UNIVERSITY OF CALIFORNIA SAN DIEGO

Type-directed Program Synthesis

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

in

Computer Science

by

Tristan Knoth

Committee in charge:

Professor Nadia Polikarpova, Chair
Professor Sam Buss
Professor Cormac Flanagan
Professor Ranjit Jhala
Professor Sorin Lerner

2023

The Dissertation of Tristan Knoth is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

DEDICATION

To Mom, Dad, and Anya.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First I would like to thank my advisor Nadia Polikarpova for her insight, patience, and enthusiasm when I needed them. I would also like to thank my committee and my collaborators, particularly Zheng Guo, Di Wang, Jan Hoffmann, and Adam Reynolds. A short-lived collaboration with Michael James inspired much of the work in Section 3.6.2. Thanks as well to the PL group at UCSD for always pushing me – especially the 3148 crew, even though y'all left me behind.

I could not have done it without my friends here in San Diego and elsewhere. Thank you all – especially Erik, Kyle, Mia, John, Andi, Anish, Elizabeth, Alex, Mario, Matt, Josh, Sunil, and Zé – for making these years (mostly) a blast. The burritos, beers, shows, memories, pick-and-rolls, and dawn patrols we shared made San Diego a special place. That said, it's been a tough journey: I needed everybody I mentioned and many others along the way.

Less personally, thanks to whoever first put french fries in a burrito for fueling my body and mind. I'd also like to thank every UC union organizer for empowering me and so many others to fight.

Chapter 1 is, in part, a reprint of the material as it appears in Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. ACM, 2019. The dissertation author was a primary author and investigator of this work.

Chapter 2 is, in part, a reprint of the material as it appears in Proceedings of the ACM on Programming Languages, Volume 4, Issue ICFP Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann, and Nadia Polikarpova. ACM, 2020. The dissertation author was a primary author and investigator of this work.

Chapter 3 is, in part, a reprint of material currently in submission. Tristan Knoth, Zheng Guo, and Nadia Polikarpova. The dissertation author was a primary author and investigator of this work.

VITA

2017	Bachelor of Arts, Computer Science and Mathematics Grinnell College
2019	Masters of Science, Computer Science University of California San Diego
2023	Doctor of Philosophy, Computer Science University of California San Diego

ABSTRACT OF THE DISSERTATION

Type-directed Program Synthesis

by

Tristan Knoth

Doctor of Philosophy in Computer Science

University of California San Diego, 2023

Professor Nadia Polikarpova, Chair

Program synthesis tools automate programming itself, generating executable code from a high-level specification. In recent years, type systems have proved an effective specification style for the synthesis of a wide variety of functional programs. Nonetheless, a number of problems remain.

In this dissertation we address holes in the type-directed synthesis literature. First, we consider the problem of generating *efficient* programs: we design a resource analysis enabling the implementation of a synthesizer that can produce more efficient programs than other tools. This approach to resource analysis is automatic and highly expressive, subsuming a number of existing techniques as well as automating the verification of bounds that previously would have required

manual proofs. Second, we address the problem of *specification*: despite recent successes in type-directed synthesis, existing specification styles each have their own shortcomings. In order to enable further experimentation with new approaches to specification we design and implement a framework for building type-directed synthesizers. This work reduces the problem of implementing an efficient synthesizer to that of implementing a typechecker: the framework handles search. Our framework makes search techniques reusable across compatible type systems, facilitating the implementation of an efficient synthesizer.

Introduction

Program synthesis tools offer users the ability to automate certain aspects of software development. A synthesizer takes a declarative specification and searches for a satisfying program. There are synthesis tools that allow users to provide input-output examples, natural language, or formal specifications describing their intent [143, 37, 109, 42, 108].

Type systems are particularly well-suited for specifying synthesis problems of recursive functional programs. They are very flexible, allowing users to specify a variety of functional properties. Moreover, they provide unbounded guarantees about the behavior of the resulting program. While a number of tools [101, 43, 107, 72] have shown that types can specify a variety of properties, there are still issues with existing techniques. First, existing specification styles often fall short of describing interesting programs, or they get unwieldy for more complex programs. Second, search remains a hard problem: these tools often rely on domain-specific insight to generate complex programs.

This dissertation claims that types can be a viable foundation for more realistic synthesis tools. First, we show how a type-directed approach can fill in one of the primary gaps in the synthesis literature: the synthesis of *efficient* programs. Then, we build a framework that unifies a number of existing synthesis tools in order to enable reuse of search techniques and easy experimentation with specification styles.

Chapter 1: Resource-guided Program Synthesis

A useful synthesizer will need to reason about *resource consumption* while generating programs. Programmers know that analyzing a program's runtime or memory consumption

is vital. Chapter 1 presents the first such resource-aware program synthesizer. We discuss the design and implementation of Resyn, which synthesizes programs from a functional specification and an upper-bound on resource usage. Resyn is able to generate more efficient implementations of a number of benchmarks than other tools can.

Chapter 2: Liquid Resource Types

Chapter 1 introduces a novel resource analysis designed to automatically verify resource bounds on recursive functional programs. This technique is automatic and expressive: the typechecker can automatically verify dependent resource bounds that previously required manual proof. In Chapter 2 we introduce *liquid resource types*, a technique that *unifies* our work on Resyn with existing techniques for automated amortized resource analysis (AARA) [62]. The resulting tool can automatically verify a much wider variety of resource bounds. Not only does it unify the analyses provided by several instantiations of AARA – each previously requiring different tools – it also automates for the first time the validation of rich dependent bounds.

Chapter 3: Type-Directed Program Synthesis

Chapter 1 showed how to implement a synthesizer for a resource type system in order to fill a gap in the synthesis literature. A number of other works [101, 43, 107, 72] make analagous contributions, developing a synthesizer for some rich type system.

In this chapter we unify these projects via a *franework* for turning a typechecker into a synthesizer. Under our model, a programmer can implement a compatible typechecker, and the framework provides search capabilities turning it into a synthesizer. This helps address the two most important problems in synthesis: it makes it easier to experiment with new styles of specification, and it allows the reuse of novel search techniques between tools. We use our framework to implement clones of three existing synthesizers, each of which is competitive if not faster than the original while requiring significantly less engineering effort. We also show how the framework opens the door to further experimentation with specifications by *combining* type systems in order to address some issues with existing specification techniques.

Chapter 1

Resource-Guided Program Synthesis

1.1 Introduction

In recent years, *program synthesis* has emerged as a promising technique for automating low-level aspects of programming [50, 126, 132]. One of the greatest challenges in software development is to write programs that are not only correct but also efficient with respect to memory usage, execution time, or domain specific resource metrics. For this reason, automatically optimizing program performance has long been a goal of program synthesis, and several existing techniques tackle this problem for *low-level straight-line code* [117, 104, 119, 105, 20] or add efficient synchronization to concurrent programs [24, 55, 25, 41]. However, the developed techniques are not applicable to recent advances in the synthesis of *high-level* looping or recursive programs manipulating custom data structures [82, 101, 42, 107, 74, 109]. These techniques lack the means to analyze and understand the resource usage of the synthesized programs. Consequently, they cannot take into account the program's efficiency and simply return the first program that arises during the search and satisfies the functional specification.

In this work, we study the problem of synthesizing high-level recursive programs given both *a functional specification of a program and a bound on its resource usage.* A naive solution would be to first generate a program using conventional program synthesis and then use existing automatic static resource analyses [66, 103, 29] to check whether its resource usage satisfies the bound. Note, however, that for recursive programs, both synthesis and resource analysis are

undecidable in theory and expensive in practice. Instead, in this chapter we propose *resource-guided synthesis*: an approach that tightly integrates program synthesis and resource analysis, and uses the resource bound to guide the synthesis process, generating programs that are efficient by construction.

Type-Driven Synthesis

In a nutshell, the idea of this work is to combine type-driven program synthesis*, pioneered in the work on* SYNQUID *[107], with type-based automatic amortized resource analysis (AARA) [71, 77, 64, 67] as implemented in Resource Aware ML (RaML) [63].* Type-driven synthesis and AARA are a perfect match because they are both based on decidable, constraint-based type systems that can be easily checked with off-the-shelf constraint solvers.

In SYNQUID, program specifications are written as *refinement types* [134, 86]. The key to efficient synthesis is *round-trip type checking*, which uses an SMT solver to aggressively prune the search space by rejecting partial programs that do not meet the specification (see Section 3.2.2). Until now, types have only been used in the context of synthesis to specify functional properties.

AARA is a type-based technique for automatically deriving symbolic resource bounds for functional programs. The idea is to add resource annotations to data types, in order to specify a potential function that maps values of that type to non-negative numbers. The type system ensures that the initial potential is sufficient to cover the cost of the evaluation. By a priori fixing the shape of the potential functions, type inference can be reduced to linear programming (see Section 1.2.2).

The Re^2 Type System

The first contribution of this chapter is a new type system, which we dub Re^2*—for* re*finements and* re*sources—that combines polymorphic refinement types with AARA (Section 1.3).* Re^2 is a conservative extension of SYNQUID's refinement type system and RaML's affine type system with linear potential annotations. As a result, Re^2 can express logical assertions that are

required for effectively specifying program synthesis problems. In addition, the type system features annotations of numeric sort in the same refinement language to express potential functions. Using such annotations, programmers can express precise resource bounds that go beyond the template potential functions of RaML.

The features that distinguish Re^2 from other refinement-based type systems for resource analysis [103, 29, 110] are (1) the combination of logical and quantitative refinements and (2) the use of AARA, which simplifies resource constraints and naturally applies to non-monotone resources like memory that can become available during the execution. These features also pose nontrivial technical challenges: the interaction between substructural and dependent types is known to be tricky [88, 90], while polymorphism and higher-order functions are challenging for AARA (one solution is proposed in [77], but their treatment of polymorphism is not fully formalized).

In addition to the design of Re^2, we prove the soundness of the type system with respect to a small-step cost semantics. In the formal development, we focus on a simple call-by-value functional language with Booleans and lists, where type refinements are restricted to linear inequalities over lengths of lists. However, we structure the formal development to emphasize that Re^2 can be extended with user-defined data types, more expressive refinements, or non-linear potential annotations. The proof strategy itself is a contribution of this chapter. The type soundness of the logical refinement part of the system is inspired by TiML [103]. The main novelty is the soundness proof of the potential annotations using a small-step cost semantics instead of RaML's big-step evaluation semantics.

Type-Driven Synthesis with Re^2

The second contribution of this chapter is a resource-guided synthesis algorithm based on Re^2. In Section 3.5, we first develop a system of *synthesis rules* that prescribe how to derive well-typed programs from Re^2 types, and prove its soundness *wrt.* the Re^2 type system. We then show how to algorithmically derive programs using a combination of backtracking search and

constraint solving. In particular this requires solving a new form of constraints we call *resource constraints*, which are constrained linear inequalities over unknown numeric refinement terms. To solve resource constraints, we develop a custom solver based on counter-example guided inductive synthesis [127] and SMT [35].

The RESYN Synthesizer

The third contribution of this chapter is the implementation and experimental evaluation of the first resource-aware synthesizer for recursive programs. We implemented our synthesis algorithm in a tool called RESYN, which takes as input (1) a *goal type* that specifies the logical refinements and resource requirements of the program, and (2) types of *components* (*i.e.* library functions that the program may call). RESYN then synthesizes a program that provably meets the specification (assuming the soundness of components).

To evaluate the scalability of the synthesis algorithm and the quality of the synthesized programs, we compare RESYN with baseline SYNQUID on a variety of general-purpose data structure operations, such as eliminating duplicates from a list or computing common elements between two lists. The evaluation (Section 2.4) shows that RESYN is able to synthesize programs that are asymptotically more efficient than those generated by SYNQUID. Moreover, the tool scales better than a naive combination of synthesis and resource analysis.

1.2 Background and Overview

This section provides the necessary background on type-driven program synthesis (Section 3.2.2) and automatic resource analysis (Section 1.2.2). We then describe and motivate their combination in Re^2 and showcase novel features of the type system (Section 1.2.3). Finally, we demonstrate how Re^2 can be used for resource-guided synthesis (Section 1.2.4).

```
1 common = λ l1 . λ l2 . match l1 with Nil → Nil
2   Cons x xs → if ¬(member x l2)
3       then common xs l2
4       else Cons x (common xs l2)
```

Figure 1.1. Synthesized program that computes common elements between two lists

1.2.1 Type-Driven Program Synthesis

Type-driven program synthesis [107] is a technique for automatically generating functional programs from their high-level specifications expressed as *refinement types* [86, 115]. For example, a programmer might describe a function that computes the common elements between two lists using the following type signature:

```
common :: l1: List a → l2: List a
  → {ν: List a | elems ν = elems l1 ∩ elems l2}
```

Here, the return type of `common` is *refined* with a predicate which restricts the set of elements of the output list ν[1] to be the intersection of the sets of elements of the two arguments. Here `elems` is a user-defined logic-level function, also called *measure* [78, 134]. In addition to the *synthesis goal* above, the synthesizer takes as input a *component library*: signatures of data constructors and functions it can use. In our example, the library includes the list constructors `Nil` and `Cons` and the function

```
member :: x: a → l: List a → {Bool | ν = (x in elems l)}
```

which determines whether a given value is in the list. Given this goal and components, the type-driven synthesizer SYNQUID [107] produces an implementation of `common` in Figure 1.1.

The Synthesis Mechanism

Type-driven synthesis works by systematically exploring the space of programs that can be built from the component library and validating candidate programs against the goal type using a variant of liquid type inference [115]. To validate a program against a refinement

[1] Hereafter the bound variable of the refinement is always called ν and the binding is omitted.

type, liquid type inference generates a system of *subtyping constraints* over refinement types. The subtyping constraints are then reduced to implications between refinement predicates. For example, checking `common xs l2` in line 3 of Figure 1.1 against the goal type reduces to validating the following implication:

$$(\mathsf{elems}\ l_1 = \{x\} \cup \mathsf{elems}\ xs) \wedge (x \notin \mathsf{elems}\ l_2) \wedge (\mathsf{elems}\ \nu = \mathsf{elems}\ xs \cap \mathsf{elems}\ l_2) \implies \mathsf{elems}\ \nu = \mathsf{elems}\ l_1 \cap \mathsf{elems}\ l_2$$

Since this formula belongs to a decidable theory of uninterpreted functions and arrays, its validity can be checked by an SMT solver [35]. In general, the generated implications may contain unknown predicates. In this case, type inference reduces to a system of *constrained horn clauses* [17], which can be solved via predicate abstraction.

Synthesis and Program Efficiency

The program in Figure 1.1 is correct, but not particularly efficient: it runs roughly in time $n{\cdot}m$, where m is the length of `l1` and n is the length of `l2`, since it calls the `member` function (a linear scan) for every element of `l1`. The programmer might realize that keeping the input lists *sorted* would enable computing common elements in linear time by scanning the two lists in parallel. To communicate this intent to the synthesizer, they can define the type of (strictly) sorted lists by augmenting a traditional list definition with a simple refinement:

```
data SList a where SNil :: SList a
  SCons :: x: a → xs: SList {a | x < ν} → SList a
```

This definition says that a sorted list is either empty, or is constructed from a head element `x` and a tail list `xs`, as long as `xs` is sorted and all its elements are larger than `x`.[2] Given an updated synthesis goal (where `selems` is a version of `elems` for `SList`)

```
common' :: l1: SList a → l2: SList a
```

[2]Following SYNQUID, our language imposes an implicit constraint on all type variables to support equality and ordering. Hence, they cannot be instantiated with arrow types. This could be lifted by adding type classes.

```
1 common' = λ l1 . λ l2 . match l1 with SNil → Nil
2   SCons x xs → match l2 with SNil → Nil
3     SCons y ys →
4       if x < y then common' xs l2
5                else if y < x then common' l1 ys
6                                  else Cons x (common' xs ys)
```

Figure 1.2. A more efficient version of the program in Figure 1.1 for sorted lists

$$\rightarrow \{\nu\texttt{: List a} \mid \texttt{elems } \nu = \texttt{selems l1} \cap \texttt{selems l2}\}$$

and a component library that includes `List`, `SList`, and $<$ (but not `member`!), SYNQUID can synthesize an efficient program shown in in Figure 1.2.

However, if the programmer leaves the function `member` in the library, SYNQUID will synthesize the inefficient implementation in Figure 1.1. In general, SYNQUID explores candidate programs in the order of size and returns the first one that satisfies the goal refinement type. This can lead to suboptimal solutions, especially as the component library grows larger and allows for many functionally correct programs. To avoid inefficient solutions, the synthesizer has to be aware of the resource usage of the candidate programs.

1.2.2 Automatic Amortized Resource Analysis

To reason about the resource usage of programs we take inspiration from *automatic amortized resource analysis (AARA)* [71, 77, 64, 67]. AARA is a state-of-the-art technique for automatically deriving symbolic resource bounds on functional programs, and is implemented for a subset of OCaml in Resource Aware ML (RaML) [67, 63]. For example, RaML is able to automatically derive the worst-case bound $2m + n{\cdot}m$ on the number of recursive calls for the function `common` and $m+n$ for `common'` [3].

[3] In this section we assume for simplicity that the resource of interest is the number of recursive calls. Both AARA and our type system support user-defined cost metrics (see Section 1.3 for details).

Potential Annotations

AARA is inspired by the *potential method* for manually analyzing the worst-case cost of a sequence of operations [131]. It uses annotated types to introduce potential functions that map program states to non-negative numbers. To derive a bound, we have to statically ensure that the potential at every program state is sufficient to cover the cost of the next transition and the potential of the following state. In this way, we ensure that the initial potential is an upper bound on the total cost.

The key to making this approach effective is to closely integrate the potential functions with data structures [71, 77]. For instance, in RaML the type $L^1(\texttt{int})$ stands for a list that contains one unit of potential for every element. This type defines the potential function $\phi(\ell{:}L^1(\texttt{int})) = 1 \cdot |\ell|$. The potential can be used to pay for a recursive call (or, in general, cover resource usage) or to assign potential to other data structures.

Bound Inference

Potential annotations can be derived automatically by starting with a symbolic type derivation that contains fresh variables for the potential annotations of each type, and applying syntax directed type rules that impose local constraints on the annotations. The integration of data structures and potential ensures that these constraints are linear even for polynomial potential annotations.

1.2.3 Bounding Resources with Re^2

To reason about resource usage in type-driven synthesis, we integrate AARA's potential annotations and refinement types into a novel type system that we call Re^2. In Re^2, a refinement type can be annotated with a *potential term* ϕ of numeric sort, which is drawn from the same logic as refinements. Intuitively, the type R^ϕ denotes values of refinement type R with ϕ units of potential. In the rest of this section we illustrate features of Re^2 on a series of examples, and delay formal treatment to Section 1.3.

With potential annotations, users can specify that `common'` must run in time at most $m+n$, by giving it the following type signature:

```
common' :: l1: SList a^1 → l2: SList a^1
  → {v: List a | elems v = selems l1 ∩ selems l2}
```

This type assigns one unit of potential to every element of the arguments `l1` and `l2`, and hence only allows making one recursive call per element of each list. Whenever resource annotations are omitted, the potential is implicitly zero: for example, the elements of the result carry no potential.

Our type checker uses the following reasoning to argue that this potential is sufficient to cover the efficient implementation in Figure 1.2. Consider the recursive call in line 4, which has a cost of one. Pattern-matching `l1` against `SCons x xs` transfers the potential from `l1` to the binders, resulting in types $\texttt{x} : a^1$ and $\texttt{xs} : \texttt{SList}\ (\{a \mid \texttt{x} < \nu\}^1)$. The unit of potential associated with `x` can now be used to pay for the recursive call. Moreover, the types of the arguments, `xs` and `l2`, match the required type $\texttt{SList}\ a^1$, which guarantees that the potential stored in the tail and the second list are sufficient to cover the rest of the evaluation. Other recursive calls are checked in a similar manner.

Importantly, the inefficient implementation in Figure 1.1 would not type-check against this signature. Assuming that `member` is soundly annotated with

```
member :: x: a → l: List a^1 → {Bool | v = (x in elems l)}
```

(requiring a unit of potential per element of `l`), the guard in line 2 consumes all the potential stored in `l2`; hence the occurrence of `l2` in line 3 has the type $\texttt{List}\ a^0$, which is not a subtype of $\texttt{List}\ a^1$.

Dependent Potential Annotations

In combination with logical refinements and parametric polymorphism, this simple extension to the SYNQUID's type system turns out to be surprisingly powerful. Unlike in RaML,

```
append :: xs: List a^1 → ys: List a
  → {List a | len ν = len xs + len ys}

triple :: l: List Int^2 → {List n | len ν = 3*(len l)}
triple = λ l . append l (append l l)

tripleSlow :: l: List Int^3 → {List n | len ν = 3*(len l)}
tripleSlow = λ l . append (append l l) l
```

Figure 1.3. Append three copies of a list. The type of `append` specifies that it returns a list whose length is the sum of the lengths of its arguments. It also requires one unit of potential on each element of the first list. Moreover, `append` has a polymorphic type and can be applied to lists with different element types, which is crucial for type-checking `tripleSlow`.

potential annotations in Re^2 can be *dependent*, *i.e.* mention program variables and the special variable ν. Dependent annotations can encode fine-grained bounds, which are out of reach for RaML. As one example, consider function `range a b` that builds a list of all integers between a and b; we can express that it takes at most $b-a$ steps by giving the argument b a type $\{\texttt{Int} \mid \nu \geq a\}^{\nu-a}$. As another example, consider insertion into a sorted list `insert x xs`; we can express that it takes at most as many steps as there are elements in *xs* that are smaller than x, by giving *xs* the type $\texttt{SList}\ \alpha^{\mathsf{ite}(\nu<x,1,0)}$ (*i.e.* only assigning potential to elements that are smaller than x). These fine-grained bounds are checked completely automatically in our system, by reduction to constraints in SMT-decidable theories.

Polymorphism

Another source of expressiveness in Re^2 is parametric polymorphism: since potential annotations are attached to types, type polymorphism gives us *resource polymorphism* for free. Consider two functions in Figure 1.3, `triple` and `tripleSlow`, which implement two different ways to append a list `l` to two copies of itself. Both of them make use of a component function `append`, whose type indicates that it makes a linear traversal of its first argument. Intuitively, `triple` is more efficient that `tripleSlow` because in the former both calls to `append` traverse a list of length n, whereas in the latter the outer call traverses a list of length $2{\cdot}n$. This difference is reflected in the signatures of the two functions: `tripleSlow` requires

three units of potential per list element, while `triple` only requires two.

Checking that `tripleSlow` satisfies this bound is somewhat nontrivial because the two applications of `append` must have *different types*: the outer application must return `List Int`, while the inner application must return `List Int`^{1} (*i.e.* carry enough potential to be traversed by `append`). RaML's monomorphic type system is unable to assign a single general type to `append`, which can be used at both call sites. So the function has be reanalyzed at every (monomorphic) call site. Re^2, on the other hand, handles this example out of the box, since the type variable `a` in the type of `append` can be instantiated with `Int` for the outer occurrence and with `Int`^{1} for the inner occurrence, yielding the type

```
xs: List Int^2 → ys: List Int^1 → {List Int^1 | ...}
```

As a final example, consider the standard `map` function:

```
map :: (a → b) → List a → List b
```

Although this type has no potential annotations, it implicitly tells us something about the resource behavior of `map`: namely, that `map` applies a function to each list element *at most once*. This is because `a` can be instantiated with a type with an arbitrary amount of potential, and the only way to pay for this potential is with a list element (which also has type `a`).

1.2.4 Resource-Guided Synthesis with RESYN

We have extended SYNQUID with support for Re^2 types in a new program synthesizer RESYN. Given a resource-annotated signature for `common'` from Section 1.2.3 and a component library that includes `member`, RESYN is able to synthesize the efficient implementation in Figure 1.2. The key to efficient synthesis is type-checking each program candidate incrementally as it is being constructed, and discarding an ill-typed program prefix as early as possible. For example, while enumerating candidates for the function `common'`, we can safely discard the inefficient version from Figure 1.1 even *before* constructing the second branch of the conditional (because the first branch together with the guard use up too many resources). Hence, as we

explain in more detail in Section 3.5, a key technical challenge in ReSyn has been a tight integration of resources into Synquid's *round-trip type checking* mechanism, which aggressively propagates type information top-down from the goal and solves constraints incrementally as they arise.

Termination Checking

In addition to making the synthesizer resource-aware, Re^2 types also subsume and generalize Synquid's termination checking mechanism. To avoid generating diverging functions, Synquid uses a simple *termination metric* (the tuple of function's arguments), and checks that this metric decreases at every recursive call. Using this metric, Synquid is not able to synthesize the function `range` from Section 1.2.3, because it requires a recursive call that decreases the *difference* between the arguments, $b - a$. In contrast, ReSyn need not reason explicitly about termination, since potential annotations already encode an upper bound on the number of recursive calls. Moreover, the flexibility of these annotations enables ReSyn to synthesize programs that require nontrivial termination metrics, such as `range`.

1.3 The Re^2 Type System

In this section, we define a subset of Re^2 as a formal calculus to prove type soundness. This subset includes Booleans that are refined by their values, and lists that are refined by their lengths. The programs in Section 3.1 and Section 3.2 use Synquid's surface syntax. The gap from the surface language to the core calculus involves inductive types and refinement-level measures. The restriction to this subset in the technical development is only for brevity and proofs carry over to all the features of Synquid.

