasonable definition.

Yet, programmers must program with intent, so our research posits that, in fact, intent is embedded in all code as *invariants*: properties that are true of sets of values, variables and the heap in all potential executions of a program. Then, we claim that *high-level* invariants are properties that relate multiple values, values and their heap interactions in all potential executions of a program.

We capture these high-level invariants in *refinement types* that extend primitive types of values and variables with *refinements* that allow us to express properties of complex, potentially recursive and algebraic, data structures of unbounded size even when they represent subtle relationships between stack allocated values and data on the heap. We then use Liquid Type Inference, due to Rondon, Kawaguchi and Jhala [52, 54] to enable automatic static verification of these invariants.

To that end, we refine our problem statement: Programmer intent is embedded in *data structure invariants*, program properties that are true over every execution, including those over complex control invariants such as pattern matching, pointer arithmetic, and polymorphic recursion.

High-Level Liquid Types. Then, the goal of this dissertation is to utilize and adapt the Liquid Type Inference [52, 54] technique and related type-based verification techniques to capture, encode and verify that source code is implemently correctly with respect to high-level invariants that accurately represent programmer intent with respect to their programs.

1.2 Contributions

As a result, this dissertation contributes the following to the research state of the art.

1. We describe a novel means of writing "high-level" data structure specifications using "Liquid" refinement types that correspond naturally to programmer intent.

2. We describe a static analysis algorithm for inferring such refinement types over functional data structures using Rondon, Kawaguchi and Jhala's Liquid Type Inference technique [52, 54] for automatic inference of refinement types.

3. We describe a type and effects system embedded in our refinement type system that allows us to describe the interaction of *merge-join* style concurrent programs over shared heaps.

4. We describe a static analysis algorithm for inferring the above refinement types over merge-join concurrent programs that subtly manipulate mutable data structures and hence prove that they are memory-safe, *i.e.,* deterministic.

5. We show that our techniques are practical and efficient by implementing two tools, DSOLVE and CSOLVE that implement the techniques and use them to verify the safety and determinism of a number of challenging benchmarks both from the literature and from production data structure libraries.

1.2.1 Chapter Summaries

To this end, in Chapter 2 we introduce Recursive Refinements, a technique for encoding and efficiently verifying universally quantified data structure properties that hold for every element of a recursive structure. This includes, but is not limited to, sorted-ness of lists, balanced-ness of trees, and compositions of properties such as balanced binary search trees.

Then, in Chapter 3 we introduce Measures and Polymorphic Refinements. First, we show that measures are a means of specifying non-boolean, well-founded properties of recursive and variant structures. We show that, when encoded using our methods, these properties can be verified directly against their implementations in source code and used to prove that code maintains complex properties syntactically expressed using the measure name.

Second, we show that Polymorphic Refinements allow a programmmer the expressiveness of existentially quantified properties without causing undecidability of verification. To this end, we demonstrate a syntactic guarantee that the existential quantifier can be eliminated during proof search, ensuring that proof search remains decidable.

In Chapter 4, we introduce Liquid Effects, a type-and-effect system for C programs that encodes memory access patterns over heaps, and admits Liquid Type Inference to enable the automatic proof of disjointness of such access patterns, allowing a programmer to write provably deterministic parallel code using fork-join concurrency.

Finally, in Chapter 5 we conclude with a detailed summary of both the concepts introduced in this dissertation, and a brief mention of the impact that our work has had on the research community attempting to address both verification of complex high-level program properties as well as synthesis of programs that provably maintain high-level program properties.

1.2.2 Dissertation Summary

This dissertation addresses the impedance mismatch that programmers face when attempting to encode high-level invariants over source code constructs that may not explicitly reflect their intent. We posit that this impedance mismatch is largely due to the need for encoding and automatic static verification of data structure invariants. Then, we show that, by piggybacking predicates over expressive base types, there exists both an intuitive encoding of complex properties such as, but not limited to, sortedness of lists, balanced-ness of trees, and acyclicity of linked structures. Further, we show that this technique extends to a significant concurrent setting with novel expressiveness.

Finally, we argue that work detailed in this dissertation has had a lasting effect on the field of automatic program verification. We argue that, prior to this work, the state of the art in automatic data structure reasoning was intractable, either requiring reasoning undecidable problems such as satisfaction of quantified first and higher-order logical sentences, or doubly exponential solvers for approximate graph domains.

We then argue that our key ideas enabling *decidable* reasoning about data structure properties have enabled new lines of work in tractable reasoning and automatic verification of complex data structure manipulating code. We justify this argument by following the direct lines of research that utilize and extend our approaches to solve previously intractable problems such as program synthesis, inference of arbitrary data structure invariants, automatic verification of imperative code manipulating linked data structures, and an extensive line of work on verifying HASKELL programs.

Chapter 2

Recursive Refinements

In this chapter, we will introduce a mechanism which we call *Recursive Refinements*. The key idea behind Recursive Refinements is that they are an intuitive way to soundly reason about "well-formed" data structures.

What is a "well-formed" data structure? In this thesis, we will say that a data structure is "well-formed" if there exists a language level mechanism for reasoning about its properties.

For example, in Java, objects instantiated of a well-defined class are "well-formed" data structures because of the existence of class declaration and definition syntax. Further, In C, records are "well-formed" data structures, because of the existence of the C "struct" syntax.

However the term "well-formed" will *not* make any statement about whether or not these data structures are guaranteed to match their compile-time definitions over every possible execution of a program which defines them. That is, the term "well-formed" says nothing about the dynamic behavior of a data structure.

In fact, *C* does not have a type system which is statically enforced, and hence a segment of memory whose life starts as a struct may end as any arbitrary binary value. In particular, this value may have no valid interpretation as a C struct. Yet, we still consider records to be well-formed data structures in C.

In fact, the goal of this thesis will be to define a set of syntactic and semantic mechanisms that allow for the definition of "well-formed" data structures of a certain complexity which are not only syntactically definable in our language, but whose properties, in which the complexity lies, are guaranteed to hold over every execution.

In particular, the complex properties I refer to are those that are often considered "semantic" or "high-level" properties of ordered data, such as sortedness for lists, balancedness for trees, acyclicity for graphs, and so on. Further, we will show that, given careful specification and use of the mechanisms we introduce, these "high-level" properties can be guaranteed to hold at runtime over every possible execution.

2.1 Recursive Algebraic Data Types

In functional languages, data structure values are given types corresponding to their structure. The key idea of data structure type is that they are typically recursively defined disjoint unions of tuples known as Algebraic Data Types, or ADTs.

Lists. We begin by examining the primitive functional list. We will refer to such structures as *recursively defined* linked structures. That is, structures which can be ordinally organized such that the $\mathtt{n}+1^{st}$ element can be defined in terms of the $\mathtt{n}^{th}$ element which precedes it. To use the instructive example of a functional list, it suffices to define a list as a basis, or initial 0^{th} element, followed by a single inductive definition which defines any given element in terms of the element which precedes it.

For example, to define a list of the natural counting numbers from one to three, or

$\texttt{range}13$ using the OCAML standard library, we produce the following definitions:

$$\begin{aligned}\texttt{type int list} &= \texttt{Cons of int} * \texttt{int list}\\ &\mid \texttt{Nil}\\ \texttt{letrange}_{1,3} &= \texttt{Cons}(1,\texttt{Cons}(2,\texttt{Cons}(3,\texttt{Nil})))\end{aligned}$$

Let us dissect this set of definitions. We first define a type in a *pattern-matching* style. This style of definition is often referred to as a *variant*, a *disjoint union*, or as we will refer to them henceforth, an *algebraic data type*. This type definition states, simply, that for any value which has type list, the basis this value is an arbitrary object which we call `Nil`, and further, every list otherwise contains a `Cons` which must be followed by either another integer `Cons` or a `Nil`, which must necessarily end the list.

To produce the $n+1^{st}$ item of this list, for $n > 1$, one need only recursively produce the n^{th} item, then apply the inductive rule. One can easily see that this can be repeated indefinitely, and hence one can "unfold" this definition an arbitrary number of times to produce an arbitrarily long list of integers. Conversely, with a simple recursive function, one can produce the set of naturals up to any integer n, or more succinctly stated using the OCAML standard library $\texttt{range}1n$.

However, the type we have just defined, `int list`, does not only describe intervals of the naturals. In fact, in OCAML, we can think of types as sets of values, and the set described by `int list` is the set of all lists of integers in any order or sign. In practice, we say that every list of integers *inhabits* the type `int list`.

this is a convenient mechanism for summarization: we give all lists the same type, and then we can have functions which take as input only lists but any list at all. That is, we can create a function

$$\texttt{list\_sum} : \texttt{int list} \to \texttt{int}$$

and in many ways this is very convenient. We have described a function which takes any list of integers, and most likely produces an integer which is the sum of all the integers in the input list.

However, my claim is that this is actually a suboptimal mechanism when trying to do what we will refer to in this thesis as “precise reasoning” about programs at compile time. To see what “precise reasoning means”, consider the following function signature:

$$\texttt{pos\_list\_sum} : \texttt{int list} \to \texttt{int}$$

That is, we would like to describe a function which takes only lists of positive integers and returns an expression which is a sum of the list’s elements. As it turns out, ML types alone do not let one check this property at compile time or runtime. The ML type system will allow this function to take any integer list at all and return any integer it may compute.

One option for checking this property might might be to embed a property checking piece of code, which is often called a “code contract” in the literature, into the function. This code contract could detect a violation of the property in the course of any execution which did violate the property. However, such a method has several drawbacks. First, a code contract cannot tell you whether there exists an execution which might violate the property. Second, it may be that the code contract is forced to simply halt the program or otherwise do something bad if the property is violated.

This raises the following question: why does ML not include such a type and enforce it at runtime? As aforementioned, it is because of fundamental limits on computability. The problem of determining "high-level-properties" on values of this kind is directly reducible to the halting program; it is easy to see why by recalling our implementation of this check as a contract that halts the program on property violation. If we can determine at compile-time whether the contract is violated, we can determine, in general, whether the program halts.

Consider the following: first, determining the length of a list at compile-time is undecidable for the same reason as above. Consequently, there is no bound on the number of list items that a property must specify as positive. Hence, we must say something of the sort "for every integer in this list, that integer is positive". In fact, this directly translates into a statement in first order logic that is a *universally quantified* and, as it turns out, satisfaction of quantified first order logic is undecidable as well.

Hence, the decision problems we are trying to reason about in order to verify the maintenance of high-level-properties over data structures are generally undecidable.

However, as it turns out, for some large classes of data structure intensive programs, we need only embed quantifier free first-order statements into structured *type refinements* that we call *recursive refinements*. This by itself does not solve our problem, since the satisfaction of quantifier-free first order logic may not be decidable, but is NP-complete.

Hence, we rely on one more mechanism: in this thesis, we assume and show experimentally that there exist efficient *Satisfiabiliy Modulo Theories* (SMT) solvers that can prove satisfaction of the kinds of formulas that the algorithms for inferring and checking recursive refinements generate efficiently.

In the remainder of this chapter, we will describe the recursive refinements mechanism and show why and how it makes the problem of checking these properties more tractable than their worst-case computability and complexity.

In the next section, we will first use a series of didactic examples to informally show how these invariants are specified and how they are then automatically proven. Then, we will fully formalize a toy language and a checkable type system around the crux of these ideas. Finally, we will show the reader a method by which recursive refinements are automatically inferred.

2.2 Verifying Software with Liquid Types

We begin with an overview of our type-based verification approach. First, we review simple refinement and liquid types. Next, we describe recursive and polymorphic refinements, and illustrate how they can be used to verify data structure invariants.

Refinement Types. Our system is built on the notion of refining *base* types with predicates over program values that specify additional constraints which are satisfied by all values of the type [6, 21]. For example, all values of integer type (denoted `int`), can be described as $\{\nu : \texttt{int} \mid e\}$ where ν is a special *value variable* not appearing in the program, and e is a boolean-valued expression constraining the value variable called the *refinement predicate*. Intuitively, the base refinement predicate specifies the set of values c of the base type B such that the predicate $e[\nu \mapsto c]$ evaluates to true. Then, consider the type $\{\nu : \texttt{int} \mid \nu \leq \texttt{n}\}$. We say that this refinement type is inhabited by the set of integers whose value is less than or equal to the value of the variable `n`. That is, the base type integer is refined by a dependent express that depends on the value of the variable `n`. **Dependent Function Types.** We use the base refinements to build up *dependent function types*, written $x\!:\!\tau_1\!\rightarrow\!\tau_2$. Here, τ_1 is the domain type of the function, such that the name x denotes the the formal parameter of the function. Then, the parameter x may appear in the base refinements of the range type τ_2. For example, $x{:}\texttt{int}{\rightarrow}\{\nu : \texttt{int} \mid x \leq \nu\}$ is the type of a function that takes an input integer and returns an integer greater than the input. Thus, the type `int` abbreviates $\{\nu : \texttt{int} \mid \top\}$, where $\top$ and $\bot$ abbreviate *true* and *false* respectively.

Liquid Types. A *logical qualifier* is a boolean-valued expression (*i.e.,* 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. 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 $i \leq \nu$ 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 $\mathbb{Q}$. In the rest of this section, let $\mathbb{Q}$ be the qualifiers

$$\mathbb{Q} = \{0 < \nu,\ \star \leq \nu\}.$$

A *liquid type over* $\mathbb{Q}$ is a dependent type where the refinement predicates are conjunctions of qualifiers from $\mathbb{Q}^\star$. We write *liquid type* when $\mathbb{Q}$ is clear from the context. We can automatically *infer* refinement types by requiring that certain expressions like recursive functions have liquid types [52].

Safety Verification. Refinement types can be used to statically prove safety properties by encoding appropriate preconditions into the types of primitive operations. For example, to prove that no divide-by-zero or assertion failures occur at run-time, we can type check the program using the types $\texttt{int}{\rightarrow}\{\nu : \texttt{int} \mid \nu \neq 0\}{\rightarrow}\texttt{int}$ and $\{\nu : \texttt{bool} \mid \nu\}{\rightarrow}\texttt{unit}$ for division and `assert` respectively.

```
let rec range i j =
  if i > j then
    []
  else
    let is = range (i+1) j in
    i::is

let harmonic n =
  let ds = range 1 n in
  List.fold_left
    (fun s k -> s + 10000/k)
    0 ds
```

Figure 2.1: Divide-by-zero

```
let rec insert x ys =
  match ys with
  | [] -> [x]
  | y::ys' ->
     if x < y then x::y::ys'
     else y::(insert x ys')

let rec insertsort xs =
  match xs with
  | [] -> []
  | x::xs' ->
     insert x (insertsort xs')
```

Figure 2.2: Insertion Sort

2.3 Overview

In this section, we begin by augmenting a vanilla ML ADT with *uniform refinements*, statements which apply to every element in an ADT and may not depend on items within the data structure. Then, we expound on this mechanism to introduce *recursive refinements*, for describing universal invariants on data structures which can depend on items within the structure.

Uniform Refinements. Recall that the ML type for integer lists is:

$$\mu t.\mathtt{Nil} + \mathtt{Cons}\langle x_1\!:\!\mathtt{int}, x_2\!:\!t\rangle$$

a recursive sum-of-products which we abbreviate to `int list`. We specify a list of integers each of which satisfies a predicate p as $(\langle\langle\rangle;\langle p;\top\rangle\rangle)$ `int list`. For example, $(\langle\langle\rangle;\langle \mathtt{i} \leq \nu;\top\rangle\rangle)$ `int list` specifies lists of integers greater than some program variable `i`. This representation reduces specification and inference to determining which logical qualifiers apply at each position of the refinement matrix. In the sequel, for any expression e and relation $\bowtie$ we define the abbreviation $\rho^e_\bowtie$ as:

$$\rho^e_\bowtie \doteq \langle\langle\rangle;\langle e \bowtie \nu;\top\rangle\rangle \tag{2.1}$$

To see how such uniform refinements can be used for verification, consider the program in Figure 2.1. The function `range` takes two integers, `i` and `j`, and returns the list of integers `i`,...,`j`. The function `harmonic` takes an integer `n` and returns the `n`th (scaled) harmonic number. To do so, `harmonic` first calls `range` to get the list of denominators 1,...,`n`, and then calls `foldl` with an accumulator to compute the harmonic number. To prove that the divisions inside `harmonic` are safe, we need to know that all the integers in the list `is` are non-zero. Using $\mathbb{Q}$, our system infers:

$$\texttt{range} : \texttt{i}{:}\texttt{int}{\rightarrow}\texttt{j}{:}\texttt{int}{\rightarrow}(\rho^{\texttt{i}}_{\leq})\ \texttt{int list}$$

By substituting the actuals for the formals in the inferred type for `range`, our system infers that the variable `ds` has the type: $(\rho^{1}_{\leq})\ \texttt{int list}$. As the polymorphic ML type for `foldl` is $\forall\alpha,\beta.(\alpha{\rightarrow}\beta{\rightarrow}\alpha){\rightarrow}\alpha{\rightarrow}\beta\ \texttt{list}{\rightarrow}\alpha$, our system infers that, at the application to `foldl` inside `harmonic`, α and β are respectively instantiated with `int` and $\{\nu : \nu \mid \texttt{int}\}0 < \nu$, and hence, that the accumulator `f` has the type:

$$\texttt{s}{:}\texttt{int}{\rightarrow}\texttt{k}{:}\{\nu : \nu \mid \texttt{int}\}0 < \nu{\rightarrow}\texttt{int}$$

As `k` is strictly greater than 0, our system successfully typechecks the application of the division operator $/$, proving the program “division safe”. Hence, by refining the base types that appear inside recursive types, we can capture invariants that hold uniformly across all the elements within the recursively defined value.

2.3.1 Recursive Refinements

The key idea behind recursive refinements is that recursive types can be refined using a matrix of predicates, where each predicate applies to a particular element of the recursive type. For ease of exposition, we consider recursive types whose body is a sum-of-products. Each product can be refined using a *product refinement*, which is a vector of predicates where the j^{th} predicate refines the j^{th} element of the product. Each sum-of-products (and hence the entire recursive type), can be refined with a *recursive refinement*, which is a vector of product refinements, where the i^{th} product refinement refines the i^{th} element of the sum.

Nested Refinements. The function `range` returns an *increasing* sequence of integers. Recursive refinements can capture this invariant by applying the recursive refinement to the μ-bound variables *inside* the recursive type. For each type typ define:

$$\mathsf{typ}\ \mathtt{list}_{\leq} \doteq \mu t.\mathtt{Nil}\ +\ \mathtt{Cons}\langle x_1 : \mathsf{typ}, x_2 : (\rho_{\leq}^{x_1})\ t\rangle \tag{2.2}$$

Thus, a list of increasing integers is $\mathtt{int}\ \mathtt{list}_{\leq}$. This succinctly captures the fact that the list is increasing, since the result of "unfolding" the type by substituting each occurrence of t with the entire recursively refined type is:

$$\mathtt{Nil}\ +\ \mathtt{Cons}\langle x_1' : \mathtt{int}, x_2' : (\rho_{\leq}^{x_1'})\ \mathtt{int}\ \mathtt{list}_{\leq}\rangle$$

where x_1', x_2' are fresh names introduced for the top-level "head" and "tail". Intuitively, this unfolded sum type corresponds to a value that is either the empty list or a cons of a head x_1' and a tail which is an increasing list of integers greater than x_1'.

For inference, we need only find which qualifiers flow into the recursive refinement matrices. Using $\mathbb{Q}$, our system infers that `range` returns an increasing list of integers no less than `i`:

$$\mathtt{range} : \mathtt{i} : \mathtt{int} \rightarrow \mathtt{j} : \mathtt{int} \rightarrow (\rho_{\leq}^{\mathtt{i}})\ \mathtt{int}\ \mathtt{list}_{\leq}$$

As another example, consider the insertion sort function from Figure 2.2. In a manner similar to the analysis for `range`, using just $\mathbb{Q}$, and no other annotations, our system infers that `insert` has the type $\mathtt{x} : \alpha \rightarrow \mathtt{ys} : \alpha\ \mathtt{list}_{\leq} \rightarrow \alpha\ \mathtt{list}_{\leq}$, and hence that program correctly sorts lists. Furthermore, the system infers that `insertsort` has the type xs:α list$\rightarrow\alpha$ list$_{\leq}$.

The main difficulty in inferring this type arises from the case where insert is recursively invoked on the tail of ys. The arguments passed to the recursive call to insert are a value no less than y (inferred from the branch) and a list of values no less than y (as the list is increasing). Thus, at this site, we can *instantiate* the α in the polymorphic ML type of insert, namely $\forall\alpha.\alpha\rightarrow\alpha$ list$\rightarrow\alpha$ list, with the monomorphic $\{\nu : \alpha \mid \mathtt{y} \leq \nu\}$ to deduce that the output of the recursive call has type $\{\nu : \nu \mid \mathtt{y} \leq \nu\}$ list, *i.e.,* is a list of values no less than y. This fact, combined with the (inductive) fact that insert preserves the increasing property, shows that the result of cons-ing y and the recursively computed value (and hence, the output of insert) is an increasing list.

2.4 Base Language

We begin by reviewing the core language *NanoML* from [52], by presenting its syntax and static semantics. *NanoML* has a strict, call-by-value semantics, formalized in Figure 2.4 using a small-step operational semantics. In this and the following chapters, we will extend this language with syntactic mechanisms and a type system that guarantees its safe behavior with regard to the complex data structure properties mentioned in the introduction.

Expressions. The expressions of *NanoML*, summarized in Figure 2.3, include variables, primitive constants, $\lambda-$abstractions and function application. In addition, *NanoML* has let-bindings and recursive function definitions using the fixpoint operator fix. Our type system conservatively extends ML-style parametric polymorphism. Thus, we assume that the ML type inference algorithm automatically places appropriate type generalization and instantiation annotations into the source expression. Finally, we assume that, via α-renaming,

e	$::=$		*Expressions:*
	$\mid$	x	variable
	$\mid$	$\mathtt{c}$	constant
	$\mid$	$\lambda x.e$	abstraction
	$\mid$	$x\,x$	application
	$\mid$	$\mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e$	if-then-else
	$\mid$	$\mathtt{let}\ x\ =\ e\ \mathtt{in}\ e$	let-binding
	$\mid$	$\mathtt{fix}\ x.e$	fixpoint
	$\mid$	$[\Lambda\alpha]e$	type-abstraction
	$\mid$	$[\mathsf{typ}]e$	type-instantiation
Q	$::=$		*Liquid Refinements*
	$\mid$	$\top$	true
	$\mid$	q	qualifier in $\mathbb{Q}^\star$
	$\mid$	$Q \wedge Q$	conjunction
B	$::=$		*Base:*
	$\mid$	$\mathtt{int}$	integers
	$\mid$	$\mathtt{bool}$	booleans
	$\mid$	α	type variable
$\mathbb{A}(\mathbb{B})$	$::=$		*Unrefined Skeletons:*
	$\mid$	B	base
	$\mid$	$x\!:\!\mathbb{T}(\mathbb{B})\rightarrow\mathbb{T}(\mathbb{B})$	function
$\mathbb{T}(\mathbb{B})$	$::=$		*Refined Skeletons:*
	$\mid$	$\{\nu :\ \mathbb{A}(\mathbb{B})\ \mid\ \mathbb{B}\}$	refined type
$\mathbb{S}(\mathbb{B})$	$::=$		*Type Schema Skeletons:*
	$\mid$	$\mathbb{T}(\mathbb{B})$	monotype
	$\mid$	$\forall\alpha.\mathbb{S}(\mathbb{B})$	polytype
typ, σ	$::=$	$\mathbb{T}(\top), \mathbb{S}(\top)$	*Types, Schemas*
τ, S	$::=$	$\mathbb{T}(E), \mathbb{S}(E)$	*Depend. Types, Schemas*
$\hat{\tau}, \hat{S}$	$::=$	$\mathbb{T}(Q), \mathbb{S}(Q)$	*Liquid Types, Schemas*

Figure 2.3: *NanoML* **Syntax**

$$
\begin{array}{rcll}
\textbf{Contexts} & & & \boxed{C} \\
v & ::= & & \textit{Values:} \\
 & & \mid \mathtt{c} & \text{constants} \\
 & & \mid \lambda x.e & \lambda\text{-terms} \\
 & & \mid \langle v \rangle & \text{tuples} \\
\mathscr{C} & ::= & & \textit{Contexts:} \\
 & & \mid \bullet & \text{hole} \\
 & & \mid \mathscr{C}\ e & \text{application left} \\
 & & \mid v\,\mathscr{C} & \text{application right} \\
 & & \mid \mathtt{if}\ \mathscr{C}\ \mathtt{then}\ e\ \mathtt{else}\ e & \text{if-then-else} \\
 & & \mid \mathtt{let}\ x\ =\ \mathscr{C}\ \mathtt{in}\ e & \text{let-binding} \\
 & & \mid \langle v_1, \ldots, v_{j-1}, \mathscr{C}, e_{j+1}, e_n \rangle & \text{tuples}
\end{array}
$$

$$
\begin{array}{rcll}
\textbf{Evaluation} & & & \boxed{e \leadsto e'} \\
\mathtt{c}\ v & \leadsto & [\![\mathtt{c}]\!](v) & \text{[E-Prim]} \\
(\lambda x.e)\ v & \leadsto & e[x \mapsto v] & [\text{E-}\beta] \\
\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ e\ \mathtt{else}\ e' & \leadsto & e & \text{[E-If-True]} \\
\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ e\ \mathtt{else}\ e' & \leadsto & e' & \text{[E-If-False]} \\
\mathtt{let}\ x\ =\ v\ \mathtt{in}\ e & \leadsto & e[x \mapsto v] & \text{[E-Let]} \\
\mathscr{C}[e] & \leadsto & \mathscr{C}[e']\ \text{if}\ e \leadsto e' & \text{[E-Compat]}
\end{array}
$$

Figure 2.4: *NanoML* **Small-Step Operational Semantics**

each variable is bound at most once in an environment.

Types and Schemas. *NanoML* has a system of base types, function types, and ML-style parametric polymorphism using type variables α and schemas where the type variables are quantified at the outermost level. We organize types into *unrefined types*, which have no top-level refinement predicates, but which may be composed of types which are themselves refined, and *refined types*, which have a top-level refinement predicate. An *ML type (schema)* is a type (schema) where all the refinement predicates are $\top$. A *liquid type (schema)* is a type (schema) where all the refinement predicates are conjunctions of qualifiers from $\mathbb{Q}^\star$. We write typ and σ for ML types and schemas, τ and S for refinement types and schemas, and $\hat{\tau}$ and $\hat{S}$ for liquid types and schemas. We write typ to abbreviate $\{\nu : \mathsf{typ} \mid \top\}$, and $\overline{\mathsf{typ}}$ to abbreviate $\{\nu : \mathsf{typ} \mid \bot\}$. When typ is clear from context, we write $\{e\}$ to abbreviate $\{\nu : \mathsf{typ} \mid e\}$.

Instantiation. We write $\mathsf{Ins}(S, \alpha, \tau)$ for the *instantiation* of the type variable α in the scheme S with the refined type τ. Intuitively, $\mathsf{Ins}(S, \alpha, \{\nu : \tau' \mid e'\})$ is the refined type obtained by replacing each occurrence of $\{\nu : \alpha \mid e\}$ in S with $\{\nu : \tau' \mid e \wedge e'\}$.

Constants. The basic units of computation in *NanoML* are primitive constants, which are given refinement types that precisely capture their semantics. Primitive constants include basic values, like integers and booleans, as well as primitive functions that define basic operations. The input types for primitive functions describe the values for which each function is defined, and the output types describe the values returned by each function. In addition to / and `assert`, the constants of *NanoML* include:

$$\begin{array}{rcl}
\texttt{true} & : & \{\nu : \texttt{bool} \mid \nu\} \\
\texttt{false} & : & \{\nu : \texttt{bool} \mid \texttt{not}\ \nu\} \\
3 & : & \{\nu : \texttt{int} \mid \nu = 3\} \\
= & : & x{:}\texttt{int}{\rightarrow}y{:}\texttt{int}{\rightarrow}\{\nu : \texttt{bool} \mid \nu \Leftrightarrow (x = y)\} \\
+ & : & x{:}\texttt{int}{\rightarrow}y{:}\texttt{int}{\rightarrow}\{\nu : \texttt{int} \mid \nu = x + y\}
\end{array}$$

Next, we give an overview of our static type system, by describing environments and summarizing the different kinds of judgments.

Environments and Shapes. A *type environment* Γ is a sequence of *type bindings* of the form $x{:}S$ and *guard predicates* e. The guard predicates are used to capture constraints about "path" information corresponding to the branches followed in if-then-else expressions. The *shape* of a refinement type schema S, written as $\mathsf{Shape}(S)$, is the ML type schema obtained by replacing all the refinement predicates with $\top$ (*i.e.,* "erasing" the refinement predicates). The shape of an environment is the ML type environment obtained by applying Shape to each type binding and removing the guards.

Judgments. Our system has four kinds of judgments that relate environments, expressions, recursive refinements and types. Well-formedness judgments ($\Gamma \vdash S$) state that a type schema S is *well-formed* under environment Γ. Intuitively, the judgment holds if the refinement predicates of S are boolean expressions in Γ. Subtyping judgments ($\Gamma \vdash S_1 <: S_2$) state that the type schema S_1 *is a subtype of* the type schema S_2 under environment Γ. Intuitively, the judgments state that, under the value-binding and guard constraints imposed by Γ, the set of values described by S_1 is contained in the set of values described by S_2. Typing judgments ($\Gamma \vdash_{\mathbb{Q}} e{:}S$) state that, using the logical qualifiers $\mathbb{Q}$, the expression e has the type schema

Well-Formed Types $\boxed{\Gamma \vdash S}$

$$\frac{\mathsf{Shape}(\tau) = \tau}{\Gamma \vdash \tau}\ [\text{WF-Reflex}]$$

$$\frac{\Gamma \vdash \tau \quad \Gamma; \nu : \mathsf{Shape}(\tau) \vdash e : \texttt{bool}}{\Gamma \vdash \{\nu :\ \tau \mid e\}}\ [\text{WF-Refine}]$$

$$\frac{\Gamma \vdash \tau \quad \Gamma; x : \mathsf{Shape}(\tau) \vdash \tau'}{\Gamma \vdash x : \tau \rightarrow \tau'}\ [\text{WF-Fun}]$$

Figure 2.5: *NanoML*: **Well-Formedness Rules**

Decidable Subtyping $\boxed{\Gamma \vdash S_1 <: S_2}$

$$\frac{}{\Gamma \vdash \tau <: \tau}\ [\text{<:-Reflex}]$$

$$\frac{\Gamma \vdash \tau <: \tau' \quad \mathsf{Valid}([\![\Gamma]\!] \wedge [\![e]\!] \Rightarrow [\![e']\!])}{\Gamma \vdash \{\nu :\ \tau \mid e\} <: \{\nu :\ \tau' \mid e'\}}\ [\text{<:-Refine}]$$

$$\frac{\Gamma \vdash \tau_2 <: \tau_1 \qquad \Gamma; x_2 : \tau_2 \vdash \tau_1'[x_1 \mapsto x_2] <: \tau_2'}{\Gamma \vdash x_1 : \tau_1 \rightarrow \tau_1' <: x_2 : \tau_2 \rightarrow \tau_2'}\ [\text{<:-Fun}]$$

Figure 2.6: *NanoML*: **Subtyping Rules**

S under environment Γ. Intuitively, the judgments state that, under the value-binding and guard constraints imposed by Γ, the expression e will evaluate to a value described by the type S.

Decidable Subtyping and Liquid Type Inference. In order to determine whether one type is a subtype of another, our system uses the subtyping rules (Figure 2.6) to generate a set of implication checks over refinement predicates. To ensure that these implication checks are decidable, we *embed* the implication checks into a decidable logic of equality, uninterpreted functions, and linear arithmetic (EUFA) that can be decided by an SMT solver [15]. As the types of branches, functions, and polymorphic instantiations are liquid, we can automatically infer liquid types for programs using abstract interpretation [52].

Liquid Type Checking $\boxed{\Gamma \vdash_{\mathbb{Q}} e : S}$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : S \quad \Gamma \vdash S <: S' \quad \Gamma \vdash_{\mathbb{Q}} S'}{\Gamma \vdash_{\mathbb{Q}} e : S'} \text{[L-SUB]}$$

$$\frac{\Gamma(x) = \{\nu : \ \tau \ \mid \ e\}}{\Gamma \vdash_{\mathbb{Q}} x : \{\nu : \ \tau \ \mid \ e \wedge \nu = x\}} \text{[L-VAR]} \qquad \frac{}{\Gamma \vdash_{\mathbb{Q}} \mathsf{c} : ty(\mathsf{c})} \text{[L-CONST]}$$

$$\frac{\Gamma \vdash x : \hat{\tau}_x \rightarrow \hat{\tau} \quad \Gamma ; x : \hat{\tau}_x \vdash_{\mathbb{Q}} e : \hat{\tau}}{\Gamma \vdash_{\mathbb{Q}} (\lambda x.e) : x : \hat{\tau}_x \rightarrow \hat{\tau}} \text{[L-FUN]}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} x_1 : x : \tau \rightarrow \tau' \quad \Gamma \vdash_{\mathbb{Q}} x_2 : \tau}{\Gamma \vdash_{\mathbb{Q}} x_1 \ x_2 : \tau'[x \mapsto x_2]} \text{[L-APP]}$$

$$\frac{\Gamma \vdash \hat{\tau} \quad \Gamma \vdash_{\mathbb{Q}} e_1 : \texttt{bool} \qquad \Gamma ; e_1 \vdash_{\mathbb{Q}} e_2 : \hat{\tau} \quad \Gamma ; \neg e_1 \vdash_{\mathbb{Q}} e_3 : \hat{\tau}}{\Gamma \vdash_{\mathbb{Q}} \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \hat{\tau}} \text{[L-IF]}$$

$$\frac{\Gamma \vdash \hat{\tau} \quad \Gamma \vdash_{\mathbb{Q}} e_1 : S_1 \quad \Gamma ; x : S_1 \vdash_{\mathbb{Q}} e_2 : \hat{\tau}}{\Gamma \vdash_{\mathbb{Q}} \texttt{let } x \ = \ e_1 \texttt{ in } e_2 : \hat{\tau}} \text{[L-LET]}$$

$$\frac{\Gamma \vdash \hat{S} \quad \Gamma ; x : \hat{S} \vdash_{\mathbb{Q}} e : \hat{S}}{\Gamma \vdash_{\mathbb{Q}} \texttt{fix } x.e : \hat{S}} \text{[L-FIX]}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : S \qquad \alpha \not\in \Gamma}{\Gamma \vdash_{\mathbb{Q}} [\Lambda \alpha] e : \forall \alpha . S} \text{[L-GEN]}$$

$$\frac{\Gamma \vdash \tau \quad \mathsf{Shape}(\tau) = \mathsf{typ} \quad \Gamma \vdash_{\mathbb{Q}} e : \forall \alpha . S}{\Gamma \vdash_{\mathbb{Q}} [\mathsf{typ}] e : \mathsf{Ins}(S, \alpha, \tau)} \text{[L-INST]}$$

Figure 2.7: **Liquid Type Checking Rules for** *NanoML*

$$\begin{array}{lcll}
e & ::= & \ldots & \textit{Expressions:} \\
 & \mid & \langle e \rangle & \text{tuple} \\
 & \mid & \texttt{C}\langle e \rangle & \text{constructor} \\
 & \mid & \texttt{match}\, e\, \texttt{with} \mid_i \texttt{C}_i\langle x_i \rangle \mapsto e_i & \text{match-with} \\
 & \mid & \texttt{unfold}\, e & \text{unfold} \\
 & \mid & \texttt{fold}\, e & \text{fold} \\
\varepsilon & ::= & \texttt{m} \mid \texttt{c} \mid x \mid \varepsilon\ \varepsilon & \textit{M-Expressions} \\
M & ::= & (\texttt{m}, \langle \texttt{C}_i\langle x_i \rangle \mapsto \varepsilon_i \rangle, \texttt{typ}, \mu t.\Sigma_i \texttt{C}_i \langle x_i : \texttt{typ}_i \rangle) & \textit{Measures} \\
\rho(\mathbb{B}) & ::= & \langle \langle \mathbb{B} \rangle^* \rangle^* & \textit{Recursive Refinement} \\
\mathbb{A}(\mathbb{B}) & ::= & \ldots & \textit{Unrefined Skeletons:} \\
 & \mid & \langle x : \mathbb{T}(\mathbb{B}) \rangle & \text{product} \\
 & \mid & \Sigma_i \texttt{C}_i \langle x : \mathbb{T}(\mathbb{B}) \rangle & \text{sum} \\
 & \mid & (\rho(\mathbb{B}))\ t & \text{recursive type variable} \\
 & \mid & (\rho(\mathbb{B}))\ \mu t.\Sigma_i \texttt{C}_i \langle x : \mathbb{T}(\mathbb{B}) \rangle & \text{recursive type}
\end{array}$$

Figure 2.8: **Recursive Refinements: Syntax**

ρ-Application $\boxed{(\rho)\ \tau \triangleright \tau'}$

$$\frac{\text{fresh } x' (\langle e \rangle [x \mapsto x'])\ \langle x : \tau[x \mapsto x'] \rangle \triangleright \langle x' : \tau' \rangle}{(e; \langle e \rangle)\ (x : \{\nu : \ \tau \ \mid \ e_x\}; \langle x : \tau \rangle) \triangleright (x' : \{\nu : \ \tau \ \mid \ e \wedge e_x\}; \langle x' : \tau' \rangle)}\ [\triangleright\text{-PROD}]$$

$$\frac{\forall i :\ (\rho_i)\ \langle x_i : \tau_i \rangle \triangleright \langle x'_i : \tau'_i \rangle}{(\rho)\ \Sigma_i \texttt{C}_i \langle x_i : \tau_i \rangle \triangleright \Sigma_i \texttt{C}_i \langle x'_i : \tau'_i \rangle}\ [\triangleright\text{-SUM}]$$

Figure 2.9: **Recursive Refinement Application**

2.5 Recursive Refinements

We now describe *NanoML$_{ds}$*, an extension our core language *NanoML* with recursively refined datatypes and corresponding values. As we proceed through this thesis, we will evolve *NanoML$_{ds}$* to provide support for an increasing number of verified features.

However, in the following we will first describe how we extend the syntax of expressions and types, and correspondingly extend the dynamic semantics to support refined recursive types using recursive refinements. Then, we extend the static type system by formalizing the derivation rules that deal with recursive values, and illustrate how the rules

Well-Formed Types $\boxed{\Gamma \vdash S}$

$$\frac{\Gamma \vdash \tau \quad \Gamma;x\!:\!\mathsf{Shape}(\tau) \vdash \langle x\!:\!\tau\rangle}{\Gamma \vdash x\!:\!\tau;\langle x\!:\!\tau\rangle}\ [\text{WF-PROD}]$$

$$\frac{\forall i:\ \Gamma \vdash \langle x_i\!:\!\tau_i\rangle}{\Gamma \vdash \Sigma_i \mathsf{C}_i\langle x_i\!:\!\tau_i\rangle}\ [\text{WF-SUM}]$$

$$\frac{(\rho)\ \tau \triangleright \tau' \quad \Gamma \vdash \tau'[t \mapsto \mathsf{Shape}(\mu t.\tau)]}{\Gamma \vdash (\rho)\ \mu t.\tau}\ [\text{WF-REC}]$$

Figure 2.10: **Recursive Refinement Well-Formedness Rules**

Decidable Subtyping $\boxed{\Gamma \vdash S_1 <: S_2}$

$$\frac{\Gamma \vdash \tau <: \tau' \quad \Gamma;x\!:\!\tau \vdash \langle x\!:\!\tau\rangle <: \langle x'\!:\!\tau'[x' \mapsto x]\rangle}{\Gamma \vdash x\!:\!\tau;\langle x\!:\!\tau\rangle <: x'\!:\!\tau';\langle x'\!:\!\tau'\rangle}\ [\text{<:-PROD}]$$

$$\frac{\forall i:\ \Gamma \vdash \langle x_i\!:\!\tau_i\rangle <: \langle x_i'\!:\!\tau_i'\rangle}{\Gamma \vdash \Sigma_i \mathsf{C}_i\langle x_i\!:\!\tau_i\rangle <: \Sigma_i \mathsf{C}_i\langle x_i'\!:\!\tau_i'\rangle}\ [\text{<:-SUM}]$$

$$\frac{\begin{array}{cc}(\rho_1)\ \tau_1 \triangleright \tau_1' & (\rho_2)\ \tau_2 \triangleright \tau_2' \\ \mathsf{typ} = \mathsf{Shape}(\mu t.\tau_1) = \mathsf{Shape}(\mu t.\tau_2) & \Gamma \vdash \tau_1'[t \mapsto \mathsf{typ}] <: \tau_2'[t \mapsto \mathsf{typ}]\end{array}}{\Gamma \vdash (\rho_1)\ \mu t.\tau_1 <: (\rho_2)\ \mu t.\tau_2}\ [\text{<:-REC}]$$

Figure 2.11: **Recursive Refinement Subtyping**

Liquid Type Checking $\boxed{\Gamma \vdash_{\mathbb{Q}} e : S}$

$$\frac{\Gamma \vdash (\hat\rho)\ \mu t.\hat\tau \quad (\rho)\ \hat\tau \triangleright \hat\tau' \quad \Gamma \vdash_{\mathbb{Q}} e : \{\nu :\ \hat\tau'[t \mapsto (\hat\rho)\ \mu t.\hat\tau] \mid e'\}}{\Gamma \vdash_{\mathbb{Q}} \texttt{fold}\, e : \{\nu :\ (\hat\rho)\ \mu t.\hat\tau \mid e'\}}\ [\text{L-FOLD}]$$

$$\frac{(\rho)\ \tau \triangleright \tau' \quad \Gamma \vdash e : \{\nu :\ (\rho)\ \mu t.\tau \mid e'\}}{\Gamma \vdash \texttt{unfold}\, e : \{\nu :\ \tau'[t \mapsto (\rho)\ \mu t.\tau] \mid e'\}}\ [\text{L-UNFOLD}]$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} \langle e\rangle : \langle x\!:\!\tau\rangle}{\Gamma \vdash_{\mathbb{Q}} \mathtt{C}_j\langle e\rangle : \mathtt{C}_j\langle x\!:\!\tau\rangle + \Sigma_{i\neq j}\mathsf{C}_i\langle x_i\!:\!\overline{\mathsf{typ}_i}\rangle}\ [\text{L-SUM}]$$

$$\frac{\Gamma \vdash e : \Sigma_i \mathsf{C}_i\langle x_i\!:\!\tau_i\rangle \quad \Gamma \vdash \tau \quad \forall i\ \Gamma;\langle x_i\!:\!\tau_i\rangle;\vdash e_i : \hat\tau}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ |_i\ \mathsf{C}_i\langle x_i\rangle \mapsto e_i : \hat\tau}\ [\text{L-MATCH}]$$

Figure 2.12: **Recursive Refinements Static Semantics**

Contexts			$\boxed{C}$
v	$::=$		*Values:*
		$\mid \mathtt{c}$	constants
		$\mid \lambda x.e$	λ-terms
		$\mid \langle v \rangle$	tuples
		$\mid \mathtt{C}_i v$	sums
		$\mid \mathtt{fold}\ v$	folds
$\mathscr{C}$	$::=$		*Contexts:*
		$\mid \bullet$	hole
		$\mid \mathscr{C}\ e$	application left
		$\mid v\,\mathscr{C}$	application right
		$\mid \mathtt{if}\ \mathscr{C}\ \mathtt{then}\ e\ \mathtt{else}\ e$	if-then-else
		$\mid \mathtt{let}\ x\ =\ \mathscr{C}\ \mathtt{in}\ e$	let-binding
		$\mid \langle v_1, \ldots, v_{j-1}, \mathscr{C}, e_{j+1}, e_n \rangle$	tuples
		$\mid \mathtt{C}_i \mathscr{C}$	sums
		$\mid \mathtt{fold}\ \mathscr{C}$	folds
		$\mid \mathtt{unfold}\ \mathscr{C}$	unfolds

Evaluation			$\boxed{e \leadsto e'}$
$\mathtt{c}\ v$	$\leadsto$	$[\![\mathtt{c}]\!](v)$	[E-PRIM]
$(\lambda x.e)\ v$	$\leadsto$	$e[x \mapsto v]$	[E-β]
$\mathtt{if\ true\ then}\ e\ \mathtt{else}\ e'$	$\leadsto$	e	[E-IF-TRUE]
$\mathtt{if\ false\ then}\ e\ \mathtt{else}\ e'$	$\leadsto$	e'	[E-IF-FALSE]
$\mathtt{let}\ x\ =\ v\ \mathtt{in}\ e$	$\leadsto$	$e[x \mapsto v]$	[E-LET]
$\mathtt{unfold}\ (\mathtt{fold}\ v)$	$\leadsto$	v	[E-UNFOLD]
$\mathtt{match}\ \mathtt{C}_j \langle v \rangle\ \mathtt{with}\ \mid_i \mathtt{C}_i \langle x_i \rangle \mapsto e_i$	$\leadsto$	$e_j[\langle x \rangle \mapsto \langle v \rangle]$	[E-MATCH]
$\mathscr{C}[e]$	$\leadsto$	$\mathscr{C}[e']$ if $e \leadsto e'$	[E-COMPAT]

Figure 2.13: *NanoML$_{ds}$*: **Small-Step Operational Semantics**

are applied to check expressions.

2.5.1 Syntax and Dynamic Semantics

Figure 2.8 describes how the expressions and types of *NanoML*$_{ds}$ are extended to include recursively-defined values. The language of expressions includes tuples, value constructors, folds, unfolds, and pattern-match expressions. The language of types is extended with product types, tagged sum types, and a system of *iso-recursive types*. We assume that appropriate `fold` and `unfold` annotations are automatically placed in the source at the standard construction and matching sites, respectively [43]. We assume that different types use disjoint constructors, and that pattern match expressions contain exactly one match binding for each constructor of the appropriate type. The run-time values are extended with tuples, tags and explicit fold/unfold values in the standard manner [43].

Similarly, Figure 2.13 defines a small-step operational semantics that extends those of *NanoML* to account for the additional expression syntax in the standard manner

Notation. We write $\langle Z \rangle$ for a sequence of values of the kind Z. We write $(Z;\langle Z \rangle)$ for a sequence of values whose first element is Z, and the remaining elements are $\langle Z \rangle$. We write $\langle\rangle$ for the empty sequence. As for functions, we write dependent tuple types using a sequence of name-to-type bindings, and allow refinements for "later" elements in the tuple to refer to previous elements. For example, $\langle x_1 : \texttt{int}; x_2 : \{\nu : \texttt{int} \mid x_1 \leq \nu\}\rangle$ is the type of pairs of integers where the second element is greater than the first.

2.5.2 Static Semantics

The derivation rules pertaining to recursively refined types, including the rules for product, sum, and recursive types, as well as construction, match-with, fold and unfold expressions are shown in Figure 2.12. The fold and unfold rules use a judgment called ρ-*application*, written $(\rho)\ \tau \triangleright \tau'$. Intuitively, this judgment states that when a recursive refinement ρ is *applied* to a type τ, the result is a sum type τ' with refinements from ρ. Next, we describe each judgment and the rules relevant to it.

Next, we describe the typing judgments of the static type system, focusing on the rules relating to recursive types. The other rules are either standard [21], or can be found in previous work [52]. We end the section by discussing various design decisions made in order to maintain a balance between expressiveness of the system and decidability of type checking and inference.

ρ-Application and Unfolding. Formally, a recursive type $(\rho)\ \mu t.\tau$ is unfolded in two steps. First, we *apply* the recursive refinement ρ to the *body* of the recursive type τ to get the result τ' (written $(\rho)\ \tau \triangleright \tau'$). Second, in the result of the application τ', we *replace* the μ-bound recursive type variable t with the entire original recursive type ($(\rho)\ \mu t.\tau$), and *normalize* by replacing adjacent refinements $(\rho)(\rho')$ with a new refinement ρ'' which contains the conjunctions of corresponding predicates from ρ and ρ'.

Example. Consider the type which describes increasing lists of integers greater than some x_1':

$$(\rho_{\leq}^{x_1'})\ \mu t.\mathtt{Nil}\ +\ \mathtt{Cons}\langle x_1\!:\!\mathtt{int}, x_2\!:\!(\rho_{\leq}^{x_1})\ t\rangle$$

To unfold this type, we first apply the recursive refinement $\rho_{\leq}^{x_1'}$ to the recursive type's body, $\mathtt{Nil}\ +\ \mathtt{Cons}\langle x_1\!:\!\mathtt{int}, x_2\!:\!(\rho_{\leq}^{x_1})t\rangle$. To apply the recursive refinement to the above sum type, we use rule [$\triangleright$-SUM] to apply the product refinements $\langle\rangle$ and $\langle x_1' \leq \nu; \top\rangle$ of $\rho_{\leq}^{x_1'}$ to the products corresponding to the `Nil` and `Cons` constructors, respectively. To apply the refinements to each product, we use the rule [$\triangleright$-PROD] to obtain the result:

$$\texttt{Nil} + \texttt{Cons}\langle x_1'':\{\nu:\ \texttt{int} \mid x_1' \leq \nu\}, x_2'':(\rho_{\leq}^{x_1''})t\rangle \tag{2.3}$$

Notice that the result is a sum type with fresh names for the "head" and "tail" of the unfolded list. Observe that this renaming allows us to soundly use the head's value to refine the tail, via a recursive refinement stipulating all elements in the tail are greater than the head, x_1''. To complete the unfolding, we replace t with the entire recursive type and normalize to get:

$$\texttt{Nil} + \texttt{Cons}\langle x_1'':\{\nu:\ \texttt{int} \mid x_1' \leq \nu\}, x_2'':(\rho)\ \texttt{int list}_{\leq}\rangle$$

where $\rho \doteq \langle\langle\rangle;\langle x_1' \leq \nu \wedge x_1'' \leq \nu;\top\rangle\rangle$. Intuitively, the result of the unfolding is a type that specifies an empty list, or a non-empty list with a head greater than x_1' and an increasing tail whose elements are greater than x_1' and x_1''.

Next, consider the type describing lists of increasing integers from (2.2). Formally, the type is

$$(\rho_{\top})\ \mu t.\texttt{Nil} + \texttt{Cons}\langle x_1:\texttt{int}, x_2:(\rho_{\leq}^{x_1})\ t\rangle \tag{2.4}$$

where $\rho_{\top}$ is the trivial recursive refinement $\langle\langle\rangle;\langle\top;\top\rangle\rangle$. To unfold this type, we first apply the (trivial) recursive refinement $\rho_{\top}$ to the body, using rules $\triangleright$-SUM and $\triangleright$-PROD to get

$$\texttt{Nil} + \texttt{Cons}\langle x_1':\texttt{int}, x_2':(\rho_{\leq}^{x_1'})\ t\rangle \tag{2.5}$$

To complete the unfolding, we replace the μ-bound t in the above with the entire recursive type, to get

$$\texttt{Nil} + \texttt{Cons}\langle x_1'\!:\!\texttt{int}, x_2'\!:\!(\rho_{\leq}^{x_1'})\ (\rho_{\top})\ \texttt{int list}_{\leq}\rangle$$

which after conjoining the adjacent $\rho_{\leq}^{x_1'}$ and $\rho_{\top}$ reduces to

$$\texttt{Nil} + \texttt{Cons}\langle x_1'\!:\!\texttt{int}, x_2'\!:\!(\rho_{\leq}^{x_1'})\ \texttt{int list}_{\leq}\rangle$$

Note that the fresh names x_1' and x_2' for the head and tail of the unfolded list allow us to soundly use the head's value to refine the tail, via a recursive refinement stipulating all elements in the tail are greater than the head x_1'. In effect this allows us to soundly "instantiate" the recursive refinement $\rho_{\leq}^{x_1}$ with the "bound" x_1 with the specific head x_1' of the list. Intuitively, the result of the unfolding is a type that specifies an empty list, or a non-empty list whose tail is a list of increasing integers, each of which is greater than the head.

Well-formedness. We require that the refinement types of program expressions are *well-formed*, *i.e.,* boolean expressions over variables in scope at that point. For example, the type of a let expression should not refer to the variable it binds (rule [L-LET]). Rule [WF-REC] checks if a recursive type $(\rho)\ \mu t.\tau$ is well-formed in an environment Γ, *i.e.,* if $\Gamma \vdash (\rho)\ \mu t.\tau$. First, the rule applies the refinement ρ to τ, the body of the recursive type, to obtain τ'. Next, the rule replaces the μ-bound variable t in τ' with the *shape* of the recursive type and checks the well-formedness of the result of the substitution.

Example. When $\rho_{\top} \doteq \langle\langle\rangle;\langle\top;\top\rangle\rangle$, the check

$$\emptyset \vdash (\rho_{\top})\ \mu t.\texttt{Nil} + \texttt{Cons}\langle x_1\!:\!\texttt{int}, x_2\!:\!(\rho_{\leq}^{x_1})\ t\rangle$$

is reduced to checking, using [WF-REFINE] that in the environment where x'_1 (the fresh name given to the unfolded list's "head") has type `int`, the refinement $\{\nu : \texttt{int} \mid x'_1 \leq \nu\}$ (applied to the elements of the unfolded list's "tail") is well-formed. Formally, the check is reduced by [WF-REC], which first applies $\rho_\top$ to the body of the type, yielding:

$$\texttt{Nil} + \texttt{Cons}\langle x'_1 : \texttt{int}, x'_2 : (\rho_{\leq}^{x'_1})\ t\rangle$$

Next, [WF-REC] substitutes the shape for t, reducing the check to:

$$\emptyset \vdash \texttt{Nil} + \texttt{Cons}\langle x'_1 : \texttt{int}, x'_2 : (\rho_{\leq}^{x'_1})\ \texttt{int list}\rangle$$

which gets reduced by [WF-SUM] and [WF-PROD] to:

$$x'_1 : \texttt{int} \vdash (\rho_{\leq}^{x'_1})\ \texttt{int list}$$

i.e., that the type that describes a list of integers greater than x'_1, is well-formed in an environment where x'_1 is an integer. This is also checked using [WF-REC], which first applies the refinement to the body of the recursive type to get (2.3). Next, the rule substitutes the shape `int list` for the μ-bound variable t to reduce the above to the following:

$$x'_1 : \texttt{int} \vdash \texttt{Nil} + \texttt{Cons}\langle x''_1 : \{\nu : \texttt{int} \mid x'_1 \leq \nu\}, x''_2 : \texttt{int list}\rangle$$

[WF-SUM], [WF-PROD], and [WF-REFINE] reduce this to:

$$x'_1 : \texttt{int}; x''_1 : \texttt{int} \vdash \texttt{int list}$$

$$x'_1 : \texttt{int}; \nu : \texttt{int} \vdash (x'_1 \leq \nu) : \texttt{bool}$$

which follow from [WF-REFLEX] and ML typing rules.

Subtyping. The subtyping relationship between two types reduces to a set of implication checks between the refinement predicates for the types. The rules for base and function types are standard [21].

Products and Sums. Rule [<:-PROD] (resp. [<:-SUM]) determines whether two products (resp. sums) satisfy the subtyping relation by checking subtyping between corresponding elements.

Example. When $\Gamma \doteq x'_1 : \texttt{int}$, the check

$$\Gamma \vdash \langle y_1 : \{x'_1 < \nu\}, y_2 : \texttt{int list}\rangle <: \langle z_1 : \{x'_1 \neq \nu\}, z_2 : \texttt{int list}\rangle \tag{2.6}$$

is reduced by [<:-PROD] to:

$$\Gamma \vdash \{\nu : \texttt{int} \mid x'_1 < \nu\} <: \{\nu : \texttt{int} \mid x'_1 \neq \nu\}$$

$$\Gamma; y_1 : \texttt{int} \vdash \texttt{int list} <: \texttt{int list}$$

[<:-REFLEX] ensures the latter. [<:-REFINE] reduces the former to checking the validity of $x'_1 < \nu \Rightarrow x'_1 \neq \nu$ in EUFA.

Recursive Types. Rule [<:-REC] determines whether the subtyping relation holds between two recursively refined types. The rule first applies the outer refinements to the bodies of the recursive types, then substitutes the μ-bound variable with the *shape* (as for well-formedness), and then checks that the resulting sums are subtypes.