Syntax

Figure 2.6 presents the grammar of terms in Re^2 via abstract binding trees [58]. The core language is basically the standard lambda calculus augmented with Booleans and lists. A *value* $v \in \mathsf{Val}$ is either a boolean constant, a list of values, or a function. Expressions in Re^2 are in

$$
\begin{array}{l}
a ::= x \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{nil} \mid \mathsf{cons}(\hat{a}_h, a_t) \\
\hat{a} ::= a \mid \lambda(x.e_0) \mid \mathsf{fix}(f.x.e_0) \\
e ::= \hat{a} \mid \mathsf{if}(a_0, e_1, e_2) \mid \mathsf{matl}(a_0, e_1, x_h.x_t.e_2) \mid \mathsf{app}(\hat{a}_1, \hat{a}_2) \\
\quad \mid \mathsf{let}(e_1, x.e_2) \mid \mathsf{impossible} \mid \mathsf{tick}(c, e_0) \\
v ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{nil} \mid \mathsf{cons}(v_h, v_t) \mid \lambda(x.e_0) \mid \mathsf{fix}(f.x.e_0)
\end{array}
$$

Figure 1.4. Syntax of the core calculus

Refinement

$$\psi, \phi ::= x \mid \top \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid n \mid \psi_1 \leq \psi_2 \mid \psi_1 + \psi_2 \mid \psi_1 = \psi_2$$

Sort

$$\Delta ::= \mathbb{B} \mid \mathbb{N} \mid \delta_\alpha$$

Base Type

$$B ::= \mathsf{bool} \mid L(T) \mid m \cdot \alpha$$

Resource-Annotated Type

$$T ::= R^\phi$$

Refinement Type

$$R ::= \{B \mid \psi\} \mid m \cdot (x : T_x \to T)$$

Type Schema

$$S ::= T \mid \forall\alpha.S$$

Figure 1.5. Syntax of the type system

a-normal-form [116], which means that syntactic forms occurring in non-tail position allow only *atoms* $\hat{a} \in \mathsf{Atom}$, i.e., variables and values; this restriction simplifies typing rules for applications, as we explain below. We identify a subset SimpAtom of Atom that contains atoms *interpretable* in the refinement logic. Intuitively, the value of an $a \in \mathsf{SimpAtom}$ should be either a Boolean or a list. The syntactic form impossible is introduced as a placeholder for unreachable code, e.g., the else-branch of a conditional whose predicate is always true.

The syntactic form $\mathsf{tick}(c, e_0)$ is used to specify resource usage, and it is intended to cost $c \in \mathbb{Z}$ units of resource and then reduce to e_0. If the cost c is negative, then $-c$ units of resource will become available in the system. tick terms support flexible user-defined cost metrics: for example, to count recursive calls, the programmer may wrap every such call in $\mathsf{tick}(1, \cdot)$; to keep track of memory consumption, they might wrap every data constructor in $\mathsf{tick}(c, \cdot)$, where c is the amount of memory that constructor allocates.

Operational Semantics

The resource usage of a program is determined by a small-step operational cost semantics. The semantics is a standard one augmented with a *resource* parameter. A step in the evaluation judgment has the form $\langle e, q\rangle \mapsto \langle e', q'\rangle$ where e and e' are expressions and $q, q' \in \mathbb{Z}_0^+$ are nonnegative integers. For example, the following is the rule for $\mathsf{tick}(c, e_0)$.

$$\frac{}{\langle \mathsf{tick}(c, e_0), q\rangle \mapsto \langle e_0, q - c\rangle}$$

The *multi-step* evaluation relation $\mapsto^*$ is the reflexive transitive closure of $\mapsto$. The judgment $\langle e, q\rangle \mapsto^* \langle e', q'\rangle$ expresses that with q units of available resources, e evaluates to e' without running out of resources and q' resources are left. Intuitively, the high-water mark resource usage of an evaluation of e to e' is the minimal q such that $\langle e, q\rangle \mapsto^* \langle e', q'\rangle$. For monotone resources like time, the cost is the sum of costs of all the evaluated tick expressions. In general, this net cost is invariant, that is, $p - p' = q - q'$ if $\langle e, p\rangle \mapsto^n \langle e', p'\rangle$ and $\langle e, q\rangle \mapsto^n \langle e', q'\rangle$, where $\mapsto^n$ is the relation obtained by self-composing $\mapsto$ for n times.

Refinements

We now combine SYNQUID's type system with AARA to reason about resource usage. Figure 1.5 shows the syntax of the Re^2 type system. Refinements ψ are distinct from program terms and classified by sorts Δ. Re^2's sorts include Booleans $\mathbb{B}$, natural numbers $\mathbb{N}$, and *uninterpreted symbols* δ_α. Refinements can be logical formulas and linear expressions, which may reference program variables. Logical refinements ψ have sort $\mathbb{B}$, while potential annotations ϕ have sort $\mathbb{N}$. Re^2 interprets a variable of Boolean type as its value, list type as its length, and type variable α as an uninterpreted symbol with a corresponding sort δ_α. We use the following

interpretation $\mathscr{I}(\cdot)$ to reflect interpretable atoms $a \in \mathsf{SimpAtom}$ in the refinement logic:

$$\begin{aligned} \mathscr{I}(x) &= x \\ \mathscr{I}(\mathsf{true}) &= \top \qquad\qquad \mathscr{I}(\mathsf{nil}) = 0 \\ \mathscr{I}(\mathsf{false}) &= \bot \qquad \mathscr{I}(\mathsf{cons}(\_, a_t)) = \mathscr{I}(a_t) + 1 \end{aligned}$$

Types

We classify types into four categories. Base types B include Booleans, lists and type variables. Type variables α are annotated with a *multiplicity* $m \in \mathbb{Z}_0^+ \cup \{\infty\}$, which denotes an upper bound on the number of usages of a variable like in bounded linear logic [47]. For example, $L(2 \cdot \alpha)$ denotes a universal list whose elements can be used at most twice.

Refinement types are *subset types* and *dependent arrow types*. The inhabitants of the subset type $\{B \mid \psi\}$ are values of type B that satisfy the refinement ψ. The refinement ψ is a logical predicate over program variables and a special *value variable* ν, which does not appear in the program and stands for the inhabitant itself. For example, $\{\mathsf{bool} \mid \nu\}$ is a type of true, and $\{L(\mathsf{bool}) \mid \nu \leq 5\}$ represents Boolean lists of length at most 5. Dependent arrow types $x : T_x \to T$ are function types whose return type may reference the formal argument x. As type variables, these function types are also annotated with a multiplicity $m \in \mathbb{Z}_0^+ \cup \{\infty\}$ restricting the number of times the function may be applied.

To apply the potential method of amortized analysis [130], we need to define potentials with respect to the data structures in the program. We introduce *resource-annotated types* as a refinement type augmented with a potential annotation, written R^ϕ. Intuitively, R^ϕ assigns ϕ units of potential to values of the refinement type R. The potential annotation ϕ may also reference the value variable ν. For example, $L(\mathsf{bool})^{5 \times \nu}$ describes Boolean lists ℓ with $5|\ell|$ units of potential where $|\ell|$ is the length of ℓ. The same potential can be expressed by assigning 5 units of potential to every element using the type $L(\mathsf{bool}^5)$.

$$\frac{\Gamma(x) = \{B \mid \psi\}^{\phi}}{\Gamma \vdash x : B}\ (\textsc{SimpAtom-Var}) \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ (\textsc{SimpAtom-True}) \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ (\textsc{SimpAtom-False}) \qquad \frac{\Gamma \vdash T\ \mathsf{type}}{\Gamma \vdash \mathsf{nil} : L(T)}\ (\textsc{SimpAtom-Nil})$$

$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash \hat{a}_h :: T \qquad \Gamma_2 \vdash a_t : L(T)}{\Gamma \vdash \mathsf{cons}(\hat{a}_h, a_t) : L(T)}\ (\textsc{SimpAtom-Cons})$$

Figure 1.6. Base typing rules: $\Gamma \vdash a : B$

Type schemas represent (possibly) polymorphic types. Note that the type quantifier $\forall$ can only appear outermost in a type.

Similar to SYNQUID, we introduce a notion of *scalar* types, which are resource-annotated base types refined by logical constraints. Intuitively, interpretable atoms are scalars and Re^2 only allows the refinement-level logic to reason about values of scalar types. We will abbreviate $1 \cdot \alpha$ as α, $\{B \mid \top\}$ as B, $\infty \cdot (x\!:\!T_x \to T)$ as $x\!:\!T_x \to T$, and R^0 as R.

Typing Rules

In Re^2, the *typing context* Γ is a sequence of variable bindings $x : S$, type variables α, path conditions ψ, and free potentials ϕ. Our type system consists of five judgments: sorting, well-formedness, subtyping, sharing, and typing. We omit sorting and well-formedness rules and include them in the technical report [83]. The sorting judgment $\Gamma \vdash \psi \in \Delta$ states that a refinement ψ has a sort Δ under a context Γ. A type S is said to be well-formed under a context Γ, written $\Gamma \vdash S$ type, if every referenced variable in it is in the correct scope.

Figure 1.7 presents selected typing rules for Re^2. The typing judgment $\Gamma \vdash e :: S$ states that the expression e has type S in context Γ. The intuitive meaning is that if there is *at least* the amount resources as indicated by the potential in the context Γ then this suffices to evaluate e to a value v, and after the evaluation there are *at least* as many resources available as indicated by the potential in S. The auxiliary typing judgment $\Gamma \vdash a : B$, defined in Figure 2.9, assigns base types to interpretable atoms. Atomic typing is useful in the rule (T-SIMPATOM), which uses the interpretation $\mathscr{I}(\cdot)$ to derive a most precise refinement type for interpretable atoms.

(T-SimpAtom)
$$\frac{\Gamma \vdash a : B}{\Gamma \vdash a :: \{B \mid \nu = \mathscr{I}(a)\}}$$

(T-Var)
$$\frac{\Gamma(x) = S}{\Gamma \vdash x :: S}$$

(T-Imp)
$$\frac{\Gamma \models \bot \qquad \Gamma \vdash T \text{ type}}{\Gamma \vdash \mathsf{impossible} :: T}$$

(T-Consume-P)
$$\frac{c \geq 0 \qquad \Gamma \vdash e_0 :: T}{\Gamma, c \vdash \mathsf{tick}(c, e_0) :: T}$$

(T-Consume-N)
$$\frac{c < 0 \qquad \Gamma, -c \vdash e_0 :: T}{\Gamma \vdash \mathsf{tick}(c, e_0) :: T}$$

(T-Cond)
$$\frac{\Gamma \vdash a_0 : \mathsf{bool} \qquad \Gamma, \mathscr{I}(a_0) \vdash e_1 :: T \qquad \Gamma, \neg\mathscr{I}(a_0) \vdash e_2 :: T}{\Gamma \vdash \mathsf{if}(a_0, e_1, e_2) :: T}$$

(T-MatL)
$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma \vdash T' \text{ type} \qquad \Gamma_1 \vdash a_0 : L(T) \qquad \Gamma_2, \mathscr{I}(a_0) = 0 \vdash e_1 :: T' \qquad \Gamma_2, x_h : T, x_t : L(T), \mathscr{I}(a_0) = x_t + 1 \vdash e_2 :: T'}{\Gamma \vdash \mathsf{matl}(a_0, e_1, x_h.x_t.e_2) :: T'}$$

(T-Let)
$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma \vdash T_2 \text{ type} \qquad \Gamma_1 \vdash e_1 :: S_1 \qquad \Gamma_2, x : S_1 \vdash e_2 :: T_2}{\Gamma \vdash \mathsf{let}(e_1, x.e_2) :: T_2}$$

(T-App-SimpAtom)
$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash \hat{a}_1 :: 1 \cdot (x : \{B \mid \psi\}^{\phi} \to T) \qquad \Gamma_2 \vdash a_2 :: \{B \mid \psi\}^{\phi}}{\Gamma \vdash \mathsf{app}(\hat{a}_1, a_2) :: [\mathscr{I}(a_2)/x]T}$$

(T-App)
$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash \hat{a}_1 :: 1 \cdot (x : T_x \to T) \qquad \Gamma_2 \vdash \hat{a}_2 :: T_x \qquad \Gamma \vdash T \text{ type}}{\Gamma \vdash \mathsf{app}(\hat{a}_1, \hat{a}_2) :: T}$$

(T-Abs)
$$\frac{\Gamma \vdash T_x \text{ type} \qquad \Gamma, x : T_x \vdash e_0 :: T \qquad \vdash \Gamma \curlyvee \Gamma \mid \Gamma}{\Gamma \vdash \lambda(x.e_0) :: x : T_x \to T}$$

(T-Abs-Lin)
$$\frac{\Gamma \vdash T_x \text{ type} \qquad \Gamma, x : T_x \vdash e_0 :: T}{m \cdot \Gamma \vdash \lambda(x.e_0) :: m \cdot (x : T_x \to T)}$$

(T-Fix)
$$\frac{R = x : T_x \to T \qquad \Gamma \vdash R \text{ type} \qquad \Gamma, f : R, x : T_x \vdash e_0 :: T \qquad \vdash \Gamma \curlyvee \Gamma \mid \Gamma}{\Gamma \vdash \mathsf{fix}(f.x.e_0) :: R}$$

(S-Gen)
$$\frac{v \in \mathsf{Val} \qquad \Gamma, \alpha \vdash v :: S \qquad \Gamma, \alpha \vdash S \curlyvee S \mid S}{\Gamma \vdash v :: \forall \alpha.S}$$

(S-Inst)
$$\frac{\Gamma \vdash e :: \forall \alpha.S \qquad \Gamma \vdash \{B \mid \psi\}^{\phi} \text{ type}}{\Gamma \vdash e :: [\{B \mid \psi\}^{\phi}/\alpha]S}$$

(S-Subtype)
$$\frac{\Gamma \vdash e :: T_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash e :: T_2}$$

(S-Transfer)
$$\frac{\Gamma' \vdash e :: S \qquad \Gamma \models \Phi(\Gamma) = \Phi(\Gamma')}{\Gamma \vdash e :: S}$$

(S-Relax)
$$\frac{\Gamma \vdash e :: R^{\phi} \qquad \Gamma \vdash \phi' \in \mathbb{N}}{\Gamma, \phi' \vdash e :: R^{\phi + \phi'}}$$

Figure 1.7. Typing rules: $\Gamma \vdash e :: S$

$\boxed{\Gamma \vdash S \curlyvee S_1 \mid S_2}$

$$\frac{}{\Gamma \vdash \mathsf{bool} \curlyvee \mathsf{bool} \mid \mathsf{bool}}\ (\textsc{Share-Bool}) \qquad \frac{\Gamma \vdash T \curlyvee T_1 \mid T_2}{\Gamma \vdash L(T) \curlyvee L(T_1) \mid L(T_2)}\ (\textsc{Share-List}) \qquad \frac{\alpha \in \Gamma \qquad m = m_1 + m_2}{\Gamma \vdash m \cdot \alpha \curlyvee m_1 \cdot \alpha \mid m_2 \cdot \alpha}\ (\textsc{Share-TVar})$$

$$\frac{\Gamma, \alpha \vdash S \curlyvee S \mid S}{\Gamma \vdash \forall \alpha . S \curlyvee \forall \alpha . S \mid \forall \alpha . S}\ (\textsc{Share-Poly})$$

$$\frac{\Gamma \vdash B \curlyvee B_1 \mid B_2 \qquad \Gamma \vdash \{B \mid \psi\}\ \mathsf{type}}{\Gamma \vdash \{B \mid \psi\} \curlyvee \{B_1 \mid \psi\} \mid \{B_2 \mid \psi\}}\ (\textsc{Share-Subset}) \qquad \frac{\Gamma \vdash (x : T_x \to T)\ \mathsf{type} \qquad m = m_1 + m_2}{\Gamma \vdash (m \cdot (x : T_x \to T)) \curlyvee (m_1 \cdot (x : T_x \to T)) \mid (m_2 \cdot (x : T_x \to T))}\ (\textsc{Share-Arrow})$$

$$\frac{\Gamma \vdash R \curlyvee R_1 \mid R_2 \qquad \Gamma, \nu : R \models \phi = \phi_1 + \phi_2}{\Gamma \vdash R^{\phi} \curlyvee {R_1}^{\phi_1} \mid {R_2}^{\phi_2}}\ (\textsc{Share-Pot})$$

$\boxed{\Gamma \vdash T_1 <: T_2}$

$$\frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash L(T_1) <: L(T_2)}\ (\textsc{Sub-List}) \qquad \frac{\alpha \in \Gamma \qquad m_1 \geq m_2}{\Gamma \vdash m_1 \cdot \alpha <: m_2 \cdot \alpha}\ (\textsc{Sub-TVar}) \qquad \frac{\Gamma \vdash B_1 <: B_2 \qquad \Gamma, \nu : B_1 \models \psi_1 \implies \psi_2}{\Gamma \vdash \{B_1 \mid \psi_1\} <: \{B_2 \mid \psi_2\}}\ (\textsc{Sub-Subset})$$

$$\frac{\Gamma \vdash T_x' <: T_x \qquad \Gamma, x : T_x' \vdash T <: T' \qquad m \geq m'}{\Gamma \vdash m \cdot (x : T_x \to T) <: m' \cdot (x : T_x' \to T')}\ (\textsc{Sub-Arrow}) \qquad \frac{\Gamma \vdash R_1 <: R_2 \qquad \Gamma, \nu : R_1 \models \phi_1 \geq \phi_2}{\Gamma \vdash {R_1}^{\phi_1} <: {R_2}^{\phi_2}}\ (\textsc{Sub-Pot})$$

Figure 1.8. Sharing and subtyping rules

The *subtyping* judgment $\Gamma \vdash T_1 <: T_2$ in Figure 1.8 is defined in a standard way, with the extra requirement that the potential in T_1 should be greater than or equal to that in T_2. Subtyping is often used to "forget" some program variables in the type to ensure the result type does not reference any locally introduced variable, e.g., the result type of $\mathsf{let}(e_1, x.e_2)$ cannot have x in it and the result type of $\mathsf{matl}(a_0, e_1, x_h.x_t.e_2)$ cannot reference x_h or x_t.

To reason about logical refinements, we introduce *validity checking*, written $\Gamma \models \psi$, to state that a logical refinement ψ is always true under any instance of the context Γ. The validity checking relation is established upon a denotational semantics for refinements. Validity checking in Re^2 is decidable because it can be reduced to Presburger arithmetic. The full development of validity checking is included in the technical report [83].

We reason about inductive invariants for lists in rule (T-MATL), using interpretation $\mathscr{I}(\cdot)$. In our formalization, lists are refined by their length thus the invariants are: (i) nil has length 0, and (ii) the length of $\mathsf{cons}(\_, a_t)$ is the length of a_t plus one. The type system can be easily enriched with more refinements and data types (e.g., the elements of a list are the union of its head and those of its tail) by updating the interpretation $\mathscr{I}(\cdot)$ as well as the premises of rule (T-MATL).

Finally, notable as well are the two typing rules for applications: (T-APP) and (T-APP-SIMPATOM). In the former case, the function return type T does not mention x, and hence can be directly used as the type of the application (this is the case *e.g.* for all higher-order applications, since our well-formedness rules prevent functions from appearing in refinements). In the latter case, T mentions x, but luckily any argument of a scalar type must be a simple atom a, so we can substitute x with its interpretation $\mathscr{I}(a)$. The ability to derive precise types for dependent applications motivates the use of a-normal-form in Re^2.

Resources

The rule (T-CONSUME-P) states that an expression $\mathsf{tick}(c, e_0)$ is only well-typed in a context that contains a free potential term c. To transform the context into this form, we can

use the rule (S-TRANSFER) to transfer potential within the context between variable types and free potential terms, as long as we can prove that the total amount of potential remains the same. For example, the combination of (S-TRANSFER) and (S-RELAX) allows us to derive both $x : \mathsf{bool}^1 \vdash x :: \mathsf{bool}^1$ and $x : \mathsf{bool}^1 \vdash \mathsf{tick}(1,x) :: \mathsf{bool}$ (but not $x : \mathsf{bool}^1 \vdash \mathsf{tick}(1,x) :: \mathsf{bool}^1$).

The typing rules of Re^2 form an *affine* type system [137]. To use a program variable multiple times, we have to introduce explicit *sharing* in Figure 1.8 to ensure that the program cannot gain potential. The sharing judgment $\Gamma \vdash S \curlyvee S_1 \mid S_2$ means that in the context Γ, the potential indicated by S is apportioned into two parts to be associated with S_1 and S_2. We extend this notion to *context sharing*, written $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, which states that Γ_1, Γ_2 has the same sequence of bindings as Γ, but the potentials of type bindings in Γ are shared point-wise, and the free potentials in the Γ are also split. A special context sharing $\vdash \Gamma \curlyvee \Gamma \mid \Gamma$ is used in the typing rules (T-ABS) and (T-FIX) for functions. The self-sharing indicates that the function can only reference potential-free free variables in the context. This is also used to ensure that the program cannot gain more potential through free variables by applying the same function multiple times.

Restricting functions to be defined under potential-free contexts is undesirable in some situations. For example, a curried function of type $x{:}T_x \to y{:}T_y \to T$ might require nonzero units of potential on its first argument x, which is not allowed by rule (T-ABS) or (T-FIX) on the inner function type $y{:}T_y \to T$. We introduce another rule (T-ABS-LIN) to relax the restriction. The rule associates a multiplicity m with the function type, which denotes the number of times that the function could be applied. Instead of context self-sharing, we require the potential in the context to be enough for m function applications. Note that in RESYN's surface syntax used in the Section 3.2, every curried function type implicitly has multiplicity 1 on the inner function: $x{:}T_x \to 1 \cdot (y{:}T_y \to T)$.

Example

Recall the function `triple` from Figure 1.3, which can be written as follows in Re^2 core syntax:

$$\begin{aligned}\texttt{triple} \;&::\; \ell{:}L(\mathsf{bool}^2) \to \{L(\mathsf{bool}) \mid \nu = 3 \times \ell\} \\ \texttt{triple} \;&=\; \lambda(\ell.\mathsf{let}(\mathsf{app}(\mathsf{app}(\texttt{append},\ell),\ell),\ell'. \\ &\qquad \mathsf{app}(\mathsf{app}(\texttt{append},\ell),\ell'))\end{aligned}$$

Next, we illustrate how Re^2 uses the signature of `append`:

$$\texttt{append} :: \forall\alpha.xs{:}L(\alpha^1) \to 1 \cdot (ys{:}L(\alpha) \to \{L(\alpha) \mid \nu = xs + ys\})$$

to justify the resource bound $2|\ell|$ on `triple`. Suppose Γ is a typing context that contains the signature of `append`. The argument ℓ is used three times, so we need to use sharing relations to apportion the potential of ℓ. We have $\Gamma \vdash L(\mathsf{bool}^2) \curlyvee L(\mathsf{bool}^1) \mid L(\mathsf{bool}^1)$, $\Gamma \vdash L(\mathsf{bool}^1) \curlyvee L(\mathsf{bool}^1) \mid L(\mathsf{bool}^0)$, and we assign $L(\mathsf{bool}^1)$, $L(\mathsf{bool}^0)$, and $L(\mathsf{bool}^1)$ to the three occurrences of ℓ respectively in the order they appear in the program. To reason about $e_1 = \mathsf{app}(\mathsf{app}(\texttt{append},\ell),\ell)$, we instantiate `append` with $\alpha \mapsto \mathsf{bool}^0$, inferring its type as

$$xs{:}L(\mathsf{bool}^1) \to 1 \cdot (ys{:}L(\mathsf{bool}^0) \to \{L(\mathsf{bool}^0) \mid \nu = xs + ys\})$$

and by (T-App-SimpAtom) we derive the following:

$$\Gamma, \ell : L(\mathsf{bool}^1) \vdash e_1 :: \{L(\mathsf{bool}^0) \mid \nu = \ell + \ell\}.$$

We then can typecheck $e_2 = \mathsf{app}(\mathsf{app}(\texttt{append},\ell),\ell')$ with the same instantiation of `append`:

$$\Gamma, \ell : L(\mathsf{bool}^1), \ell' : T_1 \vdash e_2 :: \{L(\mathsf{bool}^0) \mid \nu = xs + (xs + xs)\}.$$

(where T_1 is the type of e_1). Finally, by subtyping and the following valid judgment in the refinement logic

$$\Gamma, \ell : L(\mathsf{bool}^2), \nu : L(\mathsf{bool}^0) \models \nu = \ell + (\ell + \ell) \implies \nu = 3 \times \ell,$$

we conclude $\Gamma \vdash \texttt{triple} :: \ell{:}L(\mathsf{bool}^2) \to \{L(\mathsf{bool}) \mid \nu = 3 \times \ell\}$.

Soundness

The type soundness for Re^2 is based on progress and preservation. The progress theorem states that if we derive a bound q for an expression e with the type system and $p \geq q$ resources are available, then $\langle e, p \rangle$ can make a step if e is not a value. In this way, progress shows that resource bounds are indeed bounds on the high-water mark of the resource usage since states $\langle e, p \rangle$ in the small step semantics can be stuck based on resource usage if, for instance, $p = 0$ and $e = \mathsf{tick}(1, e')$.

Theorem 1 (Progress)**.** If $q \vdash e :: S$ and $p \geq q$, then either $e \in \mathsf{Val}$ or there exist e' and p' such that $\langle e, p \rangle \mapsto \langle e', p' \rangle$.

Proof. By strengthening the assumption to $\Gamma \vdash e :: S$ where Γ is a sequence of type variables and free potentials, and then induction on $\Gamma \vdash e :: S$. □

The preservation theorem accounts for resource consumption by relating the left over resources after a computation to the type judgment of the new term.

Theorem 2 (Preservation)**.** If $q \vdash e :: S$, $p \geq q$ and $\langle e, p \rangle \mapsto \langle e', p' \rangle$, then $p' \vdash e' :: S$.

Proof. By strengthening the assumption to $\Gamma \vdash e :: S$ where Γ is a sequence of free potentials, and then induction on $\Gamma \vdash e :: S$, followed by inversion on the evaluation judgment $\langle e, p \rangle \mapsto \langle e', p' \rangle$. □

The proof of preservation makes use of the following crucial substitution lemma.

Lemma 1 (Substitution)**.** If $\Gamma_1, x : \{B \mid \psi\}^{\phi}, \Gamma' \vdash e :: S, \Gamma_2 \vdash t :: \{B \mid \psi\}^{\phi}, t \in \mathsf{Val}$ and $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, then $\Gamma, [\mathscr{I}(t)/x]\Gamma' \vdash [t/x]e :: [\mathscr{I}(t)/x]S$.

Proof. By induction on $\Gamma_1, x : \{B \mid \psi\}^{\phi}, \Gamma' \vdash e :: S$. □

Since we found the purely syntactic soundness statement about results of computations (they are well-typed values) somewhat unsatisfactory, we also introduced a denotational notation of consistency. For example, a list of values $\ell = [v_1, \cdots, v_n]$ is consistent with $q \vdash \ell :: L(\{\mathsf{bool} \mid \neg \nu\})^{\nu+5}$, if $q \geq n+5$ and each value v_i of the list is false. We then show that well-typed values are *consistent* with their typing judgement.