Example. Consider the subtyping check

$$x_1' : \texttt{int} \vdash (\rho_<^{x_1'})\ \texttt{int list} <: (\rho_{\neq}^{x_1'})\ \texttt{int list} \tag{2.7}$$

that is, the list of integers greater than x_1' is a subtype of the list of integers distinct from x_1'. Rule [<:-REC] applies the refinements to the bodies of the recursive types and then substitutes the shapes, reducing the above (after using [<:-SUM], [<:-PROD] and [<:-REFLEX]) to (2.6). Finally, applying the rule [<:-REC] yields the judgment

$$\emptyset \vdash \texttt{int list}_< <: \texttt{int list}_{\neq} \tag{2.8}$$

$$\texttt{int list}_< \doteq (\rho_\top)\ \mu t.\texttt{Nil} + \texttt{Cons}\langle x_1 : \texttt{int}, x_2 : (\rho_<^{x_1})\ t\rangle$$

$$\texttt{int list}_{\neq} \doteq (\rho_\top)\ \mu t.\texttt{Nil} + \texttt{Cons}\langle x_1 : \texttt{int}, x_2 : (\rho_{\neq}^{x_1})\ t\rangle$$

that is, that the list of strictly increasing integers is a subtype of the list of distinct integers. To see why, observe that after applying the trivial top-level recursive refinement $\rho_\top$ to the body, substituting the shapes, and applying the [<:-SUM], [<:-PROD] and [<:-REFLEX] rules, the check reduces to (2.7).

Local Subtyping. Our recursive refinements represent universally quantified properties over the elements of a structure. Hence, we reduce subtyping between two recursively-refined structures to *local* subtype checks between corresponding elements of two *arbitrarily* chosen values of the recursive types. The judgment [<:-REC] carries out this reduction via ρ-application and unfolding and enforces the local subtyping requirement with the final antecedent.

Well-foundedness. Although the rule [<:-REC] unfolds the bodies of the recursive types, our subtyping check is well-founded, *i.e.,* terminates, as the $\mu-$bound variable is substituted with the shape, not the dependent recursive type. Thus, our system unfolds the recursive type twice; once to apply the outer recursive refinement, and once to apply the inner recursive refinements after unfolding.

Fresh names. Note that the ρ-application introduces fresh names for the "top-level" elements of the recursive type. However, as with the names of formal parameters in function types, the fresh names are unified using substitution as shown in [<:-PROD].

Typing. We now turn to the rules for type checking expressions. We start with the rules for matching (unfolding) and construction (folding) recursive values.

Match and Unfold. Rules [L-UNFOLD] and [L-MATCH] describe how values *extracted* from recursively constructed values are type checked. First, [L-UNFOLD] is applied, which performs one unfolding of the recursive type, as described above. Second, the rule [L-MATCH] is used on the resulting sum type. This rule stipulates that the entire expression has some type $\hat{\tau}$ if the type is well-formed in the current environment, and that, for each case of the match (*i.e.,* for each element of the sum), the body expression has type $\hat{\tau}$ in the environment extended with the corresponding match bindings.

Example. Consider the following function:

```
let rec sortcheck xs =
  match xs with
  | x::(x'::_ as xs') ->
     assert (x <= x'); sortcheck xs'
  | _ -> ()
```

Let Γ be $\texttt{xs}:\alpha\ \texttt{list}_{\leq}$. From rule [L-UNFOLD], we have

$$\Gamma \vdash_{\mathbb{Q}} \texttt{unfold}\,\texttt{xs} : \texttt{Nil} + \texttt{Cons}\langle \texttt{x}:\alpha;\texttt{xs}':(\rho^{\texttt{x}}_{\leq})\ \alpha\ \texttt{list}_{\leq}\rangle$$

For clarity, we assume that the fresh names are those used in the match-bindings. For the $\texttt{Cons}$ pattern in the outer match, we use the rule [L-MATCH-M] to get the environment Γ', which is Γ extended with $\texttt{x}:\alpha$, $\texttt{xs}':(\rho^{\texttt{x}}_{\leq})\ \alpha\ \texttt{list}_{\leq}$. Thus, [L-UNFOLD] yields

$$\Gamma' \vdash_{\mathbb{Q}} \texttt{unfold}\,\texttt{xs}' : \texttt{Nil} + \texttt{Cons}\langle \texttt{x}' : \{\nu : \ \alpha \ \mid \ \texttt{x} \leq \nu\}; \texttt{xs}'' : \ldots\rangle$$

Hence, in the environment:

$$\texttt{xs}:\alpha\ \texttt{list}_{\leq};\texttt{x}:\alpha;\texttt{xs}':(\rho^{\texttt{x}}_{\leq})\ \alpha\ \texttt{list}_{\leq};\texttt{x}':\{\nu : \ \alpha \ \mid \ \texttt{x} \leq \nu\}$$

which corresponds to the extension of Γ' with the (inner) pattern-match bindings, the argument type $\{\nu = \texttt{x} \leq \texttt{x}'\}$ is a subtype of the input type $\{\nu\}$ and system verifies that the $\texttt{assert}$ cannot fail.

Construct and Fold. Rules [L-SUM] and [L-FOLD] describe how recursively constructed values are type checked. First, [L-SUM] is applied, which uses the constructor $\texttt{C}_j$ to determine which sum type the tuple should be injected into. Notice that, in the resulting sum, the refinement predicate for all other sum elements is $\perp$, capturing the fact that $\texttt{C}_j$ is the only inhabited constructor within the sum value. Second, the rule [L-FOLD] folds the (sum) type into a recursive type.

This is analogous to the classical fold [43] for iso-recursive types, which checks that the unfolded version of the typeis in fact the type of the expression prior to applying the $\texttt{fold}$ operator [43].

Example. Consider the expression $\texttt{Cons}(\texttt{i},\texttt{is})$ which is returned by the function $\texttt{range}$

from Figure 2.1. Let $\Gamma \doteq \mathtt{i}:\mathtt{int};\mathtt{is}:(\rho_{\leq}^{\mathtt{i}+1})\ \mathtt{int\ list}_{\leq}$. Rule [<:-REFINE] yields:

$$\Gamma;x_1':\{\mathtt{i}=\nu\} \vdash \{\mathtt{i}+1 \leq \nu\} <: \{x_1' \leq \nu \wedge i \leq \nu\}$$

Consequently, using the rules for subtyping, we derive:

$$\Gamma;x_1':\{\mathtt{i}=\nu\} \vdash (\rho_{\leq}^{\mathtt{i}+1})\ \mathtt{int\ list}_{\leq} <: (\rho')\ \mathtt{int\ list}_{\leq} \tag{2.9}$$

where ρ' is $\langle\langle\rangle;\langle x_1' \leq \nu \wedge i \leq \nu;\top\rangle\rangle$. Using [L-PROD]:

$$\Gamma \vdash_{\mathbb{Q}} (\mathtt{i},\mathtt{is}) : \langle x_1':\{\mathtt{i}=\nu\};x_2':(\rho')\ \mathtt{int\ list}_{\leq}\rangle$$

and so, using the subsumption rule [L-SUB] and (2.9) we have:

$$\Gamma \vdash_{\mathbb{Q}} (\mathtt{i},\mathtt{is}) : \langle x_1':\{\mathtt{i}\leq\nu\};x_2':(\rho')\ \mathtt{int\ list}_{\leq}\rangle$$

i.e., the first element of the pair is greater than `i` and the second element is an increasing list of values greater than than the first element and `i`. Thus, applying [L-SUM], we get:

$$\Gamma \vdash_{\mathbb{Q}} \mathtt{Cons}(\mathtt{i},\mathtt{is}) : \mathtt{Nil}\ +\ \mathtt{Cons}\ \langle x_1':\{\mathtt{i}\leq\nu\};x_2':(\rho')\ \mathtt{int\ list}_{\leq}\rangle$$

As $(\rho_{\leq}^{\mathtt{i}})\ \mathtt{int\ list}_{\leq}$ unfolds to the above type, [L-FOLD] yields

$$\Gamma \vdash_{\mathbb{Q}} \mathtt{fold}(\mathtt{Cons}(\mathtt{i},\mathtt{is})) : (\rho_{\leq}^{\mathtt{i}})\ \mathtt{int\ list}_{\leq}$$

i.e., `range` returns an increasing list of integers greater than `i`.

2.5.3 Type Inference

Next, we summarize the main issues that had to be addressed to extend the liquid type inference algorithm of [52] to the setting of recursive refinements. For details, see [51].

Polymorphic Recursion. Recall the function `insert` from Figure 2.2. Suppose that `ys` is an increasing list, *i.e.,* it has the type $\alpha\ \texttt{list}_{\leq}$. In the case when `ys` is not empty, and `x` is not less than `y`, the system must infer that the list $\texttt{Cons}(\texttt{y}, \texttt{insert x ys}')$ is an increasing list. To do so, it must reason that the recursive call to `insert` returns a list of values that are (a) increasing and, (b) greater than `y`. Fact (a) can be derived from the (inductively inferred) output type of `insert`. However, fact (b) is specific to this particular call site – `y` is not even in scope at other call sites, or at the point at which `insert` is defined. Notice, though, that the branch condition at the recursive call tells us that the first parameter passed to `insert`, namely, `x`, is greater than `y`. As `y` and $\texttt{ys}'$ are the head and tail of the increasing list `ys`, we know also that every element of $\texttt{ys}'$ is greater than `y`. The ML type of `insert` is $\forall\alpha.\alpha\rightarrow\alpha\ \texttt{list}\rightarrow\alpha\ \texttt{list}$. By instantiating α at this call site with $\{\nu : \alpha \mid \texttt{y} \leq \nu\}$, we can deduce fact (b).

This kind of reasoning is critical for establishing recursive properties. In our system it is formalized by the combination of [L-INST], a rule for dependent polymorphic instantiation, and [L-FIX], a dependent version of Mycroft's rule [39] that allows the instantiation of the polymorphic type of a recursive function within the body of the function. Although Mycroft's rule is known to render ML type inference undecidable [25], this is not so for our system, as it conservatively extends the ML type system. In other words, since only well-typed ML programs are typable in our system, we can use Milner's rule to construct an ML type derivation tree in a first pass, and subsequently use Mycroft's rule and the generalized types inferred in the first pass to instantiate dependent types at (polymorphic) recursive call sites.

Conjunctive Templates. Recall from Section 2.4 that polymorphic instantiation using Ins (rule [L-INST]) results in types whose refinements are a conjunction of the refinements applied to the original type variables (*i.e.,* α) within the scheme (*i.e.,* S) and the top-level refinements applied to the type used for instantiation (*i.e.,* τ). Similarly, the normalizing of recursive refinements (*i.e.,* collapsing adjacent refinements $(\rho)(\rho')$) results in new refinements that conjoin refinements from ρ and ρ'. Due to these mechanisms, the type inference engine must solve implication constraints over *conjunctive refinement templates* described by the grammar

$$\begin{array}{lcll} \theta & ::= & \varepsilon \mid [x \mapsto e];\theta & \text{(Pending Substitutions)} \\ L & ::= & e \mid \theta \cdot \kappa \wedge L & \text{(Refinement Template)} \end{array}$$

where κ are *liquid type variables* [52]. We use the fact that

$$P \Rightarrow (Q \wedge R) \text{ is valid iff } P \Rightarrow Q \text{ and } P \Rightarrow R \text{ are valid}$$

to reduce each implication constraint over a conjunctive template into a set of constraints over simple templates (with a single conjunct). The iterative weakening algorithm from [52] suffices to solve the reduced constraints, and hence infer liquid types.

3. Liquid Type Restriction. In our system, two key ingredients make *inference* decidable. First, we have ensured that *all* dependent constraints are captured explicitly using refinement predicates. Second, we have ensured that for certain critical kinds of expressions, for which types must be inferred (as they cannot be locally synthesized [44]), the types are *liquid*, *i.e.,* all refinement predicates are conjunctions of logical qualifiers. This ensures that the space of possible refinements is finite and can be efficiently searched, thereby making inference decidable.

1. Path-sensitivity. In order to precisely check and infer properties of recursive values, it is critical that the system track *explicit* value flow due to name bindings, as well as *implicit* value relationships effected through the use of branches in if-then-else expressions and match-with expressions. Our system captures explicit value flow via subtyping and implicit relationships via environment guard predicates. For example, in the `insert` function from Figure 2.2, the comparison is crucial for establishing that in the `then` branch, as `y` is greater than `x`, and all the elements of $\mathtt{ys}'$ are greater than `y` (as $\mathtt{Cons}(\mathtt{y}, \mathtt{ys}')$ is increasing), it must be that all the elements of $\mathtt{ys}'$ are greater than `x` and so $\mathtt{Cons}(\mathtt{x}, \mathtt{ys}')$ is increasing.

2.6 Summary

In this chapter we introduced Recursive Refinements, an invariant encoding mechanism that places quantifier-free logical refinements within the algebraic structure of recursive ADTs. We then provided an overview demonstrating their use and a formalism for reasoning about their expressiveness. Finally, we discussed the use of type inference to enable automatic program verification using recursive ADTs augmented with Recursive Refinements.

However, we note that Recursive Refinements are not expressive enough to represent properties that cannot be recursively, transitively defined, properties that are non-boolean, or properties that require the use of existentional quantification *i.e.,* properties that are true of one or more, but not all values in a structur.

In Chapter 3 we introduce Measures and Polymorphic Refinements; We then show how Measures allow encoding of potentially non-recursive, non-transitive and non-boolean properties. Next, we show how Polymorphic Refinements allow encoding of properties requiring exponential quantification. Further, we show that these mechanisms both admit Liquid Type Inference for automatic program verification. Finally, we present our experimental results using DSOLVE, a static program verification tool that uses Recursive Refinements, Measures and Polymorphic Refinements to automatically verify the correctness of a number of challenging data structure benchmarks.

Acknowledgements

This chapter contains material adapted from the publication: Ming Kawaguchi, Patrick Rondon, Ranjit Jhala. “Type-Based Data Structure Verification”, *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 159–169, 2009. The dissertation author was principal researcher and author on this publication. This chapter also contains material adapted from the publication: 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. The dissertation author was a co-author on this publication.

Chapter 3

Measures and Polymorphic Refinements

3.1 Measures: Introduction

Motivation. In practice, data structure invariants are not always “well formed” in that they cannot always be recursively, transitively defined, or they may not be strict invariants at all; to check many data structure manipulating programs it turns out that we often need to reason about properties of data structures which are not boolean, and may be represented by nearly arbitrary computations over values.

Hence, in this chapter, we present a mechanism we call Measures that can be used to specify “programmatic” properties of structures. That is, Measures encode properties of structures that require non-declarative computational expressiveness. These properties may not be transitive, if they are even inductive, and they are likely to be non-boolean.

Measures. Invariants may rely on non-boolean valued properties of structures or boolean valued properties which cannot be inductively, transitively described with recursive refinements. For example, an invariant on lists may refer to the *length* of the list, or an invariant on trees may refer to the *height*. Worse, there are common invariants that rely on complex

```
measure len =
  | []    -> 0
  | x::xs -> 1 + len xs
```

Figure 3.1: Measure for the length of a list

definitions such as tree *balanced-ness*, meaning the set of subtrees both have something called a height, and there is a strict inequality on these heights. In fact, these are not uncommon properties; in the last chapter we saw that recursive refinements were unable to describe the properties of an elementary red black tree due to the alternation of *red* and *black* nodes.

To implement this mechanism within the regime of quantifier-free first order logic, we use the logic of uninterpreted functions combined with a loose restriction on the domain of expressible properties such that the properties can be overapproximated within the logic.

For example, consider the example of a length properties over lists. We require the programmer to state her intention of using such a property by defining what we call a *measure*. Then, she need simply state a short *measure annotation*, specifically that shown in Figure 3.1, within the specification file inductively expressing the list length property.

The crux of this methodology is that we restrict the language used in measure definitions such that they can be easily embedded within our decidable `EUFA` refinement embedding framework.

However, this alone is not enough to allow an automated prover to reason about such a function. Hence, at appropriate points, we insert witnesses skolemized about values of type $\mu t.\texttt{Nil} + \texttt{Cons}\langle x_1 : \texttt{int}, x_2 : t\rangle$ into prover queries. The combination of skolemization of witnesses and restriction of definition allows for tractable automated proving despite the use of potentially unbounded data structures.

As usual, an astute reader will notice that such definitions can cause a refinement

```
let rec insert x ys =
  match ys with
  | [] -> [x]
  | y::ys' ->
     if x < y then x::y::ys'
     else y::(insert x ys')

let rec insertsort xs =
  match xs with
  | [] -> []
  | x::xs' ->
     insert x (insertsort xs')
```

Figure 3.2: Insertion Sort

type system to be unsound, even with careful application of witnesses such that any proof that `length` *a* is true requires that the measure defined above actually be true.

In fact, measures are simply approximations of program functions that may or may not exist in the programmer's written code. If the code for such a function `length` does exist, then some measure definitions would result in an unsound type system.

Informally, we note that, for well-founded recursive definitions and quantifier-free overapproximations made in a restricted pattern-matching syntax, measures *can* be defined and used soundly, we will not prove such a theorem.

Hence, it is arguably more important to note that measures need *not* be sound. They are simply user-defined uninterpreted functions whose user-defined definitions are applied at a fixed finite set of points as part of proof obligations.

3.2 Measures: Overview

Structure Refinements via Measures. Suppose we wish to check that `insertsort`'s output list has the same *set of elements* as the input list. We first specify what we mean

```
measure len =
  | []    -> 0
  | x::xs -> 1 + len xs
```

Figure 3.3: Measure for the length of a list

```
measure elts =
  | []    -> empty
  | x::xs -> union (single x) (elts xs)
```

Figure 3.4: Measure for the set of elements in a list

by "the set of elements" of the list using a *measure*, an inductively-defined, terminating function that we can soundly use in refinements. The measure in Figure 3.4 specifies the set of elements in a list, where `empty`, `union` and `single` are primitive constants corresponding to the respective set values and operations. Using just the measure specification and the SMT solver's decidable theory of sets our system infers that `insertsort` has the type:

$$\texttt{xs}{:}\alpha\ \texttt{list}{\rightarrow}\{\nu:\ \alpha\ \texttt{list}_{\leq}\ \mid\ \texttt{elts}\ \nu = \texttt{elts}\ \texttt{xs}\}$$

i.e., the output list is sorted and has the same elements as the input list. In Section 3.7 we will show how properties like balancedness can be verified by combining measures (to specify heights) and recursive refinements (to specify balancedness at each level).

Suppose we wish to check that the output list has the same number of elements as the input list. To do so, we first specify what we mean by "the number of elements" of the list using a *measure*: a set of expressions which defines, for each constructor, the measure of that constructor in terms of its arguments. Recall the measure for the length of a list from Figure 3.1, with

$$\mathbb{Q} = \{\texttt{len}\ \nu = \texttt{len}\ \star, \texttt{len}\ \nu = 1 + \texttt{len}\ \star\}$$

Using just the measure specification, our system infers that `insert` and `insertsort` have the types:

$$\mathtt{x}:\alpha\rightarrow\mathtt{ys}:\alpha\ \mathtt{list}\rightarrow\{\nu:\ \alpha\ \mathtt{list}\ \mid\ \mathtt{len}\ \nu = 1+\mathtt{len}\ \mathtt{ys}\}$$

$$\mathtt{xs}:\alpha\ \mathtt{list}\rightarrow\{\nu:\ \alpha\ \mathtt{list}\ \mid\ \mathtt{len}\ \nu = \mathtt{len}\ \mathtt{xs}\}$$

and thus prove that the number of elements in the list returned by `insertsort` list is the same as that of the input lis

Summary. In this section, we introduced measures, a mechanism for using refinement type inference over the logic including uninterpreted functions that can be used as a knob to accomplish one or both of the following tasks: First, measures can be used to universally embed complex inductive properties *present in the code* about recursive structures into refinement types. Second, measures can be used to *assume* complex inductive properties do hold over recursive structures, and then to embed that assumption into refinement types. As we note above, when measures are used to assume inductive properties that are either non-total, not well-founded, or do not have witnesses in the program code, the measures mechanism is a knob for introducing *unsoundness* into the verification tool. One can think of this as an ”escape hatch“ for proving properties in an, *e.g.* rely-guarantee model in which verification is not whole-program; further, trusted assumptions [17] can be combined with *e.g.* hybrid type checking [21] to defer static checking of inductive properties that may be intractable at compile-time as checks performed at runtime.

However, Measures cannot express existential properties. In the sequel, we introduce Polymorphic Refinements to express properties that require existential statements, such as properties that relate the keys and correspondent values in map structures.

3.3 Polymorphic Refinements: Introduction

Motivation. As was our motivation for formulating measures, data structures are not always "well formed". However, in this section we consider data structures that cannot always be recursively , or otherwise defined by an ADT. For example, arrays, vectors and hashtables, while they may be textbook parts of a programmer's toolkit, are not definable using ADTs.

In this chapter, we present a mechanism that we call "Polymorphic Refinements" which can be used to specify properties of linked structures which *may not* have or maintain an inductive or recursive structure, and some which may even be mutable such as hashtables.

Polymorphic Refinements. In ML, arbitrary linked data structures are generally represented using maps. In fact, as every student learns an elementary principle of maps and graphs is that an appropriately interpreted map can represent any arbitrary directed graph using a simple adjacency list scheme.

However, arbitrary maps are difficult to reason about for several reasons. First, most invariants over such structures are *existential* in nature. To wit, consider a directed graph which has the property that it is acyclic. Then, it is an elementary result that there exists a strict ordering among the elements of the adjacency map, and in practice many directed acyclic graphs are constructed in programs by adding adjacent edges only when a strict ordering holds. To represent this invariant over a map, we would like to say the following: "each element of this map is adjacent only to elements which are greater than the element". However, expressing such invariants using predicate logic is quite tricky. As it turns out, it is most expedient to simply say of every list in the adjacency map that there exists some integer, say `n`, such that every item in the list is greater than `n`, then we have expressed our invariant.

The problem with this scheme is that, much like universal quantification, existential quantification is quite difficult to reason about automatically. In fact, recall that negation carries through a universal quantifier by turning it into an existential and v.v. Hence, just as we obviated universal quantifiers using recursive refinements, we would like to concoct a mechanism to obviate existentials as well. Further, we would like to formulate a mechanisms to both specify and check existential properties.

To this end, we present *Polymorphic Refinements*, a means of specifying existential invariants which has exactly these two properties. First, it is easy to specify existential properties, adding only a small amount of type annotation to polymorphic type variable declarations and instantiations which represent user intent. Second, every existential invariant specified using our simple scheme has decidable satisfaction and is hence checkable.

The key idea behind this mechanism is that when an existentially quantified variable is introduced, the refinement it is introduced in must state that it has a *polymorphic* existential introduction. Then, the elimination rules for such quantified variables *must contain a skolem variable* such that the result of substituting the quantified variable with the skolem results in a valid type.

More succinctly, if one introduces an existentially quantified variable to use in refinement types, every usage of that variable *must* contain a witness for which the refinement holds. As usual, we provision our mechanism such that annotations are checked using our standard solving mechanism.

Again, the key idea is that every variable which may be existentially quantified in a refinement type must have been declared in an outer scope over which the existentially quantified variable is meant to refer to. Further, each usage of the quantified variable in a type must include a skolem which is a valid witness.

The astute reader will notice that only trivial existentials can be introduced; this is not a mistake! In fact, the motivation for our design is due to observation of how existentially quantified properties are used in practical program verification. Most often, a variable is only existentially quantified for one of two reasons: first, because the actual program variable which a quantified variable refers to is inconveniently placed syntactically in the program text; Second, because a quantified variable may refer to multiple actual program variables due to parametric polymorphism.

Chapter Summary. Hence, this chapter will have the following form. First, we will give an overview of how to use these two mechanisms to prove challenging properties over linked structures that may or may not be definable using recursive ADTs. Then, we will redefine our toy *ML*-like language $NanoML_{ds}$ and give its type systems polymorphic refinements and measures using a simple and natural specification syntax very similar to that shown above. Once we have formally defined our mechanisms, we will show how how to check them and, as usual, how to use the liquid types framework to infer properties involving them. Finally, we will describe our experimental evaluation of all three mechanisms: Recursive Refinements, Polymorphic Refinements, and Measures, working in harmony to prove challenging "high-level" properties on data structure-related benchmarks in OCAML.

3.4 Polymorphic Refinements: Overview

Maps. In the last section, we discussed how finite maps, such as arrays, vectors, hash tables, *etc.* are an essential tool in any programmer's toolbox for representing arbitrary graphs. The classical way to model maps is using the array axioms [37], and by using algorithmically problematic, universally quantified formulas to capture properties over *all* key-value bindings.

```
let fib i =
  let rec f t0 n =
    if mem t0 n then
      (t0, get t0 n)
    else if n <= 2 then
      (t0, 1)
    else
      let (t1,r1) = f t0 (n-1) in
      let (t2,r2) = f t1 (n-2) in
      let r       = r1 + r2 in
      (set t2 n r, r) in
  snd (f (new 17) i)
```

Figure 3.5: Memoization

```
let rec build_dag (n, g) =
  let node = random () in
  if (0>node || node>=n) then
    (n, g)
  else
    let succs  = get g node in
    let succs' = (n+1)::succs in
    let g'     = set g node succs' in
    build_dag (n+1, g')

let g0     = set (new 17) 0 []
let (_,g1) = build_dag (1, g0)
```

Figure 3.6: Acyclic Graph

Polymorphic refinements allow the implicit representation of universal map invariants within the type system and provide a strategy for generalizing and instantiating quantifiers in order to verify and infer universal map invariants.

To this end, we extend the notion of parametric polymorphism to include refined polytype variables and schemas. Using polymorphic refinements, we can give a polymorphic map the type $(\texttt{i}\!:\!\alpha,\beta)$ `Map.t`. Intuitively, this type describes a map where every key `i` of type α is mapped to a value of type β and β can refer to `i`. For example, if we instantiate α and β with `int` and $\{\nu : \texttt{int} \mid 1 < \nu \wedge \texttt{i} - 1 < \nu\}$, respectively, the resulting type describes a map from integer keys `i` to integer values strictly greater than 1 and $\texttt{i} - 1$.

Memoization. Consider the memoized fibonacci function `fib` in Figure 3.5. The example is shown in the SSA-converted style of Appel [5], with `t0` being the input name of the memo table, and `t1` and `t2` the names after updates. To verify that `fib` always returns a value greater than 1 and (the argument) $\texttt{i} - 1$, we require the universally quantified invariant that every key `j` in the memo table `t0` is mapped to a value greater than 1 and $\texttt{j} - 1$. Using the qualifiers $\{1 \leq \nu, \star - 1 \leq \nu\}$, our system infers that the polytype variables α and β can be instantiated as described above, and so the map `t0` has type $(\texttt{i}\!:\!\texttt{int}, \{\nu : \texttt{int} \mid 1 \leq \nu \wedge \texttt{i} - 1 \leq \nu\})$ `Map.t`, *i.e.,* every integer key *i* is mapped to a value greater than 1 and $\texttt{i} - 1$. Using this, the system infers that `fib` has type $\texttt{i}\!:\!\texttt{int} \rightarrow \{\nu : \texttt{int} \mid 1 \leq \nu \wedge i - 1 \leq \nu\}$.

Directed Graphs. Consider the function `build_dag` from Figure 3.6, which represents a directed graph with the pair $(\texttt{n}, \texttt{g})$ where `n` is the number of nodes of the graph, and `g` is a map that encodes the link structure by mapping each node, an integer less than `n`, to the list of its directed successors, each of which is also an integer less than `n`. In each iteration, the function `build_dag` randomly chooses a node in the graph, looks up the map to find its successors, creates a new node $\mathtt{n}+1$ and adds $\mathtt{n}+1$ to the successors of the node in the graph. As each node's successors are greater than the node, there can be no cycles in the graph. Using just the qualifiers $\{\nu \leq \star, \star < \nu\}$ our system infers that the function `build_dag` inductively constructs a directed acyclic graph. Formally, the system infers that `g1` has type

$$DAG \doteq (\mathtt{i}\!:\!\mathtt{int}, (\langle\langle\rangle;\langle \mathtt{i} < \nu;\top\rangle\rangle)\ \mathtt{int\ list})\ \mathtt{Map.t} \tag{3.1}$$

which specifies that each node `i` has successors that are *strictly greater* than `i`. Thus, by combining recursive and polymorphic refinements over simple quantifier-free predicates, our system infers complex shape invariants about linked structures.

3.5 NanoML_{ds} Base Language: Extended

We now describe how the toy language *NanoML* is extended to capture invariants of finite maps and complex non-boolean properties by describing the syntax and static semantics of Polymorphic Refinements and Measures. The syntax of expressions and dynamic semantics are unchanged. Note that we have reprinted the syntax for NanoML_{ds} for reference and highlighted the changes to the static syntax needed to support Polymorphic Refinements and Measures.

3.5.1 Syntax

Figure 3.7 shows how the syntax of types is extended to include polymorphically refined types (in addition to standard polymorphic types). First, type schemas can be universally quantified over *polyrefined polytype variables* $\alpha\langle x\!:\!\mathtt{typ}\rangle$. Second, the body of

a	$::=$		*Specification:*
	$\mid$	$\texttt{measure}\, x : B \rightarrow B = p$	
p	$\mid$	$p \mid p$	
p	$\mid$	$c \rightarrow ep$	
	$\mid$		
e	$::=$		*Expressions:*
	$\mid$	x	variable
	$\mid$	$\texttt{c}$	constant
	$\mid$	$\lambda x.e$	abstraction
	$\mid$	$x\, x$	application
	$\mid$	$\texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e$	if-then-else
	$\mid$	$\texttt{let}\, x\ =\ e\, \texttt{in}\, e$	let-binding
	$\mid$	$\texttt{fix}\, x.e$	fixpoint
	$\mid$	$[\Lambda\alpha]e$	type-abstraction
	$\mid$	$[\mathsf{typ}]e$	type-instantiation
Q	$::=$		*Liquid Refinements*
	$\mid$	$\top$	true
	$\mid$	q	qualifier in $\mathbb{Q}^{\star}$
	$\mid$	$Q \wedge Q$	conjunction
B	$::=$		*Base:*
	$\mid$	$\texttt{int}$	integers
	$\mid$	$\texttt{bool}$	booleans
	$\mid$	α	type variable
$\mathbb{A}(\mathbb{B})$	$::=$		*Unrefined Skeletons:*
	$\mid$	B	base
	$\mid$	$x{:}\mathbb{T}(\mathbb{B}){\rightarrow}\mathbb{T}(\mathbb{B})$	function
$\mathbb{T}(\mathbb{B})$	$::=$		*Refined Skeletons:*
	$\mid$	$\alpha[x \mapsto y]$	**polyrefined polytype variable**
	$\mid$	$\{\nu :\ \mathbb{A}(\mathbb{B})\ \mid\ \mathbb{B}\}$	refined type
$\mathbb{S}(\mathbb{B})$	$::=$		*Type Schema Skeletons:*
	$\mid$	$\mathbb{T}(\mathbb{B})$	monotype
	$\mid$	$\forall\alpha.\mathbb{S}(\mathbb{B})$	polytype
	$\mid$	$\forall\alpha\langle x{:}\mathsf{typ}\rangle.\mathbb{S}(\mathbb{B})$	**polyrefined polytype schema**
typ, σ	$::=$	$\mathbb{T}(\top), \mathbb{S}(\top)$	*Types, Schemas*
τ, S	$::=$	$\mathbb{T}(E), \mathbb{S}(E)$	*Depend. Types, Schemas*
$\hat{\tau}, \hat{S}$	$::=$	$\mathbb{T}(Q), \mathbb{S}(Q)$	*Liquid Types, Schemas*

Figure 3.7: *NanoML$_{ds}$*: **Syntax**

the type schema can contain *polyrefined polytype variable instances* $\alpha[x \mapsto y]$.

Intuitively, the quantification over $\alpha\langle x\!:\!\mathsf{typ}\rangle$ indicates that α can be instantiated with a refined type containing a free variable x of type typ. It is straightforward to extend the system to allow multiple free variables; we omit this for clarity. A refined polytype instance $\alpha[x \mapsto y]$ is a standard polymorphic type variable α with a *pending substitution* $[x \mapsto y]$ that gets applied *after* α is instantiated.

Note that in fact a polytype variable instance *is* inherently existential. That is, the polytype variable instance is syntactic sugar for $\exists x.\{\nu :\ \alpha\ \mid\ y = x\}$. That is, y serves as a *witness* for the existentially bound x. To keep the quantification implicit, the instantiation function Ins eagerly applies the pending substitution whenever each polytype variable is instantiated. When the polytype variable is instantiated with a type containing other polytype instances, it suffices to telescope the pending substitutions by replacing $\alpha[x \mapsto x_1][x_2 \mapsto y]$ with $\alpha[x \mapsto y]$ if $x_1 \equiv x_2$ and with $\alpha[x \mapsto x_1]$ otherwise.

Example. The following refined polytype schema signature specifies the behavior of the key operations of a *finite dependent map*.

$$\mathtt{new} : \forall \alpha, \beta\langle x\!:\!\alpha\rangle.\mathtt{int} \to (\mathtt{i}\!:\!\alpha, \beta[x \mapsto \mathtt{i}])\ \mathtt{t}$$

$$\mathtt{set} : \forall \alpha, \beta\langle x\!:\!\alpha\rangle.(\mathtt{i}\!:\!\alpha, \beta[x \mapsto \mathtt{i}])\ \mathtt{t} \to \mathtt{k}\!:\!\alpha \to \beta[x \mapsto \mathtt{k}] \to (\mathtt{j}\!:\!\alpha, \beta[x \mapsto \mathtt{j}])\ \mathtt{t}$$

$$\mathtt{get} : \forall \alpha, \beta\langle x\!:\!\alpha\rangle.(\mathtt{i}\!:\!\alpha, \beta[x \mapsto \mathtt{i}])\ \mathtt{t} \to \mathtt{k}\!:\!\alpha \to \beta[x \mapsto \mathtt{k}]$$

$$\mathtt{mem} : \forall \alpha, \beta\langle x\!:\!\alpha\rangle.(\mathtt{i}\!:\!\alpha, \beta[x \mapsto \mathtt{i}])\ \mathtt{t} \to \mathtt{k}\!:\!\alpha \to \mathtt{bool}$$

In the signature, `t` is a polymorphic type constructor that takes two arguments corresponding to the types of the keys and values stored in the map. If the signature is implemented using association lists, the type $(\mathtt{i}\!:\!\alpha, \beta[x \mapsto \mathtt{i}])\ \mathtt{t}$ is an abbreviation for $\langle \mathtt{i}\!:\!\alpha, \beta[x \mapsto \mathtt{i}]\rangle\ \mathtt{list}$.

In general, this signature can be implemented by any appropriate data structure, such as balanced search trees, hash tables, *etc.* The signature specifies that certain relationships must hold between the keys and values when new elements are added to the map (using `set`), and it ensures that subsequent reads (using `get`) return values that satisfy the relationship. For example, when α and β are instantiated with `int` and $\{\nu : \texttt{int} \mid x \leq \nu\}$ respectively, we get a finite map $(\texttt{i}:\texttt{int}, \{\nu : \texttt{int} \mid \texttt{i} \leq \nu\})\ \texttt{t}$ where, *for each* key-value binding, the value is greater than the key. For this map, the type for `set` ensures that when a new binding is added to the map, the value to be added has the type $\{x \leq \nu\}[x \mapsto \texttt{k}]$, which is $\{\texttt{k} \leq \nu\}$, *i.e.,* is greater than the key `k`. Dually, when `get` is used to query the map with a key `k`, the type specifies that the returned value is guaranteed to be greater than the key `k`.

Measures. As we previously discussed, many properties of structures used in invariants are *non-boolean*. To allow these properties to be used in invariants, we must extend the language to facilitate a type of annotation which we explicitly call a specification. These specifications take the form of a function definition.

However, we must be careful in calling these specifications functions, as they are not exactly as expressive as functions. In particular, measures are strictly defined as functions that take values of the language's base ADTs and, via explicitly defined pattern matching expressions, output values of another base type.

Further, for soundness we require several restrictions on measure functions. First, it may be that a function with an identical name as the measure function is defined within the program itself. If an identically named function is defined within the program, then for soundness we require that the annotated measure function *overapproximate* the function defined in the program. Second, regardless of whether a function with a similar name is defined within the program, all recursive functions must be *well-founded*. Note that, although we require this in the metatheory, we do not check for these properties within the type system; these restrictions must be minded by the programmer herself.

Figure 3.7 shows the syntax of measure functions under the non-terminal a. A *measure name* $\mathtt{m}$ is a special variable drawn from a set of measure names. A *measure expression* ε is an expression drawn from a restricted language of variables, constants, measure names and application. A *measure for a recursive (ML) type* $\mu t.\Sigma_i \mathtt{C}_i\langle x_i : \mathsf{typ}_i\rangle$ of type typ is a quadruple: $(\mathtt{m}, \langle \mathtt{C}_i\langle x_i\rangle \mapsto \varepsilon_i\rangle, \mathsf{typ}, \mu t.\Sigma_i \mathtt{C}_i\langle x_i : \mathsf{typ}_i\rangle)$. We note that when measures are strictly defined by structural induction, they are well-founded and hence total for the purpose of reasoning about soundness.

3.6 *NanoML$_{ds}$* Type Checking: Extended

We now describe our extended static type system in detail, by discussing each of its judgments in turn and the treatment of both Polymorphic Refinement and Measure annotations. For brevity, we do not rehash those rules covered in Chapter 2, although we reprint the type checking rules for reference.

The structure of this section will proceed as follows. First, we will discuss the checking and static semantics of polymorphic refinements. Then, because the two mechanisms can be examined independently, we will discuss the checking and static semantics of measures. Note again that neither of these mechanisms affect the dynamic semantics of *NanoML$_{ds}$*.

3.6.1 Static Semantics

Figure 3.9 summarizes the rules for polymorphic refinements.

Well-formedness. We extend our well-formedness rules with a rule [WF-REFVAR] which

Liquid Type Checking $\boxed{\Gamma \vdash_{\mathbb{Q}} e : S}$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : S \quad \Gamma \vdash S <: S' \quad \Gamma \vdash_{\mathbb{Q}} S'}{\Gamma \vdash_{\mathbb{Q}} e : S'}\ [\text{L-SUB}]$$

$$\frac{\Gamma(x) = \{\nu : \tau \mid e\}}{\Gamma \vdash_{\mathbb{Q}} x : \{\nu : \tau \mid e \wedge \nu = x\}}\ [\text{L-VAR-B}]$$

$$\frac{}{\Gamma \vdash_{\mathbb{Q}} \mathtt{c} : ty(\mathtt{c})}\ [\text{L-CONST}]$$

$$\frac{\Gamma \vdash x : \hat{\tau}_x \rightarrow \hat{\tau} \quad \Gamma; x : \hat{\tau}_x \vdash_{\mathbb{Q}} e : \hat{\tau}}{\Gamma \vdash_{\mathbb{Q}} (\lambda x.e) : x : \hat{\tau}_x \rightarrow \hat{\tau}}\ [\text{L-FUN}]$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e_1 : x : \tau \rightarrow \tau' \quad \Gamma \vdash_{\mathbb{Q}} e_2 : \tau}{\Gamma \vdash_{\mathbb{Q}} e_1\ e_2 : \tau'[x \mapsto e_2]}\ [\text{L-APP}]$$

$$\frac{\Gamma \vdash \hat{\tau} \quad \Gamma \vdash_{\mathbb{Q}} e_1 : \mathtt{bool} \qquad \Gamma; e_1 \vdash_{\mathbb{Q}} e_2 : \hat{\tau} \quad \Gamma; \neg e_1 \vdash_{\mathbb{Q}} e_3 : \hat{\tau}}{\Gamma \vdash_{\mathbb{Q}} \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau}\ [\text{L-IF}]$$

$$\frac{\Gamma \vdash \hat{\tau} \quad \Gamma \vdash_{\mathbb{Q}} e_1 : S_1 \quad \Gamma; x : S_1 \vdash_{\mathbb{Q}} e_2 : \hat{\tau}}{\Gamma \vdash_{\mathbb{Q}} \mathtt{let}\ x\ =\ e_1\ \mathtt{in}\ e_2 : \hat{\tau}}\ [\text{L-LET}]$$

$$\frac{\Gamma \vdash \hat{S} \quad \Gamma; x : \hat{S} \vdash_{\mathbb{Q}} e : \hat{S}}{\Gamma \vdash_{\mathbb{Q}} \mathtt{fix}\ x.e : \hat{S}}\ [\text{L-FIX}]$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} e : S \qquad \alpha \notin \Gamma}{\Gamma \vdash_{\mathbb{Q}} [\Lambda\alpha]e : \forall\alpha.S}\ [\text{L-GEN}]$$

$$\frac{\Gamma \vdash \tau \quad \mathsf{Shape}(\tau) = \mathsf{typ} \quad \Gamma \vdash_{\mathbb{Q}} e : \forall\alpha.S}{\Gamma \vdash_{\mathbb{Q}} [\mathsf{typ}]e : S[\alpha \mapsto \tau]}\ [\text{L-INST}]$$

$$\frac{\Gamma \vdash (\hat{\rho})\ \mu t.\hat{\tau} \quad (\hat{\rho})\ \hat{\tau} \triangleright \hat{\tau}' \qquad \Gamma \vdash_{\mathbb{Q}} e : \hat{\tau}'[t \mapsto (\hat{\rho})\ \mu t.\hat{\tau}]}{\Gamma \vdash_{\mathbb{Q}} \mathtt{fold}\ e : (\hat{\rho})\ \mu t.\hat{\tau}}\ [\text{L-FOLD}]$$

$$\frac{(\rho)\ \tau \triangleright \tau' \quad \Gamma \vdash_{\mathbb{Q}} e : (\rho)\ \mu t.\tau}{\Gamma \vdash_{\mathbb{Q}} \mathtt{unfold}\ e : \tau'[t \mapsto (\rho)\ \mu t.\tau]}\ [\text{L-UNFOLD}]$$

$$\frac{\text{fresh } x \quad \Gamma \vdash_{\mathbb{Q}} e : \tau \quad \Gamma; x : \tau \vdash_{\mathbb{Q}} \langle e \rangle : \langle x : \tau \rangle}{\Gamma \vdash_{\mathbb{Q}} (e, \langle e \rangle) : (x : \tau, \langle x : \tau \rangle)}\ [\text{L-PRODUCT}]$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} \langle e \rangle : \langle x : \tau \rangle}{\Gamma \vdash_{\mathbb{Q}} \mathtt{C}_j\langle e \rangle : \mathtt{C}_j\langle x : \tau \rangle + \Sigma_{i \neq j}\mathtt{C}_i\langle x_i : \overline{\mathsf{typ}_i} \rangle}\ [\text{L-SUM}]$$

$$\frac{\Gamma \vdash \hat{\tau} \quad \Gamma \vdash_{\mathbb{Q}} e : \Sigma_i \mathtt{C}_i\langle x_i : \tau_i \rangle \quad \forall i :\ \Gamma; \langle x_i : \tau_i \rangle \vdash_{\mathbb{Q}} e_i : \hat{\tau}}{\Gamma \vdash_{\mathbb{Q}} \mathtt{match}\ e\ \mathtt{with}\ |_i\ \mathtt{C}_i\langle x_i \rangle \mapsto e_i : \hat{\tau}}\ [\text{L-MAT}]$$

Figure 3.8: **Liquid Type Checking Rules for** *NanoML$_{ds}$*

Well-Formed Types $\boxed{\Gamma \vdash S}$

$$\frac{\alpha\langle x\!:\!\mathsf{typ}\rangle \in \Gamma \quad \Gamma \vdash y\!:\!\mathsf{typ}}{\Gamma \vdash \alpha[x \mapsto y]}\ [\text{WF-RefVar}]$$

$$\frac{\Gamma;\alpha\langle x\!:\!\mathsf{typ}\rangle \vdash S}{\Gamma \vdash \forall\alpha\langle x\!:\!\mathsf{typ}\rangle.S}\ [\text{WF-RefPoly}]$$

Decidable Subtyping $\boxed{\Gamma \vdash S_1 <: S_2}$

$$\frac{\alpha\langle x\!:\!\mathsf{typ}\rangle \in \Gamma \quad \Gamma \vdash \{\nu : \mathsf{typ} \mid \nu = y_1\} <: \{\nu : \mathsf{typ} \mid \nu = y_2\}}{\Gamma \vdash \alpha[x \mapsto y_1] <: \alpha[x \mapsto y_2]}\ [\text{<:-RefVar}]$$

$$\frac{\Gamma;\alpha\langle x\!:\!\mathsf{typ}\rangle \vdash S_1 <: S_2}{\Gamma \vdash \forall\alpha\langle x\!:\!\mathsf{typ}\rangle.S_1 <: \forall\alpha\langle x\!:\!\mathsf{typ}\rangle.S_2}\ [\text{<:-RefPoly}]$$

Liquid Type Checking $\boxed{\Gamma \vdash_{\mathbb{Q}} e : S}$

$$\frac{\Gamma;\alpha\langle x\!:\!\mathsf{typ}\rangle \vdash_{\mathbb{Q}} e : S \quad \alpha\langle x\!:\!\mathsf{typ}\rangle \notin \Gamma}{\Gamma \vdash_{\mathbb{Q}} [\Lambda\alpha\langle x\!:\!\mathsf{typ}\rangle]e : \forall\alpha\langle x\!:\!\mathsf{typ}\rangle.S}\ [\text{L-RefGen}]$$

$$\frac{\Gamma;x\!:\!\mathsf{typ}_x \vdash \tau \quad \mathsf{Shape}(\tau) = \mathsf{typ} \quad \Gamma \vdash_{\mathbb{Q}} e : \forall\alpha\langle x\!:\!\mathsf{typ}_x\rangle.S}{\Gamma \vdash_{\mathbb{Q}} [\mathsf{typ}]e : \mathsf{Ins}(S,\alpha,\tau)}\ [\text{L-RefInst}]$$

Figure 3.9: **Polymorphic Refinements: Syntax, Static Semantics**

```
let rec get xs k =
  match xs with
  | [] -> diverge ()
  | (k',d')::xs' -> if k=k' then d' else get xs' k
```

Figure 3.10: Implementation of `get` function

states that a polytype instance $\alpha[x \mapsto y]$ is well-formed if: (a) the instance occurs in a schema quantified over $\alpha\langle x\!:\!\mathtt{typ}\rangle$, *i.e.,* where α can have a free occurrence of x of type typ, and (b) the free variable y that replaces x is bound in the environment to a type typ.

Subtyping. We extend our subtyping rules with a rule [<:-REFVAR] for subtyping refined polytype variable instances. The intuition behind the rule follows from the existential interpretation of the refined polytype instances and the fact that $\forall \nu.(\nu = y_1) \Rightarrow (\nu = y_2)$ implies $y_1 = y_2$, which implies $(\exists x.P \wedge x = y_1) \Rightarrow (\exists x.P \wedge x = y_2)$ for every logical formula P in which x occurs free. Thus, to check that the subtyping holds *for any* possible instantiation of α containing a free x, it suffices to check that the replacement variables y_1 and y_2 are equal.

Example. It is straightforward to check that each of the schemas for set, get, *etc.* are well-formed. Next, let us see how the following implementation of the get function in Figure 3.10 implements the refined polytype schema shown above. That is, let us see how our rules derive:

$$\emptyset \vdash_{\mathbb{Q}} \mathtt{get} : \forall \alpha, \beta\langle x\!:\!\alpha\rangle.(\mathtt{i}\!:\!\alpha, \beta[x \mapsto \mathtt{i}])\ \mathtt{t} \to \mathtt{k}\!:\!\alpha \to \beta[x \mapsto k]$$

From the input assumption that xs has the type $\langle \mathtt{i}\!:\!\alpha, \beta[x \mapsto \mathtt{i}]\rangle$ list, and the rules for unfolding and pattern matching ([L-UNFOLD-M] and [L-MATCH-M]) we have that, at the point where $\mathtt{d}'$ is returned, the environment Γ contains the type binding

$$\mathtt{d}'\!:\!\beta[x \mapsto \mathtt{i}][\mathtt{i} \mapsto \mathtt{k}']$$

```
let fib m =
  let rec fibmemo t0 n =
    if n <= 2 then (t0, 1) else
      if mem t0 n then (t0, get t n) else
       let (t1, r1) = fibmemo t  (n-1) in
       let (t2, r2) = fibmemo t1 (n-2) in
       (set t2 n (r1 + r2), r1 + r2) in
  snd (fibmemo (create 17) m)
```

Figure 3.11: Memoized Implementation of the Fibonacci Function.

which, after telescoping the substitutions, is equivalent to the binding

$$\mathtt{d}' : \beta[x \mapsto \mathtt{k}'].$$

Due to [L-IF], the branch condition $\mathtt{k} = \mathtt{k}'$ is in Γ, and so

$$\Gamma \vdash \{\nu : \alpha \mid \nu = \mathtt{k}'\} <: \{\nu : \alpha \mid \nu = \mathtt{k}\}.$$

Thus, from rule [<:-REFVAR], and subsumption, we derive that the `then` branch has the type $\beta[x \mapsto \mathtt{k}]$ from which the schema follows. The reasoning for checking the other functions like `set` and `mem`, and also implementations using different underlying data structures like balanced maps or hash tables is similar.

Typing. We extend the typing rules with rules that handle refined polytype generalization ([L-REFGEN]) and instantiation ([L-REFINST]). A refined polytype variable $\alpha\langle x : \mathsf{typ}\rangle$ can be instantiated with a dependent type τ that contains a free occurrence of x of type typ. This is ensured by the well-formedness antecedent for [L-REFINST] which checks that τ is well-formed in the environment extended with the appropriate binding for x. However, once τ is substituted into the body of the schema S, the different pending substitutions at each of the refined polytype *instances* of α are applied and hence *x does not appear free* in the instantiated type, which is consistent with the existential interpretation of polymorphic refinements.

The soundness of treating a refined polytype variable instantiation as an unrefined variable instantiation proceeded by a substitution follows from the fact that for all P and Q where x appears free, $(\exists x.(P \sim Q) \wedge x = y)$ is logically equivalent to $(\exists x.P \wedge x = y) \sim (\exists x.Q \wedge x = y)$ for $\sim \in \{\wedge, \vee\}$. This equivalence allows us to "push" the pending substitutions "inside" the (monomorphic) dependent type τ used for instantiation.

Example: [Checking Clients]. Consider the code in Figure 3.11, that *uses* dependent maps to implement a memoized version of the fibonacci function. Let us see how our system uses polymorphic refinements to derive that

$$\emptyset \vdash_{\mathbb{Q}} \mathtt{fib} : \mathtt{m}{:}\mathtt{int} \to \{\nu : \mathtt{int} \mid 1 \leq \nu \wedge \mathtt{m} - 1 \leq \nu\}$$

First we deduce that

$$\emptyset \vdash_{\mathbb{Q}} \mathtt{fibmemo} : \mathtt{t0}{:}\mathit{Fib}_{\mathtt{i}} \to \mathtt{int} \to \langle \mathit{Fib}_{\mathtt{i}}, \{\nu : \mathtt{int} \mid 1 \leq \nu \wedge \mathtt{n} - 1 \leq \nu\}\rangle$$

where

$$\mathit{Fib}_{\mathtt{i}} \doteq (\mathtt{i}{:}\mathtt{int}, \{\nu : \mathtt{int} \mid 1 \leq \nu \wedge \mathtt{i} - 1 \leq \nu\})\ \mathtt{t}.$$

To derive the above, we instantiate the refined polytype variables α and $\beta\langle x{:}\alpha\rangle$ in the signatures for `mem`, `get`, `set` with `int` and $\{1 \leq \nu \wedge x - 1 \leq \nu\}$ respectively, after which the rules from Section 3.5 suffice to establish the types of `fibmemo` and `fib`.

Polymorphic Refinements vs. Array Axioms. Our technique of specifying the behavior of finite maps using polymorphic refinements is orthogonal, and complementary, to the classical approach that uses McCarthy's array axioms [37]. In this approach, one models reads and writes to arrays, or, more generally, finite maps, using two operators. The first, $Sel(m,i)$, takes a *map* m and an *address* i and returns the value stored in the map at that address. The second, $Upd(m,i,v)$, takes a map m, an address i and a *value* v and returns the new map which corresponds to m updated at the address i with the new value v. The two efficiently decidable axioms

$$\forall m,i,v.\ Sel(Upd(m,i,v),i) = v$$

$$\forall m,i,j,v.\ i = j \vee Sel(Upd(m,i,v),j) = Sel(m,j)$$

specify the behavior of the operators. Thus, an analysis can use the operators to algorithmically reason about the exact contents of explicitly named addresses within a map. For example, the predicate $Sel(\texttt{m},\texttt{i}) = 0$ specifies that `m` maps the key `i` to the value 0. However, to capture invariants that hold for all key-value bindings in the map, one must use universally quantified formulas, which make algorithmic reasoning brittle and unpredictable. For example, to verify `fib` from Figure 3.11, we need the invariant

$$\forall \texttt{i}.(1 \leq Sel(\texttt{t0},\texttt{i}) \wedge \texttt{i} - 1 \leq Sel(\texttt{t0},i)).$$

Unfortunately, quantifiers make algorithmic reasoning brittle and unpredictable as ad-hoc heuristics are required to generalize and instantiate the quantifiers.

In contrast, polymorphic refinements can smoothly capture and reason about the relationship between *all* the addresses and values but do not, as described so far, let us refer to particular named addresses.

We can have the best of both worlds in our system by combining these techniques. Using polymorphic refinements, we can reason about universal relationships between keys and values and by refining the output types of `set` and `get` with the predicates $(\nu = \mathit{Upd}(\mathtt{m},\mathtt{k},\mathtt{v}))$ and $(\nu = \mathit{Sel}(\mathtt{m},\mathtt{k}))$, respectively, we can simultaneously reason about the specific keys and values in a map.

Formally, we give the `set` function the signature

$$\begin{aligned}&\mathtt{set} : \mathtt{m}{:}(\mathtt{i}{:}\alpha,\beta[x \mapsto \mathtt{i}])\ \mathtt{t}{\rightarrow}\mathtt{k}{:}\alpha{\rightarrow}\mathtt{v}{:}\beta[x \mapsto \mathtt{k}]{\rightarrow}\\ &\{\nu :\ (\mathtt{j}{:}\alpha,\beta[x \mapsto \mathtt{j}])\ \mathtt{t}\ \mid\ \nu = \mathit{Upd}(\mathtt{m},\mathtt{k},\mathtt{v})\}\end{aligned}$$

and the `get` function the signaure

$$\begin{aligned}&\mathtt{get} : \mathtt{m}{:}(\mathtt{i}{:}\alpha,\beta[x \mapsto \mathtt{i}])\ \mathtt{t}{\rightarrow}\mathtt{k}{:}\alpha{\rightarrow}\\ &\{\nu :\ \beta[x \mapsto k]\ \mid\ \nu = \mathit{Sel}(\mathtt{m},\mathtt{k})\}.\end{aligned}$$

Example: [Mutable Linked Structures]. Polymorphic refinements can be used to verify properties of linked structures, as each link field corresponds to a map from the set of source structures to the set of link targets. For example, a field `f` corresponds to a map `f`, a field read `x.f` corresponds to `get f x`, and a field write `x.f` $\leftarrow e$ corresponds to `set f x e`. Consider an SSA-converted [5] implementation of the textbook `find` function for the *union-find* data structure shown in Figure 3.12.