Lemma 2 (Consistency)**.** If $q \vdash v :: S$, then v satisfies the conditions indicated by S and q is greater than or equal to the potential stored in v with respect to S.

As a result, we derive the following theorem.

Theorem 3 (Soundness)**.** If $q \vdash e :: S$ and $p \geq q$ the either

- $\langle e, p \rangle \mapsto^* \langle v, p' \rangle$ and v is consistent with $p' \vdash v :: S$ or
- for every n there is $\langle e', p' \rangle$ such that $\langle e, p \rangle \mapsto^n \langle e', p' \rangle$.

Complete proofs can be found in the technical report [83].

Inductive Datatypes and Measures

We can generalize our development of list types for inductive types $\mu X.\overrightarrow{C : T \times X^k}$, where C is the constructor name, T is the element type that does not contain X, and X^k is the k-element product type $X \times X \times \cdots \times X$. The introduction rules and elimination rules are almost the same as (T-NIL), (T-CONS) and (T-MATL), respectively, except that we need to capture inductive invariants for each constructor C in the rules correspondingly. In SYNQUID, these invariants are specified by inductive *measures* that map values to refinements. We can introduce new sorting rules for inductive types to embed values as their related measures in the refinement logic.

$$
\begin{array}{l}
D ::= \cdot \mid D; x \leftarrow e \\
\mathring{e} ::= e \mid \circ \mid \mathsf{app}(x, \circ) \mid \mathsf{if}(x, \circ, \circ) \mid \mathsf{matl}(x, \circ, x_h.x_t.\circ) \mid \mathsf{lets}(D.\mathring{e}) \\
T ::= R^{\phi} \mid ?
\end{array}
$$

Figure 1.9. Extended syntax

Constant Resource

Our type system infers *upper bounds* on resource usage. Recently, AARA has been generalized to verify *constant-resource* behavior [97]. A program is said to be constant-resource if its executions on inputs of the same size consume the same amount of resource. We can adapt the technique in [97] to Re^2 by (i) changing the subtyping rules to keep potentials invariant (*i.e.* replacing $\geq$ with $=$ in (SUB-TVAR), (SUB-ARROW), (SUB-POT)), and (ii) changing the rule (SIMP-ATOM-VAR) to require $\phi = 0$. Based on the modified type system, our synthesis algorithm can also synthesize constant-time implementations (see Section 1.5.2 for more details).

1.4 Type-Driven Synthesis with Re^2

In this section, we first show how to turn the type checking rules of Re^2 into *synthesis rules*, and then leverage these rules to develop a *synthesis algorithm*.

1.4.1 Synthesis Rules

Extended Syntax

To express synthesis rules, we extend Re^2 with a new syntactic form $\mathring{e}$ for *expression templates*. As shown in Figure 1.9, templates are expressions that can contain holes $\circ$ in certain positions. The *flat let* form $\mathsf{lets}(D.\mathring{e})$, where D is a sequence of bindings, is a shortcut for a nest of let-expressions $\mathsf{let}(x_1, d_1 \ldots \mathsf{let}(x_n, d_n.\mathring{e}))$; we write $\mathsf{fold}(\mathsf{lets}(D.e))$ to convert a flat let (without holes) back to the original syntax. We also extend the language of types with an *unknown type* ?, which is used to build partially defined goal types, as explained below.

$\boxed{\Gamma \vdash \mathring{e} :: S \rightsquigarrow e}$

$$\frac{\Gamma, \alpha \vdash S \curlyvee S \mid S \qquad \Gamma, \alpha \vdash \circ :: S \rightsquigarrow e}{\Gamma \vdash \circ :: \forall \alpha . S \rightsquigarrow e}\ (\textsc{Syn-Gen})$$

$$\frac{\Gamma, f : (x : T_x \to T), x : T_x \vdash \circ :: T \rightsquigarrow e \qquad \vdash \Gamma \curlyvee \Gamma \mid \Gamma}{\Gamma \vdash \circ :: (x : T_x \to T) \rightsquigarrow \mathsf{fix}(f.x.e)}\ (\textsc{Syn-Fix})$$

$$\frac{\Gamma, x : T_x \vdash \circ :: T \rightsquigarrow e}{\Gamma \vdash \circ :: (1 \cdot (x : T_x \to T)) \rightsquigarrow \lambda(x.e)}\ (\textsc{Syn-Abs})$$

$$\frac{\Gamma \vdash \circ :: \mathsf{bool} \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x) \qquad \Gamma \vdash \mathsf{lets}(D.\mathsf{if}(x, \circ, \circ)) :: T \rightsquigarrow e}{\Gamma \vdash \circ :: T \rightsquigarrow e}\ (\textsc{Syn-Cond})$$

$$\frac{\Gamma \vdash T\ \mathsf{type} \qquad \Gamma \vdash \circ :: L(T) \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x) \qquad \Gamma \vdash \mathsf{lets}(D.\mathsf{matl}(x, \circ, x_h.x_t.\circ)) :: T \rightsquigarrow e}{\Gamma \vdash \circ :: T \rightsquigarrow e}\ (\textsc{Syn-MatL})$$

$$\frac{\Gamma \vdash x : \mathsf{bool} \qquad \Gamma, x \vdash \circ :: T \rightsquigarrow e1 \qquad \Gamma, \neg x \vdash \circ :: T \rightsquigarrow e2}{\Gamma \vdash \mathsf{if}(x, \circ, \circ) :: T \rightsquigarrow \mathsf{if}(x, e1, e2)}\ (\textsc{Fill-Cond})$$

$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash x : L(T) \qquad \Gamma_2, x = 0 \vdash \circ :: T \rightsquigarrow e1 \qquad \Gamma_2, x_h : T, x_t : L(T), x = 1 + x_t \vdash \circ :: T \rightsquigarrow e2}{\Gamma \vdash \mathsf{matl}(x, \circ, x_h.x_t.\circ) :: T \rightsquigarrow \mathsf{matl}(x, e1, x_h.x_t.e_2)}\ (\textsc{Fill-MatL})$$

$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash e_1 :: T_1 \qquad \Gamma_2, x : T_1 \vdash \mathsf{lets}(D.\mathring{e}_2) :: T \rightsquigarrow e_2}{\Gamma \vdash \mathsf{lets}(x \leftarrow e_1; D.\mathring{e}_2) :: T \rightsquigarrow \mathsf{let}(e_1, x.e_2)}\ (\textsc{Fill-Let})$$

$$\frac{\Gamma \models \bot}{\Gamma \vdash \circ :: T \rightsquigarrow \mathsf{impossible}}\ (\textsc{Syn-Imp})$$

$$\frac{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}{\Gamma \vdash \circ :: T \rightsquigarrow \mathsf{fold}(\mathsf{lets}(D.a))}\ (\textsc{Syn-Atom})$$

$\boxed{\Gamma \vdash \mathring{e} :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}$

$$\frac{\Gamma \vdash x :: T}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(\cdot.x)}\ (\textsc{ASyn-Var})$$

$$\frac{\Gamma \vdash \mathsf{true} :: T}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(\cdot.\mathsf{true})}\ (\textsc{ASyn-True})$$

$$\frac{\Gamma \vdash \mathsf{false} :: T}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(\cdot.\mathsf{false})}\ (\textsc{ASyn-False})$$

$$\frac{\Gamma \vdash \mathsf{nil} :: T}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(\cdot.\mathsf{nil})}\ (\textsc{ASyn-Nil})$$

$$\frac{\Gamma \vdash \circ :: 1 \cdot (\_ : ? \to T) \overset{a}{\rightsquigarrow} \mathsf{lets}(D_1.x) \qquad \Gamma \vdash \mathsf{lets}(D_1.\mathsf{app}(x, \circ)) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x')}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x')}\ (\textsc{ASyn-App})$$

$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash e_1 :: T_1 \qquad \Gamma_2, x : T_1 \vdash \mathsf{lets}(D.\mathring{e}_2) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D_2.a)}{\Gamma \vdash \mathsf{lets}(x \leftarrow e_1; D.\mathring{e}_2) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(x \leftarrow e_1; D_2.a)}\ (\textsc{AFill-Let})$$

$$\frac{\Gamma \vdash x :: 1 \cdot (\_ : T_1 \to T) \qquad T_1\ \mathsf{non\text{-}scalar} \qquad \Gamma \vdash \circ :: T_1 \rightsquigarrow \hat{a}}{\Gamma, 1 \vdash \mathsf{app}(x, \circ) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(x' \leftarrow \mathsf{tick}(1, \mathsf{app}(x, \hat{a})).x')}\ (\textsc{AFill-App})$$

$$\frac{\Gamma \vdash x :: 1 \cdot (y : T_1 \to T') \qquad T_1\ \mathsf{scalar} \qquad \Gamma \vdash \circ :: T_1 \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a) \qquad \Gamma \vdash \mathsf{fold}(\mathsf{lets}(D.\mathsf{app}(x, a))) :: T}{\Gamma, 1 \vdash \mathsf{app}(x, \circ) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D; x' \leftarrow \mathsf{tick}(1, \mathsf{app}(x, a)).x')}\ (\textsc{AFill-App-SimpAtom})$$

Figure 1.10. Selected synthesis rules

Synthesis for A-Normal-Form

Our synthesis relation consists of two mutually recursive judgments: the *synthesis* judgment $\Gamma \vdash \mathring{e} :: S \rightsquigarrow e$ intuitively means that the template $\mathring{e}$ can be completed into an expression e such that $\Gamma \vdash e :: S$; the purpose of the auxiliary *atomic synthesis* judgment is explained below. Selected rules for both judgments are given in Figure 1.10; the full technical development can be found in the technical report [83].

The synthesis rule (SYN-GEN) handles polymorphic goal types. The rules (SYN-FIX) and (SYN-ABS) handle arrow types and derive either a fixpoint term or an abstraction. The rule (SYN-IMP) derives $\mathsf{impossible}$ in an inconsistent context (which may arise *e.g.* in a dead branch of a pattern match). The rest of the rules handle the common case when the goal type T is scalar and the context is consistent; in this case the target expression can be either a conditional, a match, or an *E-term* [107], *i.e.* a term made of variables, applications, and constructors. Special care must be taken to ensure that these expressions are in a-normal-form: generally, a-normalizing an expression requires introducing fresh variables and let-bindings for them. To retain completeness, our synthesis rules need to do the same: intuitively, in addition to an expression e, a rule might also need to produce a sequence of let-bindings D that define fresh variables in e. To this end, we introduce the *atomic synthesis* judgment $\Gamma \vdash \mathring{e} :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)$, which synthesizes *normalized E-terms*, where a is an atom and each definition in D is an application or a constructor in a-normal-form.

As an example, consider the rule (SYN-COND) for synthesizing conditionals: ideally, we would like to synthesize a guard e of type bool, and then synthesize the two branches under the assumptions that e evaluates to true and false, respectively. Recall, however, that the guard must be atomic; hence, to synthesize a well-formed conditional, we use atomic synthesis to produce a guard $\mathsf{lets}(D.x)$. Now to get a well-scoped program we must place the whole conditional *inside* the bindings D; to that end, the second premise of (SYN-COND) uses a nontrivial template $\mathsf{lets}(D.\mathsf{if}(x, \circ, \circ))$. The rules (FILL-LET) and (FILL-COND) handle this template by integrating it into the typing context and exposing the hole; along the way (FILL-LET) takes care of context

sharing, which accounts for the potential consumed by the definitions in D. Synthesis of matches works similarly using (Syn-MatL) and (Fill-MatL).

Atomic Synthesis

The first four rules of atomic synthesis generate a simple atom if its type matches the goal; the rest of the rules deal with the hardest part: normalized applications. Consider the rule (ASYN-APP): given a goal type T for the application $\mathsf{app}(e_1, e_2)$, we need to construct goal types for e_1 and e_2, to avoid enumerating them blindly. Following SYNQUID's *round-trip type checking* idea, we use the type $\_ : ? \to T$ as the goal for e_1 (*i.e.* a function from unknown type to T). The subtyping rules for ? are such that $\Gamma \vdash (y : T_1 \to T_2) <: (\_ : ? \to T)$ holds if T_2 and T agree in shape and those refinements that do not mention y; hence this goal type filters out those functions e_1 that cannot fulfill the desired goal type T, independently of the choice of e_2. One difference with SYNQUID is that the goal type for e_1 is *linear*, reflecting that we intend to use e_1 only once and allowing it to capture positive potential.

Similarly to the conditional case explained above, the synthesized left-hand side of the application, e_1, has the form $\mathsf{lets}(D_1.x)$, and the argument e_2 must be synthesized inside the bindings D_1. These bindings are processed by (AFILL-LET), and the actual argument synthesis happens in either (AFILL-APP) or (AFILL-SIMPATOM), depending on whether the argument type is a scalar. The former corresponds to a higher-order application: here T_1 is an arrow type, and hence the argument cannot occur in the function's return type; in this case, synthesizing an expression of type T_1 must yield an abstraction or fixpoint (since T_1 is an arrow), both of which are atoms. The latter corresponds to a first-order application: here the return type T' can mention y, so after synthesizing an argument of type T_y, we still need to check whether the resulting application $\mathsf{lets}(D.\mathsf{app}(x, a))$ has the right type T. Note how both (AFILL-APP) or (AFILL-SIMPATOM) return normalized E-terms by generating a fresh variable and binding it to an application.

Cost Metrics

In the context of synthesis we cannot rely on programmer-written tick terms to model cost. Instead in our formalization we use a simple cost metric where each function application consumes one unit of resource; hence every application generated by (AFILL-APP) or (AFILL-SIMPATOM) is wrapped in $\mathsf{tick}(1,\cdot)$. Our implementation provides more flexibility and allows the programmer to annotate any arrow type with a non-negative cost c to denote that applying a function of this type should incur cost c.

Soundness

The synthesis rules always produce a well-typed expression (proof can be found in the technical report [83]).

Theorem 4 (Soundness of Synthesis)**.** If $\Gamma \vdash \circ :: S \rightsquigarrow e$ then $\Gamma \vdash e :: S$.

1.4.2 Synthesis Algorithm

In this section we discuss how to turn the declarative synthesis rules of Section 1.4.1 into a *synthesis algorithm*, which takes as input a *goal type* S, a context Γ, and a bound k on the program depth, and either returns a program e of depth at most k such that $\Gamma \vdash e :: S$, or determines that no such program exists. The core algorithm follows the recipe from prior work on type-driven synthesis [107, 101] and performs a fairly standard goal-directed backtracking proof search with $\Gamma \vdash \circ :: e \rightsquigarrow S$ as the top-level goal. In the rest of this section, we explain how to make such proof search feasible by reducing the core sources of non-determinism to constraint solving.

Typing constraints

The main sources of non-determinism in a synthesis derivation stem from the following premises of synthesis and typing rules: (1) whenever a given context Γ is shared as $\Gamma \vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, we need to guess how to apportion potential annotations in Γ; (2) whenever potential in a given context Γ is transfered, we need to guess potential annotations in Γ' such that $\Phi(\Gamma) = \Phi(\Gamma')$; and

Subtyping constraints

$$\mathbb{C}(\Gamma \vdash m_1 \cdot \alpha <: m_2 \cdot \alpha) = \{\Gamma \models m_1 - m_2 \geq 0\}$$
$$\mathbb{C}(\Gamma \vdash \{B_1 \mid \psi_1\} <: \{B_2 \mid \psi_2\}) = \{\Gamma, \nu : B_1 \models \psi_1 \implies \psi_2\} \cup \mathbb{C}(\Gamma \vdash B_1 <: B_2)$$
$$\mathbb{C}(\Gamma \vdash {R_1}^{\phi_1} <: {R_2}^{\phi_2}) = \{\Gamma, \nu : R_1 \models \phi_1 - \phi_2 \geq 0\} \cup \mathbb{C}(\Gamma \vdash R_1 <: R_2)$$

Sharing constraints

$$\mathbb{C}(\Gamma \vdash m \cdot \alpha \curlyvee m_1 \cdot \alpha \mid m_2 \cdot \alpha) = \{\Gamma \models m - (m_1 + m_2) \geq 0,$$
$$\Gamma \models m_1 + m_2 - m \geq 0\}$$
$$\mathbb{C}(\Gamma \vdash R^{\phi} \curlyvee {R_1}^{\phi_1} \mid {R_2}^{\phi_2}) = \{\Gamma \models \phi - (\phi_1 + \phi_2) \geq 0, \Gamma \models \phi_1 + \phi_2 - \phi \geq 0\}$$
$$\cup \mathbb{C}(\Gamma \vdash R \curlyvee R_1 \mid R_2)$$

Transfer constraints

$$\mathbb{C}(\Phi(\Gamma) = \Phi(\Gamma')) = \{\Gamma \models \Phi(\Gamma) - \Phi(\Gamma') \geq 0, \Phi(\Gamma') - \Phi(\Gamma) \geq 0\}$$

Figure 1.11. Selected cases for translating typing constraints to validity constraints.

finally (3) whenever $\{B \mid \psi\}^{\phi}$ is used to instantiate a type variable, we need to guess both ϕ and ψ. All three amount to inference of unknown refinement terms of either Boolean or numeric sort. To infer these terms efficiently, we use the following constraint-based approach. First, we build a symbolic synthesis derivation, which may contain *unknown refinement terms* U^{Δ}_{Γ}, and collect all subtyping, sharing, and transfer premises from the derivation into a system of *typing constraints*. Here Δ records the desired sort of the unknown refinement term, and Γ records the context in which it must be well-formed. A *solution* to a system of typing constraints, is a map $\mathscr{L} : U \to \psi$ such that for every unknown U^{Δ}_{Γ}, $\Gamma \vdash \mathscr{L}(U) \in \Delta$ and substituting $\mathscr{L}(U)$ for U within the typing constraints yields valid subtyping, sharing, and transfer judgments.

Constraint Solving

To solve typing constraints, the algorithm first transforms them into validity constraints of one of two forms: $\Gamma \models \psi \implies \psi'$ or $\Gamma \models \phi \geq 0$; the interesting cases of this translation are shown in Figure 1.11. Then, using the definition of validity (in the technical report [83]), we further reduce these into a system of:

1. *Horn constraints* of the form $\psi_1 \wedge \ldots \wedge \psi_n \implies \psi_0$, and

2. *resource constraints* of the form $\psi_1 \wedge \ldots \wedge \psi_n \implies \phi \geq 0$.

Here any ψ_i can be either a Boolean unknown $U_\Gamma^{\mathbb{B}}$ or a known refinement term, and ϕ is a sum of zero or more numeric unknowns $U_\Gamma^{\mathbb{N}}$ and a known (linear) refinement term. While prior work has shown how to efficiently solve Horn constraints using predicate abstraction [115, 107], resource constraints present a new challenge, since they contain unknown terms of both Boolean and numeric sorts. In the interest of efficiency, our synthesis algorithm does not attempt to solve for both Boolean and numeric terms at the same time. Instead, it uses existing techniques to find a solution for the Horn constraints, and then plugs this solution into the resource constraints. Note that this approach does not sacrifice completeness, as long as the Horn solver returns the least-fixpoint (*i.e.* strongest) solution for each $U_\Gamma^{\mathbb{B}}$, since Boolean unknowns only appear negatively in resource constraints[4].

Resource Constraints

The main new challenge then is to solve a system of resource constraints of the form $\psi \implies \phi \geq 0$, where ψ is now a known formula of the refinement logic. Since potential annotations in Re^2 are restricted to linear terms over program variables, we can replace each unknown term $U_\Gamma^{\mathbb{N}}$ in ϕ with a linear template $\sum_{x \in X} C_i \cdot x$, where each C_i is an unknown integer coefficient and X is the set of all variables in Γ such that $\Gamma \vdash x \in \mathbb{N}$. After normalization, the system of resource constraints is reduced to the following doubly-quantified system of linear inequalities:

$$\exists \overrightarrow{C_i}. \forall \overrightarrow{x}. \bigwedge_{r \in R} r(\overrightarrow{C_i}, \overrightarrow{x})$$

where each clause r is of the form $\psi(\overrightarrow{x}) \implies \sum f(\overrightarrow{C_i}) \cdot x \geq 0$, ψ is a known formula over the program variables $\overrightarrow{x}$, and each f is a linear function over unknown integer coefficients $\overrightarrow{C_i}$.

Note a crucial difference between these constraints and those generated by RaML: since RaML's potential annotations are not dependent—*i.e.* r cannot mention program variables $\overrightarrow{x}$—

[4]Our implementation uses SYNQUID's default greatest-fixpoint Horn solver, which technically renders this technique incomplete, however we observed that it works well in practice.

Algorithm 1. Incremental solver for resource constraints

Input: Constraints R, current solution $\mathscr{C}$, examples $\mathscr{E}$
Output: New solution and examples $(\mathscr{C}, \mathscr{E})$ or $\bot$ if no solution

```
procedure SOLVE(R, 𝒞, ℰ)
    e ← SMT(∃x⃗.¬R(𝒞, x⃗))
    if e = ⊥ then                                   ▷ No counter-example
        return (𝒞, ℰ)
    else
        ℰ' ← ℰ ∪ e
        R' ← {r ∈ R | ¬r(𝒞, e)}
        𝒞' ← SMT(∃C⃗_i. ⋀_{e∈ℰ'} R'(C⃗_i, e))
        if 𝒞' = ⊥ then return ⊥                     ▷ No solution
        else SOLVE(R, 𝒞 ∪ 𝒞', ℰ')
```

its resource constraints reduce to plain linear inequalities: $\exists\overrightarrow{C_i}.\bigwedge\sum C_i \geq c$ (where c is a known constant), which can be handled by an LP solver. In our case, the challenge stems both from the double quantification and the fact that individual clauses r are *bounded* by formulas ψ, which are often nontrivial. For example, synthesizing the function `range` from Section 3.2 gives rise to the following (simplified) resource constraints:

$$\exists C_0 \ldots C_3.\forall a, b, \nu.$$
$$(\neg(a \geq b) \wedge \nu = b) \implies (C_0 + 1)\cdot a + C_1 \cdot b + (C_2 - 1)\cdot \nu + C_3 \geq 0$$
$$(\neg(a \geq b) \wedge \nu = b) \implies C_0 \cdot a + C_1 \cdot b + C_2 \cdot \nu + C_3 \geq 0$$

where a solution only exists if the bounds are taken into account. One solution is $[C_0 \mapsto -1, C_1 \mapsto 0, C_2 \mapsto 1, C_3 \mapsto 0]$, which stands for the potential term $\nu - a$.

Incremental Solving

Constraints of this form can be solved using *counter-example guided inductive synthesis* (CEGIS) [127], which is, however, relatively expensive. We observe that in the context of synthesis we have to repeatedly solve similar systems of resource constraints because a program candidate is type-checked *incrementally* as it is being constructed, which corresponds to an incrementally growing set of clauses R. Moreover, we observe that as new clauses are added,

only a few existing coefficients C_i are typically invalidated, so we can avoid solving for all the coefficients from scratch. To this end, we develop an incremental version of the CEGIS algorithm, shown in Algorithm 1.

The goal of the algorithm is to find a solution $\mathscr{C} : C_i \to \mathbb{Z}$ that maps unknown coefficients to integers such that $\overrightarrow{\forall x}.R(\mathscr{C}, \overrightarrow{x})$ holds (we write $R(\mathscr{C}, \overrightarrow{x})$ as a shorthand for $\bigwedge_{r \in R} r(\mathscr{C}, \overrightarrow{x})$). The algorithm takes as input a set of clauses R (which includes both old and new clauses), the current solution $\mathscr{C}$ (new coefficients C_i are mapped to 0) and the current set of *examples* $\mathscr{E}$, where an example $e \in \mathscr{E}$ is a partial assignment to universally-quantified variables $e : X \to \mathbb{N}$.

The algorithm first queries the SMT solver for a counter-example e to the current solution. If no such counter-example exists, the solution is still valid (this happens surprisingly often, since many resource constraints are trivial). Otherwise, the current solution needs to be updated. To this end, a traditional CEGIS algorithm would query the SMT solver with the following *synthesis constraint*: $\exists \overrightarrow{C_i}. \bigwedge_{e \in \mathscr{E}'} R(\overrightarrow{C_i}, e)$, which enforces that all clauses are satisfied on the extended set of examples. Instead, our incremental algorithm picks out only those clauses R' that are actually violated by the new counter-example; since in our setting R' is typically small, this optimization significantly reduces the size of the synthesis constraint and synthesis times for programs with dependent annotations (as we demonstrate in Section 2.4).

1.4.3 Implementation

We implemented the resource-guided synthesis algorithm in ReSyn, which extends Synquid with support for resource-annotated types and a resource constraint solver. Note that while our formalization is restricted to Booleans and length-indexed lists, our implementation supports the full expressiveness of Synquid's types: types include integers and user-defined algebraic datatypes, and refinement formulas support sets and can mention arbitrary user-defined measures. More importantly, resource terms in ReSyn can mention integer variables and use subtraction, multiplication, conditional expressions, and numeric measures; finally, multiplicities on type variables can be dependent (mention variables). These changes have the following

implications: (1) resource terms are not syntactically guaranteed to be non-negative, so we emit additional well-formedness constraints to enforce this; (2) resource terms are not syntactically restricted to be linear; our implementation is incomplete, and simply rejects the program if a nonlinear term arises; (3) subtyping and sharing constraints with conditional resource terms are decomposed into unconditional ones by moving the guard to the context, so the search space for all numeric unknowns remains unconditional; (4) to handle measure applications in resource constraints, we replace them with fresh integer variables, and avoid spurious counter-examples by explicitly instantiating the congruence axiom with all applications in the constraint.

1.5 Evaluation

We evaluated RESYN using the following criteria:

Relative performance: How do RESYN's synthesis times compare to SYNQUID's? How much does the additional burden of solving resource constraints affect its performance?

Efficacy of resource analysis: Can RESYN discover more efficient programs than SYNQUID?

Value of round-trip type checking: Does round-trip type checking afforded by the tight integration of resource analysis into SYNQUID effective at pruning the search space? How does it compare to the naive combination of synthesis and resource analysis?

Value of incremental solving: To what extent does incremental solving of resource constraints improve RESYN's performance?

1.5.1 Relative Performance

To evaluate RESYN's performance relative to SYNQUID, we selected 43 problems from SYNQUID's original suite, annotated them with resource bounds, and re-synthesized them with RESYN. The rest of the original 64 benchmarks require non-linear bounds, and thus are out of

scope of Re^2. The details of this experiment are shown in Table 1.1, which compares ReSyn's synthesis times against Synquid's on these linear-bounded benchmarks.

Unsurprisingly, due to the additional constraint-solving, ReSyn generally performs worse than Synquid: the median synthesis time is about 2.5× higher. Note, however, that in return it provides provable guarantees about the performance of generated code. ReSyn was able to discover a more efficient implementation for only *one* of the original Synquid benchmarks (`compress`, discussed below). In general, these benchmarks contain only the minimal set of components required to produce a valid implementation, which makes it hard for Synquid to find a non-optimal version. *Four* of the benchmarks in Table 1.1 use advanced features of Re^2: for example, any function using natural numbers to index or construct a data structure requires dependent potential annotations.

1.5.2 Case Studies

The value of resource-guided synthesis becomes clear when the library of components grows. To confirm this intuition, we assembled a suite of 16 case studies shown in Table 1.2, each exemplifying some feature of ReSyn.

Optimization

The first six benchmarks showcase ReSyn's ability to generate faster code than Synquid (the cost metric in each case is the number of recursive calls). Benchmark 1 is `triple` from Section 1.2.3, where both Synquid and ReSyn generate the same efficient solution; benchmark 2 is slight modification of this example: it uses a component `append'`, which traverses its second argument (unlike `append`, which traverses its first). In this case, ReSyn generates the efficient solution, associating the two calls to `append'` *to the left*, while Synquid still generates the same—now inefficient—solution, associating these calls *to the right*. In benchmark 3 ReSyn makes the optimal choice of accumulator to avoid a quadratic-time implementation. Benchmark 4 is `compress` from Table 1.1: the task is to remove adjacent duplicated from a

list. Here SYNQUID makes an unnecessary recursive call, resulting in a solution that is slightly shorter but runs in exponential time!

In other cases, RESYN drastically changes the structure of the program to find an optimal implementation. Benchmark 5 is `common` from Section 3.2.2, where RESYN must find an implementation that does not call `member`. Benchmark 6 works similarly, but computes the difference between two lists instead of their intersection. On these benchmarks, the performance disparity between RESYN and SYNQUID is much worse, as RESYN must reject many more programs before it finds an appropriate implementation. On the other hand, these benchmarks also showcase the value of *round-trip type checking*: the column *T-EAC* reports synthesis times for a naive combination of synthesis and resource analysis, where we simply ask SYNQUID to enumerate functionally correct programs until one type-checks under Re^2. As you can see, for benchmarks 5 and 6 this naive version times out after ten minutes.

Dependent Potentials

Benchmarks 7–13 showcase fine-grained bounds that leverage dependent potential annotations. The first three of those synthesize a function `insert` that inserts an element into a sorted list. In benchmark 7 we use a simple linear bound (the length of the list), while benchmarks 8 and 9 specify a tighter bound: `insert x xs` can only make one recursive call per element of `xs` larger than `x`. These two examples showcase two different styles of specifying precise bounds: in 8 we define a custom measure `numgt` that counts list elements greater than a certain value; in 9, we instead annotate each list element with a conditional term indicating that it carries potential only if its value is larger than `x`. As discussed in Section 3.2, benchmark 13 (`range`) cannot be synthesized by SYNQUID at all, because of restrictions on its termination checking mechanism, while RESYN handles this benchmark out of the box.

For benchmarks 8–13, which make use of dependent potential annotations, we also report the synthesis times without incremental solving of resource constraints (*T-NInc*), which are up to $2\times$ higher.

Constant Resource

As discussed in Section 1.3, a simple extension to Re^2 enables it to verify constant-resource implementations. We showcase this feature in benchmarks 14–16. Benchmark 15 is an example from [97], which compares a public list *ys* with a secret list *zs*. By allotting potential only to *ys*, we guarantee that the resource consumption of the generated program is independent of the length of *zs*. If this requirement is relaxed (as in benchmark 16), the generated program indeed terminates early, potentially revealing the length of *zs* to an adversary (in case *zs* is the shorter of the two lists). Benchmark 14 is a constant-time version of benchmark 7 (`insert`), which is forced to make extra recursive calls so as not to reveal the length of the list.

1.6 Related Work

Resource Analysis

Automatic static resource analysis has been extensively studied and is an active area of research. Many advanced techniques for imperative integer programs apply abstract interpretation to generate numerical invariants. The obtained *size-change information* forms the basis for the computation of actual bounds on loop iterations and recursion depths; using counter instrumentation [53], ranking functions [8, 6, 22, 124], recurrence relations [7, 5], and abstract interpretation itself [146, 26]. Automatic resource analysis techniques for functional programs are based on sized types [133], recurrence relations [33], term-rewriting [15], and amortized resource analysis [71, 77, 64, 123]. There exist several tools that can automatically derive loop and recursion bounds for imperative programs including SPEED [53, 54], KoAT [22], PUBS [5], Rank [8], ABC [18] and LOOPUS [146, 124]. These techniques are passive in the sense that they provide feedback about a program without actively synthesizing or repairing programs.

Domain-Specific Program Synthesis

Most program synthesis techniques [101, 42, 125, 39, 40, 38, 140, 138, 128, 82, 107, 74, 109] do not explicitly take resource usage into account during synthesis. Many of them,

however, leverage *domain knowledge* to restrict the search space to only include efficient programs [52, 27] or to encode domain-specific performance considerations as part of the functional specification [75, 92, 91].

Synthesis with Quantitative Objectives

Two lines of prior work on synthesis are explicitly concerned with optimizing resource usage. One is quantitative *automata-theoretic synthesis*, which has been used to synthesize optimal Mealy machines [19] and place synchronization in concurrent programs [24, 55, 25]. In contrast, we focus on synthesis of high-level programs that can manipulate custom data structures, which are out of reach for automata-theoretic synthesis.

The second relevant line of work is *synthesis-aided compilation* [117, 104, 119, 105]. This work is limited to generating low-level straight-line code, which is an easy target for correctness validation and cost estimation. Perhaps the closest work to ours is the Synapse tool [20], which supports a richer space of programs, but requires extensive guidance from the user (in the form of meta-sketches), and relies on bounded reasoning, which can only provide correctness and optimality guarantees for a finite set of inputs. In contrast, we use type-based verification and resource analysis techniques, which enable RESYN to handle high-level recursive programs and provide guarantees for an unbounded set of inputs.

Table 1.1. Comparison of RESYN and SYNQUID. For each benchmark, we report the set of provided *Components*; cumulative size of synthesized *Code* (in AST nodes) for all goals; as well as running times (in seconds) for RESYN (*Time*) and SYNQUID (*TimeNR*).

Group	*Description*	*Components*	*Code*	*Time*	*TimeNR*
List	is empty	true, false	16	0.2	0.2
	member	true, false, $=$, $\neq$	41	0.2	0.2
	duplicate each element		39	0.5	0.3
	replicate	0, inc, dec, $\leq$, $\neq$	31	2.9	0.2
	append two lists		38	1.5	0.5
	take first *n* elements	0, inc, dec, $\leq$, $\neq$	34	2.4	0.2
	drop first *n* elements	0, inc, dec, $\leq$, $\neq$	30	20.4	0.3
	concat list of lists	append	49	3.3	0.8
	delete value	$=$, $\neq$	49	0.8	0.3
	zip		32	0.4	0.2
	zip with		35	0.5	0.2
	i-th element	0, inc, dec, $\leq$, $\neq$	30	0.3	0.2
	index of element	0, inc, dec, $=$, $\neq$	43	0.5	0.3
	insert at end		42	0.4	0.3
	balanced split	fst, snd, abs	64	9.6	1.7
	reverse	insert at end	35	0.4	0.3
	insert (sorted)	$\leq$, $\neq$	57	2.0	0.7
	extract minimum	$\leq$, $\neq$	71	18.1	8.3
	foldr		43	1.8	0.6
	length using fold	0, inc, dec	39	0.3	0.2
	append using fold		42	0.3	0.3
	map		27	0.3	0.2
Unique list	insert	$=$, $\neq$	49	0.8	0.4
	delete	$=$, $\neq$	45	0.5	0.3
	compress	$=$, $\neq$	64	5.0	1.9
	integer range	0, inc, dec, $\leq$, $\neq$	46	88.4	5.1
	partition	$\leq$	71	13.0	5.5
Sorted list	insert	$<$	64	1.6	0.6
	delete	$<$	52	0.5	0.3
	intersect	$<$	71	17.0	0.8
Tree	node count	0, 1, +	34	3.8	0.5
	preorder	append	45	3.0	0.6
	to list	append	45	3.0	0.5
	member	false, not, or, $=$	63	2.2	0.6
BST	member	true, false, $\leq$, $\neq$	72	0.5	0.3
	insert	$\leq$, $\neq$	90	4.5	1.6
	delete	$\leq$, $\neq$	103	26.8	9.3
	BST sort	$\leq$, $\neq$	191	9.0	4.3
Binary Heap	insert	$\leq$, $\neq$	90	3.2	1.0
	member	false, not, or, $\leq$, $\neq$	78	2.3	0.8
	1-element constructor	$\leq$, $\neq$	44	0.2	0.2
	2-element constructor	$\leq$, $\neq$	91	0.7	0.3
	3-element constructor	$\leq$, $\neq$	274	21.4	4.0

Table 1.2. Case Studies. For each synthesis problem, we report: the run time of RESYN (*T*), SYNQUID (*T-NR*), naive combination of SYNQUID and resource analysis (*T-EAC*), RESYN without incremental solving (*T-NInc*); as well as the tightest resource bound for the code generated by RESYN (*B*) and by SYNQUID (*B-NR*). Here, *SL* is the type of sorted lists, and *CL* refers to the type of lists without adjacent duplicates. TO is 10 min; all benchmarks count recursive calls.

	Description	*Type Signature*	*Components*	*T*	*T-NR*	*T-EAC*	*T-NInc*	*B*	*B-NR*
1	triple	$\forall\alpha.xs{:}L(\alpha^2) \to \{L(\alpha) \mid \texttt{len}\ \nu = \texttt{len}\ xs + \texttt{len}\ xs + \texttt{len}\ xs\}$	append	0.9	0.4	0.4	-	$\lvert xs \rvert$	$\lvert xs \rvert$
2	triple'	$\forall\alpha.xs{:}L(\alpha^2) \to \{L(\alpha) \mid \texttt{len}\ \nu = \texttt{len}\ xs + \texttt{len}\ xs + \texttt{len}\ xs\}$	append'	2.8	0.4	1.2	-	$\lvert xs \rvert$	$\lvert xs \rvert^2$
3	concat list of lists	$\forall\alpha.xss{:}L(L(\alpha^1)) \to acc{:}L(\alpha) \to \{L(\alpha) \mid \texttt{sumLen}\ xss = \texttt{len}\nu\}$	append	3.2	0.9	1.1	-	$\lvert xss \rvert$	$\lvert xss \rvert^2$
4	compress	$\forall\alpha.xs{:}L(\alpha^1) \to \{CL(\alpha) \mid \texttt{elems}\ xs = \texttt{elems}\ \nu\}$	$=,\neq$	3.8	1.1	4.1	-	$\lvert xs \rvert$	$2^{\lvert xs \rvert}$
5	common	$\forall\alpha.ys{:}SL(\alpha^1) \to zs{:}SL(\alpha^1) \to \{L(\alpha) \mid \texttt{elems}\ \nu = \texttt{elems}\ ys \cap \texttt{elems}\ zs\}$	$<$, member	30.8	1.1	TO	-	$\lvert ys \rvert + \lvert zs \rvert$	$\lvert ys \rvert \lvert zs \rvert$
6	list difference	$\forall\alpha.ys{:}SL(\alpha^1) \to zs{:}SL(\alpha^1) \to \{L(\alpha) \mid \texttt{elems}\ \nu = \texttt{elems}\ ys - \texttt{elems}\ zs\}$	$<$, member	173.5	1.3	TO	-	$\lvert ys \rvert + \lvert zs \rvert$	$\lvert ys \rvert \lvert zs \rvert$
7	insert	$\forall\alpha.x{:}\alpha \to xs{:}SL(\alpha^1) \to \{SL(\alpha) \mid \texttt{elems}\ \nu = [x] \cup \texttt{elems}\ xs\}$	$<$	1.3	0.4	-	-	$\lvert xs \rvert$	$\lvert xs \rvert$
8	insert'	$\forall\alpha.x{:}\alpha \to xs{:}SL(\alpha)^{\texttt{numgt}(x,\nu)} \to \{SL(\alpha) \mid \texttt{elems}\ \nu = [x] \cup \texttt{elems}\ xs\}$	$<$	49.6	0.7	-	102.2	$\texttt{numgt}(x,xs)$	$\lvert xs \rvert$
9	insert''	$\forall\alpha.x{:}\alpha \to xs{:}SL(\alpha^{\texttt{ite}(x>\nu,1,0)}) \to \{SL(\alpha) \mid \texttt{elems}\ \nu = [x] \cup \texttt{elems}\ xs\}$	$<$	7.7	0.4	-	13.7	$\texttt{numgt}(x,xs)$	$\lvert xs \rvert$
10	replicate	$\forall\alpha.n{:}\texttt{Nat} \to x{:}n \times \alpha^n \to \{L(\alpha) \mid \texttt{len}\ \nu = n\}$	zero, inc, dec	1.4	0.2	-	2.7	n	n
11	take	$\forall\alpha.n{:}\texttt{Nat} \to xs{:}\{L(\alpha) \mid \texttt{len}\nu \geq n\}^n \to \{L(\alpha) \mid \texttt{len}\nu = n\}$	zero, inc, dec	1.2	0.1	-	2.4	n	n
12	drop	$\forall\alpha.n{:}\texttt{Nat} \to xs{:}\{L(\alpha) \mid \texttt{len}\nu \geq n\}^n \to \{L(\alpha) \mid \texttt{len}\nu = \texttt{len}xs - n\}$	zero, inc, dec	12.9	0.2	-	17.1	n	n
13	range	$lo{:}\texttt{Int} \to hi{:}\{\texttt{Int}^{\nu-lo} \mid \nu \geq lo\} \to \{SL(\{\texttt{Int} \mid lo \leq \nu \leq hi\}) \mid \texttt{len}\nu = hi - lo\}$	inc,dec,$\geq$	11.8	0.2	-	-	$hi-lo$	-
14	CT insert	$\forall\alpha.x{:}\alpha \to xs{:}SL(\alpha^1) \to \{SL(\alpha) \mid \texttt{elems}\ \nu = [x] \cup \texttt{elems}\ xs\}$	$<$	2.2	0.6	0.8	-	$\lvert xs \rvert$	$\lvert xs \rvert$
15	CT compare	$\forall\alpha.ys{:}L(\alpha^1) \to zs{:}L(\alpha) \to \{\texttt{bool} \mid \nu = (\texttt{len}\ ys = \texttt{len}\ zs)\}$	true, false, and	14.3	0.5	9.1	-	$\lvert ys \rvert$	$\lvert ys \rvert$
16	compare	$\forall\alpha.ys{:}L(\alpha^1) \to zs{:}L(\alpha) \to \{\texttt{bool} \mid \nu = (\texttt{len}\ ys = \texttt{len}\ zs)\}$	true, false, and	1.0	0.3	-	-	$\lvert ys \rvert$	$\lvert ys \rvert$

Chapter 2

Liquid Resource Types

2.1 Introduction

Open any algorithms textbook and one will read about a number of sorting algorithms, all functionally equivalent. Why then, are there so many algorithms that do the same thing? The answer is that there are subtle differences in their performance characteristics. Consider, for example, the choice between quicksort and insertion sort. In the worst case, both algorithms run in quadratic time. Insertion sort, however, only needs to move the values that are out of place, so it can perform much better on mostly-sorted data.

Resource analysis

Choosing between implementations of seemingly simple functions like these requires precise resource analysis. Thus, there has been a lot of existing work in both inferring and verifying bounds on a program's resource consumption. In general existing approaches must trade automation for flexibility and precision.

On one end of the spectrum, Resource-Aware ML (RAML) [68] automatically infers polynomial bounds on recursive programs by allocating *potential* amongst data structures. RAML reduces least upper bound inference to finding a minimal solution to a system of linear constraints corresponding to the program's resource demands. On the other hand, RELCOST [110] offers greater flexibility at the expense of automation. RELCOST allows users to prove precise resource bounds that depend on program values, but requires hand-written proofs.

```
insert = λx. λxs.
  match xs with
    Nil →Cons x xs
    Cons hd tl →if hd <x
      then Cons hd (tick 1 (insert x tl))
      else Cons x (Cons hd tl)
```

```
sort = λxs.
  match xs with
    Nil →Nil
    Cons hd tl →
      insert hd (tick 1 (sort tl))
```

Figure 2.1. Insertion sort

For example, consider insertion sort: Figure 2.1 shows a recursive implementation of this sorting algorithm in a functional language. In this example we adopt a simple cost model where recursive calls incur unit cost, and all other operations do not require resources; we indicate this by wrapping recursive calls in a special operation tick, which consumes a given amount of resources. RAML can infer a tight quadratic bound on the cost of evaluating sort : $0.5(n^2+n)$, where n is the length of the input list. RELCOST allows one to prove a more complex bound: insertion sort requires resources proportional to the number of out-of-order pairs in the input. However, the proof must be written by hand. *Is it possible to develop a technique that admits both automation and expressiveness and can automatically verify these kinds of fine-grained bounds?*

Liquid Types and Resources

Recent work on RESYN [84] takes a first step in this direction by extending a *liquid type system* with resource analysis. Liquid types [115] support automatic verification of nontrivial functional properties with an SMT solver. RESYN augments an existing liquid type system [107] with a single construct: types can be annotated with a numeric quantity called *potential*. For example, a value of type Int^1 carries a single unit of potential, which can be used to pay for an operation with unit cost. Combined with polymorphic datatypes, this mechanism can describe uniform assignment of potential to the elements of a data structure. For example, instantiating a polymorphic list type $\mathsf{List}\,a$ with $a \mapsto \mathsf{Int}^1$ yields $\mathsf{List}\,\mathsf{Int}^1$, a type of lists where every element has a single unit of potential.

The RESYN type checker verifies that a program has enough potential to pay for all operations that may occur during evaluation. For example, RESYN can check the implementation of insert in Figure 2.1 against the (polymorphic) type $x : a \to xs : \mathsf{List\,a}^1 \to \mathsf{List\,a}$ to verify that the function makes one recursive call per element in the input xs. Here $\mathsf{List\,a}^1$ stands for the type of lists where each element has one more unit of potential than prescribed by type a.

More interestingly, the combination of refinements and potential annotations allows RESYN to verify *value-dependent* resource bounds. To this end, RESYN supports the use of conditional linear arithmetic (CLIA) terms as potential annotations, as opposed to just constants. For example, RESYN can also check insert against the type $x : a \to xs : \mathsf{List\,a}^{\mathsf{ite}(x>\nu,1,0)} \to \mathsf{List\,a}$, which states that insert only makes a recursive call for each element in xs smaller than x. The annotation on the type of the list elements conditionally assigns potential to a value in the list only when it is smaller than x[1]. RESYN reduces this type checking problem to a system of second-order CLIA constraints, which can be solved relatively efficiently using existing program synthesis techniques [9].

Challenge: Analyzing super-linear bounds

A major limitation of the RESYN type system is that it only supports *linear bounds*. In particular, a type of the form $\mathsf{List\,a}^p$ distributes the potential p *uniformly* throughout the list, and hence cannot express resource consumption of a super-linear function like insertion sort, which traverses the end of the input list *more often* than the beginning (recall that insertion sort recursively sorts the tail of the list and traverses the newly sorted tail again to insert an element). To verify this function, we need a type that allots more potential to elements in the tail of a list than the head. In this chapter, we propose two simple extensions to the RESYN type system to support the verification of super-linear resource bounds, while still generating only second-order CLIA constraints to keep type checking efficiently decidable.

[1] Throughout the paper, the special variable ν refers to an arbitrary inhabitant of the annotated type.

```
data QList a where
  QNil ::QList a
  QCons ::a →QList a^1 →QList a
```

```
data ISList a where
  ISNil ::ISList a
  ISCons ::x: a →xs: ISList a^ite(x>v,1,0) →ISList a
```

Figure 2.2. Two list types defined with inductive potentials: QList carries quadratic potential; in ISList, elements in the tail only have potential when they are larger than the head.

Super-linear Resource Analysis with Inductive Potentials

Our first insight is that we can describe non-uniform allocation of potential in a data structure by embedding potential annotations *into datatype definitions*. We dub this mechanism *inductive potentials*. For example, the datatype QList in Figure 2.2 (left) represents lists where every element has one more unit of potential than the one before it (the total amount of potential in the list is thus *quadratic* in its length). We express this non-uniform distribution of potential with the type of QCons: the elements in the tail of the list are of type a^1 instead of a, indicating that they must contain one more unit of potential than the head does. The datatype ISList in Figure 2.2 (right) is similar, but only assigns extra potential to those elements of the tail that are smaller than the head. Using these custom datatypes we can specify a coarse-grained (with QList) and fine-grained (with ISList) resource bound for insertion sort. Importantly, all potential annotations are still expressed in CLIA, so we can verify super-linear resource bounds while reusing RESYN's constraint-solving infrastructure.

Flexibility via Abstract Potentials

Inductive potentials, as descried so far, are somewhat restrictive. One must define a custom datatype for every resource bound. In the insertion sort example, we had to define QList to perform a coarse-grained analysis and ISList to perform a fine-grained analysis; moreover, both types have a fixed constant 1 embedded in their definition, so if the cost of tick inside insert were to increase, these types would no longer work. This is clearly unwieldy: instead, we would like to be able to write *libraries* of reusable data structures, each able to express a broad family of resource bounds.

To address this limitation, our second insight is to *parameterize* datatypes by numeric logic-level functions, which can then be used inside the datatype definition to allocate potential. We dub this second type system extension *abstract potentials*. With abstract potentials, the programmer can define a single datatype that represents a family of resource bounds, and then instantiate it with appropriate potential functions to verify different concrete bounds. For example, instead of defining QList and ISList separately, we can define a more general type $\mathsf{List}\,a\,\langle q :: a \to a \to \mathsf{Nat}\rangle$, where the parameter q abstracts over the potential annotation in the constructor. We can then instantiate q with different logic-level functions to perform different analyses. Importantly, type checking still generates constraints in the same logic fragment as RESYN. This design enables our type checker to automate resource analyses that would have previously required a handwritten proof.

Contributions

In summary, this chapter makes the following technical contributions:

1. *Liquid resource types* (LRT), a flexible type system for automatic resource analysis. With inductive and abstract potentials, programmers can analyze a variety of resource bounds by specifying how potential is allocated within a data structure.

2. *Semantics and a soundness proof* for the type system, including user-defined inductive data types.

3. A *prototype implementation*, LRTCHECKER, that automatically checks precise value-dependent resource bounds with existing constraint solving technology.

4. A *library of data types* corresponding to families of resource bounds, such as lists admitting polynomial or exponential bounds over their length, and trees admitting linear combinations of their size and height.

5. An *evaluation* on a set of challenging examples showing that LRTCHECKER automatically performs resource analyses out of scope of prior approaches.

2.2 Overview

We begin with examples to better illustrate how liquid resource types enable the automatic verification of precise resource bounds. First, we show how RESYN integrates resource analysis into a liquid type system. Second, we show how inductive potentials enable the analysis of super-linear bounds. Finally, we show how abstract potentials make this paradigm flexible and reusable.

2.2.1 Background: RESYN

Liquid Types

In a refinement type system [129, 36], types are annotated with logical predicates that constrain the range of their values. For instance, the type of natural numbers can be expressed as **type** $\mathsf{Nat} = \{\mathsf{Int} \mid \nu \geq 0\}$, where the special variable ν, as before, denotes an inhabitant of the type. Liquid types [115, 134] are a kind of refinement types that restrict logical refinements to only appear on *scalar* (*i.e.* non-function) types, and be expressed in decidable logics. Due to these restrictions, liquid types support fully automatic verification of nontrivial functional properties with the help of an SMT solver.

Potential Annotations

RESYN [84] extends liquid types with the ability to reason about the resource consumption of programs in addition to their functional properties. To this end, a type can also be annotated with a numeric logic expression called *potential*, as well as a logical refinement. For example, the type Nat^1 ranges over natural numbers that carry a single unit of potential. Intuitively, potential can be used to "pay" for evaluating special tick terms, which are placed throughout the program to encode a cost model. For example, the context $[x : \mathsf{Nat}^1]$ has a total of 1 unit of *free potential*, which is sufficient to type-check a term like tick 1 (). Because duplicating potential would lead to unsound resource analysis, RESYN's type system is *affine*, which means that creating two copies of a context—for example, to type-check both sides of an

application—requires distributing the available potential between them.

Simple potential annotations can be combined with other features of the type system, such as polymorphic datatypes, to specify more complex allocation of resources. For example, instantiating a polymorphic datatype $\mathsf{List\,a}$ with $a \mapsto \mathsf{Nat}^1$ yields the type $\mathsf{List\,Nat}^1$ of natural-number lists that carry one unit of potential per element. Here and throughout the paper, a missing potential annotation defaults to zero, so the type above stands for $(\mathsf{List\,Nat}^1)^0$. This default annotation hints at our more general notion of type substitution, where potential annotations are added together: instantiating a polymorphic datatype $\mathsf{List\,a}^m$ with $a \mapsto \mathsf{Nat}^n$ yields the type $\mathsf{List\,Nat}^{m+n}$.

Note that only "top-level" potential in a type contributes to the free potential of the context: for example, the context $[xs : \mathsf{List\,Nat}^1]$ has no free potential (which makes sense, since xs could be empty). The potential bundled inside an inductive datatype can be freed via pattern matching: for example, matching the xs variable above against $\mathsf{Cons\ hd\ tl}$ extends the context with new bindings $hd :: \mathsf{Nat}^1$ and $\mathsf{tl} :: \mathsf{List\,Nat}^1$; this new context has a single unit of free potential attached to hd (which also makes sense, since we now know that xs had at least one element).