The function `find` takes two maps as input: `rank` and `parent0`, corresponding to the rank and parent fields in an imperative implementation, and an element `x`, and finds the "root" of `x` by transitively following the `parent` link, until it reaches an element that is its own parent. The function implements *path-compression*, *i.e.,* it destructively updates the

```
let rec find rank parent0 x =
  let px = get parent0 x in
  if px = x then (parent0, x) else
    let (parent1, px') = find rank parent0 px in
    let parent2        = set parent1 x px' in
    (parent2, px')

let union r0 p0 x y =
  let (p1, x')  = find r p  x in
  let (p2,y')   = find r p' y in
  let _         = assert (get p2 x' = x') in
  let _         = assert (get p2 y' = y') in
  if x' != y' then begin
    let rx' = get r0 x' in
    let ry' = get r0 y' in
    if rx' > ry' then
      (r0, set p2 y' x')
    else if rx' < ry' then
      (r, set p2 x' y')
    else
      let r1 = set r0 x' (rx' + 1) in
      let p3 = set p2 y' x' in
      (r1, p3)
  end else
    (r0, p2)
```

Figure 3.12: Functional Union-Find With Mutation and Path Compression

parent map so that subsequent queries jump straight to the root. The data structure maintains the acyclicity invariant that each non-root element's rank is strictly smaller than the rank of the element's parent. The acyclicity invariant of the `parent` map is captured by the type

$$(\mathtt{i}{:}\mathtt{int}, \{\nu : \mathtt{int} \mid (\mathtt{i} = \nu) \vee Sel(\mathtt{rank}, \mathtt{i}) < Sel(\mathtt{rank}, \nu)\})\ \mathtt{t}$$

which states that for each key `i`, the parent ν is such that, either the key is its own parent or the key's rank is less than the parent's rank.

Our system verifies that when `find` is called with a parent map that satisfies the invariant, the output map also satisfies the invariant. To do so, it automatically instantiates the refined polytype variables in the signatures for `get` and `set` with the appropriate refinement types, after which the rules from Section 3.5 (shown in Figure 3.9) suffice to establish the invariant. Similarly, our system verifies the `union` function where the rank of a root is incremented when two roots of equal ranks are linked. Thus, polymorphic refinements enable the verification of complex acyclicity invariants of mutable data structures.

$$UF_{\mathtt{i}} \doteq \mathtt{i} = \nu \vee Sel(\mathtt{rank}, \mathtt{i}) < Sel(\mathtt{rank}, \nu)$$

$$UF_{\mathtt{i}} \doteq (\mathtt{i}{:}\mathtt{int}, \{\nu : \mathtt{int} \mid UF_{\mathtt{i}}\})\mathtt{t}$$

The type $UF_{\mathtt{i}}$ captures the key acyclicity invariant by stating that the rank of each parent `i` is either a self-loop or a child with a strictly smaller rank. Our system verifies the invariant by proving that when `find` is called with arguments

$$\texttt{rank} : (\texttt{int},\texttt{int})\texttt{t}$$

$$\texttt{parent} : \mathit{UF}_{\texttt{i}}$$

$$\texttt{x} : \texttt{int}$$

it returns as output a pair of type

$$\langle \mathit{UF}_{\texttt{i}}, \{\nu : \texttt{int} \mid x = \nu \vee \mathit{Sel}(\texttt{rank},\texttt{x}) < \mathit{Sel}(\texttt{rank},\nu)\}\rangle$$

$$\emptyset \vdash_{\mathbb{Q}} \texttt{rank}{:}(\texttt{int},\texttt{int})\texttt{t} \rightarrow \texttt{rank}{:}\mathit{UF}_{\texttt{i}} \rightarrow \texttt{x}{:}\texttt{int} \rightarrow \langle \mathit{UF}_{\texttt{i}}, \{\nu : \texttt{int} \mid \mathit{UF}_{\texttt{x}}\}\rangle$$

That is, the function returns a pair of an updated parent map that satisfies the key invariant, and an element that is either the same as the input `x` (if `x` was a root), or whose rank exceeds that of `x`. To obtain this judgment, our system automatically instantiates the refined polytype variables α and $\beta\langle x{:}\alpha\rangle$ in the signatures for `get` and `set` with `int` and $\{\mathit{UF}_x\}$ respectively, type of `find`.

Typechecking with Measures. Figure 3.14 shows the rule used to check that a measure specification is well-formed. We assume that the specified sequence of measures $\langle M\rangle$ is such that $\emptyset \vdash \langle M\rangle$. Measures are like "ghost" or "auxiliary" variables common in program verification; they exist purely for specification and verification.

Figure 3.13 shows the rules for typechecking using measures. The only rules that are changed are those for fold, unfold, construction and pattern-matching. [L-FOLD-M] and [L-UNFOLD-M] ensure that the refinements for the recursive types are the same as those for the corresponding unfolded type. [L-SUM-M] ensures that, when a sum type is constructed, a predicate specifying the value of the measure `m` is conjoined to the refinement

Measured Type Checking $\boxed{\Gamma \vdash e : S}$

$$\frac{\Gamma \vdash (\hat{\rho})\ \mu t.\hat{\tau} \quad (\rho)\ \hat{\tau} \rhd \hat{\tau}' \qquad \Gamma \vdash_{\mathbb{Q}} e : \{\nu :\ \hat{\tau}'[t \mapsto (\hat{\rho})\ \mu t.\hat{\tau}]\ \mid\ e'\}}{\Gamma \vdash_{\mathbb{Q}} \texttt{fold}\, e : \{\nu :\ (\hat{\rho})\ \mu t.\hat{\tau}\ \mid\ e'\}}\ \text{[L-FOLD-M]}$$

$$\frac{(\rho)\ \tau \rhd \tau' \quad \Gamma \vdash e : \{\nu :\ (\rho)\ \mu t.\tau\ \mid\ e'\}}{\Gamma \vdash \texttt{unfold}\, e : \{\nu :\ \tau'[t \mapsto (\rho)\ \mu t.\tau]\ \mid\ e'\}}\ \text{[L-UNFOLD-M]}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} \texttt{C}_j\langle e\rangle : \Sigma_i \texttt{C}_i\langle x_i : \tau_i\rangle}{\Gamma \vdash_{\mathbb{Q}} \texttt{C}_j\langle e\rangle : \{\nu :\ \Sigma_i \texttt{C}_i\langle x_i : \tau_i\rangle\ \mid\ \wedge_m m(\nu) = \texttt{C}_{j,m}(\langle e\rangle)\}}\ \text{[L-SUM-M]}$$

$$\frac{\Gamma \vdash e : \Sigma_i \texttt{C}_i\langle x_i : \tau_i\rangle \quad \Gamma \vdash \tau \qquad \forall i\ \Gamma;\langle x_i : \tau_i\rangle;\wedge_m m(e) = \texttt{C}_{i,m}(\langle x_i\rangle) \vdash e_i : \hat{\tau}}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ |_i\ \texttt{C}_i\langle x_i\rangle \mapsto e_i : \hat{\tau}}\ \text{[L-MATCH-M]}$$

Figure 3.13: **Measured Type Checking Rules**

Measure Well-formedness $\boxed{\Gamma \vdash \langle M\rangle}$

$$\frac{\begin{array}{c}\Gamma;\texttt{m} : \mu t.\Sigma_i \texttt{C}_i\langle x_i : \mathsf{typ}_i\rangle \to \mathsf{typ} \vdash \langle M\rangle \\ \forall i :\ \Gamma;\texttt{m} : \mu t.\Sigma_i \texttt{C}_i\langle x_i : \mathsf{typ}_i\rangle \to \mathsf{typ};\langle x_i : \mathsf{typ}_i\rangle \vdash \varepsilon_i : \mathsf{typ}\end{array}}{\Gamma \vdash (\texttt{m}, \langle \texttt{C}_i\langle x_i\rangle \mapsto \varepsilon_i\rangle, \mathsf{typ}, \mu t.\Sigma_i \texttt{C}_i\langle x_i : \mathsf{typ}_i\rangle);\langle M\rangle}\ \text{[WF-M]}$$

Figure 3.14: **Measured Well-Formedness Rule**

for the constructed type for each measure m defined for the corresponding recursive type. [L-MATCH-M] ensures that when a sum type is accessed using a match expression, the environment is extended with a guard predicate that captures the relationship between the measure of the matched expression and the variables bound by the corresponding pattern for each case of the match.

That is, we insert witnesses to measure properties at the following points:

(a) the corresponding match bindings and (b) the guard predicate that captures the relationship between the measure of the matched expression and the variables bound by the matched pattern.

(b) for each measure m defined for the corresponding recursive type, a predicate specifying the value of the measure m is conjoined to the refinement for the constructed sum.

Example: Measuring Length. Consider the expression:

```
let a = [] in let b = 1::a in
match b with x::xs -> () | [] -> assert false
```

Using the measure for list length from Section 3.2 (shown in Figure 3.3), and rules [L-SUM-M] and [L-FOLD-M] (and [L-LET]), we can derive:

$$\emptyset \vdash_{\mathbb{Q}} \texttt{Nil} : \{\texttt{len}\ \nu = 0\}$$

$$\texttt{a} : \{\texttt{len}\ \nu = 0\} \vdash_{\mathbb{Q}} \texttt{Cons}(1, \texttt{a}) : \{\texttt{len}\ \nu = 1 + \texttt{len}\ \texttt{a}\}$$

We omit the ML type int list being refined for brevity. Due to rules [L-UNFOLD-M] and [L-MATCH-M], the assert in the Nil pattern-match body is checked in an environment Γ containing $\texttt{a}:\{\texttt{len}\ \nu = 0\}$, $\texttt{b}:\{\texttt{len}\ 1 + \texttt{len}\ \texttt{a}\}$ and the guard $(\texttt{len}\ \texttt{b} = 0)$ from the Nil case in the measure definition for len. As Γ is inconsistent, the argument type $\{\texttt{not}\ \nu\}$ is a subtype of the input type $\{\}$ under Γ, and so the call to assert type checks, proving the `Nil` case is not reachable.

These mechanisms combine to allow the specification and verification of inductive properties in a local manner. For example, to specify that a tree is *balanced*, one can define a a `height` measure and thereafter, use a recursive refinement that states that at each point in the tree, the difference between the heights of the left and right trees is bounded. The measure functions are structured so as to be locally instantiated at construction and match expressions.

3.7 Evaluation

We have implemented our type-based data structure verification techniques in DSOLVE, which takes as input an OCAML program (a `.ml` file), a property specification (a `.mlq` file), and a set of logical qualifiers (a `.quals` file). The program corresponds to a module, and the specification comprises measure definitions and types against which the interface functions of the module should be checked. DSOLVE combines the manually supplied qualifiers (`.quals`) with qualifiers scraped from the properties to be proved (`.mlq`) to obtain the set $\mathbb{Q}$ used to infer types for verification. DSOLVE produces as output the list of possible refinement type errors, and a `.annot` file containing the inferred liquid types for all the program expressions.

Benchmarks. We applied DSOLVE to the following set of benchmarks, designed to demonstrate: *Expressiveness*—that our approach can be used to verify a variety of complex properties across a diverse set of data structures, including textbook structures, and structures designed for particular problem domains; *Efficiency*—that our approach scales to large, realistic data structure implementations; and *Automation*—that, due to liquid type inference, our approach requires a small set of manual qualifier annotations.

- `List-sort`: a collection of textbook list-based sorting routines, including insertion-sort, merge-sort and quick-sort,
- `Map`: an ordered AVL-tree based implementation of finite maps, (from the OCAML standard library)
- `Ralist`: a random-access lists library, (due to Xi [64])
- `Redblack`: a red-black tree insertion implementation (without deletion), (due to Dunfield [18])
- `Stablesort`: a tail recursive mergesort, (from the OCAML standard library)
- `Vec`: a tree-based vector library (due to de Alfaro [14])
- `Heap`: a binary heap library, (due to Filliâtre [20])
- `Splayheap`: a splay tree based heap, (due to Okasaki [42])
- `Malloc`: a resource management library,
- `Bdd`: a binary decision diagram library (due to Filliâtre [20])
- `Unionfind`: the textbook union-find data structure,
- `Subvsolve`: a DAG-based type inference algorithm [27]

On the programs, we check the following properties: Sorted, the output list is sorted, Elts, the output list has the same elements as the input, Balance, the output trees are balanced, BST, the output trees are binary search ordered, Set, the structure implements a set interface, *e.g.* the outputs of the `add`, `remove`, `merge` functions correspond to the addition of, removal of, union of, (resp.) the elements or sets corresponding to the inputs, Len, the various operations appropriately change the length of the list, Color, the output trees satisfy the red-black color invariant, Heap, the output trees are heap-ordered, Min, the `extractmin` function returns the smallest element, Alloc, the used and free resource lists only contain used and free resources,VariableOrder, the output BDDs have the variable ordering property, Acyclic, the output graphs are acyclic. The complete benchmark suite is available in [51].

Results. The results of running DSOLVE on the benchmarks are summarized in Table 3.1. The columns of Table 3.1 are as follows: **LOC** is the number of lines of code without comments, **Property** is the properties verified, **Ann.** is the number of manual qualifier annotations, and **T(s)** is the time in seconds DSOLVE requires to verify each property.

Annotation Burden. Even for the larger benchmarks, very few qualifiers are required for verification. These qualifiers capture simple relationships between variables that are not difficult to specify after understanding the program. Due to liquid type inference, the total amount of manual annotation remains extremely small — just 3% of code size, which is acceptable given the complexity of the implementation and the properties being verified. Next, we describe the subtleties of some of the benchmarks and describe how DSOLVE verifies the key invariants.

Sorting. We used DSOLVE to verify that implementations of various list sorting algorithms returned sorted lists whose elements were the same as the input lists', *i.e.,* that the sorting functions had type $\texttt{xs}:\alpha\ \texttt{list}\rightarrow\{\nu:\ \alpha\ \texttt{list}_{\leq}\mid \texttt{elts}\ \nu = \texttt{elts}\ \texttt{xs}\}$. We applied DSOLVE to verify several functions that sort lists. For each function, we checked that the returned list was a list of increasing elements with the same element set as the input list.

Table 3.1: **DSOLVE: Data Structure Experimental Results**

Program	LOC	Ann.	T(s)	Property
`List-sort`	110	7	11	Sorted, Elts
`Map`	95	3	23	Balance, BST, Set
`Ralist`	91	3	3	Len
`Redblack`	105	3	32	Balance, Color, BST
`Stablesort`	161	1	6	Sorted
`Vec`	343	9	103	Balance, Len1, Len2
`Heap`	120	2	41	Heap, Min, Set
`Splayheap`	128	3	7	BST, Min, Set
`Malloc`	71	2	2	Alloc
`Bdd`	205	3	38	VariableOrder
`Unionfind`	61	2	5	Acyclic
`Subvsolve`	264	2	26	Acyclic
Total	1754	40	297	

The properties are specified by the types (written separately for clarity):

$$\texttt{xs}:\alpha\ \texttt{list}\rightarrow\alpha\ \texttt{list}_{\leq} \tag{Sorted}$$

$$\texttt{xs}:\alpha\ \texttt{list}\rightarrow\{\nu:\ \alpha\ \texttt{list}\ \mid\ \texttt{len}\ \nu=\texttt{len}\ \texttt{xs}\} \tag{Len}$$

We checked the above properties on `insertsort` (shown in Figure 2.2), `mergesort` [64] which recursively halves the lists, sorts, and merges the results, `mergesort2` [64] which chops the list into a list of (sorted) lists of size two, and then repeatedly passes over the list of lists, merging adjacent lists, until it is reduced to a singleton which is the fully sorted list, `quicksort`, shown in Figure 3.16, which partitions the list around a pivot, then sorts and appends the two partitions, and `stablesort`, from the OCAML standard library's `List` module, which is a tail-recursive mergesort that uses two mutually recursive functions, one which returns an increasing list, another a decreasing list. For each benchmark, DSOLVE infers that the sort function has type Sorted using only the qualifier $\{\nu\leq\star\}$. To prove Elts, we need a few simple qualifiers relating the elements of the output list to those of the input. DSOLVE cannot check Elts for `stablesort` due to its (currently) limited handling of OCAML's pattern syntax. For example, to capture concatenation, the qualifier $\texttt{elts}\ \nu = \texttt{union}\ (\texttt{elts}\ \star)\ (\texttt{elts}\ \star)$ `mergesort2` requires a measure specifying the elements of a list of lists.

Non-aliasing. Nested refinements can be useful not just to verify properties like sortedness, but also to ensure non-aliasing across an unbounded and unordered collection of values. As an example, consider the two functions `alloc` and `free` of Figure 3.15. The functions manipulate a "world" which is a triple comprised of: `m`, a bitmap indicating whether addresses are free (0) or used (1), `us`, a list of addresses marked used, and `fs`, a list of addresses marked free. Suppose that the map functions have types:

$$\texttt{set} : \texttt{m}{:}(\alpha,\beta)\ \texttt{t}{\rightarrow}\texttt{k}{:}\alpha{\rightarrow}\texttt{d}{:}\beta{\rightarrow}\{\nu : (\alpha,\beta)\ \texttt{t} \mid \nu = \mathit{Upd}(\texttt{m},\texttt{k},\texttt{d})\}$$

$$\texttt{get} : \texttt{m}{:}(\alpha,\beta)\ \texttt{t}{\rightarrow}\texttt{k}{:}\alpha{\rightarrow}\{\nu : \beta \mid \nu = \mathit{Sel}(\texttt{m},\texttt{k})\}$$

We can formalize the invariants on the lists using the product type:

$$\rho_c \doteq \langle\langle\rangle;\langle \mathit{Sel}(m,\nu) = c;\top\rangle\rangle$$

$$\sigma^c_{\bowtie} \doteq (\rho_c)\ \texttt{int list}_{\bowtie}$$

$$\mathit{RES}_{\bowtie} \doteq \langle \texttt{m}{:}(\texttt{int},\texttt{int})\,\texttt{t}, \texttt{us}{:}\sigma^1_{\bowtie}, \texttt{fs}{:}\sigma^0_{\bowtie}\rangle$$

The function `alloc` picks an address from the free list, sets the used bit of the address in the bitmap, adds it to the used list and returns this address together with the updated world. Dually, the function `free` checks if the given address is in use and, if so, removes it from the used list, unsets the used bit of the address, and adds it to the free list. We would like to verify that that the functions preserve the invariant on the world above, *i.e.,*

$$\texttt{alloc} : \mathit{RES} \rightarrow \langle \mathit{RES}, \texttt{int} \rangle, \ \texttt{free} : \mathit{RES} \rightarrow \texttt{int} \rightarrow \mathit{RES}.$$

However, the functions *do not* have these types. Suppose that the free list, `fs`, passed to `alloc`, contains *duplicates*. In particular, suppose that the head element `p` also appears inside the tail $\texttt{fs}'$. In that case, setting `p`'s used bit will cause there to be an element of the output free list $\texttt{fs}'$, namely `p`, whose used bit is set, violating the output invariant. Hence, we need to capture the invariant that there are no duplicates in the used or free lists, *i.e.,* that no two elements of the used or free lists are *aliases* for the same address. In our system, this is expressed by the type $\texttt{int list}_{\neq}$, as defined by (2.1). Hence, If the input world has type $\mathit{RES}_{\neq}$, then when `p`'s bit is set, the SMT solver uses the array axioms to determine that for each $\nu \neq \texttt{p}$, $\mathit{Sel}(\texttt{m}', \nu) = \mathit{Sel}(\texttt{m}, \nu)$ and hence, the used bit of each address in $\texttt{fs}'$ remains unset in the new map $\texttt{m}'$. Dually, our system infers that the no-duplicates invariant holds on `p` :: `us` as `p` (whose bit is unset in `m`) is different from all the elements of `us` (whose bits are set in `m`). Thus, our system automatically verifies:

$$\texttt{alloc} : \mathit{RES}_{\neq} \rightarrow \langle \mathit{RES}_{\neq}, \texttt{int} \rangle, \ \texttt{free} : \mathit{RES}_{\neq} \rightarrow \texttt{int} \rightarrow \mathit{RES}_{\neq}$$

Maps. We applied DSOLVE to verify OCAML's tree-based functional `Map` library. The trees have the ML type:

```
type ('a,'b) t =
  E | N of 'a * 'b * ('a,'b) t * ('a,'b) t * int
```

The `N` constructor takes as input a key of type `'a`, a datum of type `'b`, two subtrees, and an integer representing the *height* of the resulting tree. The library implements a variant of

```
let alloc (m, us, fs) =
  match fs with
  | [] ->
     assert false
  | p::fs' ->
     let m'  = set m p 1 in
     let us' = p::us in
     ((m', us', fs'), p)
```

```
let free (m, us, fs) p =
  if get m p = 0 then
   (m, us, fs)
  else
   let m'  = set m p 0 in
   let us' = delete p us in
   let fs' = p::fs in
   (m', us', fs')
```

Figure 3.15: Excerpt from `Malloc`

AVL trees where, internally, the heights of siblings can differ by at most 2. A *binary search tree* (BST) is one where, for each node, the keys in the left (resp. right) subtree are smaller (resp. greater) than the node's key. A tree is *balanced* if, at each node, the heights of its subtrees differ by *at most* two. Formally, after defining a height measure `ht`:

```
measure ht =  E -> 0 | N (_,_,l,r,_) ->
  if ht l < ht r then 1 + ht r else 1 + ht l
```

the balance and BST invariants are respectively specified by:

$$(\rho_{\mathsf{bal}})\ \mu t.\ \mathtt{E} + \mathtt{N}\langle k\!:\!\alpha, d\!:\!\beta, l\!:\!t, r\!:\!t, h\!:\!\mathtt{int}\rangle \qquad \text{(Balance)}$$

$$\mu t.\ \mathtt{E} + \mathtt{N}\langle k\!:\!\alpha, d\!:\!\beta, l\!:\!(\rho_{<})\ t, r\!:\!(\rho_{>})\ t, h\!:\!\mathtt{int}\rangle \qquad \text{(BST)}$$

$$\rho_{\bowtie} \doteq \langle\langle\rangle; \langle \nu \bowtie k; \top; \top; \top; \top\rangle\rangle \text{ for } \bowtie \in \{<, >\}$$

$$\rho_{\mathsf{bal}} \doteq \langle\langle\rangle; \langle \top; \top; \top; e_b; e_h\rangle\rangle$$

$$e_h \doteq (\mathtt{ht}\ l < \mathtt{ht}\ r)?(\nu = 1 + \mathtt{ht}\ r) : (\nu = 1 + \mathtt{ht}\ l)$$

$$e_b \doteq (\mathtt{ht}\ l - \mathtt{ht}\ \nu \leq 2) \wedge (\mathtt{ht}\ \nu - \mathtt{ht}\ l \leq 2)$$

DSOLVE verifies that all trees returned by API functions are balanced binary search trees, that no programmer-specified assertion fails at run-time, and that the library implements a set interface. Even though various "rotations" are performed to ensure balancedness, DSOLVE infers the types required to prove BST automatically using qualifiers generated from the specification. To prove Balance, acDSOLVE requires some manually specified qualifiers to infer the appropriate type for the rebalancing functions.

Vectors. We applied DSOLVE to verify various invariants in a library that uses binary trees to represent C++-style extensible vectors [14]. Formally, vectors are represented as:

```
type 'a t =
  Emp | Node of 'a t * int * 'a * 'a t * int * int
```

The elements of the tuple for the `Node` constructor correspond to the left tree, the number of elements in the left tree, the element at the node, the right tree, the number of elements in the right tree, and the height of the tree, respectively. The i^{th} element of the vector is the root if the size of the left tree is i, the i^{th} element of the left tree if i is less than the number of elements of the left tree, and the $(i-n-1)^{th}$ element of the right tree if the left tree has n elements. To ensure that various operations are efficient, the heights of the subtrees at each level are allowed to differ by at most two. DSOLVE verifies that that all the trees returned by API functions, which include appending to the end of a vector, updating values, deleting sub-vectors, concatenation, *etc.* , are balanced, vector operations performed with valid index operands, *i.e.,* indices between 0 and the number of elements in the vector don't fail (Len1), and that all functions passed as arguments to the iteration, fold and map procedures are called with integer arguments in the appropriate range (Len2). For example, DSOLVE proves:

$$\texttt{iteri} : \texttt{v}{:}\alpha\ \texttt{t} \rightarrow (\{0 \leq \nu < \texttt{len v}\} \rightarrow \alpha \rightarrow \texttt{unit}) \rightarrow \texttt{unit}$$

That is, the second argument passed to the higher-order iterator is only called with inputs between 0 and the length of the vector, *i.e.,* the number of elements in each vector. DSOLVE found a subtle bug in the rebalancing procedure; by using the inferred types, we were able to find a minimal and complete fix which the author adopted.

Fixing bugs with type inference. DSOLVE was able to help find and debug a subtle flaw in the recursive `recbal` procedure used to efficiently merge two balanced trees of arbitrarily different heights into a single balanced tree. Initially, DSOLVE inferred that the difference between the heights of siblings may be no greater than 4 (violating Balance by 2). Yet, the inferred type provided a clue: the height of the output at recursive call-sites was no less than the max of the input heights minus one. By mutating the code and re-inferring types, we were able to determine the code path for which the latter bound was tight, and thus the exact set of paths which broke the balance property. Using this information, we could find test inputs whose outputs matched the inferred bounds, as well as a minimal and complete fix. DSOLVE then verified that the fixed code satisfied Balance, Len1, and Len2.

Binary Decision Diagrams. A *Binary Decision Diagram* (BDD) is a reduced decision tree used to represent boolean formulas. Each node is labeled by a propositional variable drawn from some ordered set $x_1 < \ldots < x_n$. The nodes satisfy a *variable ordering* invariant that if a node labeled x_i has a child labeled x_j then $x_i < x_j$. A combination of hash-consing/memoization and variable ordering ensures that each formula has a canonical BDD representation. Using just three similar, elementary qualifiers, DSOLVE verifies the variable ordering invariant in Filliâtre's OCAML BDD library [20]. The verification requires recursive refinements to handle the ordering invariant and polymorphic refinements to handle the memoization that is crucial for efficient implementations. The type used to encode BDDs, simplified for exposition, is:

```
type var = int
type bdd = Z of int | O of int
         | N of var * bdd * bdd * int
```

where `var` is the variable at a node, and the `int` elements are hash-cons tags for the corresponding sub-BDDs. To capture the order invariant, we write a measure that represents the index of the root variable of a BDD:

```
measure var = Z _ | O _  -> maxvar + 1
            | N (x,_,_,_) -> x
```

where `maxvar` is the total number of propositional variables being used to construct BDDs. After defining

$$\mathtt{bdd} \doteq \mu t.\ \mathtt{Z}\,\mathtt{int} + \mathtt{O}\,\mathtt{int} + \mathtt{N}\langle x\!:\!\mathtt{int}, t, t, \mathtt{int}\rangle$$

$$\rho_{\mathtt{V}} \doteq \langle\langle\top\rangle;\langle\top\rangle;\langle\top;(x < \mathtt{var}\,\nu);(x < \mathtt{var}\,\nu);\top\rangle\rangle$$

we can specify BDDs satisfying the variable ordering VariableOrder invariant as $(\rho_{\mathtt{V}})$ `bdd`. The following code shows the function that computes the BDD corresponding to the negation of the input `x`, by using the table `cache` for memoization.

```
let mk_not x =
  let cache = Hash.create cache_default_size in
  let rec mk_not_rec x =
    if Hash.mem cache x then Hash.find cache x else
      let res = match x with
        | Z _ -> one | O _ -> zero
        | N (v, l, h, _) ->
            mk v (mk_not_rec l) (mk_not_rec h) in
      Hash.add cache x res; res in
  mk_not_rec x
```

Using the polymorphically refined signatures for the hash table operations (`set`, `get`, *etc.* from Section 3.6.1), with the proviso, already enforced by OCAML, that the key's type be treated as invariant, DSOLVE is able to verify that variable ordering (VariableOrder) is preserved on the entire library. To do so, DSOLVE uses the qualifier $\mathtt{var}\ \star \leq \mathtt{var}\ \nu$ to automatically instantiate the refined polytype variables α and $\beta\langle x\!:\!\alpha\rangle$ in the signatures for

`Hash.find` and `Hash.add` with `bdd` and $\{\nu : \texttt{bdd} \mid \texttt{var}\ x \leq \texttt{var}\ \nu\}$, which, with the other rules, suffices to infer that:

$$\texttt{mk\_not} : \texttt{x}{:}\texttt{bdd} \rightarrow \{\nu : \texttt{bdd} \mid \texttt{var}\ x \leq \texttt{var}\ \nu\}$$

Bit-level Type Inference. We applied DSOLVE to verify an implementation of a graph-based algorithm for inferring bit-level types from the bit-level operations of a C program [27]. The bit-level types are represented as a sequence of *blocks*, each of which is represented as a node in a graph. Mask or shift operations on the block cause the block to be *split* into sub-blocks, which are represented by the list of successors of the block node. Finally, the fact that value-flow can cause different bit-level types to have unified subsequences of blocks is captured by having different nodes *share* successor blocks. A similar algorithm was proposed to recover record structure in legacy COBOL programs [19]. The key invariant maintained by the algorithm is that the graph contains no cycles. In the implementation, the graph is represented as pair of an integer representing the number of nodes, and a hash map from nodes to lists of successor nodes. DSOLVE combines recursive and polymorphic refinements to verify that the graph satisfies an acyclicity invariant like *DAG* from Equation (3.1) in Section 3.4.