Using potential annotations and tick terms, RESYN is able to specify upper bounds on resource consumption of recursive functions. Consider, for example, the function insert that inserts a value into a sorted list xs, as shown in Figure 2.1 (left). We wish to check that insert traverses the list linearly: more precisely, that it only makes a single recursive call per list element. To this end, we wrap the recursive call in a tick with unit cost, and annotate insert with the following type signature, which allocates one unit of potential per element of the input list:

$$insert :: x : \mathsf{a} \to xs : \mathsf{List\,a}^1 \to \mathsf{List\,a}$$

Type checking

We now describe how RESYN checks insert against this specification. At a high level, type checking reduces to generating a system of linear arithmetic constraints asserting that it is

1	[insert: x:a $\to$ xs:List a^P $\to$ List a]	insert = λx. λxs.	1
2	[insert: ..., x: a, xs: List a^P]	**match**	2
3	[insert: ..., x: a, xs: List a^P]	xs **with**	3
4	[insert: ..., x: a, xs: List a]	Nil $\to$Cons x Nil	4
5	[insert: ..., x: a, xs: List a, hd: a^P, tl: List a^P]	Cons hd tl $\to$	5
6	[insert: ..., x: a, xs: List a, hd: a^{p_1}, tl: List a^{q_1}]	**if** hd <x	6
7	[insert: ..., x: a, xs: List a, hd: a^{p_2}, tl: List a^{q_2}, hd <x]	**then** Cons hd (tick 1	7
8	[insert: ..., x: a, xs: List a, hd: a$^{p_2-1}$, tl: List a^{q_2}, hd <x]	(insert x tl))	8
9	[insert: ..., x: a, xs: List a, hd: a^{p_2}, tl: List a^{q_2}, ¬(hs <x)]	**else** Cons x (Cons hd tl)	9

Figure 2.3. On the right, the implementation of insert alongside the contexts used for type checking. Each line of the program corresponds to a subexpression that generates resource constraints, with the typing context relevant for constraint generation alongside it to the left. The start of the match expression is split between two lines to separate the context used to type the entire match expression from the context used to type the scrutinee. P is used as a symbolic resource annotation, as we will check this program against different bounds by providing concrete valuations for P.

possible to partition the potential available in the context amongst all expressions that need to be evaluated. If this system of constraints is satisfiable, the given resource bound is sufficient. We generate three kinds of constraints: *sharing* constraints, which nondeterministically partition resources between subexpressions, *subtyping* constraints, which check that a given term has enough potential to be used in a given context, and *well-formedness* constraints, which assert that potential annotations are non-negative.

Figure 2.3 illustrates type-checking of insert: its left-hand side shows the context in which various subexpressions are checked (for now you can ignore the *path constraints*, shown in red). The annotations in the figure are abstract; we will use the same figure to describe how we check both dependent and constant resource bounds. For this first example, we set $P = 1$ in the top-level type annotation of insert – we are checking that insert only makes one recursive call per element in xs.

The body of insert starts with a pattern match, which requires distributing the resources in the context on line 2 between the match scrutinee and the branches. This context has no free potential, but it does have some bundled potential in xs:List a^1; bundled potential also has to be

shared between the two copies of the context, since it could later be freed by pattern matching. In this case, however, xs is not mentioned in either of the branches, so for simplicity we elide the sharing constraints and assign all its potential to line 3, leaving $\mathsf{List}\,\mathsf{a}^1$ in the context of the match scrutinee and $\mathsf{List}\,\mathsf{a}^0$ in the context of the branches. Matching the scrutinee type $\mathsf{List}\,\mathsf{a}^1$ against the type of the Cons constructor introduces new bindings $hd :: \mathsf{a}^1$ and $\mathsf{tl} :: \mathsf{List}\,\mathsf{a}^1$ into the context: now we have 1 unit of free potential at our disposal, as the input list has at least one element.

When checking the conditional, we must again partition all available resources between the guard and either of the two branches. In particular, we partition the hd binding from line 5 into $hd : \mathsf{a}^{p_1}$ and $\mathsf{hd} : \mathsf{a}^{p_2}$, generating a *sharing constraint* that reduces to $1 = p_1 + p_2$. Similarly, we also partition the remaining potential in tl into $tl : \mathsf{List}\,\mathsf{a}^{q_1}$ and $\mathsf{tl} : \mathsf{List}\,\mathsf{a}^{q_2}$, which produces a constraint $1 = q_1 + q_2$ preventing us from reusing potential still contained in the list. ReSyn partitions resources non-deterministically and offloads the work of finding a concrete partitioning to the constraint solver. Neither the guard nor the else branch contains a tick expression, so they generate only trivial constraints. The then branch is more involved, as it does contain a tick with a unit cost. We must pay for this tick using the free potential p_2 on hdleaving $hd : \mathsf{a}^{p_2-1}$ in the context when checking the expression inside the tick on line 8. Like all bindings in the context, this binding generates a *well-formedness constraint* on its type, which reduces to the arithmetic constraint $p_2 - 1 \geq 0$, thereby implicitly checking that p_2 is sufficient to pay for the tick.

Finally, type-checking the application of insert x to tl produces a *subtyping constraint* between the actual and the formal argument types: $\Gamma \vdash \mathsf{List}\,\mathsf{a}^{q_2} <: \mathsf{List}\,\mathsf{a}^1$. This in turn reduces to an arithmetic constraint $q_2 \geq 1$, asserting that tl contains enough potential to execute the recursive call.

Now, consider the complete system of generated arithmetic constraints:

$$\exists p_1, p_2, q_1, q_2 \in \mathbb{N}. 1 = p_1 + p_2 \wedge 1 = q_1 + q_2 \wedge p_2 - 1 \geq 0 \wedge q_2 \geq 1$$

Though elided above, recall that all symbolic annotations are also required to be non-negative.

This system of constraints is satisfiable by setting $p_2, q_2 = 1$ and the rest of the unknowns to 0, which RESYN automatically infers using an SMT solver.

Value-dependent resource bounds

RESYN also supports verification of dependent resource bounds. We can use a logic-level conditional to give the following more precise bound for insert:

$$insert :: x : \mathsf{a} \to xs : \mathsf{List\,a}^{\mathsf{ite}(x>\nu,1,0)} \to \mathsf{List\,a}$$

The dependent annotation on xs indicates that only those list elements smaller than x carry potential, reflecting the fact that the implementation does not make any recursive calls once it has found the appropriate place to insert x.

Type checking proceeds similarly to the non-dependent case, except that we set $P = ite(\mathsf{x} > \nu, 1, 0)$ and treat all other symbolic potential annotations as unknown *logic-level terms* over the program variables (including the special variable ν). As a result, type checking generates second-order CLIA constraints, which are universally quantified over the program variables, and may contain assumptions on these variables, derived from their logical refinements or from *path constraints* of branching expressions. For example, Figure 2.3 shows in red the path constraints derived from the conditional. In particular, when checking the first branch, we can assume that $hd < x$ holds and thus conclude that hd has potential 1 in this branch and is able to pay the cost of tick. When we check that an annotation is well-formed, we must also assume that the relevant variable's logical refinements hold. For example, to check that the annotation $p_2(x, \nu)$ on hd is non-negative we must assert that $\nu = hd$.

More precisely, the full system of constraints (omitting irrelevant program variables)

becomes:

$$
\begin{aligned}
&\exists p_1, p_2, q_1, q_2 \in \mathbb{N} \times \mathbb{N} \to \mathbb{N}. \forall x, \mathsf{hd}, \nu. \\
&\quad \mathsf{ite}(x > \nu, 1, 0) = p_1(\mathsf{x}, \nu) + p_2(\mathsf{x}, \nu) && \text{Sharing hd (line 5)} \\
&\quad \wedge \mathsf{ite}(x > \nu, 1, 0) = q_1(\mathsf{x}, \nu) + q_2(\mathsf{x}, \nu) && \text{Sharing tl (line 5)} \\
&\quad \wedge (\nu = hd \wedge \mathsf{hd} < x) \implies p_2(\mathsf{x}, \nu) - 1 \geq 0 && \text{Well-formedness of hd (line 8)} \\
&\quad \wedge hd < \mathsf{x} \implies q_2(\mathsf{x}, \nu) \geq \mathsf{ite}(\mathsf{x} > \nu, 1, 0) && \text{Subtyping of tl (from recursive call)}
\end{aligned}
$$

RESYN satisfies these constraints by setting $p_2, q_2 = \lambda(x, \nu).\mathsf{ite}(\mathsf{x} > \nu, 1, 0)$, and the rest of the unknowns to to $\lambda(x, \nu).0$. Synthesis of CLIA expressions is a well-studied problem [9, 113], and RESYN uses counterexample-guided inductive synthesis (CEGIS) [127] to solve the particular form of constraints that arise.

Limitations

While RESYN's type system enables the analysis of the resource consumption of a wide variety of functions, and can automatically check value-dependent resource bounds, it still falls short of analyzing many useful programs. The system only expresses linear bounds, which are sufficient for many data structure traversals, but not sufficient for programs that compose several traversals. Thus, RESYN cannot check the resource consumption of sort. We need a way to extend this technique to programs with more complex recursive structure. RESYN also formalizes the technique only for lists, while we would like to be able to analyze programs that manipulate arbitrary algebraic data types.

2.2.2 Our Contribution: Liquid Resource Types

To address these limitations and enable verification of super-linear bounds, this work extends the RESYN type system with two powerful mechanisms: *inductive potentials* allow the programmer to define inductively how potential is allocated within a datatype, while *abstract*

potentials support parameterizing datatype definitions by potential functions. We dub the extended type system *liquid resource types* (LRT).

Inductive Potentials

Inductive potentials are expressed simply as potential annotations on constructors of a datatype. Figure 2.2 (left) shows a simple example of a datatype, QList, with inductive potentials. Here the QCons constructor mandates that the tail of the list (a) carries at least one more unit of potential in each element than the head, and (b) is itself a QList. As a result, the total potential in a value $L = [a_1, a_2, \ldots, a_n]$ of type *QList T* is *quadratic* in n and given by the following expression (where p is the potential of type T):

$$\Phi(L) = \sum_i p + \sum_i \sum_{j>i} 1 = np + \sum_i i = \frac{n(n+2p-1)}{2}$$

We can now specify that insertion sort runs in quadratic time by giving it the type:

$$sort :: \mathsf{xs} : \mathsf{QList\ a}^1 \rightarrow \mathsf{List\ a}$$

According to the formula above, this type assigns xs the total potential of $0.5(n^2+n)$, which is precisely the bound inferred by RAML, as we mentioned in the introduction. More interestingly, we can use *value-dependent* inductive potentials to specify a tighter bound for sort, by the replacing QList in the type signature above with ISList defined in Figure 2.2 (right). In an ISList, the elements in the tail only carry the extra potential when their value is less than the head. Hence, the total potential stored in an $ISLista^1$ is equal to the number of list elements plus the number of *out-of-order pairs* of list elements. Verifying sort against this bound implies, for example, that insertion sort behaves linearly on a fully sorted list (with no decreasing element pairs) and takes the full $0.5(n^2+n)$ steps on a list sorted in reverse order.

While inductive potentials are able to express non-linear bounds, on their own, they are difficult to use: the non-linear coefficient of a resource bound is built into the datatype definition,

```
data List t <q::t →t →Nat> where
  Nil ::List t <q>
  Cons ::x: t →xs: List t^q(x,v) <q> →List t <q>
```

Figure 2.4. A list datatype parameterized by a value-dependent, quadratic abstract potential.

and hence any slight change in the analysis or the cost model—such as changing the cost of a recursive call from 1 to 2—requires defining a new datatype. We would like to be able to reuse the *structure* of these types without relying on the precise potential annotations embedded within.

Abstract potentials

To make inductive potentials reusable, we introduce the second new feature of LRT, which we dub *abstract potentials*. This feature is inspired by abstract refinement types [134], which parameterize datatypes by a refinement predicate; similarly, LRT allows parameterizing a datatype a potential function. Consider the definition of the List datatype in Figure 2.4: this datatype is parameterized by a numeric logic-level function q, which represents the additional potential contained in every element of every proper suffix of the list. This interpretation is revealed in the Cons constructor, where the value $q(x, v)$ is *added* to the linear potential annotation on the tail of the list. Note that since q is a function, this datatype subsumes both QSort and ISSort, as well as a broad range of value-dependent "quadratic" potential functions. More precisely, if a list element v of type T carries $p(v)$ units of potential, then the total potential in a list $L = [a_1, a_2, \ldots, a_n]$ of type *List T* is given by the following formula:

$$\Phi(L) = \sum_i p(a_i) + \sum_i \sum_{j>i} q(a_i, a_j)$$

Note that we can add higher-arity abstract potentials to extend the List datatype to support higher-degree polynomials. Similarly, we can add a unary abstract potential $p(v)$ to express the linear component of the list potential more explicitly (as opposed to relying on polymorphism in

1	$[\mathsf{insert}\colon \forall \mathsf{b}.x\colon \mathsf{b} \to xs\colon \mathsf{List}\,\mathsf{b}^1 \to \mathsf{List}\,\mathsf{b},\ \mathsf{sort}\colon \forall \mathsf{c}.xs\colon \mathsf{List}\,\mathsf{c}^1\,\langle Q\rangle \to \mathsf{List}\,\mathsf{c}]$	sort = λxs.	1
2	$[\mathsf{insert},\ \mathsf{sort}\colon \ldots,\ \mathsf{xs}\colon \mathsf{List}\,\mathsf{a}^1\,\langle Q\rangle]$	**match**	2
3	$[\mathsf{insert},\ \mathsf{sort}\colon \ldots,\ \mathsf{xs}\colon \mathsf{List}\,\mathsf{a}^1\,\langle Q\rangle]$	xs **with**	3
4	$[\mathsf{insert},\ \mathsf{sort}\colon \ldots,\ \mathsf{xs}\colon \mathsf{List}\,\mathsf{a}]$	Nil →Nil	4
5	$[\mathsf{insert},\ \mathsf{sort}\colon \ldots,\ \mathsf{xs}\colon \mathsf{List}\,\mathsf{a},\ \mathsf{hd}\colon \mathsf{a}^1,\ \mathsf{tl}\colon \mathsf{List}\,\mathsf{a}^{1+Q(\mathsf{hd},v)}\,\langle Q\rangle]$	Cons hd tl →	5
6	$[\mathsf{insert},\ \mathsf{sort}\colon \ldots,\ \mathsf{xs}\colon \mathsf{List}\,\mathsf{a},\ \mathsf{hd}\colon \mathsf{a}^{p_1},\ \mathsf{tl}\colon \mathsf{List}\,\mathsf{a}^{q_1(\mathsf{hd},v)}\,\langle q_1\rangle]$	insert hd	6
7	$[\mathsf{insert},\ \mathsf{sort}\colon \ldots,\ \mathsf{xs}\colon \mathsf{List}\,\mathsf{a},\ \mathsf{hd}\colon \mathsf{a}^{p_2},\ \mathsf{tl}\colon \mathsf{List}\,\mathsf{a}^{q_2(\mathsf{hd},v)}\,\langle q_2\rangle]$	(tick 1	7
8	$[\mathsf{insert},\ \mathsf{sort}\colon \ldots,\ \mathsf{xs}\colon \mathsf{List}\,\mathsf{a},\ \mathsf{hd}\colon \mathsf{a}^{p_2-1},\ \mathsf{tl}\colon \mathsf{List}\,\mathsf{a}^{q_2(\mathsf{hd},v)}\,\langle q_2\rangle]$	(sort tl))	8

Figure 2.5. Similar to Figure 2.3, the evolution of the typing context while checking different subexpressions of sort. Q is used as a symbolic resource annotation, as we will check this program against different bounds by providing concrete valuations for Q.

the type of the elements).

Type checking

With abstract potentials, we can use the same List datatype from Figure 2.4 to verify both coarse- and fine-grained bounds for insertion sort. For the coarse-grained case, we can give this function the following type signature:

$$sort :: \mathsf{xs}\colon \mathsf{List}\,\mathsf{a}^1\,\langle \lambda(\_,\_).1\rangle \to \mathsf{List}\,\mathsf{a}$$

As before, omitted potential annotations are zero by default, so the return type $\mathsf{List}\,\mathsf{a}$ is short for $(\mathsf{List}\,\mathsf{a}^0\,\langle\lambda(\_,\_).0\rangle)^0$ The type checking process is illustrated in Figure 2.5, where we set $Q = \lambda(\_,\_).1$. The initial context contains bindings for both the helper function insert and the function sort itself, which can be used to make a recursive call. More precisely, the binding for sort is added to the context as a result of type-checking the implicit fixpoint construct that wraps the lambda abstraction. Importantly for this example, LRT supports *polymorphic recursion*: the type c of list elements in the recursive call can be different from the type a of list elements in the body.

The top-level term in the body of sort is a pattern-match, so, as before, we have to split the context between the scrutinee and the branches. Since neither of the branches mentions

xs, for simplicity we omit the sharing constraints and leave all of its potential with line 3, thus inferring the type $\mathsf{List}\,\mathsf{a}^1\,\langle 1\rangle$ for the scrutinee. Matching this type against the return type of the Cons constructor in Figure 2.4, yields the substitution $t \mapsto \mathsf{a}^1, q \mapsto 1$, adding the following two new bindings to the context of the Cons branch: $hd : \mathsf{a}^1$ and $\mathsf{tl} : \mathsf{List}\,\mathsf{a}^2\,\langle \lambda(\_,\_).1\rangle$. Importantly, the tail list tl ends up with more linear potential than the original list xs, which is precisely the purpose of the inductive potential annotations in Figure 2.4, and is necessary to afford *both* the recursive call and the call to insert.

Proceeding with type-checking the Cons branch, note that there are three terms that consume resources: the application of insert hs, the tick expression, and the recursive call. We can use the free unit of potential attached to hd to pay for tick. As for tl, recall that it has twice the potential that the recursive call to sort consumes, and we would like to "save up" this extra potential to pay for the application of insert hs to the result of the recursive call. This is where polymorphic recursion comes in: the type checker is free to instantiate c in the type of the recursive call with a^s, essentially giving every list element some amount of extra potential s which is simply "piped through" the call; LRT leaves the exact value of s for the solver to find.

All together, type checking leaves us the following system of arithmetic constraints:

$$\exists p_1, p_2, q_1, q_2, s \in \mathbb{N}.\, p_1 + p_2 = 1 \wedge p_2 - 1 \geq 0$$
$$\wedge\, q_1 + q_2 = 2 \wedge q_2 \geq s + 1 \wedge s \geq 1$$

which is satisfiable with $p_2, q_2, s = 1$ and the rest of unknowns set to 0. Note that while the annotations in Figure 2.5 involve applications of abstract potentials, all potential functions involved in the coarse-grained version of the example are constants, so we can treat these as simple first-order numerical constraints.

Value-dependent resource bounds

Instantiating the abstract potentials with non-constant functions allows us to use the exact same List datatype to verify a fine-grained bound for insertion sort. To this end, we give it the type signature:

$$sort :: xs : \mathsf{List\,a}^{1}\,\langle \lambda(x_1,x_2).\,\mathsf{ite}(x_1 > x_2, 1, 0)\rangle \to \mathsf{List\,a}$$

Type checking still proceeds as illustrated in Figure 2.5, except we set $Q = \lambda(x_1,x_2).\,\mathsf{ite}(x_1 > x_2, 1, 0)$. One key difference is that matching the type of the scrutinee xs against the return type of Cons requires applying the abstract potential function, yielding:

$$tl : \mathsf{List\,a}^{1+\mathsf{ite}(\mathsf{x}>\nu,1,0)}\,\langle \lambda(x_1,x_2).\,\mathsf{ite}(x_1 > x_2, 1, 0)\rangle$$

in the typing context. The generated arithmetic constraints are similar to the coarse-grained case, but now symbolic potentials can be functions, so the constraints are second-order and must quantify over the program variables hd, ν and parameters x_1, x_2 of abstract potentials:

$$\exists p_1, p_2, q_1, q_2, s \in \mathbb{N}\times\mathbb{N}\to\mathbb{N}.\,\forall hd, \nu, x_1, x_2 \in \mathbb{N}.$$

$p_1(hd,\nu) + p_2(\mathsf{hd},\nu) = 1$	Sharing hd (line 5)
$\wedge\, p_2(hd,\nu) - 1 \geq 0$	Well-formedness of hd (line 8)
$\wedge\, q_1(hd,\nu) + q_2(\mathsf{hd},\nu) = 1 + \mathsf{ite}(\mathsf{hd} > \nu, 1, 0)$	Sharing tl (line 5)
$\wedge\, q_2(hd,\nu) \geq s(\mathsf{hd},\nu) + 1$	Subtyping from the call to sort
$\wedge\, s(hd,\nu) \geq \mathsf{ite}(\mathsf{hd} > \nu, 1, 0)$	Subtyping from the call to insert

The solver can validate these constraints by setting $p_2, \lambda(x_1,x_2).1$, $q_2, s = \lambda(x_1,x_2).\mathsf{ite}(x_1 > x_2, 1, 0)$, and the rest of the unknowns to $\lambda(x_1,x_2).0$. Importantly, even though inductive and abstract potentials significantly increase the expressiveness of the type system, the generated constraints still belong to the same logic fragment (second-order CLIA), as constraints generated

by RESYN, and hence are efficiently decidable. This is a consequence of the core design principle that differentiates LRT from other fine-grained resource analysis techniques [111, 139, 57]: to encode complex resource consumption, rather than increasing the complexity of the resource annotations, we embed *simple annotations* into *complex types*.

Although in this section we focused solely on the resource consumption of insertion sort, LRT is also able to specify and verify its functional properties—that the output list is sorted and contains the same number and/or set of elements as the input list. To this end, LRT relies on existing liquid type checking techniques [134, 107]. Additionally, while this section only shows the use of inductive and abstract potentials for expressing quadratic potentials on lists, Section 2.4 further demonstrates the flexibility of this specification style. In particular, we show how to use abstract potentials to analyze exponential-time algorithms, as well as reason about the resource consumption of tree-manipulating programs in terms of their height and size.

2.3 Technical Details

In this section, we formulate a substantial subset of our type system as a core calculus and prove type soundness. This subset features natural numbers and Booleans that are refined by their values, as well as user-defined inductive datatypes that can be refined by user-defined measures. The gap from the core calculus to our full type system involves abstract refinements and polymorphic datatypes. The restriction to this subset in the technical development is only for brevity and proofs carry over to all the features of our tool.

2.3.1 Setting the Stage: A Resource-Aware Core Language

Syntax

Figure 2.6 presents the grammar of terms in the core calculus via abstract binding trees [58]. We extend the core language of Re^2 [84] with natural numbers, null tuples, ordered pairs, and replace lists with general inductive data structures. Expressions are in *a-normal-form* [116], which means that syntactic forms for non-tail positions allow only *atoms* $\hat{a} \in \mathsf{Atom}$,

$$
\begin{array}{rcl}
a \in \mathsf{SimpAtom} & ::= & x \mid \overline{n} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{triv} \mid \mathsf{pair}(a_1, a_2) \mid C(a_0, \langle a_1, \cdots, a_m \rangle) \\
\hat{a} \in \mathsf{Atom} & ::= & a \mid \lambda(x.e_0) \mid \mathsf{fix}(f.x.e_0) \\
e \in \mathsf{Exp} & ::= & a \mid \mathsf{if}(a_0, e_1, e_2) \mid \mathsf{matp}(a_0, x_1.x_2.e_1) \\
& \mid & \mathsf{matd}(a_0, \overrightarrow{C_j(x_0, \langle x_1, \cdots, x_{m_j} \rangle).e_j}) \\
& \mid & \mathsf{app}(\hat{a}_1, \hat{a}_2) \mid \mathsf{let}(e_1, x.e_2) \mid \mathsf{impossible} \mid \mathsf{tick}(c, e_0) \\
v \in \mathsf{Val} & ::= & \overline{n} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{triv} \mid \mathsf{pair}(v_1, v_2) \mid C(v_0, \langle v_1, \cdots, v_m \rangle) \\
& \mid & \lambda(x.e_0) \mid \mathsf{fix}(f.x.e_0)
\end{array}
$$

Figure 2.6. Syntax of the core calculus

which are irreducible terms, *e.g.*, variables and values, without loss of expressivity. The restriction simplifies typing rules in our system, as we will explain in Section 2.3.4. We further identify a subset SimpAtom of Atom that contains *interpretable* atoms in the refinement logic. Intuitively, the type of an interpretable atom $a \in \mathsf{SimpAtom}$ admits a well-defined *interpretation* that maps the value of a to its logical refinements, *e.g.*, lists can be refined by their lengths. A *value* $v \in \mathsf{Val}$ is an atom without reference to any program variable. An inductive data structure $C(v_0, \langle v_1, \cdots, v_m \rangle)$ is represented by the constructor name C, the stored data v_0 in this constructor, and a sequence of child nodes $\langle v_1, \cdots, v_m \rangle$. Note that the core language has two kinds of match expressions: matp for pairs and matd for inductive data structures.

The syntactic form impossible is used as a placeholder for unreachable code, *e.g.*, the then-branch of a conditional expression whose predicate is always false. The syntactic form $\mathsf{tick}(c, e_0)$ is introduced to define the cost model, and it is intended to cost $c \in \mathbb{Z}$ units of resource and then reduce to e_0. A negative c means that $-c$ units of resource will become available. The tick expressions support flexible user-defined resource metrics. For example, the programmers can wrap every recursive call in $\mathsf{tick}(1, \cdot)$ to count those function calls; alternatively, they may wrap every data constructor in $\mathsf{tick}(c, \cdot)$ to keep track of memory consumption, where c is the amount of memory allocated by the constructor.