DSOLVE was able to verify this invariant completely automatically, using three simple inequality qualifiers. The algorithm works by representing bitvectors as directed acyclic graphs.

Red-Black Trees. We applied DSOLVE to verify the following three critical invariants of a Red-Black Tree implementation due to Dunfield [18]. First, the *color* invariant, that a Red node has no Red children. Second, the *black height* invariant, that along all paths from root to leaf the number of Black nodes is the same. Third, the *binary search tree* invariant, that at each node the keys in the left subtree are smaller than the keys in the right subtree. To prove the black height property, we first specify a black height measure bht (similar to the ht measure). Next, we specify a recursive refinement using a predicate bht $l =$ bht ν, *i.e.,* the black height of the right tree is the same as that of the left tree. DSOLVE verifies this property automatically using qualifiers gleaned from the specification. The binary search property is analogous to BST, and DSOLVE proves it automatically using qualifiers generated from the specification. For the color invariant, the challenge lies in the fact that, at intermediate points, a node is created that violates the color invariant.

Previous approaches to verifying the color invariant capture the broken tree, either using indices [65] which encode the number of consecutive red nodes, or by specifying a hierarchy of datasort refinements [18]. In either case, one must manually specify the delicate interaction between the type constructors and the indices or datasorts. While our system can permit such a proof, it also enables a more “declarative” approach to specification and verification.

First, we extend the ML type with a *Purple* constructor P that encodes Red nodes with Red children. Thus, the tree is defined as:

```
type 'a t =
  | E
  | B of 'a * 'a t * 'a t
  | R of 'a * 'a t * 'a t
  | P of 'a * 'a t * 'a t
```

There is a single point in the code where a tree with two consecutive Red nodes is built, and here, instead the code is changed to use P. Next, we define a measure, col, for the color of the tree’s root:

```
measure col = E -> 0 |B _ -> 1 |R _ -> 2 |P _ -> 3
```

Third, we specify the following refinement vectors

$$e_{\mathtt{EBR}} \doteq \mathtt{col}\ \nu \leq 2 \qquad\qquad e_{\mathtt{EB}} \doteq \mathtt{col}\ \nu \leq 1$$

$$\rho_{\mathtt{B}} \doteq \langle \top; e_{\mathtt{EBR}}; e_{\mathtt{EBR}} \rangle \qquad\qquad \rho_{\mathtt{R}} \doteq \langle \top; e_{\mathtt{EB}}; e_{\mathtt{EB}} \rangle$$

$$\rho_{\mathtt{P}} \doteq \langle \top; \top; \top \rangle \qquad\qquad \rho_{\mathsf{col}} \doteq \langle \langle\rangle; \rho_{\mathtt{R}}; \rho_{\mathtt{B}}; \rho_{\mathtt{P}}; \rho_{\mathtt{P}} \rangle$$

The recursive refinement ρ_{col} states that Black nodes ($\rho_{\mathtt{B}}$) have Empty, Black or Red sub-trees ($e_{\mathtt{EBR}}$), Red nodes ($\rho_{\mathtt{R}}$) have only Empty or Black sub-trees ($e_{\mathtt{EB}}$), and Purple nodes ($\rho_{\mathtt{P}}$) are unconstrained. Thus, a tree satisfying the color invariant has type:

$$\{\nu : (\rho_{\mathsf{col}})\ \alpha\ \mathtt{t} \mid e_{\mathtt{EBR}}\} \qquad \text{(Color)}$$

where $\alpha\ \mathtt{t}$ abbreviates the recursive (ML) type corresponding to the tree. This type also ensures there are no purple nodes in the trees returned to clients as the root is not purple and $\rho_{\mathtt{R}}$ and $\rho_{\mathtt{B}}$ ensure that neither Red nor Black nodes have any Purple children. Using the qualifiers ($e_{\mathtt{EBR}} \vee \mathtt{col} \star = 2$), *i.e.,* a tree is Purple only when another tree is Red, and $(\mathtt{col}\ \nu = 2) \oplus (\mathtt{col} \star = 2)$, *i.e.,* exactly one of two trees is Red, DSOLVE proves that all created trees have type Color, and hence satisfy the color invariant.

3.7.1 Limitations and Future Work

Our case studies reveal several expressiveness limitations of our system. Currently, simple modifications allow each program to typecheck. We intend to address these limitations in future work.

First-order Refinements. To preserve decidability, our EUFA embedding leaves all function applications in the refinement predicates uninterpreted. This prevents checking `quicksort` with the standard higher-order partition function:

$$\texttt{part} : \alpha\ \texttt{list} \rightarrow \texttt{p} : (\alpha \rightarrow \texttt{bool}) \rightarrow \{\texttt{p}(\nu)\}\ \texttt{list} * \{\neg \texttt{p}(\nu)\}\ \texttt{list}$$

where `p` is the higher-order predicate used for partitioning. The function applications $\texttt{p}(\nu)$ in `part`'s output type are left uninterpreted, so we cannot use them in verification. Instead, we change the program so that the type of `p` is $\alpha \rightarrow (\beta, \gamma)\ \texttt{either}$ where $(\beta, \gamma)\ \texttt{either}$ has constructors `T of` β and `F of` γ. The new `part` function then collects `T` and `F` values into a pair of separate lists. Thus, if `w` is the pivot, we can verify `quicksort` by passing `part` a higher-order predicate of type:

$$\alpha \rightarrow (\{\nu : \ \alpha \ \mid \ \nu \geq \texttt{w}\}, \{\nu : \ \alpha \ \mid \ \nu < \texttt{w}\})\ \texttt{either}.$$

Existential Witnesses. Recall that expressing the key acyclicity invariant in *union-find*'s `find` function (shown in Section 3.6.1) required referencing the `rank` map. Although `rank` is not used in the body of the `find` function, omitting it from the parameter list causes this acyclicity invariant to be ill-formed within `find`. Instead of complicating our system with existentially quantified types, we add `rank` as a *witness* parameter to `find`. Similarly, notice that the list obtained by appending two sorted lists `xs` and `ys` is sorted iff there exists some `w` such that the elements of `xs` (resp. `ys`) are less than (resp. greater than) `w`. In the case of `quicksort`, this `w` is exactly the pivot. Hence, we add `w` as a witness parameter to append, and pass in the pivot at the callsite, after which the system infers:

$$\text{append} : \texttt{w} : \alpha \rightarrow \{\nu \leq \texttt{w}\}\ \texttt{list}_{\leq} \rightarrow \{\texttt{w} \leq \nu\}\ \texttt{list}_{\leq} \rightarrow \alpha\ \texttt{list}_{\leq}$$

and hence that `quicksort` has type Sorted. Similar witness parameters are needed for the tail-recursive merges used in `stablesort`.

Context-Sensitivity. Finally, there were cases where functions have different behavior in different calling contexts. For example, `stablesort` uses a function that reverses a list. Depending upon the context, the function is either (1) passed an increasing list as an argument and returns a decreasing list as output, or, (2) passed a decreasing list as an argument and returns an increasing list. Our system lacks intersection types (*e.g.* [18]), and we cannot capture the above, which forces us to duplicate code at each callsite. In our experience so far, this has been rare (out of all our benchmarks, only one duplicate function in `stablesort` was required), but nevertheless, in the future, we would like to investigate how our system can be extended to intersection types.

3.8 Conclusion

In this chapter, we show two new mechanisms for describing, checking and inferring invariants on structures that may not be well-behaved: Measures and Polymorphic Refinements. For each mechanism, we show how the mechanism can be used to encode and verify correctness properties of data structures that are non-boolean, and existential, resp.

Further, we show that these mechanisms work in harmony with Recursive Refinements to automatically and efficiently prove extremely complex invariants over challenging data structure benchmarks in OCAML. To wit, we provide a case study in which our experiments discovered an exceptionally complex bug in a program that simultaneously manipulated sorted lists, balanced trees and maps.

```
type ('a, 'b) boolean = T of 'a | F of 'b

let rec partition f = function
  | [] -> ([], [])
  | x::xs ->
     let (ts,fs) = partition f xs in
     (match f x with
      | T y -> (y::ts,fs)
      | F y -> (ts,y::fs))

let rec append k xs ys =
  match xs with
  | [] -> ys
  | x::xs' -> x::(append k xs' ys)

let rec quicksort = function
  | [] -> []
  | x::xs' ->
      let f y = if y < x then T y else F y in
      let (ls,rs) = partition f xs' in
      append x (quicksort ls) (x::(quicksort rs))
```

Figure 3.16: A Simple, Functional Quicksort

Acknowledgements

This chapter contains material adapted from the publication: Ming Kawaguchi, Patrick Rondon, Ranjit Jhala. “Type-Based Data Structure Verification”, *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 159–169, 2009. The dissertation author was principal researcher and author on this publication. This chapter also contains material adapted from the publication: 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 an author on this publication.

Chapter 4

Liquid Effects

4.1 Introduction

How do we program multi-core hardware? While many models have been proposed, the model of multiple sequential threads concurrently executing over a single shared memory remains popular due to its efficiency, its universal support in mainstream programming languages and its conceptual simplicity. Unfortunately, shared memory multithreading is fiendishly hard to get right, due to the inherent non-determinism of thread scheduling. Unless the programmer is exceptionally vigilant, concurrent accesses to shared data can result in non-deterministic behaviors in the program, potentially yielding difficult-to-reproduce "heisenbugs" whose appearance depends on obscurities in the bowels of the operation system's scheduler and are hence notoriously hard to isolate and fix.

Determinism By Default. One way forward is to make parallel programs "deterministic by default" [2] such that, no matter how threads are scheduled, program behavior remains the same, eliminating unruly heisenbugs and allowing the programmer to reason about their parallel programs as if they were sequential. In recent years, many static and dynamic approaches have been proposed for ensuring determinism in parallel programs. While dynamic approaches allow arbitrary data sharing patterns, they also impose non-trivial run-time overheads and hence are best in situations where there is relatively little sharing [7]. In contrast, while static approaches have nearly no run-time overhead, they have been limited to high-level languages like Haskell [11] and Java [2] and require that shared structures be refactored into specific types or classes for which deterministic compilation strategies exist, thereby restricting the scope of sharing patterns.

Liquid Effects. In this chapter, we present *Liquid Effects*, a type-and-effect system based on refinement type inference which uses recent advances in SMT solvers to provide the best of both worlds: it allows for fine-grained shared memory multithreading in low-level languages with program-specific data access patterns, and yet statically guarantees that programs are deterministic.

Any system that meets these goals must satisfy several criteria. First, the system must support *precise and expressive effect specifications*. To precisely characterize the effect of program statements, it must precisely reason about complex heap access patterns, including patterns not foreseen by the system's designers, and it must be able to reason about relations between program values like loop bounds and array segment sizes. Second, the system must be *extensible* with user-defined effects to track effects which are domain-specific and thus could not be foreseen by the system's designers. Finally, the system must support *effect inference*, so that the programmer is not overwhelmed by the burden of providing effect annotations.

1. Precise and Expressive Effect Specifications. Our Liquid Effects type-and-effect system expresses the effect of a statement on the heap as a formula in first-order logic that classifies which addresses in the heap are accessed and with what effect they were accessed — for example, whether the data was read or written. Expressing effects as first-order formulas makes our type-and-effect system highly expressive: for example, using the decidable theory of linear arithmetic, it is simple to express heap access patterns like processing chunks of an array in parallel or performing strided accesses in a number of separate threads. Using first-order formulas for effects ensures that our system can express complex access patterns not foreseen by the system's designers, and allows us to incorporate powerful off-the-shelf SMT solvers in our system for reasoning about effects. Further, by building our type-and-effect system as an extension to an existing dependent refinement type system and allowing effect formulas to reference program variables, our system gains precise value and branch-sensitive reasoning about effects.

2. Extensibility. Our effect formulas specify how locations on the heap are affected using *effect labeling predicates* like Read and Write. These effect labeling predicates are ordinary uninterpreted predicates in first-order logic; our system does not treat effect labeling predicates specially. Thus, we are able to provide extensibility by parameterizing our system over a set of user-defined effect labeling predicates. Further, we allow the user to give *commutativity declarations* specifying which pairs of effects have benign interactions when they simultaneously occur in separate threads of execution. This enables users to track domain-specific effects not envisioned by the designers of the type-and-effect system.

3. Effect Inference. Our type rules can be easily recast as an algorithm for generating constraints over unknown refinement types and effect formulas; these constraints can then be solved using the Liquid Types technique for invariant inference, thus yielding a highly-automatic type-based method for proving determinism.

To illustrate the utility of Liquid Effects, we have implemented our type-and-effect checking techniques in CSOLVE, a refinement type inference system for C programs based on Liquid Types. We demonstrate how CSOLVE uses Liquid Effects to prove the determinism of a variety of benchmarks from the literature requiring precise, value-aware tracking of both built-in and user-defined heap effects while imposing a low annotation burden on the user. As a result, CSOLVE opens the door to efficient, deterministic programming in mainstream languages like C.

4.2 Overview

We start with a high-level overview of our approach. Figures 4.1 and 4.2 show two functions, `sum1` and `sum2` respectively, which add up the elements of an array of size 2^k *in-place* by dividing the work up into independent sub-computations that can be executed in parallel. At the end of both procedures, the first element of the array contains the final sum. In this section, we demonstrate how our system seamlessly combines path-sensitive reasoning using refinement types, heap effect tracking, and SMT-based reasoning to prove that `sum3` and `sum2` are deterministic.

4.2.1 Contiguous Partitions

The function `sum1` sums array `a` of length `len` using the auxiliary function `sumBlock`, which sums the elements of the contiguous segment of array `a` consisting of `len`-many elements and beginning at index `i`. Given an array segment of length `len`, `sumBlock` computes the sum of the segment's elements by dividing it into two contiguous subsegments which are recursively summed using `sumBlock`. The recursive calls place the sum of each subsegment in the first element of the segment; `sumBlock` computes its final result by summing the first element of each subsegment and stores it in the first element of the entire

```
void sumBlock (char *a, int i, int len) {
  if (len <= 1) return;

  int hl = len / 2;
  cobegin {
    sumBlock (a, i, hl);
    sumBlock (a, i + hl, len - hl);
  }

  a[i] += a[i + hl];
}

int sum1 (char *a, int len) {
  sumBlock (a, 0, len);

  return a[0];
}
```

Figure 4.1: Parallel Array Summation With Contiguous Partitions

array segment.

Cobegin Blocks. To improve performance, we can perform the two recursive calls inside `sumBlock` in parallel. This is accomplished using the `cobegin` construct, which evaluates each statement in a block in parallel. Note that the recursive calls both read and write to the array passed to `sumBlock`; thus, to prove that the program is deterministic, we have to show that the part of the array that is written by each call does not overlap with the portion of the array that is read by the other. In the following, we demonstrate how our system is able to show that the recursive calls access disjoint parts of the array and thus that the function `sumBlock` is deterministic.

Effect Formulas. We compute the region of the array accessed by `sumBlock` as an *effect formula* that describes which pointers into the array are accessed by a call to `sumBlock` with parameters `i` and `len`. We note that the final line of `sumBlock` writes element `i` of `a`. We state the effect of this write as the formula [1]

$$\Sigma_{\texttt{a[i]}} \doteq \nu = \texttt{a} + \texttt{i}.$$

We interpret the above formula, $\Sigma_{\texttt{a[i]}}$, as "the pointer ν accessed by the expression $\texttt{a[i]}$ is equal to the pointer obtained by incrementing pointer `a` by `i`." The final line of `sumBlock` also reads element $\texttt{i} + \texttt{hl}$ of `a`. The effect of this read is stated as a similar formula:

$$\Sigma_{\texttt{a[i+hl]}} \doteq \nu = \texttt{a} + \texttt{i} + \texttt{hl}.$$

The total effect of the final line of `sumBlock` stating which portion of the heap it accesses is then given by the disjunction of the two effect formulas:

$$\Sigma_{\texttt{a[i]}} \oplus \Sigma_{\texttt{a[i+hl]}} \doteq \nu = \texttt{a} + \texttt{i} \vee \nu = \texttt{a} + \texttt{i} + \texttt{hl}.$$

The above formula says that the heap locations accessed in the final line of `sumBlock` are exactly the locations corresponding to $\texttt{a[i]}$ and $\texttt{a}[\texttt{i} + \texttt{hl}]$. Having determined the effect of the final line of `sumBlock` as a formula over the local variables `a`, `i`, and `hl`, we make two observations to determine the effect of the entire `sumBlock` function as a formula over its arguments `a`, `i`, and `len`. First, we observe that, in the case where $\texttt{len} = 2$,

$$\texttt{a}[\texttt{i} + \texttt{hl}] = \texttt{a}[\texttt{i} + \texttt{len} - 1].$$

[1] Here we associate effects with program expressions like $\texttt{a[i]}$. However, in the technical development and our implementation, effects are associated with heap locations in order to soundly account for aliasing (refer to Section 4.4).

From this, we can see inductively that a call to `sumBlock` with index `i` and length `len` will access elements $\texttt{a}[\texttt{i}]$ to $\texttt{a}[\texttt{i}+\texttt{len}-1]$. Thus, we can soundly ascribe `sumBlock` the effect

$$\Sigma_{\texttt{sumBlock}} \doteq \texttt{a}+\texttt{i} \leq \nu \wedge \nu < \texttt{a}+\texttt{i}+\texttt{len}.$$

That is, `sumBlock` will access all elements in the given array segment.

Determinism via Effect Disjointness. We are now ready to prove that the recursive calls within `sumBlock` access disjoint regions of the heap and thus that `sumBlock` behaves deterministically when the calls are executed in parallel. We determine the effect of the first recursive call by *instantiating* `sumBlock`'s effect, *i.e.,* by replacing formals `a`, `i`, and `len` with actuals `a`, `i`, and `hl`, respectively:

$$\Sigma_{\text{Call 1}} \doteq \texttt{a}+\texttt{i} \leq \nu \wedge \nu < \texttt{a}+\texttt{i}+\texttt{hl}.$$

The resulting effect states that the first recursive call only accesses array elements $\texttt{a}[\texttt{i}]$ through $\texttt{a}[\texttt{i}+\texttt{hl}-1]$. We determine the effect of the second recursive call with a similar instantiation of `sumBlock`'s effect:

$$\Sigma_{\text{Call 2}} \doteq \texttt{a}+\texttt{i}+\texttt{hl} \leq \nu \wedge \nu < \texttt{a}+\texttt{i}+\texttt{len}-\texttt{hl}.$$

All that remains is to show that the effects $\Sigma_{\text{Call 1}}$ and $\Sigma_{\text{Call 2}}$ are disjoint. We do so by asking an SMT solver to prove the unsatisfiability of the conjoined *access intersection* formula

$$\Sigma_{\mathsf{Unsafe}} = \Sigma_{\text{Call 1}} \wedge \Sigma_{\text{Call 2}},$$

whose inconsistency establishes that the intersection of the sets of pointers ν accessed

```
declare effect Accumulate;
declare Accumulate commutes with Accumulate;

void accumLog (char *l, int j)
  effect
   (&l[j], Accumulate(v) && !Read(v) && !Write(v));

void sumStride (char *a, int stride, char *log) {
  foreach (i, 0, THREADS) {
    for (int j = i; j < stride; j += THREADS) {
        a[j] += a[j + stride];
        accumLog (log, j);
    }
  }
}

int sum2 (char *a, int len) {
  log = (char *) malloc (len);

  for (int stride = len/2; stride > 0; stride /= 2)
    sumStride (a, stride, log);

  return a[0];
}
```

Figure 4.2: Parallel Array Summation With Strided Partitions

by both recursive calls is empty.

4.2.2 Complex Partitions

In the previous example, function `sum1` computed the sum of an array's elements by recursively subdividing the array into halves and operating on each half separately before combining the results. We were able to demonstrate that `sum1` is deterministic by showing that the concurrent recursive calls to `sumBlock` operate on disjoint contiguous segments of the array. We now show that our system is capable of proving determinism even when the array access patterns are significantly more complex.

Function sum2 uses the auxiliary function sumStride, which sums an array's elements using the following strategy: First, the array is divided into two contiguous segments. Each element in the first half is added to the corresponding element in the second half, and the element in the first half is replaced with the result. The problem is now reduced to summing only the first half of the array, which proceeds in the same fashion, until we can reduce the problem no further, at which point the sum of all the array elements is contained in the first element of the array.

Foreach Blocks. To increase performance, the pairwise additions between corresponding elements in the first and second halves of the array are performed in parallel, with the level of parallelism determined by a compile-time constant THREADS. The parallel threads are spawned using the foreach construct. The statement

```
foreach (i, l, u) s;
```

executes statement s once with each possible binding of i in the range $[\mathtt{l},\mathtt{u})$. Further, all executions of statement s are performed in parallel. The foreach loop within sumStride spawns THREADS many threads. Thread i processes elements i, $\mathtt{i} + \mathtt{THREADS}$, $\mathtt{i} + 2 * \mathtt{THREADS}$, *etc.* , of array a. While this strided decomposition may appear contrived, it is commonly used in GPU programming to maximize memory bandwidth by enabling "adjacent" threads to access adjacent array cells via *memory coalescing* [1].

To prove that the foreach loop within sumStride is deterministic, we must show that no location that is written in one iteration is either read from or written to in another iteration. (We ignore the effect of the call to the function accumLog for now and return to it later.) Note that this differs from the situation with sum1: in order to demonstrate that sum1 is deterministic, it was not necessary to consider read and write effects separately; it was enough to know that recursive calls to `sumBlock` operated on disjoint portions of the heap. However, the access pattern of a single iteration of the `foreach` loop in `sumStride` is considerably more complicated, and reasoning about noninterference between loop iterations will require tracking not only which locations are affected but precisely how they are affected — that is, whether they are read from or written to.

Labeled Effect Formulas. We begin by computing the effect of each iteration of the loop on the heap, tracking which locations are accessed as well as which effects are performed at each location. We first compute the effect of each iteration of the inner `for` loop, which adds the value of $\mathtt{a}[\mathtt{j}+\mathtt{stride}]$ to $\mathtt{a}[\mathtt{j}]$. We will use the unary *effect labeling predicates* $\text{Read}(\nu)$ and $\text{Write}(\nu)$ to record the fact that a location in the heap has or has not been read from or written to, respectively. We capture the effect of the read of $\mathtt{a}[\mathtt{j}+\mathtt{stride}]$ with the effect formula

$$\Sigma_1 \doteq \text{Read}(\nu) \wedge \neg\text{Write}(\nu) \wedge \nu = \mathtt{a}+\mathtt{j}+\mathtt{stride}.$$

Note that we have used the effect label predicates Read and Write to record not only that we have accessed $\mathtt{a}[\mathtt{j}+\mathtt{stride}]$ but also that this location was only read from, not written to. We record the effect of the write to $\mathtt{a}[\mathtt{j}]$ with a similar formula:

$$\Sigma_2 \doteq \text{Write}(\nu) \wedge \neg\text{Read}(\nu) \wedge \nu = \mathtt{a}+\mathtt{j}$$

As before, the overall effect of these two operations performed sequentially, and thus the effect of a single iteration of the inner `for` loop is the disjunction of the two effect formulas above:

$$\Sigma_{\mathtt{j}} \doteq \Sigma_1 \vee \Sigma_2$$

Iteration Effect Formulas. Having computed the effect of a single iteration of `sumStride`'s inner `for` loop, the next step in proving that `sumStride` is deterministic is to compute the effect of each iteration of the outer `foreach` loop, with our goal being to show that the effects of any two distinct iterations must be disjoint. We begin by generalizing the effect formula we computed to describe the effect of a single iteration of the inner `for` loop to an effect formula describing the effect of the entire `for` loop. The effect formula describing the effect of the entire `for` is an effect formula that does not reference the loop induction variable `j`, but is implied by the conjunction of the loop body's single-iteration effect and any loop invariants that apply to the loop induction variable `j`.

Note that the induction variable `j` enjoys the loop invariant

$$\mathtt{j} < \mathtt{stride} \wedge \mathtt{j} \equiv \mathtt{i} \bmod \mathtt{THREADS}.$$

Given this invariant for `j`, we can now summarize the effect of the `for` loop as an effect formula which does not reference the loop induction variable `j` but is implied by the conjunction of the above invariant on j and the single-iteration effect formula $\Sigma_{\texttt{j}}$:

$$\begin{aligned}\Sigma_{\texttt{for}} \doteq\ & (\text{Write}(\nu) \Rightarrow (\nu - \texttt{a}) \equiv \texttt{i} \bmod \texttt{THREADS}) \\ & \wedge (\text{Write}(\nu) \Rightarrow \nu < \texttt{a} + \texttt{stride}) \\ & \wedge (\text{Read}(\nu) \Rightarrow \nu \geq \texttt{a} + \texttt{stride}) \end{aligned} \tag{4.1}$$

We have now computed the effect of `sumStride`'s inner `for` loop, and thus the effect of a single iteration of `sumStride`'s outer `foreach` loop.

Effect Projection. We define an effect projection operator $\pi(\Sigma, E)$ which returns the restriction of the effect formula Σ to the effect label E. In essence, the effect projection is the formula implied by the conjunction of Σ and $E(\nu)$. For example, the projection of the Write effect of iteration `i` of the `foreach` loop represents the addresses written by thread `i`.

Determinism via Iteration Effect Disjointness. To prove that the `foreach` loop is deterministic, we must show that values that are written in one iteration are not read or overwritten in another. First, we show that no two `foreach` iterations (*i.e.,* threads) write to the same location, or, in other words, that distinct iterations write through disjoint sets of pointers. We establish this fact by asking the SMT solver to prove the unsatisfiability of the *write-write intersection* formula:

$$\begin{aligned}\Sigma_{\mathsf{Unsafe}} = \pi(\Sigma_{\texttt{for}}, \text{Write}) \wedge \pi(\Sigma_{\texttt{for}}, \text{Write})[\texttt{i} \mapsto \texttt{i}'] \\ \wedge\ \texttt{i} \neq \texttt{i}' \wedge 0 \leq \texttt{i}, \texttt{i}' < \texttt{THREADS}.\end{aligned}$$

The formula is indeed unsatisfiable as, from Effect 4.1,

$$\begin{aligned}\pi(\Sigma_{\texttt{for}}, \text{Write}) \doteq (\nu - \texttt{a}) &\equiv \texttt{i} \bmod \texttt{THREADS} \\ &\wedge \nu < \texttt{a} + \texttt{stride} \end{aligned} \tag{4.2}$$

and so, after substituting $\texttt{i}'$ the query formula is

$$(\nu - \texttt{a}) \equiv \texttt{i} \bmod \texttt{THREADS} \wedge (\nu - \texttt{a}) \equiv \texttt{i}' \bmod \texttt{THREADS}$$

which is unsatisfiable when $\texttt{i} \neq \texttt{i}'$.

Next, we prove that no heap location is read in one iteration (*i.e.,* thread) and written in another. As before, we verify that the intersection of the pointers read in one iteration with those written in another iteration is empty. Concretely, we ask the SMT solver to prove the unsatisfiability of the *write-read intersection* formula

$$\Sigma_{\mathsf{Unsafe}} = \pi(\Sigma_{\texttt{for}}, \text{Write}) \wedge \pi(\Sigma_{\texttt{for}}, \text{Read})[\texttt{i} \mapsto \texttt{i}'] \wedge \texttt{i} \neq \texttt{i}'$$

where, from Effect 4.1, the locations *read* in a single iteration of the `foreach` loop are described by the Read projection

$$\pi(\Sigma_{\texttt{for}}, \text{Read}) \doteq \nu \geq \texttt{a} + \texttt{stride}.$$

After substituting the the write effect from Effect 4.2, we can see that the write-read-intersection formula is unsatisfiable as

$$\nu < \texttt{a} + \texttt{stride} \wedge \nu \geq \texttt{a} + \texttt{stride}$$

is inconsistent. Thus, by proving the disjointness of the write-write and write-read effects, we have proved that the `foreach` loop inside `sumStride` is deterministic.