Semantics

The resource consumption of a program is determined by a small-step operational cost semantics. The semantics is a standard structural semantics augmented with a *resource parameter*,

$\boxed{\langle e, q\rangle \mapsto \langle e', q'\rangle}$

$$\frac{}{\langle \mathsf{if}(\mathsf{true}, e_1, e_2), q\rangle \mapsto \langle e_1, q\rangle}\ (\text{E-COND-TRUE}) \qquad \frac{}{\langle \mathsf{if}(\mathsf{false}, e_1, e_2), q\rangle \mapsto \langle e_2, q\rangle}\ (\text{E-COND-FALSE}) \qquad \frac{v_1 \in \mathsf{Val}}{\langle \mathsf{let}(v_1, x.e_2), q\rangle \mapsto \langle [v_1/x]e_2, q\rangle}\ (\text{E-LET-VAL})$$

$$\frac{}{\langle \mathsf{tick}(c, e_0), q\rangle \mapsto \langle e_0, q - c\rangle}\ (\text{E-TICK}) \qquad \frac{v_1 \in \mathsf{Val} \qquad v_2 \in \mathsf{Val}}{\langle \mathsf{matp}(\mathsf{pair}(v_1, v_2), x_1.x_2.e_1), q\rangle \mapsto \langle [v_1, v_2/x_1, x_2]e_1, q\rangle}\ (\text{E-MATP-VAL})$$

$$\frac{v_0 \in \mathsf{Val} \qquad v_1 \in \mathsf{Val} \qquad \cdots \qquad v_{m_j} \in \mathsf{Val}}{\langle \mathsf{matd}(C_j(v_0, \langle v_1, \cdots, v_{m_j}\rangle), \overrightarrow{C_j(x_0, \langle x_1, \cdots, x_{m_j}\rangle).e_j}), q\rangle \mapsto \langle [v_0, v_1, \cdots, v_{m_j}/x_0, x_1, \cdots, x_{m_j}]e_j, q\rangle}\ (\text{E-MATD-VAL})$$

$$\frac{v_2 \in \mathsf{Val}}{\langle \mathsf{app}(\lambda(x.e_0), v_2), q\rangle \mapsto \langle [v_2/x]e_0, q\rangle}\ (\text{E-APP-ABS}) \qquad \frac{v_2 \in \mathsf{Val}}{\langle \mathsf{app}(\mathsf{fix}(f.x.e_0), v_2), q\rangle \mapsto \langle [\mathsf{fix}(f.x.e_0), v_2/f, x]e_0, q\rangle}\ (\text{E-APP-FIX})$$

Figure 2.7. Selected rules of the small-step operational cost semantics

which indicates the amount of available resources. The *single-step* reduction judgments have the form $\langle e, q\rangle \mapsto \langle e', q'\rangle$, where e and e' are expressions, and $q, q' \in \mathbb{Z}_0^+$ are nonnegative integers. The intuitive meaning of such a judgment is that with q units of available resources, e reduces to e' without running out of resources, and q' resources are left. Figure 2.7 shows some of the reduction rules of the small-step cost semantics. Note that all the judgments $\langle e, q\rangle \mapsto \langle e', q'\rangle$ implicitly constrain that $q, q' \geq 0$, so in the rule (E-TICK) for resource consumption, we do not need to distinguish whether the cost c is nonnegative or not.

The *multi-step* reduction relation $\mapsto^*$ is defined as the reflexive transitive closure of $\mapsto$. Multi-step reduction can be used to reason about *high-water mark* resource usage of a reduction from e to e', by finding the minimal q such that $\langle e, q\rangle \mapsto^* \langle e', q'\rangle$ for some q'. For monotone resources such as time, the high-water mark cost coincides with the *net cost*, *i.e.*, the sum of costs specified by tick expressions in the reduction. In general, net costs are invariant, *i.e.*, $p - p' = q - q'$ if $\langle e, p\rangle \mapsto^m \langle e', p'\rangle$ and $\langle e, q\rangle \mapsto^m \langle e', q'\rangle$, where $\mapsto^m$ is the m-element composition of $\mapsto$.

2.3.2 Types and Refinements

Refinements

We follow the approach of liquid types [114, 107, 84] and develop a refinement language that is distinct from the term language. Figure 2.8 formulates the syntax of the core type system. The refinement language is essentially a simply-typed lambda calculus augmented with logical connectives and linear arithmetic. As terms are classified by types, refinements ψ, ϕ are classified by *sorts* Δ. The core type system's sorts include Booleans $\mathbb{B}$, natural numbers $\mathbb{N}$, nullary $\mathbb{U}$ and binary products $\Delta_1 \times \Delta_2$, arrows $\Delta_1 \Rightarrow \Delta_2$, and *uninterpreted symbols* δ_α parametrized by type variables α. In our system, logical constraints ψ have sort $\mathbb{B}$, potential annotations ϕ have sort $\mathbb{N}$, and refinement-level functions have arrow sorts. Refinements can reference program variables. Our system interprets a program variable of Boolean, natural-number, or product type as its value, type variable α as an uninterpreted symbol of sort δ_α, and inductive datatype as its *measurement*, which is computed by a total function $\mathscr{I}_D$: (values of datatype D) $\to$ (refinements of sort Δ_D). The function $\mathscr{I}_D$ is derived by user-defined *measures* for datatypes, which we omit from the formal presentation; Although measures play an important role in specifying functional properties (*e.g.*, in [107]), they are orthogonal to resource analysis. We include the full development with measures in the technical report [85]

Formally, we define the following *interpretation* $\mathscr{I}(\cdot)$ to reflect interpretable atoms $a \in \mathsf{SimpAtom}$ as their logical refinements:

$$\mathscr{I}(x) = x$$

$$\mathscr{I}(\overline{n}) = n \qquad \mathscr{I}(\mathsf{triv}) = \star$$

$$\mathscr{I}(\mathsf{true}) = \top \qquad \mathscr{I}(\mathsf{false}) = \bot$$

$$\mathscr{I}(\mathsf{pair}(a_1, a_2)) = (\mathscr{I}(a_1), \mathscr{I}(a_2)) \qquad \mathscr{I}(C(a_0, \langle a_1, \cdots, a_m \rangle)) = \mathscr{I}_D(C(a_0, \langle a_1, \cdots, a_m \rangle))$$

Example 1 (Interpretations of datatypes)**.** Consider a natural-number list type $\mathsf{NatList}$ with

Refinement

$$\psi, \phi \;::=\; \nu \mid x \mid n \mid \star \mid \top \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \phi_1 \leq \phi_2 \mid \phi_1 + \phi_2 \mid \psi_1 = \psi_2 \mid \forall a{:}\Delta.\psi$$
$$\mid\; a \mid \lambda a{:}\Delta.\psi \mid \psi_1\ \psi_2 \mid (\psi_1, \psi_2) \mid \psi.\mathbf{1} \mid \psi.\mathbf{2}$$

Sort

$$\Delta \;::=\; \mathbb{B} \mid \mathbb{N} \mid \mathbb{U} \mid \delta_\alpha \mid \Delta_1 \times \Delta_2 \mid \Delta_1 \Rightarrow \Delta_2$$

Base Type

$$B \;::=\; \mathsf{nat} \mid \mathsf{bool} \mid \mathsf{unit} \mid B_1 \times B_2 \mid \mathsf{ind}^{\theta}_{\lhd,\pi}(\overrightarrow{C{:}(T,m)}) \mid m \cdot \alpha$$

Resource-Annotated Type

$$T \;::=\; R^{\phi}$$

Refinement Type

$$R \;::=\; \{B \mid \psi\} \mid m \cdot (x{:}T_x \to T)$$

Type Schema

$$S \;::=\; T \mid \forall\alpha.S$$

Figure 2.8. Syntax of the core type system

constructors Nil and Cons. In the core language, an empty list is encoded as $\mathsf{Nil}(\mathsf{triv}, \langle\rangle)$ and a singleton list containing a zero is represented as $\mathsf{Cons}(\overline{0}, \langle\mathsf{Nil}(\mathsf{triv}, \langle\rangle)\rangle)$. Below defines an interpretation $\mathscr{I}_{\mathsf{NatList}}$: (values of NatList) $\to$ (refinements of sort $\mathbb{N}$) that computes the length of a list:

$$\mathscr{I}_{\mathsf{NatList}}(\mathsf{Nil}(\mathsf{triv}, \langle\rangle)) \stackrel{\text{def}}{=} 0, \qquad \mathscr{I}_{\mathsf{NatList}}(\mathsf{Cons}(v_h, \langle v_t\rangle)) \stackrel{\text{def}}{=} \mathscr{I}_{\mathsf{NatList}}(v_t) + 1.$$

In the rest of this section, we will assume that the type NatList admits a length interpretation.

We will use the abbreviations $\bot, \vee, \Longrightarrow, \geq, <, >, \mathbf{ite}$ with obvious semantics; *e.g.*, $\psi_1 \vee \psi_2 \stackrel{\text{def}}{=} \neg(\neg\psi_1 \wedge \neg\psi_2)$ and $\mathbf{ite}(\psi_0, \psi_1, \psi_2) \stackrel{\text{def}}{=} (\psi_0 \implies \psi_1) \wedge (\neg\psi_0 \implies \psi_2)$. We will also abbreviate the m-element sum $\psi + \psi + \cdots + \psi$ as $m \times \psi$. We will use finite-product sorts $\Delta_1 \times \Delta_2 \times \cdots \times \Delta_m$, or $\prod_{i=1}^{m} \Delta_i$ for short, with an obvious encoding with nullary and binary products. We will also write $\psi.\mathbf{i}$ as the i-th projection from a refinement of a finite-product sort.

Types

We adapt the methodology of Re^2 [84] and classify types into four categories. Base types B are natural numbers, Booleans, nullary and binary products, inductive datatypes, and type variables. An inductive datatype $\mathsf{ind}^{\theta}_{\lhd,\pi}(\overrightarrow{C{:}(T,m)})$ consists of a sequence of constructors, each of which has a name C, a content type T (which must be a scalar type), and a finite

number $m \in \mathbb{Z}_0^+$ of child nodes. In terms of recursive types, $(\overrightarrow{C:(T,m)})$ compactly represents $\mathsf{rec}(X.\overrightarrow{C:T \times X^m})$, where X^m is the m-element product type $X \times X \times \cdots \times X$, *e.g.*, the type $\mathsf{NatList}$ in 1 can be seen as an abbreviation of $\mathsf{ind}(\mathsf{Nil}:(\mathsf{unit},0),\mathsf{Cons}:(\mathsf{nat},1))$. We will explain the resource-related parameters $\theta,\lhd$, and π later in Section 2.3.3. Type variables α are annotated with a *multiplicity* $m \in \mathbb{Z}_0^+ \cup \{\infty\}$, which specifies an upper bound on the number of references for a program variable of such a type. For example, $\mathsf{ind}(\mathsf{Nil}:(\mathsf{unit},0),\mathsf{Cons}:(2\cdot\alpha,1))$ denotes a universal list, each of whose elements can be used at most twice.

Refinement types R are *subset types* and *dependent arrow types*. Inhabitants of a subset type $\{B \mid \psi\}$ are values of type B that satisfy the refinement ψ. The refinement ψ is a logical formula over program variables and a special *value variable* ν, which is distinct from program variables and represents the inhabitant itself. For example, $\{\mathsf{bool} \mid \neg\nu\}$ is a type of false, $\{\mathsf{nat} \mid \nu > 0\}$ is a type of positive integers, and $\{\mathsf{NatList} \mid \nu = 1\}$ stands for singleton lists of natural numbers. A dependent arrow type $x:T_x \to T$ is a function type whose return type may reference its formal argument x. Similar to type variables, these arrow types are also annotated with a multiplicity $m \in \mathbb{Z}_0^+ \cup \{\infty\}$ bounding from above the number of times a function of such a type can be applied.

Resource-annotated types R^ϕ are refinement types R augmented with potential annotations ϕ. The resource annotations are used to carry out the potential method of amortized analysis [130]; intuitively, R^ϕ assigns ϕ units of potential to values of the refinement type R. The potential annotation ϕ can also reference the value variable ν. For example, $\mathsf{NatList}^{2\times\nu}$ describes natural-number lists ℓ with $2 \cdot \mathscr{I}_{\mathsf{NatList}}(\ell) = 2 \cdot |\ell|$ units of potential where $|\ell|$ is the length of ℓ. As we will show in Section 2.3.3, the same potential can also be expressed by assigning 2 units of potential to each element in the list.

Type schemas represent possibly polymorphic types, where the type quantifier $\forall$ is only allowed to appear outermost in a type. Similar to Re^2 [84], we only permit polymorphic types to be instantiated with *scalar* types, which are resource-annotated base types (possibly with subset

constraints). Intuitively, the restriction derives from the fact that our refinement-level logic is first-order, which renders our type system decidable.

We will abbreviate $1 \cdot \alpha$ as α, $\{B \mid \top\}$ as B, $\infty \cdot (x{:}T_x \to T)$ as $x{:}T_x \to T$, and R^0 as R.

2.3.3 Potentials of Inductive Data Structures

Resource-annotated types R^{ϕ} provide a mechanism to specify potential functions of inductive data structures in terms of their interpretations. However, this mechanism is not so expressive because it can only describe potential functions that are *linear* with respect to the interpretations of data structures, since our refinement logic only has linear arithmetic. One way to support non-linear potentials is to extend the refinement logic with non-linear arithmetic, which would come at the expense of decidability of the type system. In contrast, our type system adapts the idea of *univariate polynomial potentials* [68] to a refinement-type setting. This combination allows us to not only reason about polynomial resource bounds with linear arithmetic in the refinement logic, but also derive fine-grained resource bounds that go beyond the scope of prior work on typed-based amortized resource analysis [68, 65, 84].

Simple numeric annotations

We start by adding numeric annotations to datatypes, following the approach of univariate polynomial potentials [68]. Recall the type $\mathsf{NatList}$ introduced in 1. We now annotate it with a vector $\vec{q} = (q_1, \cdots, q_k) \in (\mathbb{Z}_0^+)^k$ and denote the annotated type by $\mathsf{NatList}^{\vec{q}}$. The annotation is intended to assign q_1 units of potential to every element of the list, q_2 units of potential to every element of every suffix of the list (*i.e.*, to every ordered pair of elements), q_3 units of potential to the elements of the suffixes of the suffixes (*i.e.*, to every ordered triple of elements), etc. Let ℓ be a list of type $\mathsf{NatList}$ and $\Phi(\ell : \mathsf{NatList}^{\vec{q}})$ be its potential with respect to the annotated type. Then the potential function $\Phi(\cdot)$ can be expressed as a linear combination of binomial coefficients,

where $|\ell|$ is the length of ℓ:

$$\Phi(\ell : \mathsf{NatList}^{\vec{q}}) = \sum_{i=1}^{k} \sum_{1 \le j_1 < \cdots < j_i \le |\ell|} q_i = \sum_{i=1}^{k} q_i \cdot \binom{|\ell|}{i}. \tag{2.1}$$

For example, $\mathsf{NatList}^{(2)}$ assigns 2 units of potential to each list element, so it describes lists ℓ with $2 \cdot |\ell|$ units of potential.

As shown by the proposition below, one benefit of the binomial representation in (2.1) is that the potential function $\Phi(\cdot)$ can be defined *inductively* on the data structure, and be expressed using only linear arithmetic.

Proposition 5. *Define the potential function* $\Phi(\cdot)$ *for type* $\mathsf{NatList}^{\vec{q}}$ *as follows:*

$$\Phi(\mathsf{Nil}(\mathsf{triv}, \langle\rangle) : \mathsf{NatList}^{\vec{q}}) \stackrel{\text{def}}{=} 0, \quad \Phi(\mathsf{Cons}(v_h, \langle v_t \rangle) : \mathsf{NatList}^{\vec{q}}) \stackrel{\text{def}}{=} q_1 + \Phi(v_t : \mathsf{NatList}^{\lhd(\vec{q})}),$$

where a potential shift *operator* $\lhd$ *is defined as* $\lhd(\vec{q}) \stackrel{\text{def}}{=} (q_1 + q_2, q_2 + q_3, \cdots, q_{k-1} + q_k, q_k)$. *Then* (2.1) *gives a closed-form solution to the inductive definition above.*

Based on the observation presented above, prior work [68, 65] builds an automatic resource analysis that infers polynomial resource bounds via efficient *linear programming* (LP). In this work, our main goal is not to develop an automatic inference algorithm, but rather to extend the expressivity of the potential annotations.

Dependent annotations

Our first step is to generalize numeric potential annotations to dependent ones. The idea is to express the potential annotations in the refinement language of our type system. For example, we can annotate the type NatList with a vector $\theta = (\theta_1, \cdots, \theta_k)$, where θ_i is a refinement-level abstraction of sort $\mathbb{N}^i \Rightarrow \mathbb{N}$, for every $i = 1, \cdots, k$. Intuitively, θ_i denotes the amount of potential assigned to ordered i-tuple of elements in a list, depending on the actual values of the elements, *i.e.*, let $\ell = [v_1, \cdots, v_{|\ell|}]$ be a list of natural numbers, then the potential function $\Phi(\cdot)$ with respect

to the dependently annotated type $\mathsf{NatList}^{\theta}$ can be expressed as

$$\Phi(\ell : \mathsf{NatList}^{\theta}) = \sum_{i=1}^{k} \sum_{1 \le j_1 < \cdots < j_i \le |\ell|} \theta_i(v_{j_1}, \cdots, v_{j_i}). \tag{2.2}$$

Example 2 (Dependent potential annotations)**.** Suppose we want to assign the number of ordered pairs (a,b) satisfying $a > b$ in a list ℓ of type $\mathsf{NatList}^{\theta}$ as the potential of ℓ. Then the desired potential function is $\Phi(\ell : \mathsf{NatList}^{\theta}) = \sum_{1 \le j_1 < j_2 \le |\ell|} \mathbf{ite}(v_{j_1} > v_{j_2}, 1, 0)$. Compared with (2.2), a feasible $\theta = (\theta_1, \theta_2)$ can be defined as follows:

$$\theta_1 \stackrel{\text{def}}{=} \lambda x{:}\mathbb{N}.0, \qquad\qquad \theta_2 \stackrel{\text{def}}{=} \lambda (x_1{:}\mathbb{N}, x_2{:}\mathbb{N}).\mathbf{ite}(x_1 > x_2, 1, 0).$$

Later we will show the dependent annotation given here can be used to derive a fine-grained resource bound for insertion sort at the end of Section 2.3.4.

Although dependent annotations seem to complicate the representation of potential functions, they *do* retain the benefit of numeric annotations. The key observation is that we can still express the potential *shift* operator $\lhd$ in our refinement language, which only permits linear arithmetic. Below presents a generalization of Theorem 5.

Proposition 6. *Define the potential function* $\Phi(\cdot)$ *for type* $\mathsf{NatList}^{\theta}$ *as follows:*

$$\begin{aligned}
&\Phi(\mathsf{Nil}(\mathsf{triv}, \langle\rangle) : \mathsf{NatList}^{\theta}) \stackrel{\text{def}}{=} 0, \\
&\Phi(\mathsf{Cons}(v_h, \langle v_t \rangle) : \mathsf{NatList}^{\theta}) \stackrel{\text{def}}{=} \theta_1(v_h) + \Phi(v_t : \mathsf{NatList}^{\lhd(v_h)(\theta)}),
\end{aligned}$$

where a dependent potential shift operator $\lhd$ *is defined in the refinement-level language as*

$$\lhd \stackrel{\text{def}}{=} \lambda y{:}\mathbb{N}.\lambda(\theta_1{:}\mathbb{N} \Rightarrow \mathbb{N}, \cdots, \theta_k{:}\mathbb{N}^k \Rightarrow \mathbb{N}).(\theta'_1, \cdots, \theta'_k),$$

where $\theta'_1 \stackrel{\text{def}}{=} \lambda x{:}\mathbb{N}.(\theta_1(x) + \theta_2(y,x))$, $\ldots$, $\theta'_{k-1} \stackrel{\text{def}}{=} \lambda x{:}\mathbb{N}^{k-1}.(\theta_{k-1}(x) + \theta_k(y,x))$, *and* $\theta'_k \stackrel{\text{def}}{=} \theta_k$.

Then (2.2) *gives a closed-form solution to the inductive definition above.*

Generic annotations

In general, the potential annotation θ does not need to have the form of vectors of refinement-level functions; it can be an arbitrary well-sorted refinement, as long as we know how to *extract* potentials from it (*e.g.*, a projection from $\theta = (\theta_1, \cdots, \theta_k)$ to θ_1), and how to *shift* potential annotations to get annotations for child nodes (*e.g.*, Theorem 6). This form of generic annotations formulates the notion of *abstract potentials* (introduced in Section 2.2.2), which is one major contribution of this chapter.

In our type system, we parametrize inductive datatypes with not only a potential annotation θ, but also a shift operator $\lhd$ and an extraction operator π. For natural-number lists of type $\mathsf{NatList}^{\theta}$, the potential function $\Phi(\cdot)$ is defined inductively in terms of $\lhd$ and π as follows:

$$\begin{aligned}\Phi(\mathsf{Nil}(\mathsf{triv}, \langle\rangle) : \mathsf{NatList}^{\theta}) &\stackrel{\text{def}}{=} 0,\\ \Phi(\mathsf{Cons}(v_h, \langle v_t\rangle) : \mathsf{NatList}^{\theta}) &\stackrel{\text{def}}{=} \pi(v_h)(\theta) + \Phi(v_t : \mathsf{NatList}^{\lhd(v_h)(\theta)}).\end{aligned}$$

Recall that in our type system, an inductive datatype is represented as $\mathsf{ind}^{\theta}_{\lhd,\pi}(\overrightarrow{C : (T, m)})$, where C's are constructor names, T's are content types of data stored at constructors, and m's are numbers of child nodes of constructors. Let the potential annotation θ be sorted Δ_θ, and values of content type T_j be sorted as Δ_{T_j} for each constructor $C_j : (T_j, m_j)$. Then the extraction operator π is supposed to be a tuple, the j-th component of which is a refinement-level function with sort $\Delta_{T_j} \Rightarrow \Delta_\theta \Rightarrow \mathbb{N}$, *i.e.*, extracts potential for the j-th constructor from the annotation θ. Similarly, the shift operator $\lhd$ is also a tuple whose j-th component is a refinement-level function with sort $\Delta_{T_j} \Rightarrow \Delta_\theta \Rightarrow \Delta_\theta^{m_j}$, *i.e.*, shifts potential annotations for the child nodes of the j-th constructor. With the two operators $\lhd, \pi$ and the potential annotation θ, we can now define the potential

function $\Phi(\cdot)$ for general inductive datatypes as an inductive function:

$$\begin{aligned}\Phi(C_j(v_0,\langle v_1,\cdots,v_{m_j}\rangle):\mathsf{ind}^{\theta}_{\lhd,\pi}(\overrightarrow{C:(T,m)})) \stackrel{\text{def}}{=} \Phi(v_0:T_j)& \\ +\pi.\mathbf{j}(\mathscr{I}(v_0))(\theta)& \\ +\sum_{i=1}^{m_j}\Phi(v_i:\mathsf{ind}^{\lhd.\mathbf{j}(\mathscr{I}(v_0))(\theta).\mathbf{i}}_{\lhd,\pi}(\overrightarrow{C:(T,m)})).&\end{aligned} \tag{2.3}$$

Note that (i) the definition above includes the potential of the value v_0 stored at the constructor with respect to its type T_j, because the elements in the data structure may also carry potentials, and (ii) we use the interpretation $\mathscr{I}(\cdot)$ defined in Section 2.3.2 to interpret values as their logical refinements.

Example 3 (Generic potential annotations)**.** Recall the list type $\mathsf{NatList}^{(\theta_1,\theta_2)}$ decorated with dependent potentials from 2. We can now formalize it in the core type system. Let

$$\mathsf{NatList}^{(\theta_1,\theta_2)} \stackrel{\text{def}}{=} \mathsf{ind}^{(\theta_1,\theta_2)}_{\lhd,\pi}(\mathsf{Nil}:(\mathsf{unit},0),\mathsf{Cons}:(\mathsf{nat},1)),$$

where $\lhd=(\lhd_{\mathsf{Nil}},\lhd_{\mathsf{Cons}})$ and $\pi=(\pi_{\mathsf{Nil}},\pi_{\mathsf{Cons}})$ are defined as follows:

$$\begin{aligned}\pi_{\mathsf{Nil}} &\stackrel{\text{def}}{=} \lambda\_{:}\mathbb{U}.\lambda(\theta_1:\mathbb{N}\Rightarrow\mathbb{N},\theta_2:\mathbb{N}\times\mathbb{N}\Rightarrow\mathbb{N}).0,\\ \pi_{\mathsf{Cons}} &\stackrel{\text{def}}{=} \lambda y{:}\mathbb{N}.\lambda(\theta_1:\mathbb{N}\Rightarrow\mathbb{N},\theta_2:\mathbb{N}\times\mathbb{N}\Rightarrow\mathbb{N}).\theta_1(y),\\ \lhd_{\mathsf{Nil}} &\stackrel{\text{def}}{=} \lambda\_{:}\mathbb{U}.\lambda(\theta_1:\mathbb{N}\Rightarrow\mathbb{N},\theta_2;\mathbb{N}\times\mathbb{N}\Rightarrow\mathbb{N}).\star,\\ \lhd_{\mathsf{Cons}} &\stackrel{\text{def}}{=} \lambda y{:}\mathbb{N}.\lambda(\theta_1:\mathbb{N}\Rightarrow\mathbb{N},\theta_2:\mathbb{N}\times\mathbb{N}\Rightarrow\mathbb{N}).(\lambda x{:}\mathbb{N}.\theta_1(x)+\theta_2(y,x),\theta_2).\end{aligned}$$

Different instantiations of θ_1,θ_2 lead to different potential functions. 2 presents an instantiation to count the out-of-order pairs in a natural-number list. Meanwhile, one can implement the simple numeric annotations (q_1,q_2) by setting $\theta_1 \stackrel{\text{def}}{=} \lambda x{:}\mathbb{N}.q_1$ and $\theta_2 \stackrel{\text{def}}{=} \lambda x{:}\mathbb{N}\times\mathbb{N}.q_2$ as constant functions.

2.3.4 Typing Rules

In this section, we formulate our type system as a set of derivation rules. The *typing context* Γ is a sequence of bindings for program variables x, bindings for refinement variables a, type variables α, path constraints ψ, and free potentials ϕ:

$$\Gamma ::= \cdot \mid \Gamma, x : S \mid \Gamma, a : \Delta \mid \Gamma, \alpha \mid \Gamma, \psi \mid \Gamma, \phi.$$

Our type system consists of five kinds of judgments: sorting, well-formedness, subtyping, sharing, and typing. We omit sorting and well-formedness rules and include them in the technical report [85] The sorting judgment $\Gamma \vdash \psi \in \Delta$ states that a term ψ has a sort Δ under the context Γ in the refinement language. A type S is said to be well-defined under a context Γ, denoted by $\Gamma \vdash S$ type, if every referenced variable in S is in the proper scope.