4.2.3 User-Defined Effects

By expressing our effect labeling predicates as ordinary uninterpreted predicates in first-order logic, we open the door to allowing the user to define their own effects. Such user-defined effects are first-class: we reason about them using the same mechanisms as the built-in predicates Read and Write, and effect formulas over user-defined predicates are equally as expressive as those over the built-in predicates.

We return to the `sumStride` function to demonstrate the use of user-defined effects. The `for` loop in `sumStride` tracks how many times the entry `a`[`j`] is written by incrementing the `j`-th value in the counter array `log` using the externally-defined function `accumLog`. We assume that the `accumLog` function is implemented so that its operation is atomic. Thus, using `accumLog` to increment the count of the same element in two distinct iterations of the `foreach` loop does not cause non-determinism.

Specifying Effects. We create a new user-defined effect predicate to track the effect of the `accumLog` function using the statement

```
declare effect Accumulate;
```

This extends the set of effect predicates that our system tracks to

$$\mathbb{E} \doteq \{\mathrm{Read}, \mathrm{Write}, \mathrm{Accumulate}\}.$$

We then annotate the prototype of the `accumLog` function to specify `accumLog` causes the

Accumulate effect to occur on the `j`-th entry of its parameter array `l`:

```
void accumLog (char *l, int j)
  effect
   (&l[j], Accumulate(v) && !Read(v) && !Write(v));
```

Specifying Commutativity. We specify that our user effect Accumulate does not cause nondeterminism even when it occurs on the same location in two separate threads using the *commutativity annotation*

```
declare Accumulate commutes with Accumulate;
```

This annotation extends the set of pairs of effects that commute with each other — that is, which can occur in either order without affecting the result of the computation. In particular, the commutativity annotation above extends the set of commutable pairs to

$$\mathbb{C} \doteq \{(\text{Read}, \text{Read}), (\text{Accumulate}, \text{Accumulate})\}.$$

The extended set formally specifies that in addition to pairs of Read effects (included by default), pairs of of Accumulate effects also commute, and hence may be allowed to occur simultaneously.

Generalized Effect Disjointness. Finally, we generalize our method for ensuring determinism. We check the disjointness of two effect formulas Σ_1 and Σ_2 by asking the SMT solver to prove the unsatisfiability of the *effect intersection* formula:

$$\Sigma_{\mathsf{Unsafe}} = \exists (E_1, E_2) \in (\mathbb{E} \times \mathbb{E} \setminus \mathbb{C}). \pi(\Sigma_1, E_1) \wedge \pi(\Sigma_2, E_2).$$

That is, for each possible combination of effect predicates that have not been explicitly declared to commute, we check that the two effect sets are disjoint when projected on those effects.

Thus, by declaring an Accumulate effect which is commutative, we are able to verify the determinism of `sum2`.

Effect Inference. In the preceding, we gave explicit loop invariants and heap effects where necessary. Later in this chapter, we explain how we use Liquid Type inference [53] to reduce the annotation burden on the programmer by automatically inferring the loop invariants and heap effects given above.

Outline. The remainder of this chapter is organized as follows: In Section 4.3, we give the syntax of NanoC_{eff} programs, informally explain their semantics, and give the syntax of NanoC_{eff}'s types. We define the type system of NanoC_{eff} in Section 4.4. In Section 4.5, we give an overview of the results of applying an implementation of a typechecker for NanoC_{eff} to a series of examples from the literature. We review related work in Section 4.6.

4.3 Syntax and Semantics

In this section, we present the syntax of NanoC_{eff} programs, informally discuss their semantics, and give the syntax of NanoC_{eff} types.

4.3.1 Programs

The syntax of NanoC_{eff} programs is shown in Figure 4.3. Most of the expression forms for expressing sequential computation are standard or covered extensively in previous work [53]; the expression forms for parallel computation are new to this work.

Values. The set of NanoC_{eff} values v includes program variables, integer constants n, and

v	$::=$		**Values**
	$\mid$	x	variable
	$\mid$	n	integer
	$\mid$	$\&n$	pointer
a	$::=$		**Pure Expressions**
	$\mid$	v	value
	$\mid$	$a_1 \circ a_2$	arithmetic operation
	$\mid$	$a_1 +_p a_2$	pointer arithmetic
	$\mid$	$a_1 \sim a_2$	comparison
e	$::=$		**Expressions**
	$\mid$	a	pure expression
	$\mid$	$*v$	heap read
	$\mid$	$*v_1 := v_2$	heap write
	$\mid$	$\textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2$	if-then-else
	$\mid$	$f(\overline{v})$	function call
	$\mid$	$\textbf{malloc}(v)$	memory allocation
	$\mid$	$\textbf{let } x = e_1 \textbf{ in } e_2$	let binding
	$\mid$	$\textbf{letu } x = \textbf{unfold } v \textbf{ in } e$	location unfold
	$\mid$	$\textbf{fold } l$	location fold
	$\mid$	$e_1 \parallel e_2$	parallel composition
	$\mid$	$\textbf{for each } x \textbf{ in } v_1 \textbf{ to } v_2 \ \{e\}$	parallel iteration
F	$::=$	$f\ (\overline{x_i})\ \{\ e\ \}$	**Function Declarations**
P	$::=$	$\overline{F}\ e$	**Programs**

Figure 4.3: **Syntax of** $NanoC_{eff}$ **programs**

constant pointer values $\&n$ representing addresses in the heap. With the exception of the null pointer value $\&0$, constant pointer values do not appear in source programs.

Expressions. The set of pure $NanoC_{eff}$ expressions a includes values v, integer arithmetic expressions $a_1 \circ a_2$, where $\circ$ is one of the standard arithmetic operators $+$, $-$, $*$, *etc.* , pointer arithmetic expressions $a_1 +_p a_2$, and comparisons $a_1 \sim a_2$ where $\sim$ is one of the comparison operators $=$, $<$, $>$, *etc.* Following standard practice, zero and non-zero values represent falsity and truth, respectively.

The set of $NanoC_{eff}$ expressions e includes the pure expressions a as well as all side-effecting operations. The expression forms for pointer read and write, if-then-else, function call, memory allocation, and let binding are all standard. The location unfold and fold expressions are used to support type-based reasoning about the heap using strong updates As they have no effect on the dynamic semantics of $NanoC_{eff}$ programs, we defer discussion of location unfold and fold expressions to Section 4.4.

Parallel Expressions. The set of $NanoC_{eff}$ expressions includes two forms for expressing parallel computations. The first, parallel compositioni

$$e_1 \parallel e_2,$$

evaluates expressions e_1 and e_2 in parallel and returns when both subexpressions have evaluated to values.

The second parallel expression form is the parallel iteration expression form

$$\textbf{for each } x \textbf{ in } v_1 \textbf{ to } v_2 \ \{e\},$$

where v_1 and v_2 are integer values. A parallel iteration expression is evaluated by evaluating the body expression e once for each possible binding of variable x to an integer in the range $[v_1, v_2)$. All iterations are evaluated in parallel, and the parallel iteration expression returns when all iterations have finished evaluating. Both forms of parallel expressions are evaluated solely for their side effects.

Functions and Programas. A function declaration $f\ (\overline{x_i})\ \{\ e\ \}$ declares a function f with arguments x_i whose body is the expression e. The return value of the function is the value of the expression. An *NanoC*$_{eff}$ program consists of a sequence of function declarations F followed by an expression e which is evaluated in the environment containing the previously-declared functions.

4.3.2 Types

The syntax of *NanoC*$_{eff}$ types is shown in Figure 4.4. **Base Types.** The base types b of *NanoC*$_{eff}$ include integer types $\texttt{int}(i)$ and pointer types $\texttt{ref}(l,i)$. The integer type $\texttt{int}(i)$ describes an integer whose value is in the set described by the index i; the set of indices is described below. The pointer type $\texttt{ref}(l,i)$ describes a pointer value to the heap location l whose offset from the beginning of location l is an integer belonging to the set described by index i.

Indices. The language of *NanoC*$_{eff}$ base types uses *indices* i to approximate the values of integers and the offsets of pointers from the starts of the heap locations where they point. There are two forms of indices. The first, the singleton index n, describes the set of integers $\{n\}$. The second, the sequence index n^{+m}, represents the sequence of offsets $\{n+lm\}_{l=0}^{\infty}$.

Refinement Types. The *NanoC*$_{eff}$ refinement types τ are formed by joining a base type b with a refinement formula p to form the refinement type $\{\nu : b \mid p\}$. The distinguished *value variable* ν refers to the value described by this type. The *refinement formula* p is a log-

b	$::=$		**Base Types**
	$\mid$	$\texttt{int}(i)$	integer
	$\mid$	$\texttt{ref}(l, i)$	pointer
p	$::=$	$\phi(\overline{v})$	**Refinement Formulas**
τ	$::=$	$\{\nu : b \mid p\}$	**Refinement Types**
i	$::=$		**Indices**
	$\mid$	n	constant
	$\mid$	n^{+m}	sequence
c	$::=$	$\overline{i : \tau}$	**Blocks**
l	$::=$		**Heap Locations**
	$\mid$	$\tilde{l}$	abstract location
	$\mid$	l_j	concrete location
h	$::=$		**Heap Types**
	$\mid$	ε	empty heap
	$\mid$	$h * l \mapsto c$	extended heap
Σ	$::=$		**Heap Effects**
	$\mid$	ε	empty effect
	$\mid$	$\Sigma * \tilde{l} \mapsto p$	extended effect
σ	$::=$	$(\overline{x_i : \tau_i})/h_1 \;\rightarrow\; \tau/h_2/\Sigma$	**Function Schemes**

Figure 4.4: **Syntax of** $NanoC_{eff}$ **types**

ical formula $\phi(\nu, \overline{v})$ over $NanoC_{eff}$ values $\overline{v}$ and the value variable ν. The type $\{\nu : b \mid p\}$ describes all values v of base type b that satisfy the formula $p[\nu \mapsto v]$. Our refinements are first-order formulas with equality, uninterpreted functions, and linear arithmetic.

When it is unambiguous from the context, we use b to abbreviate $\{\nu : b \mid \text{true}\}$.

Blocks. A block c describes the contents of a heap location as a sequence of bindings from indices i to refinement types τ. Each binding $i : \tau$ states that a value of type τ is contained in the block at each offset n from the start of the block which is described by the index i. Bindings of the form $m : \tau$, where types are bound to singleton indices, refer to exactly one element within the block; as such, we allow the refinement formulas within the same block to refer to the value at offset m using the syntax $@m$. We require all indices in a block to be disjoint.

Heap Types. Heap types h statically describe the contents of run-time heaps as sets of bindings from heap locations l to blocks c. A heap type is either the empty heap ε or a heap h extended with a binding from location l to block c, written $h * l \mapsto c$. The location l is either an *abstract location* $\tilde{l}$, which corresponds to arbitrarily many run-time heap locations, or a *concrete location* l_j, which corresponds to exactly one run-time heap location. The distinction between abstract and concrete locations is used to implement a form of sound strong updates on heap types in the presence of aliasing, unbounded collections, and concurrency; we defer detailed discussion to Section 4.4. Locations may be bound at most once in a heap.

Heap Effects. A heap effect Σ describes the effect of an expression on a set of heap locations as a set of bindings from heap locations l to effect formulas p. A heap effect is either the empty effect ε or an effect Σ extended with a binding from location $\tilde{l}$ to an *effect formula* p over the in-scope program variables and the value variable ν, written $\Sigma * \tilde{l} \mapsto p$. Intuitively, an effect binding $\tilde{l} \mapsto p$ describes the effect of an expression on location $\tilde{l}$ as the set of pointers v such that the formula $p[\nu \mapsto v]$ is valid. Note that we only bind abstract locations in heap effects.

Effect Formulas. The formula portion of an effect binding can describe the effect of an expression with varying degrees of precision, from simply describing whether the expression accesses the location at all, to describing which offsets into a location are accessed, to describing which offsets into a location are accessed and how they are accessed (*e.g.* whether they are read or written).

For example, if we use the function $\mathrm{BB}(\nu)$ to refer to the beginning of the block where ν points, we can record the fact that expression e accesses the first ten items in the block at location $\tilde{l}$ (for either reading or writing) with the effect binding

$$\tilde{l} \mapsto \mathrm{BB}(\nu) \leq \nu \wedge \nu < \mathrm{BB}(\nu) + 10,$$

i.e., by stating that all pointers used by e to accesses location $\tilde{l}$ satisfy the above formula. To record that an expression does not access location $\tilde{l}$ at all, we ascribe it the effect formula false.

We add further expressiveness to our language of effect formulas by enriching it with effect-specific predicates used to describe the particular effects that occurred within a location. We use the unary predicates $\mathrm{Read}(v)$ and $\mathrm{Write}(v)$ to indicate that pointer v was read from or written to, respectively. Using these predicates, we can precisely state how an expression depends upon or affects the contents of a heap location.

For example, we can specify that an expression does not write to a location $\tilde{l}$ using the effect binding

$$\tilde{l} \mapsto \neg\mathrm{Write}(\nu).$$

On the other hand, we may specify precisely which offsets into $\tilde{l}$ are written by using Write to guard a predicate describing a set of pointers into $\tilde{l}$. For example, the effect binding

$$\tilde{l} \mapsto \text{Write}(v) \Rightarrow (\text{BB}(v) \leq v \wedge v < \text{BB}(v) + 10)$$

describes an expression which writes to the first ten elements of the block at $\tilde{l}$.

Effect predicates like Read and Write may only appear in heap effect bindings; they may not appear in type refinements.

Function Schemes. A *NanoC$_{eff}$ function scheme*

$$(\overline{x_i : \tau_i})/h_1 \ \rightarrow \ \tau/h_2/\Sigma$$

is the type of functions which take arguments x_i of corresponding types τ_i in a heap of type h_1, return values of type τ in a heap of type h_2, and incur side-effects described by the heap effect Σ. The types of function parameters may depend on the other function parameters, and the type of the input heap may depend on the parameters. The types of the return value, output heap, and heap effect may depend on the types of the parameters. We implicitly quantify over all the location names appearing in a function scheme.

4.4 Type System

In this section, we present the typing rules of *NanoC$_{eff}$*. We begin with a discussion of *NanoC$_{eff}$*'s type environments. We then discuss *NanoC$_{eff}$*'s expression typing rules, paying particular attention to how the rules carefully track side effects. Finally, we describe how we use tracked effects to ensure determinism.

Type Environments. Our typing rules make use of three types of environments. A *local environment*, Γ, is a sequence of *type bindings* of the form $x : \tau$ recording the types of in-scope variables and *guard formulas* p recording any branch conditions under which an expression is evaluated. A global environment, Φ, is a sequence of function type bindings of the form $f : S$ which maps functions to their refined type signatures. Finally, an effect environment Π is a pair $(\mathbb{E}, \mathbb{C})$. The first component in the pair, $\mathbb{E}$, is a set of effect label predicates. The second component in the pair, $\mathbb{C}$, is a set of pairs of effect label predicates such that $(E_1, E_2) \in \mathbb{C}$ states that effect E_1 *commutes* with E_2. By default, $\mathbb{E}$ includes the built-in effects, Read and Write, and $\mathbb{C}$ includes the pair $(\text{Read}, \text{Read})$, which states that competing reads to the same address in different threads produce a result that is independent of their ordering. In our typing rules, we implicitly assume a single global effect environment Π.

Local environments are well-formed if each bound type or guard formula is well-formed in the environment that precedes it. An effect environment $(\mathbb{E}, \mathbb{C})$ is well-formed as long as $\mathbb{C}$ is symmetric and only references effects in the set $\mathbb{E}$, *i.e.,* $\mathbb{C} \subseteq \mathbb{E} \times \mathbb{E}$, and $(E_1, E_2) \in \mathbb{C}$ iff $(E_2, E_1) \in \mathbb{C}$.

$$\Gamma ::= \varepsilon \mid x : \tau ; \Gamma \mid p ; \Gamma \qquad \text{(Local Environment)}$$

$$\Phi ::= \varepsilon \mid \mathtt{f} : S ; \Phi \qquad \text{(Global Environment)}$$

$$\mathbb{E} ::= \{\text{Read}, \text{Write}\} \mid E ; \mathbb{E}$$

$$\mathbb{C} ::= (\text{Read}, \text{Read}) \mid (E, E) ; \mathbb{C}$$

$$\Pi ::= (\mathbb{E}, \mathbb{C}) \qquad \text{(Effect Environment)}$$

4.4.1 Typing Judgments

We now discuss the rules for ensuring type well-formedness, subtyping, and typing expressions. Due to space constraints, we only present the formal definitions of the most pertinent rules, and defer the remainder to the accompanying technical report [29].

Subtyping Judgments. The rules for subtyping refinement types are straightforward: type τ_1 is a subtype of τ_2 in environment Γ, written $\Gamma \vdash \tau_1 <: \tau_2$, if 1) τ_1's refinement implies τ_2 under the assumptions recorded as refinement types and guard predicates in Γ and 2) τ_1's index is included in τ_2's when both are interpreted as sets.

Heap type h_1 is a subtype of h_2 in environment Γ, written $\Gamma \vdash h_1 <: h_2$, if both heaps share the same domain and, for each location l, the block bound to l in h_1 is a subtype of the block bound to l in h_2. Subtyping between blocks is pairwise subtyping between the types bound to each index in each block; we use substitutions to place bindings to offsets $@i$ in the environment.

Because effect formulas are first-order formulas, subtyping between effects is simply subtyping between refinement types: an effect Σ_1 is a subtype of Σ_2 in environment Γ, written $\Gamma \vdash \Sigma_1 <: \Sigma_2$, if each effect formula bound to a location in Σ_1 implies the formula bound to the same location in Σ_2 using the assumptions in Γ.

Type Well-Formedness. The type well-formedness rules of $NanoC_{eff}$ ensure that all refinement and effect formulas are well-scoped and that heap types and heap effects are well-defined maps over locations. These rules are straightforward; we briefly discuss them below, deferring their formal definitions [29].

A heap effect Σ is well-formed with respect to environment Γ and heap h, written $\Gamma, h \vdash \Sigma$, if it binds only abstract locations which are bound in h, binds a location at most once, and binds locations to effect formulas which are well-scoped in Γ. Further, an effect formula p is well-formed in an effect environment $(\mathbb{E}, \mathbb{C})$ if all effect names in p are present in $\mathbb{E}$.

A refinement type τ is well-formed with respect to Γ, written $\Gamma \vdash \tau$, if its refinement predicate references only variables contained in Γ. A heap type h is well-formed in environment Γ, written $\Gamma \vdash h$, if it binds any location at most once, binds a corresponding abstract location for each concrete location it binds, and contains only well-formed refinement types.

A "world" consisting of a refinement type, heap type, and heap effect is well-formed with respect to environment Γ, written $\Gamma \vdash \tau/h/\Sigma$, if, with respect to the environment Γ, the type and heap type are well-formed and the heap effect Σ is well-formed with respect to the heap h.

The rule for determining the well-formedness of function type schemes is standard.

Pure Expression Typing. The rules for typing a pure expression a in an environment Γ, written $\Gamma \vdash a : \tau$, are straightforward, and are deferred to [29]. These rules assign each pure expression a refinement type that records the exact value of the expression.

Expression Typing and Effect Tracking. Figure 4.5 shows the rules for typing expressions which explicitly manipulate heap effects to track or compose side-effects. Each expression is typed with a refinement type, a refinement heap assigning types to the heap's contents after evaluating the expression, and an effect which records how the expression accesses each heap location as a mapping from locations to effect formulas. We use the abbreviation `void` to indicate the type $\{\nu : \texttt{int}(0) \mid \text{true}\}$.

Pointer dereference expressions $*v$ are typed by the rule [T-READ]. The value v is typed to find the location and index into which it points in the heap; the type of the expression is the type at this location and index. The rule records the read's effect in the heap by creating a new binding to v's abstract location, $\tilde{l}$, and uses the auxiliary function

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

$$\frac{\Gamma \vdash v : \mathtt{ref}(l_j, i) \qquad h = h_1 * l_j \mapsto \ldots, i : \tau, \ldots}{\Phi,\Gamma,h \vdash *v : \tau/h/\tilde{l} \mapsto \mathit{logEffect}(v, \text{Read})}\ [\text{T-Read}]$$

$$\frac{\begin{array}{c} h = h_1 * l_j \mapsto \ldots, n{:}b, \ldots \qquad \Gamma \vdash v_1 : \mathtt{ref}(l_j, n) \\ \Gamma \vdash v_2 : b \qquad h' = h_1 * l_j \mapsto \ldots, n{:}\{\nu :\ b \mid \nu = v_2\}, \ldots \end{array}}{\Phi,\Gamma,h \vdash *v_1 := v_2 : \mathtt{void}/h'/\tilde{l} \mapsto \mathit{logEffect}(v_1, \text{Write})}\ [\text{T-SUpd}]$$

$$\frac{\Gamma \vdash v_1 : \mathtt{ref}(l_j, n^{+m}) \qquad \Gamma \vdash v_2 : \hat{\tau} \qquad h = h_1 * l_j \mapsto \ldots, n^{+m} : \hat{\tau}, \ldots}{\Phi,\Gamma,h \vdash *v_1 := v_2 : \mathtt{void}/h/\tilde{l} \mapsto \mathit{logEffect}(v_1, \text{Write})}\ [\text{T-WUpd}]$$

$$\frac{\Phi,\Gamma,h \vdash e_1 : \tau_1/h_1/\Sigma_1 \qquad \Phi,\Gamma;x{:}\tau_1,h_1 \vdash e_2 : \hat{\tau}_2/\hat{h}_2/\hat{\Sigma}_2 \qquad \Gamma \vdash \hat{\tau}_2/\hat{h}_2/\hat{\Sigma}_2}{\Phi,\Gamma,h \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \hat{\tau}_2/\hat{h}_2/\Sigma_1 \oplus \hat{\Sigma}_2}\ [\text{T-Let}]$$

$$\frac{\begin{array}{c} \Phi,\Gamma,h \vdash e_1 : \tau_1/\hat{h}'/\hat{\Sigma}_1 \qquad \Phi,\Gamma,h \vdash e_2 : \tau_2/\hat{h}'/\hat{\Sigma}_2 \\ \Phi,\Gamma,\hat{h}' \vdash \hat{\Sigma}_{\{1,2\}} \qquad \Phi,\Gamma \vdash \mathit{OK}(\hat{\Sigma}_1,\hat{\Sigma}_2) \qquad h \text{ abstract} \end{array}}{\Phi,\Gamma,h \vdash e_1 \parallel e_2 : \mathtt{void}/\hat{h}'/\hat{\Sigma}_1 \oplus \hat{\Sigma}_2}\ [\text{T-Par}]$$

$$\frac{\begin{array}{c} \Gamma \vdash v_1 : \mathtt{int}(i) \qquad \Gamma \vdash v_2 : \mathtt{int}(i) \\ \Gamma_1 = \Gamma;x{:}\{\nu :\ \mathtt{int}(i) \mid v_1 \le \nu < v_2\} \qquad \Phi,\Gamma_1,h \vdash e : \tau/h/\Sigma \\ y \text{ fresh} \qquad \Gamma_2 = \Gamma_1;y{:}\{\nu :\ \mathtt{int}(i) \mid v_1 \le \nu < v_2 \wedge \nu \ne x\} \\ \Phi,\Gamma_2 \vdash \mathit{OK}(\Sigma, \Sigma[x \mapsto y]) \\ \Phi,\Gamma,h \vdash \tau/h/\hat{\Sigma}' \qquad \Gamma_1 \vdash \Sigma <: \hat{\Sigma}' \qquad h \text{ abstract} \end{array}}{\Phi,\Gamma,h \vdash \textbf{for each}\ x\ \textbf{in}\ v_1\ \textbf{to}\ v_2\ \{e\} : \mathtt{void}/h/\hat{\Sigma}'}\ [\text{T-Foreach}]$$

$$\frac{\begin{array}{c} \Gamma \vdash v : \{\nu :\ \mathtt{ref}(\tilde{l}, i_y) \mid \nu \ne 0\} \\ h = h_0 * \tilde{l} \mapsto \overline{n_k : \tau_k}, \overline{i^+ : \tau^+} \qquad \theta = [\overline{@n_k \mapsto x_k}] \qquad \overline{x_k} \text{ fresh} \\ \Gamma_1 = \Gamma;\overline{x_k : \theta\tau_k} \qquad l_j \text{ fresh} \qquad h_1 = h * l_j \mapsto \overline{n_k : \{\nu = x_k\}}, \overline{i^+ : \theta\tau^+} \\ \Phi,\Gamma_1;x{:}\{\nu :\ \mathtt{ref}(l_j, i_y) \mid \nu = v\}, h_1 \vdash e : \hat{\tau}_2/\hat{h}_2/\hat{\Sigma} \qquad \Gamma_1 \vdash h_1 \\ \Gamma \vdash \hat{\tau}_2/\hat{h}_2/\hat{\Sigma} \qquad \Sigma = \hat{\Sigma} \oplus \{(\tilde{l} \mapsto \mathit{logEffect}(\text{BB}(\nu) + n_k, \text{Read}))\}_{n_k} \end{array}}{\Phi,\Gamma,h \vdash \textbf{letu}\ x = \textbf{unfold}\ v\ \textbf{in}\ e : \hat{\tau}_2/\hat{h}_2/\Sigma}\ [\text{T-Unfold}]$$

$$\frac{h = h_0 * \tilde{l} \mapsto \hat{c}_1 * l_j \mapsto c_2 \qquad \Gamma \vdash c_2 <: \hat{c}_1}{\Phi,\Gamma,h \vdash \textbf{fold}\ L : \mathtt{void}/h_0 * \tilde{l} \mapsto \hat{c}_1/\varepsilon}\ [\text{T-Fold}]$$

Figure 4.5: **Typing Rules for** $\mathit{NanoC}_{\mathit{eff}}$

Effects Checking $\boxed{\Gamma \vdash OK(\Sigma_1, \Sigma_2)}$

$$\frac{\Psi(E_1,E_2) = \pi(p_1,E_1) \wedge \pi(p_2,E_2) \qquad \forall (E_1,E_2) \in (\mathbb{E} \times \mathbb{E}) \setminus \mathbb{C}.\mathsf{Unsat}([\![\Gamma]\!] \wedge \Psi(E_1,E_2))}{\Gamma \vdash OK(p_1,p_2)}$$

$$\frac{\forall l \in Dom(\Sigma_1 \oplus \Sigma_2).\Gamma \vdash OK(\mathrm{LU}(\Sigma_1,l),\mathrm{LU}(\Sigma_2,l))}{\Gamma \vdash OK(\Sigma_1,\Sigma_2)}$$

Effects Operations

$$\begin{aligned}
\Sigma_1 \oplus \Sigma_2 &\doteq \{l \mapsto \mathrm{LU}(\Sigma_1,l) \vee \mathrm{LU}(\Sigma_2,l)\}_{l \in \bigcup Dom(\Sigma_{1,2})} \\
\mathrm{LU}(\Sigma,l) &\doteq \begin{cases} \Sigma(l) & l \in Dom(\Sigma) \\ \text{false} & \text{o.w.} \end{cases} \\
\pi(p,E) &\doteq p[E \mapsto \mathrm{Check}] \wedge \mathrm{Check}(\nu) \\
\mathit{logEffect}(v,E) &\doteq E(\nu) \wedge \nu = v \wedge_{E' \in \mathbb{E} \setminus \{E\}} \neg E'(\nu)
\end{aligned}$$

Figure 4.6: NanoC_{eff} **Rules for Effectful Operations**

logEffect to create an effect formula which states that the only location that is accessed by this dereference is exactly that pointed to by v (*i.e.,* $\nu = v$), that the location is read (*i.e.,* $\mathrm{Read}(\nu)$), and that no other effect occurs.

The rules for typing heap-mutating expressions of the form $*v_1 := v_2$, [T-SUPD] and [T-WUPD], are similar to [T-READ]. The two rules for mutation differ only in whether they perform a strong update on the type of the heap, *i.e.,* writing through a pointer with a singleton index strongly updates the type of the heap, while writing through a pointer with a sequence index does not.

Expressions are sequenced using the **let** construct, typed by rule [T-LET]. The majority of the rule is standard; we discuss only the portion concerning effects. The rule types expressions e_1 and e_2 to yield their respective effects Σ_1 and Σ_2. The effect of the entire **let** expression is the composition of the two effects, $\Sigma_1 \oplus \Sigma_2$, defined in Figure 4.6. We check that Σ_2 is well-formed in the initial environment to ensure that the variable x does not escape its scope.

Rule [T-PAR] types the parallel composition expression $e_1 \parallel e_2$. Expressions e_1 and e_2 are typed to obtain their effects Σ_1 and Σ_2. We then use the *OK* judgment, defined in Figure 4.6, to verify that effects Σ_1 and Σ_2 commute, *i.e.,* that the program remains deterministic regardless of the interleaving of the two concurrently-executing expressions. We give $e_1 \parallel e_2$ the effect $\Sigma_1 \oplus \Sigma_2$. We require that the input heap for a parallel composition expression must be *abstract*, that is, contain only bindings for parallel compositions; this forces the expressions which are run in parallel to unfold any locations they access, which in turn enforces several invariants that will prevent the type system from unsoundly assuming invariants in one thread which may be broken by the heap writes performed in another. We elaborate on this below.

Rule [T-FOREACH] types the **foreach** parallel loop expression. The loop induction variable i ranges over values between v_1 and v_2, exclusive; we type the body expression e in the environment enriched with an appropriate binding for i and compute the resulting heap and per-iteration effect Σ. We require that the heap is loop-invariant. To ensure that the behavior of the **foreach** loop is deterministic, we must check that the effects of distinct iterations do not interfere. Hence, we check non-interference at two arbitrary, distinct iterations by adding a binding for a fresh name j to the environment with the invariant that $i \neq j$, and verifying with *OK* that the effect at an iteration i, Σ, commutes with the effect at a distinct iteration j, $\Sigma[i \mapsto j]$. We return Σ', which subsumes Σ but is well-formed in the original environment, ensuring that the loop induction variable i does not escape its scope. As with [T-PAR], we require that the input heap is abstract, for the same reasons.

Strong Updates and Concurrency The rules [T-UNFOLD] and [T-FOLD] are used to implement a local non-aliasing discipline for performing strong updates on the types of heap locations when only one pointer to the location is accessed at a time. The mechanism is similar to freeze/thaw and adopt/focus [63, 16]. These rules are treated in previous work [53]; we now discuss how these rules handle effects and briefly recap their handling of strong updates.

Rule [T-UNFOLD] types the **letu** construct for unfolding a pointer to an abstract location, $\tilde{l}$, to obtain a pointer to a corresponding concrete location, l_j, bound to the variable x. [T-UNFOLD] constructs the block bound to l_j by creating a skolem variable x_j for each binding of a singleton index n_j to a type τ_j; the binding is a singleton index in a concrete location, so it corresponds to exactly one datum on the heap. We apply an appropriate substitution to the elements within the block, then typecheck the body expression e with respect to the extended heap and enriched environment. Well-formedness constraints ensure that an abstract location is never unfolded twice in the same scope and that x does not escape its scope.

We allow two concurrently-executing expressions to unfold, and thus simultaneously access, the same abstract location. When a location is unfolded, our type system records the refinement types bound to each of its singleton indices in the environment. If we do not take care, the refinement types bound to singleton indices and recorded in the environment may be invalidated by an effect (*e.g.* a write) occurring in a simultaneously-executing expression.Thus, an expression which unfolds a location *implicitly* depends on the data whose invariants are recorded in its environment at the time of unfolding. To capture these implicit dependencies, when a pointer v is unfolded, we conservatively record a pseudo-read [62] at each singleton offset n_k within the block into which v points by recording in the heap effect that the pointer $\mathrm{BB}(v) + n_k$ is read, where $\mathrm{BB}(v)$ is the beginning of the memory block where v points. Then, the invariant recorded in the environment at the unfolding is preserved, as any violating writes would get caught by the determinism check.

Rule [T-FOLD] removes a concrete location from the heap so that a different pointer to the same abstract location may be unfolded. The rule ensures that the currently-unfolded concrete location's block is a subtype of the corresponding abstract location's so that we can be sure that the abstract location's invariant holds when it is unfolded later. The **fold** expression has no heap effect.

Program Typing. The remaining expression forms do not manipulate heap effects; as their typing judgments are straightforward and covered in previous work [53], we defer their definition [29]. We note that the rule for typing **if** expressions records the value of the branch condition in the environment when typing each branch. The judgments for typing function declarations and programs in $NanoC_{eff}$ are straightforward and are deferred to [29].

4.4.2 Handling Effects

We now describe the auxiliary definitions used by the typing rules of Section 4.4.1 for checking effect noninterference.

Combining Effects. In Figure 4.6, we define the $\oplus$ operator for combining effects. Our decision to encode effects as formulas in first-order logic makes the definition of this operator especially simple: for each location l, the combined effect of Σ_1 and Σ_2 on location l is the disjunction of of the effects on l recorded in Σ_1 and Σ_2. We use the auxiliary function $\mathrm{LU}(\Sigma, l)$ to look up the effect formula for location l in Σ; if there is no binding for l in Σ, the false formula, indicating no effect, is returned instead.

Effects Checking. We check the noninterference of heap effects Σ_1, Σ_2 using the OK judgment, defined in Figure 4.6. The effects are noninterfering if the formulas bound to locations in both heap effects are pairwise noninterfering. Two effect formulas p_1 and p_2

Table 4.1: **CSOLVE: Liquid Effects Experimental Results**

Program	LOC	Quals	Annot	Changes	T (s)
Reduce	39	7	7	N/A	8.8
SumReduce	39	1	3	0	1.8
QuickSort	73	3	4	N/A	5.9
MergeSort	95	6	7	0	32.4
IDEA	222	7	5	3	59.7
K-Means	458	7	10	16	63.7

are noninterfering as follows. For each pair of non-commuting effects E_1 and E_2 in the set $\mathbb{E}$, we project the set of pointers in p_1 (resp., p_2) which are accessed with effect E_1 (resp., E_2) using the effect projection operator π. Now, if the conjunction of the projected formulas is unsatisfiable, the effect formulas are noninterfering.

User-Defined Effects and Commutativity. We allow our set of effects $\mathbb{E}$ to be extended with additional effect label predicates by the user. To specify how these effects interact, commutativity annotations can be provided that indicate which effects do not result in nondeterministic behavior when run simultaneously. Then, the user may override the effect of any function by providing an effect Σ as an annotation, allowing additional flexibility to specify domain-specific effects or override the effect system if needed.

4.5 Evaluation

In Table 4.1, **(LOC)** is the number of source code lines as reported by *sloccount*, **(Quals)** is the number of logical qualifiers required to prove memory safety and determinism, **(Annot)** is the number of annotated function signatures plus the number of effect and commutativity declarations. **(Changes)** is the number of program modifications, **(T)** is the time in seconds taken for verification.

We have implemented the techniques in this chapter as an extension to CSOLVE, a

```
<region r1,r2,r3 | r1:* # r3:*, r2:* # r3:*> void
merge(DPJArrayInt<r1> a, DPJArrayInt<r2> b, DPJArrayInt<r3> c)
  reads r1:*, r2:* writes r3:* {
  if (a.length <= merge_size)
    seq_merge(a, b, c);
  else {
    int ha=a.length/2, sb=split(a.get(ha),b);
    final DPJPartitionInt<r1> a_split=new DPJPartitionInt<r1>(a,hd);
    final DPJPartitionInt<r2> a_split=new DPJPartitionInt<r2>(b,sb);
    final DPJPartitionInt<r3> c_split=new DPJPartitionInt<r3>(c,ha+sb);
    cobegin { merge(a_split.get(0),b_split.get(0),c_split.get(0));
              merge(a_split.get(1),b_split.get(1),c_split.get(1)); }}}
```

Figure 4.7: DPJ merge function

```
qualif Q(V: ptr) : &&[_ <= V; V {<, >=} (_ + _ + _)]

void merge(int* ARRAY LOC(L) a,
           int* ARRAY LOC(L) b,
           int la, int lb, int* ARRAY c) {
  if (la <= merge_size){
    seq_merge(a, b, la, lb, c);
  } else {
    int ha = la / 2, sb = split(a[ha],b,lb);
    cobegin {
      merge(a,b,ha,sb,c);
      merge(a+ha,b+sb,la-ha,lb-sb,c+ha+sb); }}}
```

Figure 4.8: CSOLVE merge function

refinement type-based verifier for C. CSOLVE takes as input a C program and a set of logical qualifiers, or formulas over the value variable v, to use in refinement and effect inference. CSOLVE then checks the program both for memory safety errors (*e.g.* out-of-bounds array accesses, null pointer dereferences) and non-determinism. If CSOLVE determines that the program is free from such errors, it outputs an annotation file containing the types of program variables, heaps, functions, and heap effects. If CSOLVE cannot determine that the program is safe, it outputs an error message with the line number of the potential error and the inferred types of the program variables in scope at that location.

Type and Effect Inference. CSOLVE infers refinement types and effect formulas using the predicate abstraction-based Liquid Types [53] technique; we give a high-level overview here. We note that there are expressions whose refinement types and heap effects cannot be constructed from the types and heap effects of subexpressions or from bindings in the heap and environment but instead must be *synthesized*. For example, the function typing rule requires us to synthesize types, heaps, and heap effects for functions. This is comparable to inferring pre- and post-conditions for functions (and similarly, loop invariants), which reduces to determining appropriate formulas for refinement types and effects. To make inference tractable, we require that formulas contained in synthesized types, heaps, and heap effects are *liquid*, *i.e.,* conjunctions of *logical qualifier* formulas provided by the user. These qualifiers are templates for predicates that may appear in types or effects. We read our inference rules as an algorithm for constructing subtyping constraints which, when solved, yield a refinement typing for the program.

Methodology To evaluate our approach, we drew from the set of Deterministic Parallel Java (DPJ) benchmarks reported in [2]. For those which were parallelized using the methods we support, namely heap effects and commutativity annotations, we verified determinism and memory safety using our system. Our goals for the evaluation were to show that our system was as expressive as DPJ's for these benchmarks while requiring less program restructuring.

Results The results of running CSOLVE on the following benchmarks are presented in Table 4.1. `Reduce` is the program given in Section 4.2. `SumReduce` initializes an array using parallel iteration, then sums the results using a parallel divide-and-conquer strategy. `MergeSort` divides an array into quarters, recursively sorting each in parallel. It then merges pairs of sorted quarters in parallel, and finally merges the two sorted halves to yield a single sorted array. The final merges are performed recursively in parallel. `QuickSort` is a standard in-place quicksort adapted from a sequential version included with DPJ. We parallelized the algorithm by partitioning the input array around its median, then recursively sorting each partition in parallel. `K-Means` was adapted from the STAMP benchmarks [38], the C implementation that was ported for DPJ. `IDEA` is an encryption/decryption kernel ported to C from DPJ.

Changes Some benchmarks required modifications in order to conform to the current implementation of CSOLVE; this does not indicate inherent limitations in the technique. These include: multi-dimensionalizing flattened arrays, writing stubs for matrix **malloc**, expanding structure fields (`K-Means`), and soundly abstracting non-linear operations and `#define`-ing configuration parameters to constants to work around a bug in the SMT solver (`IDEA`).

Annotations CSOLVE requires two classes of annotations to compute base types for the program. First, the annotation `ARRAY` is required to distinguish between pointers to singletons and pointers to arrays. Second, as base type inference is intraprocedural, we must additionally specify which pointer parameters may be aliased by annotating them with a location parameter of the form `LOC(L)`.

Qualitative Comparison Since the annotations used by CSOLVE and DPJ are very different, a quantitative comparison between them is not meaningful. Instead, we illustrate the kinds of annotations used by each tool and qualitatively compare them. Figures 4.7, 4.8 contain DPJ, and CSOLVE code, resp., implementing the recursive `Merge` routine from `MergeSort`, including region annotations and required wrapper classes. In the latter, `Merge` takes two arrays `a` and `b` and recursively splits them, sequentially merging them into array `c` at a target size. In particular, each statement in the cobegin writes to a different contiguous interval of `c`.

In DPJ, verifying this invariant requires: First, wrapping all arrays with an API class `DPJArrayInt`, and then wrapping each contiguous subarray with another class `DPJArrayPartitionInt`. This must be done because DPJ does not support precise effects over partitions of native Java arrays. Second, the method must be explicitly declared to be polymorphic over named regions `r1`, `r2`, and `r3`, corresponding to the memory locations in which the formals reside. Finally, `Merge` must be explicitly annotated with the appropriate effects, *i.e.,* it reads `r1` and `r2` and writes `r3`.

In CSOLVE, verifying this invariant requires: First, specifying that `a` and `b` are potentially aliased arrays (the annotation `LOC(L) ARRAY`). Second, we specify the qualifiers used to synthesize refinement types and heap effects via the line starting with `qualif`. This specification says that a predicate of the given form may appear in a type or effect, with the wildcard _ replaced by any program identifier in scope for that type or effect. Using the given qualifier, CSOLVE infers that `a` and `b`'s location is only read, and that `c`'s location is only written at indices described by the formula $c \leq \nu < c + la + lb$, which suffices to prove determinism.

Unlike DPJ, CSOLVE does not require invasive changes to code (*e.g.* explicit array partitioning), and hence supports domain- and program-specific sharing patterns (*e.g.* memory coalescing from Figure 4.2). However, this comes at the cost of providing qualifiers. In future work, abstract interpretation may help lessen this burden.

4.6 Related Work

The literature on checking determinism (and other properties) of multithreaded programs generally fall into two categories: static and dynamic. While there has been much exciting work on dynamic mechanisms, including work at the architecture [41], operating system [7] and language runtime [9, 34] levels, for relevance, we limit our discussion to static mechanisms for checking and enforcing determinism for relevance. In particular the literature that pertains to reasoning about heap disjointness: type-and-effect systems, semantic determinism checking, and other closely related techniques.

Named Regions. Region-based approaches assign references (or objects) to distinct segments of the heap, which are explicitly named and manipulated by the programmer. This approach was introduced in the context of adding impure computations to a functional language, and developed as a means of controlling the side-effects incurred by segments of code in sequential programs [33, 35]. Effects were also investigated in the context of safe, programmer-controlled memory management [58, 28, 32]. Ideas from this work led to the notion of abstract heap locations [63, 24, 16] and our notion of fold and unfold.

Ownership Types. In the OO setting, regions are closely related to *ownership types* which use the class hierarchy of the program to separate the heap into disjoint, nested regions [13, 56]. In addition to isolation, ownership types can be used to track effects [12], and to reason about data races and deadlocks [10, 4, 36].

Such techniques can be used to enforce determinism [57], but regions and ownership relations are not enough to enforce fine-grained separation. Instead, we must precisely track relations between program variables. We are inspired by DPJ [2], which shows how some sharing patterns can be captured in a dependent region system. However, we show the full expressiveness of refinement type inference and SMT solvers can be brought to bear to enable complex, low-level sharing with static determinism guarantees.

Checking Properties of Multithreaded Programs. Several authors have looked into type-and-effect systems for checking other properties of multithreaded programs. For example, [22, 47] show how types can be used to prevent races, [23] describes an effect discipline that encodes Lipton's Reduction method for proving atomicity. Our work focuses on the higher-level semantic property of determinism. Nevertheless, it would be useful to understand how race-freedom and atomicity could be used to establish determinism. Others [50, 3] have looked at proving that different blocks of operations *commute*. In future work, we could use these to automatically generate effect labels and commutativity constraints.

Program Logics and Abstract Interpretation. There is a vast literature on the use of logic (and abstract interpretation) to reason about sets of addresses (*i.e.,* the heap). The literature on logically reasoning about the heap includes the pioneering work in TVLA [66], separation logic [49], and the direct encoding of heaps in first-order logic [30, 31], or with explicit sets of addresses called dynamic frames [26]. These logics have also been applied to reason about concurrency and parallelism: [48] looks at using separation logic to obtain (deterministic) parallel programs, and [66] uses abstract interpretation to find loop invariants for multithreaded, heap-manipulating Java programs. The above look at analyzing disjointness for linked data structures. The most closely related to our work is [61], which uses intra-procedural numeric abstract interpretation to determine the set of array indices used by different threads, and then checks disjointness over the domains.

Our work differs in that we show how to consolidate all the above lines of work into a uniform location-based heap abstraction (for separation logic-style disjoint structures) with type refinements that track finer-grained invariants. Unlike [48], we can verify access patterns that require sophisticated arithmetic reasoning, and unlike [61] we can check separation between disjoint structures, and even indices drawn from compound structures like arrays, lists and so on. Our type system allows "context-sensitive" reasoning about (recursive) procedures. Further, first-order refinements allow us to verify domain-specific sharing patterns via first class effect labels [35].

4.7 Conclusion

In this chapter, we broadened the scope of Liquid Types from the verification of pure functional programs in a high-level language to the verification of impure imperative programs in a low-level language. We did this by carefully combining several powerful constructs: precise models of typed heaps with concrete and abstract locations, a fold and unfold mechanism for coping with temporary invariant violation, and Liquid Types for inferring precise data structure invariants. We demonstrated that this combination enables the largely-automatic verification of memory safety properties through several realistic examples requiring precise reasoning about invariants of in-memory data structures.

Acknowledgements

This chapter contains material adapted from the publication: Ming Kawaguchi, Patrick Rondon, Ranjit Jhala. "Deterministic Parallelism via Liquid Effects", *Proceedings of*

the 2012 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), pages 45–54, 2012. The dissertation author was principal researcher and author on this publication. This chapter also contains material adapted from the publication: 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 a co-author on this publication.

Chapter 5

Conclusions and Impact

In the introduction to this dissertation, we claimed that there exists an impedance mismatch between the high-level properties that programmers would like to be true of their programs and the tools, including language-level mechanisms, that they use to implement their programs. We then posited that such high-level property specifications, and hence the impedance mismatch, is due to the low level of source code mechanisms available for implementing and specifying properties of data structures such as sorted lists, balanced trees, acyclic graphs, disjoint in-heap arrays, and so on.

Encoding. In the preceding chapters, we introduced four methods for encoding high-level invariants of data structure as refinement types for the purpose of program verification.

In Chapter 2, we described Recursive Refinements for encoding properties of recursive structures such as sorted lists and binary search trees. In Chapter 3, we described Measures for encoding *non-boolean*, but well-founded inductive properties of structures such as the size of sets and height of trees. Then, we described Polymorphic Refinements for encoding properties of linked structures such as the acyclicity of graphs and dependencies between keys and values in arrays and hashmaps. In Chapter 4, we described Liquid Effects, a type-and-effects system for C programs that encodes memory access patterns over heaps, and how it can be used to guarantee the determinism of fork-join concurrency.

Verification. Further, we showed that each method was compatible with Liquid Type Inference [52, 54], an algorithm for inferring refinement types such that a program is safe if a "Liquid" typing for the program can be inferred. We then showed that, with minimal programmer annotation, our refinement type encoding and Liquid Type Inference enable static program verification.

Finally, we showed that our techniques are practical and efficient by implementing them as two tools: first, DSOLVE for verifying complex data structure properties in OCAML programs, and second, CSOLVE for verifying complex data structure properties in the presence of fork-join concurrency in *C* programs.

Practicality. Using DSOLVE, we verified the safety of thousands of lines of challenging benchmarks, including both canonical data structure implementations and production data structure libraries, such as the OCAML standard library's `Map` module. Further, we described how DSOLVE found a subtle bug in a production library, `vec`, and was used to find a minimal test case and fix that was reported to and adopted by the library authors.

Efficiency. Using CSOLVE and DSOLVE, we automatically verified 20 challenging OCAML and *C* benchmarks (totaling 1754 lines and 926 lines, resp.) in a cumulative total of 470 seconds. Further, we showed that this performance could be reached with an extremely low programmer burden, totaling only 126 total lines of annotation, or 5% of total code size.

Contributions. In this dissertation, we claim that we have produced lasting solutions that mitigate the fundamental impedance mismatch between high-level programmer intent and the limits of language expressiveness. In particular, we posit that this mismatch is due to the difficulty of expressing and verifying the maintenance of high-level data structure invariants and propose novel, practical and efficient solutions for both encoding data structure invariants and automatically verifying that source code maintains these invariants in two realistic settings.

Finally, we experimentally show that these methods are practical, efficient and novel by implementing them into two tools, DSOLVE and CSOLVE to prove, for the first time: First, the safety of multiple OCAML standard library and production modules. Second, we describe how we discovered a bug in production OCAML module `vec`, such that a programmer-intended data structure invariant was not preserved due to a subtle programmer error on a single path in a complex piece of source code. Third, we describe a method for automatically proving the safety of complex concurrent memory access patterns that use pointer arithmetic to enable code performance in environments such as general purpose GPU programming.

Prior to our work. the state of the art in data structure invariant encoding and verification relied on intractable or manual, tedious methods: First, universal quantification over predicates in first order logic for which no complete decision procedure exists and heuristic solvers are impractically slow and unpredictable; Second, hand-written proofs in domain-specific logics [40] or worse, higher-order logical sentences [67]; Third, multiply exponential abstract interpretation using graph domains [55].

Our work. In this dissertation, we showed that a significant class of data structure invariants can be encoded and automatically verified by piggybacking quantifier-free logical predicates over a source language's base types. Our method enables the following novel capabilities. First, we allow quantifier-free predicates to be used to describe unbounded structures in which previous methods require universal quantification. Second, we introduce measures to automatically describe proof terms using predicate-free statements in well-founded procedures verified against a source program's syntax, rather than a specialized interactive proof language. Third, we introduce polymorphic refinements to enable decidable use of a crucial class of existentially quantified predicates for maps and linked structures. We then show experimentally that verification in this framework is practical and efficient.

Finally, we introduce Liquid Effects, a type-and-effect system that precisely and expressively describes the footprint of fork-join style concurrently executing threads. We show qualitatively and quantitatively that programming with deterministic parallelism in this framework has a low programmer burden while remaining provably correct.

Impact. In particular, we argue that results derived from our three key mechanisms — Recursive Refinements, Polymorphic Refinements, and most importantly, Measures — have enabled a wide range of subsequent advancement inthe automatic inference of arbitrary, non-Liquid, refinement types in data structure manipulating code [68], decidable synthesis of programs performing challenging tasks such as manipulating data structures and effectful tracking of information flow [45, 46], verify numerous challenging HASKELL benchmarks under lazy evaluation and a non-Hindley-Milner base type system [59, 60], and automatic verification linked data structure invariants in imperative programs using logically refined alias types with a separation logic fragment to establish physical typing of the heap [8].

Bibliography

[1] Nvidia cuda programming guide.

[2] Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA*, 2009.

[3] Farhana Aleen and Nathan Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *ASPLOS*, 2009.

[4] Zachary R. Anderson, David Gay, Robert Ennals, and Eric A. Brewer. Sharc: checking data sharing strategies for multithreaded c. In *PLDI*, 2008.

[5] A. W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4), 1998.

[6] L. Augustsson. Cayenne - a language with dependent types. In *ICFP*, 1998.

[7] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[8] Alexander Bakst and Ranjit Jhala. Predicate abstraction for linked data structures. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, pages 65–84, 2016.

[9] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, 2010.

[10] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, 2002.

[11] Manuel Chakravarty, Gabriele Keller, Roman Lechtchinsky, and W. Pfannenstiel. Nepal: Nested data parallelism in haskell. In *Euro-Par*, 2001.

[12] David G. Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, 2002.

[13] David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. In *ECOOP*, 2001.

[14] Luca de Alfaro. Vec: Extensible, functional arrays for ocaml. http://www.dealfaro.com/vec.html.

[15] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[16] R. DeLine and M.A. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, 2001.

[17] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report Research Report 159, Compaq Systems Research Center, December 1998.

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

[19] John Field and G. Ramalingam. Identifying procedural structure in cobol programs. In *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE '99, Toulouse, France, September 6, 1999*, pages 1–10, 1999.

[20] J.C. Filliâtre. Ocaml software. http://www.lri.fr/~filliatr/software.en.html.

[21] C. Flanagan. Hybrid type checking. In *POPL*. ACM, 2006.

[22] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *PLDI*, pages 219–232, 2000.

[23] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.

[24] J.S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI*, 2002.

[25] F. Henglein. Type inference with polymorphic recursion. *ACM TOPLAS*, 15(2):253–289, 1993.

[26] Bart Jacobs, Frank Piessens, Jan Smans, K. Rustan M. Leino, and Wolfram Schulte. A programming model for concurrent object-oriented programs. *TOPLAS*, 2008.

[27] Ranjit Jhala and Rupak Majumdar. Bit-level types for high-level reasoning. In *FSE 2006: Foundations of Software Engineering*, page to appear. ACM, 2006.

[28] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX*, 2002.

[29] M. Kawaguchi, P. Rondon, A. Bakst, and R. Jhala. Liquid effects: Technical report. http://goto.ucsd.edu/~rjhala/liquid.

[30] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL*, 2008.

[31] Shuvendu K. Lahiri, Shaz Qadeer, and David Walker. Linear maps. In *PLPV*, 2011.

[32] Chris Lattner and Vikram S. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI*, 2005.

[33] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects, 2002.

[34] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In *SOSP*, pages 327–336, 2011.

[35] Daniel Marino and Todd D. Millstein. A generic type-and-effect system. In Andrew Kennedy and Amal Ahmed, editors, *TLDI*, pages 39–50. ACM, 2009.

[36] Jean-Phillipe Martin, Michael Hicks, Manuel Costa, Periklis Akritidis, and Miguel Castro. Dynamically checking ownership policies in concurrent c/c++ programs. In *POPL*, pages 457–470, 2010.

[37] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[38] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[39] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Symposium on Programming*, pages 217–228, 1984.

[40] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP*, 2008.

[41] Adrian Nistor, Darko Marinov, and Josep Torrellas. Instantcheck: Checking the determinism of parallel programs using on-the-fly incremental hashing. In *MICRO*, pages 251–262, 2010.

[42] C. Okasaki. *Purely Functional Data Structures*. CUP, 1999.

[43] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[44] B. C. Pierce and D. N. Turner. Local type inference. In *POPL*, pages 252–265, 1998.

[45] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 522–538, 2016.

[46] Nadia Polikarpova, Jean Yang, Shachar Itzhaky, and Armando Solar-Lezama. Type-driven repair for information flow security. *CoRR*, abs/1607.03445, 2016.

[47] Polyvios Pratikakis, Jeffrey S. Foster, and Michael W. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI*, 2006.

[48] Mohammad Raza, Cristiano Calcagno, and Philippa Gardner. Automatic parallelization with separation logic. In *ESOP*, pages 348–362, 2009.

[49] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[50] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *TOPLAS*, 19(6), 1997.

[51] P. Rondon, M. Kawaguchi, and R. Jhala. Recursive refinements. http://pho.ucsd.edu/rjhala/recursive-refinements-techrep.pdf.

[52] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[53] P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, 2010.

[54] Patrick Rondon. *Liquid Types*. PhD thesis, University of California, San Diego, 2012.

[55] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[56] Matthew Smith. Towards an effects system for ownership domains. In *In ECOOP Workshop - FTfJP 2005*, 2005.

[57] Tachio Terauchi and Alex Aiken. A capability calculus for concurrency and determinism. *TOPLAS*, 30, 2008.

[58] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages, 1993.

[59] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. Abstract refinement types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 209–228, 2013.

[60] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 269–282, 2014.

[61] Martin T. Vechev, Eran Yahav, Raghavan Raman, and Vivek Sarkar. Automatic verification of determinism for structured parallel programs. In *SAS*, 2010.

[62] Jan Voung, Ravi Chugh, Ranjit Jhala, and Sorin Lerner. Dataflow analysis for concurrent programs using data race detection. In *PLDI*, 2008.

[63] D. Walker and J.G. Morrisett. Alias types for recursive data structures. 2000.

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

[65] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.

[66] Eran Yahav and Mooly Sagiv. Verifying safety properties of concurrent heap-manipulating programs. *TOPLAS*, 32(5), 2010.

[67] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, 2008.

[68] He Zhu, Aditya V. Nori, and Suresh Jagannathan. Learning refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 400–411, 2015.