Typing with refinements

Figure 2.10 presents the typing rules of the core type system. The typing judgment $\Gamma \vdash e :: S$ states that the expression e has type S under context Γ. Its intuitive meaning is that if all path constraints in Γ are satisfied, and there is *at least* the amount resources as indicated by the potential in Γ then this suffices to evaluate e to a value v that satisfies logical constraints indicated by S, and after the evaluation there are *at least* as many resources available as indicated by the potential in S. The rules can be organized into syntax-directed and structural rules. Structural rules (S-*) can be applied to every expression; in the implementation, we apply these rules strategically to avoid redundant proof search.

The auxiliary *atomic-typing* judgment $\Gamma \vdash a : B$, defined in Figure 2.9, assigns base types to interpretable atoms $a \in \mathsf{SimpAtom}$. Atomic typing is useful in the rule (T-SIMPATOM), which uses the interpretation $\mathscr{I}(\cdot)$ to derive a most precise refinement type for interpretable atoms, *e.g.*, true is typed $\{\mathsf{bool} \mid \nu = \top\}$, $\overline{5}$ is typed $\{\mathsf{nat} \mid \nu = 5\}$, and a singleton list $\mathsf{Cons}(\overline{5}, \langle \mathsf{Nil}(\mathsf{triv}, \langle\rangle)\rangle)$ is typed $\{\mathsf{NatList}^{\theta} \mid \nu = 1\}$ with some appropriate θ (recall that $\mathsf{NatList}$ admits a length inter-

pretation).

The *subtyping* judgment $\Gamma \vdash T_1 <: T_2$ is defined via a common approach for refinement types, with the extra requirement that the potential in T_1 should be not less than that in T_2. Figure 2.11 shows the subtyping rules. A canonical use of subtyping is to "forget" locally introduced program variables in the result type of an expression, *e.g.*, to "forget" x in the type of e_2 when typing $\mathsf{let}(e_1, x.e_2)$. In rule (SUB-DTYPE), we introduce a partial order $\sqsubseteq_{\Delta_\theta}$ over potential annotations θ of sort Δ_θ. For example, if θ_1 and θ_2 are sorted $\mathbb{N}$, then $\theta_1 \sqsubseteq_{\mathbb{N}} \theta_2$ is encoded as $\theta_1 \leq \theta_2$ in the refinement language. We carefully define the partial order, in a way that the partial-order relation can be encoded as a first-order fragment of the refinement language. Notable is that we introduce *validity-checking* judgments $\Gamma \models \psi$ to reason about logical constraints, *i.e.*, to state that the Boolean-sorted refinement ψ is always true under any instance of the context Γ. We formalize the validity-checking relation via a set-based denotational semantics for the refinement language. Validity checking is then reduced to Presburger arithmetic, making it decidable. The full development of validity checking is included in the technical report [85]

The rule (T-MATD) reasons about *invariants* for inductive datatypes. These invariants come from the associated interpretation of inductive data structures, *e.g.*, the length of a list $\mathsf{Cons}(a_h, \langle a_t \rangle)$ is one plus the length of its tail a_t. Intuitively, if the data structure a_0 can be deconstructed as $C_j(x_0, \langle x_1, \cdots, x_{m_j} \rangle)$ of a datatype D with the form $\mathsf{ind}^{\theta}_{\lhd,\pi}(\overrightarrow{C : (T, m)})$, then by the definition of the interpretation $\mathscr{I}(\cdot)$, we can derive

$$\mathscr{I}(a_0) = \mathscr{I}(C_j(x_0, \langle x_1, \cdots, x_{m_j} \rangle)) = \mathscr{I}_D(C_j(x_0, \langle x_1, \cdots, x_{m_j} \rangle)),$$

which is exactly the path constraint required by the rule (T-MATD) to type the j-th branch e_j. For example, if a_0 has type $\mathsf{NatList}^{\theta}$, then the path constraints for the $\mathsf{Nil}(\_, \langle\rangle)$ and $\mathsf{Cons}(x_h, \langle x_t \rangle)$ constructors become $\mathscr{I}(a_0) = 0$ and $\mathscr{I}(a_0) = x_t + 1$, respectively.

The type system has two rules for applications: (T-APP) and (T-APP-SIMPATOM). In

$$\frac{\Gamma(x) = \{B \mid \psi\}^{\phi}}{\Gamma \vdash x : B}\ (\textsc{SimpAtom-Var}) \qquad \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash b : \mathsf{bool}}\ (\textsc{SimpAtom-Bool}) \qquad \frac{}{\Gamma \vdash \overline{n} : \mathsf{nat}}\ (\textsc{SimpAtom-Nat}) \qquad \frac{}{\Gamma \vdash \mathsf{triv} : \mathsf{unit}}\ (\textsc{SimpAtom-Unit})$$

$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash a_1 : B_1 \qquad \Gamma_2 \vdash a_2 : B_2}{\Gamma \vdash \mathsf{pair}(a_1, a_2) : B_1 \times B_2}\ (\textsc{SimpAtom-Pair})$$

$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash a_0 :: T_j \qquad \Gamma_2 \vdash \langle a_1, \cdots, a_{m_j} \rangle : \prod_{i=1}^{m_j} \mathsf{ind}_{\lhd,\pi}^{\lhd.\mathbf{j}(\mathscr{I}(a_0))(\theta).\mathbf{i}}(\overrightarrow{C : (T, m)})}{\Gamma, \pi.\mathbf{j}(\mathscr{I}(a_0))(\theta) \vdash C_j(a_0, \langle a_1, \cdots, a_{m_j} \rangle) : \mathsf{ind}_{\lhd,\pi}^{\theta}(\overrightarrow{C : (T, m)})}\ (\textsc{SimpAtom-ConsD})$$

Figure 2.9. Base typing rules: $\Gamma \vdash a : B$

the former case, the function return type T does not mention x, and thus can be directly used as the type of the application. This rule deals with cases *e.g.* for all applications with higher-order arguments, since our sorting rules prevent functions from showing up in the refinements language. In the latter case, the function return type T mentions x, but the argument has a scalar type, and thus must be an interpretable atom $a \in \mathsf{SimpAtom}$, so we can substitute x in T with its interpretation $\mathscr{I}(a)$. Note that it is the use of a-normal-form that brings us the ability to derive precise types for dependent function applications.

Resources

There are two typing rules for the syntactic form $\mathsf{tick}(c, e_0)$, one for nonnegative costs and the other for negative costs. The rule (T-TICK-N) assumes $c < 0$ and adds $-c$ units of free potential to the context for typing e_0. The rule (T-TICK-P) behaves differently; it states that $\mathsf{tick}(c, e_0)$ is only typable in a context containing a free-potential term c. Nevertheless, we can use the rule (S-TRANSFER) to rearrange free potentials within the context into this form, as long as the total amount of free potential stays unchanged. In the rule (S-TRANSFER), $\Phi(\Gamma)$ extracts all the free potentials in the context Γ, while $|\Gamma|$ removes all the free potentials, *i.e.*, $|\Gamma|$ keeps the functional specifications of Γ.

To carry out amortized resource analysis [130], our type system is supposed to properly reason about potentials, that is, potentials cannot be generated from nothing. This *linear* nature of potentials motivates us to develop an *affine* type system [137]. As in Re^2 [84], we have to

$$\text{(T-SimpAtom)}\ \frac{\Gamma \vdash a : B}{\Gamma \vdash a :: \{B \mid \nu = \mathscr{I}(a)\}} \qquad \text{(T-Var)}\ \frac{\Gamma(x) = S}{\Gamma \vdash x :: S} \qquad \text{(T-Imp)}\ \frac{\Gamma \models \bot \qquad \Gamma \vdash T\ \mathsf{type}}{\Gamma \vdash \mathsf{impossible} :: T} \qquad \text{(T-Tick-P)}\ \frac{c \geq 0 \qquad \Gamma \vdash e_0 :: T}{\Gamma, c \vdash \mathsf{tick}(c, e_0) :: T}$$

$$\text{(T-Tick-N)}\ \frac{c < 0 \qquad \Gamma, -c \vdash e_0 :: T}{\Gamma \vdash \mathsf{tick}(c, e_0) :: T} \qquad \text{(T-Cond)}\ \frac{\Gamma \vdash a_0 : \mathsf{bool} \qquad \Gamma, \mathscr{I}(a_0) \vdash e_1 :: T \qquad \Gamma, \neg\mathscr{I}(a_0) \vdash e_2 :: T}{\Gamma \vdash \mathsf{if}(a_0, e_1, e_2) :: T}$$

$$\text{(T-MatP)}\ \frac{\vdash \Gamma \between \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash a_0 : B_1 \times B_2 \qquad \Gamma \vdash T\ \mathsf{type} \qquad \Gamma_2, x_1 : B_1, x_2 : B_2, \mathscr{I}(a_0) = (x_1, x_2) \vdash e_1 :: T}{\Gamma \vdash \mathsf{matp}(a_0, x_1.x_2.e_1) :: T}$$

$$\text{(T-MatD)}\ \frac{\begin{array}{c}\vdash \Gamma \between \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash a_0 : \mathsf{ind}^{\theta}_{\lhd,\pi}(\overrightarrow{C : (T, m)}) \qquad \Gamma \vdash T'\ \mathsf{type} \\ \text{for each } j,\ \Gamma_{2,j} \stackrel{\text{def}}{=} \Big(\Gamma_2, x_0 : T_j, \\ x_1 : \mathsf{ind}^{\lhd, \mathbf{j}(x_0)(\theta).\mathbf{1}}_{\lhd,\pi}(\overrightarrow{C : (T, m)}), \ldots, x_{m_j} : \mathsf{ind}^{\lhd, \mathbf{j}(x_0)(\theta).\mathbf{m_j}}_{\lhd,\pi}(\overrightarrow{C : (T, m)}), \mathscr{I}(a_0) = \mathscr{I}_D(C_j(x_0, \langle x_1, \ldots, x_{m_j}\rangle)), \boldsymbol{\pi}.\mathbf{j}(x_0)(\theta)\Big), \\ \Gamma_{2,j} \vdash e_j :: T'\end{array}}{\Gamma \vdash \mathsf{matd}(a_0, \overrightarrow{C_j(x_0, \langle x_1, \cdots, x_{m_j}\rangle).e_j}) :: T'}$$

$$\text{(T-Let)}\ \frac{\vdash \Gamma \between \Gamma_1 \mid \Gamma_2 \qquad \Gamma \vdash T_2\ \mathsf{type} \qquad \Gamma_1 \vdash e_1 :: S_1 \qquad \Gamma_2, x : S_1 \vdash e_2 :: T_2}{\Gamma \vdash \mathsf{let}(e_1, x.e_2) :: T_2}$$

$$\text{(T-App-SimpAtom)}\ \frac{\vdash \Gamma \between \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash \hat{a}_1 :: 1 \cdot (x : \{B \mid \psi\}^{\phi} \to T) \qquad \Gamma_2 \vdash a_2 :: \{B \mid \psi\}^{\phi}}{\Gamma \vdash \mathsf{app}(\hat{a}_1, a_2) :: [\mathscr{I}(a_2)/x]T}$$

$$\text{(T-App)}\ \frac{\vdash \Gamma \between \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash \hat{a}_1 :: 1 \cdot (x : T_x \to T) \qquad \Gamma_2 \vdash \hat{a}_2 :: T_x \qquad \Gamma \vdash T\ \mathsf{type}}{\Gamma \vdash \mathsf{app}(\hat{a}_1, \hat{a}_2) :: T}$$

$$\text{(T-Abs)}\ \frac{\Gamma \vdash T_x\ \mathsf{type} \qquad \Gamma, x : T_x \vdash e_0 :: T \qquad \vdash \Gamma \between \Gamma \mid \Gamma}{\Gamma \vdash \lambda(x.e_0) :: x : T_x \to T} \qquad \text{(T-Abs-Lin)}\ \frac{\Gamma \vdash T_x\ \mathsf{type} \qquad \Gamma, x : T_x \vdash e_0 :: T}{m \times \Gamma \vdash \lambda(x.e_0) :: m \cdot (x : T_x \to T)}$$

$$\text{(T-Fix)}\ \frac{S = \forall \overrightarrow{\alpha}.x : T_x \to T \qquad \Gamma \vdash S\ \mathsf{type} \qquad \Gamma, f : S, \overrightarrow{\alpha}, x : T_x \vdash e_0 :: T \qquad \vdash \Gamma \between \Gamma \mid \Gamma}{\Gamma \vdash \mathsf{fix}(f.x.e_0) :: S} \qquad \text{(S-Gen)}\ \frac{v \in \mathsf{Val} \qquad \Gamma, \alpha \vdash v :: S \qquad \Gamma, \alpha \vdash S \between S \mid S}{\Gamma \vdash v :: \forall \alpha.S}$$

$$\text{(S-Inst)}\ \frac{\Gamma \vdash e :: \forall \alpha.S \qquad \Gamma \vdash \{B \mid \psi\}^{\phi}\ \mathsf{type}}{\Gamma \vdash e :: [\{B \mid \psi\}^{\phi}/\alpha]S} \qquad \text{(S-Subtype)}\ \frac{\Gamma \vdash e :: T_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash e :: T_2} \qquad \text{(S-Transfer)}\ \frac{\Gamma' \vdash e :: S \qquad \Gamma \models \Phi(\Gamma) = \Phi(\Gamma') \qquad |\Gamma| = |\Gamma'|}{\Gamma \vdash e :: S}$$

$$\text{(S-Relax)}\ \frac{\Gamma \vdash e :: R^{\phi} \qquad \Gamma \vdash \phi' \in \mathbb{N}}{\Gamma, \phi' \vdash e :: R^{\phi + \phi'}}$$

Figure 2.10. Typing rules: $\Gamma \vdash e :: S$

introduce explicit *sharing* to use a program variable multiple times. The sharing judgment takes the form $\Gamma \vdash S \curlyvee S_1 \mid S_2$ and is intended to state that under the context Γ, the potential associated with type S is apportioned into two parts to be associated with type S_1 and type S_2. Figure 2.11 also presents the sharing rules. In rule (SHARE-DTYPE), we introduce a notation $\theta = \theta_1 \oplus_{\Delta_\theta} \theta_2$, which means that the annotation θ is the "sum" of two annotations θ_1, θ_2 that have sort Δ_θ. For example, we define $\theta_1 \oplus_{\mathbb{N}} \theta_2$ by $\theta_1 + \theta_2$ in the refinement language. Similar to the partial order $\sqsubseteq_{\Delta_\theta}$, which is used in the subtyping rules, we encode the "sum" operator $\oplus_{\Delta_\theta}$ using a first-order fragment of the refinement language. The sharing relation is further extended to *context sharing*, written $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, which means that Γ_1 and Γ_2 have the same sequence of bindings as Γ, but the free potentials in Γ are split into two parts to be associated with Γ_1 and Γ_2. Context sharing is used extensively in the typing rules where the expression has at least two sub-expressions to evaluate, *e.g.*, in the rule (T-LET) for an expression $\mathsf{let}(e_1, x.e_2)$, we apprortion Γ into Γ_1 and Γ_2, use Γ_1 for typing e_1 and Γ_2 for typing e_2. Note that the rule (T-ABS) and (T-FIX) has self-sharing $\vdash \Gamma \curlyvee \Gamma \mid \Gamma$ as a premise, which means that the function can only use free variables with zero potential in the context. This restriction ensures that the program cannot gain potential through free variables by repeatedly applying a function of type $\infty \cdot (x : T_x \to T)$ with an infinite multiplicity.

The rule (T-ABS-LIN) is introduced for typing functions with upper bounds on the number of applications. The rule associates a multiplicity $m \in \mathbb{Z}_0^+$ with the function type as the upper bound. We use a finer-grained premise than context self-sharing to state that the potential of the free variables in the function is enough to pay for m function applications. This rule is useful for deriving types of curried functions *e.g.* a function of type $x : T_x \to y : T_y \to T$ that require nonzero units of potential in its first argument x. In that case, a function f can be assigned a type $x : T_x \to m \cdot (y : T_y \to T)$, which means that the potential stored in the first argument x is enough for the partially applied function $\mathsf{app}(f, x)$ to be invoked for m times.

The elimination rule (T-MATD) realizes the inductively defined potential function in (2.3): for typing the j-th branch e_j, one has to add bindings of the content type $x_0 : T_j$ and

properly shifted types for child nodes $x_i : \mathsf{ind}_{\lhd,\pi}^{\lhd.\mathbf{j}(x_0)(\theta).\mathbf{i}}(\overrightarrow{C:(T,m)})$, as well as a free-potential term $\pi.\mathbf{j}(x_0)(\theta)$ indicated by the potential-extraction operator $\pi.\mathbf{j}$, to the context. The introduction rule (SIMPATOM-CONSD) stores the amount of potentials required for deconstructing data structures. For typing $C_j(a_0, \langle a_1, \cdots, a_{m_j}\rangle)$ with type $\mathsf{ind}_{\lhd,\pi}^{\theta}(\overrightarrow{C:(T,m)})$, the rule requires $\pi.\mathbf{j}(\mathscr{I}(a_0))(\theta)$ as free potential in the context, which is used to pay for potential extraction $\pi.\mathbf{j}$, and a premise stating that each child node a_i has a corresponding properly-shifted annotated datatype $\mathsf{ind}_{\lhd,\pi}^{\lhd.\mathbf{j}(\mathscr{I}(a_0))(\theta).\mathbf{i}}(\overrightarrow{C:(T,m)})$.

Finally, the structural rule (S-RELAX) is usually used when we are analyzing function applications. Both the rule (T-APP) and the rule (T-APP-SIMPATOM) use up all the potential in the context, but in practice it is necessary to pass some potential through the function call to analyze non-tail-recursive programs. This is achieved by using the rule (S-RELAX) at a function application with ϕ' as the potential threaded to the computation that continues after the function returns.

Example 4 (Insertion sort)**.** As shown in Section 2.2.2, our type system is able to verify that an implementation of insertion sort performs exactly the same amount of insertions as the number of out-of-order pairs in the input list. We rewrite the function `insert` as follows in the core calculus, using the dependently annotated list type $\mathsf{NatList}^{(\theta_1,\theta_2)}$ from 2:

$$\texttt{insert} :: y{:}\mathsf{nat} \to \ell{:}\mathsf{NatList}^{(\lambda x{:}\mathbb{N}.\mathbf{ite}(y>x,1,0),\lambda x{:}\mathbb{N}\times\mathbb{N}.0)} \to \mathsf{NatList}^{(\lambda x{:}\mathbb{N}.0,\lambda x{:}\mathbb{N}\times\mathbb{N}.0)}$$

$$\begin{aligned}
\texttt{insert} = \lambda(&y.\mathsf{fix}(f.\ell.\mathsf{matd}(\ell,\\
&\quad \mathsf{Nil}(_,\langle\rangle).\mathsf{Cons}(y,\langle\mathsf{Nil}(\mathsf{triv},\langle\rangle)\rangle),\\
&\quad \mathsf{Cons}(h,\langle t\rangle).\mathsf{let}(y>h,b.\\
&\qquad \mathsf{if}(b,\mathsf{tick}(1,\mathsf{let}(\mathsf{app}(f,t),t'.\mathsf{Cons}(h,\langle t'\rangle))),\mathsf{Cons}(y,\langle\mathsf{Cons}(h,\langle t\rangle)\rangle))))
\end{aligned}$$

We assume that a comparison function $>$ with signature $a{:}\mathsf{nat} \to b{:}\mathsf{nat} \to \{\mathsf{bool} \mid \nu = (a > b)\}$ is provided in the typing context. Next, we illustrate how our type system justifies the number

$\boxed{\Gamma \vdash S \curlyvee S_1 \mid S_2}$

$$\frac{}{\Gamma \vdash \mathsf{nat} \curlyvee \mathsf{nat} \mid \mathsf{nat}}\ (\textsc{Share-Nat}) \qquad \frac{}{\Gamma \vdash \mathsf{bool} \curlyvee \mathsf{bool} \mid \mathsf{bool}}\ (\textsc{Share-Bool}) \qquad \frac{}{\Gamma \vdash \mathsf{unit} \curlyvee \mathsf{unit} \mid \mathsf{unit}}\ (\textsc{Share-Unit})$$

$$\frac{\Gamma, \alpha \vdash S \curlyvee S \mid S}{\Gamma \vdash \forall \alpha.S \curlyvee \forall \alpha.S \mid \forall \alpha.S}\ (\textsc{Share-Poly})$$

$$\frac{\Gamma \vdash B_1 \curlyvee B_{11} \mid B_{12} \qquad \Gamma \vdash B_2 \curlyvee B_{21} \mid B_{22}}{\Gamma \vdash B_1 \times B_2 \curlyvee B_{11} \times B_{21} \mid B_{12} \times B_{22}}\ (\textsc{Share-Prod})$$

$$\frac{\Gamma \vdash \overrightarrow{T} \curlyvee \overrightarrow{T_1} \mid \overrightarrow{T_2} \qquad \Gamma \vdash \theta, \theta_1, \theta_2 \in \Delta_\theta \qquad \Gamma \models \theta = \theta_1 \oplus_{\Delta_\theta} \theta_2}{\Gamma \vdash \mathsf{ind}^{\theta}_{\lhd,\pi}(\overrightarrow{C : (T, m)}) \curlyvee \mathsf{ind}^{\theta_1}_{\lhd,\pi}(\overrightarrow{C : (T_1, m)}) \mid \mathsf{ind}^{\theta_2}_{\lhd,\pi}(\overrightarrow{C : (T_2, m)})}\ (\textsc{Share-Dtype})$$

$$\frac{\alpha \in \Gamma \qquad m = m_1 + m_2}{\Gamma \vdash m \cdot \alpha \curlyvee m_1 \cdot \alpha \mid m_2 \cdot \alpha}\ (\textsc{Share-Tvar})$$

$$\frac{\Gamma \vdash B \curlyvee B_1 \mid B_2 \qquad \Gamma \vdash \{B \mid \psi\}\ \mathsf{type}}{\Gamma \vdash \{B \mid \psi\} \curlyvee \{B_1 \mid \psi\} \mid \{B_2 \mid \psi\}}\ (\textsc{Share-Subset})$$

$$\frac{\Gamma \vdash (x : T_x \to T)\ \mathsf{type} \qquad m = m_1 + m_2}{\Gamma \vdash (m \cdot (x : T_x \to T)) \curlyvee (m_1 \cdot (x : T_x \to T)) \mid (m_2 \cdot (x : T_x \to T))}\ (\textsc{Share-Arrow})$$

$$\frac{\Gamma \vdash R \curlyvee R_1 \mid R_2 \qquad \Gamma, \nu : R \models \phi = \phi_1 + \phi_2}{\Gamma \vdash R^{\phi} \curlyvee {R_1}^{\phi_1} \mid {R_2}^{\phi_2}}\ (\textsc{Share-Pot})$$

$\boxed{\Gamma \vdash T_1 <: T_2}$

$$\frac{}{\Gamma \vdash \mathsf{nat} <: \mathsf{nat}}\ (\textsc{Sub-Nat}) \qquad \frac{}{\Gamma \vdash \mathsf{unit} <: \mathsf{unit}}\ (\textsc{Sub-Unit}) \qquad \frac{}{\Gamma \vdash \mathsf{bool} <: \mathsf{bool}}\ (\textsc{Sub-Bool})$$

$$\frac{\Gamma \vdash B_1 <: B_1' \qquad \Gamma \vdash B_2 <: B_2'}{\Gamma \vdash B_1 \times B_2 <: B_1' \times B_2'}\ (\textsc{Sub-Prod})$$

$$\frac{\Gamma \vdash \overrightarrow{T} <: \overrightarrow{T'} \qquad \Gamma \vdash \theta, \theta' \in \Delta_\theta \qquad \Gamma \models \theta' \sqsubseteq_{\Delta_\theta} \theta}{\Gamma \vdash \mathsf{ind}^{\theta}_{\lhd,\pi}(\overrightarrow{C : (T, m)}) <: \mathsf{ind}^{\theta'}_{\lhd,\pi}(\overrightarrow{C : (T', m)})}\ (\textsc{Sub-Dtype})$$

$$\frac{\alpha \in \Gamma \qquad m_1 \geq m_2}{\Gamma \vdash m_1 \cdot \alpha <: m_2 \cdot \alpha}\ (\textsc{Sub-Tvar})$$

$$\frac{\Gamma \vdash B_1 <: B_2 \qquad \Gamma, \nu : B_1 \models \psi_1 \implies \psi_2}{\Gamma \vdash \{B_1 \mid \psi_1\} <: \{B_2 \mid \psi_2\}}\ (\textsc{Sub-Subset})$$

$$\frac{\Gamma \vdash T_x' <: T_x \qquad \Gamma, x : T_x' \vdash T <: T' \qquad m \geq m'}{\Gamma \vdash m \cdot (x : T_x \to T) <: m' \cdot (x : T_x' \to T')}\ (\textsc{Sub-Arrow})$$

$$\frac{\Gamma \vdash R_1 <: R_2 \qquad \Gamma, \nu : R_1 \models \phi_1 \geq \phi_2}{\Gamma \vdash {R_1}^{\phi_1} <: {R_2}^{\phi_2}}\ (\textsc{Sub-Pot})$$

Figure 2.11. Sharing and subtyping

of recursive calls in `insert` is bounded by the number of elements in ℓ that are less than the element y that is being inserted to ℓ. Suppose Γ is a typing context that contains the signature of $>$, as well as type bindings for y, f, and ℓ. To reason about the pattern match on the list ℓ, we apply the (T-MATD) rule, where $T \stackrel{\text{def}}{=} \mathsf{NatList}^{(\lambda x:\mathbb{N}.0,\lambda x:\mathbb{N}\times\mathbb{N}.0)}$:

$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash \ell : \mathsf{NatList}^{(\lambda x:\mathbb{N}.\mathbf{ite}(y>x,1,0),\lambda x:\mathbb{N}\times\mathbb{N}.0)} \qquad \Gamma_2, \ell = 0 \vdash e_1 :: T \qquad \Gamma_2, h : \mathsf{nat}, t : \mathsf{NatList}^{(\lambda x:\mathbb{N}.\mathbf{ite}(y>x,1,0),\lambda x:\mathbb{N}\times\mathbb{N}.0)}, \ell = t+1, \mathbf{ite}(y>h,1,0) \vdash e_2 :: T}{\Gamma \vdash \mathsf{matd}(\ell, \mathsf{Nil}(\_, \langle\rangle).e_1, \mathsf{Cons}(h, \langle t \rangle).e_2) :: T}$$

For the context sharing, we apportion all the potential of ℓ to Γ_1 and the rest of potential of Γ to Γ_2. In fact, since y and f do not carry potentials, the context Γ_2 is potential-free *i.e.* $\vdash \Gamma_2 \curlyvee \Gamma_2 \mid \Gamma_2$. For the Nil-branch, e_1 is a value that describes a singleton list containing y, thus we can easily conclude this case by rule (SIMPATOM-CONSD) and the fact that the return type T is potential-free. For the Cons-branch, we first apply the (T-LET) rule with (T-APP-SIMPATOM) rule to derive a precise refinement type for the comparison result b:

$$\frac{\vdash \Gamma_2 \curlyvee \Gamma_2 \mid \Gamma_2 \qquad \Gamma_2, h : \cdots, t : \cdots, \ell = t+1, 0 \vdash y > h :: \{\mathsf{bool} \mid \nu = (y > h)\} \qquad \Gamma_2, h : \cdots, t : \cdots, \ell = t+1, \mathbf{ite}(y>h,1,0), b : \{\mathsf{bool} \mid \nu = (y>h)\} \vdash e_3 :: T}{\Gamma_2, h : \cdots, t : \cdots, \ell = t+1, \mathbf{ite}(y>h,1,0) \vdash \mathsf{let}(y > h, b.e_3) :: T}$$

Then we use the rule (T-COND) to reason about the conditional expression e_3:

$$\frac{\Gamma_2, h : \cdots, t : \cdots, \ell = t+1, \mathbf{ite}(y>h,1,0), b : \{\mathsf{bool} \mid \nu = (y>h)\}, b \vdash e_4 :: T \qquad \Gamma_2, h : \cdots, t : \cdots, \ell = t+1, \mathbf{ite}(y>h,1,0), b : \{\mathsf{bool} \mid \nu = (y>h)\}, \neg b \vdash e_5 :: T}{\Gamma_2, h : \cdots, t : \cdots, \ell = t+1, \mathbf{ite}(y>h,1,0), b : \{\mathsf{bool} \mid \nu = (y>h)\} \vdash \mathsf{if}(b, e_4, e_5) :: T}$$

By validity checking, we can show that $y : \mathsf{nat}, h : \mathsf{nat}, b : \{\mathsf{bool} \mid \nu = (y > h)\}, b \models y > h$, thus $y : \mathsf{nat}, h : \mathsf{nat}, b : \{\mathsf{bool} \mid \nu = (y > h)\}, b \models \mathbf{ite}(y > h, 1, 0) = 1$. Then, by the (S-TRANSFER) rule on the goal involving the then-branch e_4, it suffices to show that $\Gamma_2, h : \cdots, t : \cdots, \ell = t+1, b : \{\mathsf{bool} \mid \nu = (y > h)\}, b, 1 \vdash e_4 :: T$. Note that we now have one unit of free potential in

the context, so we can use it for typing the tick expression by (T-TICK-P):

$$\frac{\Gamma_2, h:\cdots, t:\cdots, \ell = t+1, b:\{\mathsf{bool} \mid \nu = (y > h)\}, b \vdash \mathsf{let}(\mathsf{app}(f,t), t'.\mathsf{Cons}(h,\langle t'\rangle)) :: T}{\Gamma_2, h:\cdots, t:\cdots, \ell = t+1, b:\{\mathsf{bool} \mid \nu = (y > h)\}, b, 1 \vdash \mathsf{tick}(1, \mathsf{let}(\mathsf{app}(f,t), t'.\mathsf{Cons}(h,\langle t'\rangle))) :: T}$$

It remains to derive the type of the recursive function application $\mathsf{app}(f,t)$, and the list construction $\mathsf{Cons}(h,\langle t'\rangle)$ where t' is the return of the application. The derivation is straightforward as f has type $\ell : \mathsf{NatList}^{(\lambda x:\mathbb{N}.\mathbf{ite}(y>x,1,0), \lambda x:\mathbb{N}\times\mathbb{N}.0)} \to T$, t has type $\mathsf{NatList}^{(\lambda x:\mathbb{N}.\mathbf{ite}(y>x,1,0), \lambda x:\mathbb{N}\times\mathbb{N}.0)}$, thus the returned list t' has type T and so does $\mathsf{Cons}(h,\langle t'\rangle)$.

We now turn to the function `sort` that makes use of `insert`:

$$\begin{aligned}
&\texttt{sort} :: \ell : \mathsf{NatList}^{(\lambda x:\mathbb{N}.1, \lambda(x_1:\mathbb{N}, x_2:\mathbb{N}).\mathbf{ite}(x_1>x_2,1,0))} \to \mathsf{NatList}^{(\lambda x:\mathbb{N}.0, \lambda x:\mathbb{N}\times\mathbb{N}.0)} \\
&\texttt{sort} = \mathsf{fix}(f.\ell.\mathsf{matd}(\ell, \\
&\qquad \mathsf{Nil}(_, \langle\rangle).\mathsf{Nil}(\mathsf{triv}, \langle\rangle), \\
&\qquad \mathsf{Cons}(h,\langle t\rangle).\mathsf{tick}(1, \mathsf{let}(\mathsf{app}(f,t), t'.\mathsf{let}(\mathsf{app}(\texttt{insert}, h), ins.\mathsf{app}(ins, t'))))))
\end{aligned}$$

Recall that in 2, we explain that the type of the argument list ℓ defines a potential function in terms of the number of out-of-order pairs in ℓ. Let Γ' be a typing context that contains the signature of `insert`, as well as potential-free type bindings for f and ℓ. Using the shift operation $\lhd$ for NatList, we are supposed to derive the following judgment for the Cons-branch of the pattern match:

$$\Gamma', h : \mathsf{nat}, t : \mathsf{NatList}^{(\lambda x:\mathbb{N}.1+\mathbf{ite}(h>x,1,0), \lambda(x_1:\mathbb{N}, x_2:\mathbb{N}).\mathbf{ite}(x_1>x_2,1,0))}, \ell = t+1 \vdash \mathsf{let}(\mathsf{app}(f,t), t'.\cdots) :: T.$$

However, we get stuck here, because there is a mismatch between the argument type of f *i.e.* `sort`, and the shifted type of the tail list t in the context.

Polymorphic recursion

In general, it is often necessary to type recursive function calls with a type that has different potential annotations from the declared types of the recursive functions. We achieve this using polymorphic recursion that allows recursive calls to be instantiated with types that have different potential annotations. Although we get stuck when typing `sort` in 4, we will show how our system is able to type a polymorphic version of `sort`, which has been informally demonstrated in Section 2.2.2.

Example 5 (Insertion sort with polymorphic recursion)**.** We start with a polymorphic list type, which is supported by our implementation but not formulated in the core calculus:

$$\mathsf{List}^{\theta}(\alpha) \equiv \mathsf{ind}^{\theta}_{\lhd,\pi}(\mathsf{Nil} : \mathsf{unit}, \mathsf{Cons} : (x{:}\,\alpha) \times \mathsf{List}^{\lhd_{\mathsf{Cons}}(x)(\theta)}(\alpha^{\theta(x,\nu)})),$$

where $\lhd = (\lhd_{\mathsf{Nil}}, \lhd_{\mathsf{Cons}})$, $\pi = (\pi_{\mathsf{Nil}}, \pi_{\mathsf{Cons}})$ are defined as follows:

$$\pi_{\mathsf{Nil}} \stackrel{\text{def}}{=} \lambda\_.\lambda\theta.0, \qquad \pi_{\mathsf{Cons}} \stackrel{\text{def}}{=} \lambda y.\lambda\theta.0,$$
$$\lhd_{\mathsf{Nil}} \stackrel{\text{def}}{=} \lambda\_.\lambda\theta.\star, \qquad \lhd_{\mathsf{Cons}} \stackrel{\text{def}}{=} \lambda y.\lambda\theta.\theta.$$

We then generalize the type signatures of `insert` and `sort` with the polymorphic list type:

$$\texttt{insert} :: \forall\alpha.y{:}\,\alpha \to \ell{:}\,\mathsf{List}^{\lambda(x_1,x_2).0}(\alpha^{\mathbf{ite}(y>\nu,1,0)}) \to \mathsf{List}^{\lambda(x_1,x_2).0}(\alpha), \tag{2.4}$$

$$\texttt{sort} :: \forall\alpha.\ell{:}\,\mathsf{List}^{\lambda(x_1,x_2).\mathbf{ite}(x_1>x_2,1,0)}(\alpha^{1}) \to \mathsf{List}^{\lambda(x_1,x_2).0}(\alpha) \tag{2.5}$$

Similar to the type derivation in 4, we are supposed to derive the following judgment for the Cons-branch of the pattern match in the implementation of `sort`:

$$\Gamma', h : \alpha, t : \mathsf{List}^{\lambda(x_1,x_2).\mathbf{ite}(x_1>x_2,1,0)}(\alpha^{1+\mathbf{ite}(h>\nu,1,0)}), \ell = t+1 \vdash \mathsf{let}(\mathsf{app}(f,t), t'.\cdots) :: T.$$

Now the function f is bound to the polymorphic type in (2.5). To type the function call $\mathsf{app}(f,t)$, we instantiate f with $\alpha^{\mathbf{ite}(h>v,1,0)}$, *i.e.*, f has type $\ell\colon \mathsf{List}^{\lambda(x_1,x_2).\mathbf{ite}(x_1>x_2,1,0)}(\alpha^{1+\mathbf{ite}(h>v,1,0)}) \to \mathsf{List}^{\lambda(x_1,x_2).0}(\alpha^{\mathbf{ite}(h>v,1,0)})$. Thus, the type of the return value t' of $\mathsf{app}(f,t)$ matches the argument type of `insert`, so we can derive the function application $\mathsf{let}(\mathsf{app}(\texttt{insert},h), ins.\mathsf{app}(ins,t'))$ has the desired return type $\mathsf{List}^{\lambda(x_1,x_2).0}(\alpha)$.

2.3.5 Soundness

We now extend Re^2's type soundness [84] to new features we introduced in previous sections, including refinement-level computation and user-defined inductive datatypes. The soundness of the type system is based on progress and preservation, and takes resources into account. The progress theorem states that if $q \vdash e :: S$, then either e is already a value, or we can make a step from e with at least q units of available resource. Intuitively, progress indicates that our type system derives bounds that are indeed upper bounds on the high-water mark of resource usage.

Lemma 3 (Progress)**.** If $q \vdash e :: S$ and $p \geq q$, then either $e \in \mathsf{Val}$ or there exist e' and p' such that $\langle e, p\rangle \mapsto \langle e', p'\rangle$.

Proof. By strengthening the assumption to $\Gamma \vdash e :: S$ where Γ is a sequence of type variables and free potentials, and then induction on $\Gamma \vdash e :: S$. □

The preservation theorem then relates leftover resources after a step in computation and the typing judgment for the new term to reason about resource consumption.

Lemma 4 (Preservation)**.** If $q \vdash e :: S$, $p \geq q$ and $\langle e, p\rangle \mapsto \langle e', p'\rangle$, then $p' \vdash e' :: S$.

Proof. By strengthening the assumption to $\Gamma \vdash e :: S$ where Γ is a sequence of free potentials, and then induction on $\Gamma \vdash e :: S$, followed by inversion on the evaluation judgment $\langle e, p\rangle \mapsto \langle e', p'\rangle$. □

As in other refinement type systems, purely syntactic soundness statement about results of computations (*i.e.*, they are well-typed values) is unsatisfactory. Thus, we also formulate a denotational notation of *consistency*. For example, the literal $b = \mathsf{true}$, but not $b = \mathsf{false}$, is consistent with $0 \vdash b :: \{\mathsf{bool} \mid \nu\}$; A list of values $\ell = [v_1, \cdots, v_n]$ is consistent with $q \vdash \ell :: \mathsf{NatList}^{\lambda x:\mathbb{N}.x}$, if $q \geq \sum_{i=1}^{n} v_i$. We then show that well-typed values are *consistent* with their typing judgement.

Lemma 5 (Consistency)**.** If $q \vdash v :: S$, then v satisfies the conditions indicated by S and q is greater than or equal to the potential stored in v with respect to S.

Proof. By inversion on the typing judgment we have $q \vdash v : B$ for some base type B or v is an abstraction. The latter case is easy as the refinement language cannot mention function values. For the former case, we proceed by strengthening the assumption to $\Gamma \vdash v : B$ where Γ is a sequence of type variables and free potentials, then induction on $\Gamma \vdash v : B$. □

As a result of the lemmas above, we derive the following main technical theorem of this chapter.

Theorem 7 (Soundness)**.** If $q \vdash e :: S$ and $p \geq q$ then either

- $\langle e, p\rangle \mapsto^* \langle v, p'\rangle$ and v is consistent with $p' \vdash v :: S$ or
- for every n there is $\langle e', p'\rangle$ such that $\langle e, p\rangle \mapsto^n \langle e', p'\rangle$.

Detailed proofs are included in the technical report [85]

2.4 Evaluation

We have implemented the new features of liquid resource types, inductive and abstract potentials, on top of the RESYN type checker; we refer to the resulting implementation as LRTCHECKER. In this section, we evaluate LRTCHECKER according to three metrics:

Expressiveness: Can LRTCHECKER express non-linear and dependent bounds? To what extent can LRTCHECKER express bounds that systems like RESYN and RAML could not?

Automation: Can LRTCHECKER automatically verify expressive bounds which other tools cannot? Are the verification times reasonable?

Flexibility: Can we define reusable datatypes that can express a variety of resource bounds across different programs?

2.4.1 Reusable Datatypes

We first describe a small library of resource-annotated datatypes we created, which we will use to specify type signatures for our benchmark functions. The definitions of the four datatypes are listed in 2.1. Since potential is only specified inductively in these definitions, we also provide a closed form expression for the potential associated with each such data structure (omitting the potential stored in the element type a). The proofs of these closed forms can be found in the technical report [85].

List and EList are general purpose list data structures that contain quadratic and exponential potential, respectively. In particular, List admits dependent potential expressions, as the abstract potential parameter is a function of the list elements. This list type can be adapted to express higher-degree polynomial potential functions via the generalized left shift operation, described in Section 2.3.3. EList can be modified to express exponential potential for any positive integer base k by modifying the type of the second argument to Cons:

$$Cons :: \mathsf{x} : \mathsf{a}^{q} \rightarrow \mathsf{xs} : \mathsf{EList\,a}\,\langle k \cdot q \rangle \rightarrow \mathsf{EList\,a}\,\langle q \rangle$$

Such a list contains $q \cdot (k^n - 1)$ units of potential; k has to be fixed for annotations to remain linear.

LTree is a binary tree with values (and thus, potential) stored in its leaves. We show that the total potential stored in the tree is $q \cdot n \cdot h$, where n is the number of leaves in the tree

Table 2.1. Annotated data structures with their corresponding potential functions. n is taken to be the number of elements in the data structure. In PTree, $|\ell|$ is the length of the path specified by the predicate p.

	Datatype	Potential Interpretation
1	**data** List a <q::a→a→Int> **where** Nil ::List a <q> Cons ::x: a →List a^{qx} <q> →List a <q>	$\sum_{i<j} q(a_i, a_j)$
2	**data** EList a <q::Int> **where** Nil ::EList a <q> Cons ::x: a^q →EList a <2∗q> →EList a <q>	$q \cdot (2^n - 1)$
3	**data** LTree a <q::Int> **where** Leaf ::a →LTree a <q> Node ::LTree a^q <q> →LTree a^q <q> → LTree a <q>	$\approx q \cdot n \log_2(n)$
4	**data** PTree a <p::a→Bool, q::Int> **where** Leaf ::PTree a <p,q> Node ::x: a^q →PTree a <p, ite(p(x), q, 0)> →PTree a <p, ite(p(x), 0, q)> → PTree a <p,q>	$q \cdot \|\ell\|$

and h is its height. If we additionally assume that the tree is balanced, then $h = O(\log(n))$, and hence the amount of potential in the tree is $O(n \cdot \log(n))$. In Section 2.4.2 we use this tree as an intermediate data structure in order to reason about logarithmic bounds.

PTree is a binary tree with elements in the nodes, which uses dependent potential annotations to specify the exact *path* through the tree that carries potential; we refer to this data structure as *pathed potential tree*. PTree is parameterized by a boolean-sorted *abstract refinement* [134], p, which is then used in the potential annotations to conditionally allocate potential either to the left or to the right subtree, depending on the element in the node. Since p is used to pick exactly one subtree at each step, it specifies a path from root to leaf.

These data structures showcase a variety of ways in which liquid resource types can be used to reason about a program's performance. Additionally, because the interpretation of abstract potentials is left entirely to the user, one can define custom data structures to describe

Table 2.2. Functional benchmarks. For each benchmark, we list its type signature, verification time (t), and source for the benchmark – either RAML [68], SYNQUID [107], or RELCOST [110].

Type	No.	Description	Type Signature	t (s)	Source
Polynomial Quadratic Potential	1	All ordered pairs	$\mathsf{List}\,\mathsf{a}^2\,\langle 2\rangle \to \mathsf{List}\,(\mathsf{Pair}\,\mathsf{a})$	0.5	RAML
	2	List Reverse	$\mathsf{List}\,\mathsf{a}^2\,\langle 1\rangle \to \mathsf{List}\,\mathsf{a}$	0.4	SYNQUID
	3	List Remove Duplicates	$\mathsf{List}\,\mathsf{a}^2\,\langle 1\rangle \to \mathsf{List}\,\mathsf{a}$	0.4	SYNQUID
	4	Insertion Sort (Coarse)	$\mathsf{List}\,\mathsf{a}^2\,\langle 1\rangle \to \mathsf{List}\,\mathsf{a}$	0.6	SYNQUID
	5	Selection Sort	$\mathsf{List}\,\mathsf{a}^4\,\langle 3\rangle \to \mathsf{List}\,\mathsf{a}$	0.5	SYNQUID
	6	Quick Sort	$\mathsf{List}\,\mathsf{a}^3\,\langle 3\rangle \to \mathsf{List}\,\mathsf{a}$	1.0	SYNQUID
	7	Merge Sort	$\mathsf{List}\,\mathsf{a}^2\,\langle 2\rangle \to \mathsf{List}\,\mathsf{a}$	0.9	SYNQUID
Non-Polynomial Potential	8	Subset Sum	$\mathsf{EList}\,\mathsf{Int}\,\langle 2\rangle \to \mathit{Int} \to \mathsf{Bool}$	0.3	–
	9	Merge Sort Flatten	$\mathsf{LTree}\,\mathsf{a}^1\,\langle 1\rangle \to \mathsf{List}\,\mathsf{a}$	0.9	–
Value-Dependent Potential	10	Insertion Sort (Fine)	$\mathsf{List}\,\mathsf{a}^1\,\langle \lambda x_1, x_2.\,\mathsf{ite}(x_1 < x_2, 1, 0)\rangle \to \mathsf{List}\,\mathsf{a}$	5.4	RELCOST
	11	BST Insert	$x{:}a \to \mathsf{PTree}\,\mathsf{a}\,\langle \lambda x_1.x < x_1, 1\rangle \to \mathsf{PTree}\,\mathsf{a}\,\langle \lambda x_1.x < x_1, 0\rangle$	2.4	–
	12	BST Member	$x{:}a \to \mathsf{PTree}\,\mathsf{a}\,\langle \lambda x_1.x < x_1, 1\rangle \to \mathsf{Bool}$	6.0	–

other resource bounds as needed.

2.4.2 Benchmark Programs

We evaluate the expressiveness of LRTCHECKER on a suite of 12 benchmark programs listed in Table 2.2. The resource consumptions of these benchmarks covers a wide range of complexity classes. We choose functions with quadratic, exponential, logarithmic, and value-dependent resource bounds in order to showcase the breadth of bounds LRTCHECKER can verify. We are able to express these bounds using only the datatypes from 2.1, showing the flexibility and reusability of these datatype definitions. The cost model in all benchmarks is the number of recursive calls (as in Section 3.3).

Benchmarks 1-7 require only standard quadratic bounds. Benchmarks 2-7 are those programs from the original SYNQUID benchmark suite [107] that RESYN could not handle, because they require non-linear bounds. Some of the analyses, such as merge sort, are over-approximate. Benchmark 8 moves beyond polynomials, solving the well-known subset sum problem. The function runs in exponential time, so we can write a resource bound using our EList data structure to require exponential potential in the input. Once we have verified that $\mathit{subsetSum} :: \mathsf{EList}\,\mathsf{Int}\,\langle 2\rangle \to \mathsf{Int} \to \mathsf{Bool}$, we can use the provided closed-form potential func-

tion to calculate total resource usage: $2(2^n - 1)$, exactly the number of recursive calls made at runtime.

Benchmark 9 illustrates how LRTCHECKER can verify a more precise $O(n\log(n))$ bound for a version of merge sort. LRT is unable to allocate logarithmic amount of potential directly to a list, hence we specify this benchmark using LTree as an intermediate data structure. Prior work has shown [13] that merge sort can be written more explicitly as a composition of two function: build, which converts a list into a tree (where each internal node represents a split of a list into halves), and flatten, which takes a tree and recursively merges its subtrees into a single sorted list. In the traditional implementation of merge sort, the two passes are fused, and the intermediate tree is never constructed; however, keeping this tree explicit, enables us to specify a logarithmic bound on the flatten phase of merge sort, which performs the actual sorting. We accomplish this by typing its input as $\mathsf{LTree}\,\mathsf{a}^{1}\,\langle 1\rangle$; because build always constructs balanced trees, this tree carries approximately $n\log(n)$ units of potential, where n is the number of leaves in the tree, which coincides with the number of list elements. Unfortunately, LRT is unable to express a precise resource specification for the build phase of merge sort, or for the traditional, fused version without the intermediate tree.

Benchmarks 10 through 12 show the expressiveness of value-dependent potentials. Benchmark 10 is the dependent version of insertion sort from Section 3.3. Benchmarks 11 and 12 use the PTree data structure to allocate linear potential along a value-dependent path in a binary tree. We use PTree to specify the resource consumption of inserting into and checking membership in a binary search tree. PTree allows us to assign potential only along the specific path taken while searching for the relevant node in the tree. As a result, we can endow our tree with exactly the amount of potential required to execute member or insert on an arbitrary BST. If we have the additional guarantee that our BST is balanced, we can also conclude that these bounds are logarithmic, as the relevant path is the same length as the height of the tree.

2.4.3 Discussion and Limitations

Table 2.2 confirms that LRTCHECKER is reasonably efficient: verification takes under a second for simple numeric bounds (benchmarks 1-9). The precision of value-dependent bounds (benchmarks 10-12) comes with slightly higher verification times—up to six seconds; these benchmarks generate second-order CLIA constraints and require the use of RESYN's CEGIS solver (as opposed to first-order constraints that can be handled by an SMT solver).

No other automated resource analysis system can verify all of our benchmarks in Section 2.4.2. RELCOST can be used to verify all of these bounds, but provides no automation. RESYN cannot verify any of our benchmarks, as it can only reason about linear resource consumption. RAML can infer an appropriate bound for benchmarks 1-7, which all require quadratic potential. However, RAML cannot reason about the other examples, as it cannot reason about program values, and only support polynomials. RAML relies on a built-in definition of potential in a data structure, while LRTCHECKER exposes allocation of potential via datatype declarations, allowing the programmer to easily configure it to handle non-polynomial bounds. In particular, a LRTCHECKER user can adopt RAML's treatment of polynomial resource bounds via our List type, and can also write non-polynomial specifications with other datatypes from our library or with a custom datatype.

The RELCOST formalism presented in [110] allows one to manually verify all of the bounds in Section 2.4.2. [30] presents an implementation of a subset RELCOST. This tool can be used to automatically verify non-linear bounds that are dependent only on the length of a list. To verify non-linear bounds, the system still generates non-linear constraints, and thus relies on incomplete heuristics for constraint solving. Benchmarks 10-12 in Table 2.2 all consist of conditional bounds, which are not supported by the implementation of RELCOST.

Despite LRTCHECKER's flexibility, it has some limitations. Firstly, our resource bounds must be defined inductively over the function's input, and hence we cannot express bounds that do not match the structure of the input type. A prototypical example is the logarithmic bound for

merge sort: we can specify this bound for the flatten phase, which operates over a tree (where the logarithm is "reified" in the tree height), but not for merge sort as a whole that operates over a list.

Secondly, LRTCHECKER cannot express multivariate resource bounds. Consider a function that takes two lists and returns a list of every pair in the cartestian product of the two inputs. This function runs in $O(m \cdot n)$, where m and n are the lengths of the two input lists. There is no way to express this bound by annotating the types of input lists with terms form CLIA.

Finally, LRTCHECKER can verify, but not infer resource bounds. So while verification is automatic, finding the correct type signature must be done manually, even if the correct data structure has been selected. Simple modifications would allow the system to infer non-dependent resource bounds following the approach of RAML [68], but this technique does not generalize to the dependent case.

2.5 Related work

Verification and inference techniques for resource analysis have been extensively studied. Traditionally, automatic techniques for resource analysis are based on a two-phase process: (1) extract recurrence relations from a program and (2) solve recurrence relations to obtain a closed-form bound. This strategy has been pioneered by Wegbreit [141] and has been later been studied for imperative programs [3, 4] using techniques such as abstract interpretation and symbolic analysis [79, 80]. The approach can also be used for higher-order functional programs by extracting higher-order recurrences [32]. Other resource analysis techniques are based on static analysis [49, 51, 145, 124] and term rewriting [14, 70, 98, 21].

Most closely related to our work are type-based approaches to resource bound analysis. We biuld upon type-based automated amortized resource analysis (AARA). AARA has been introduced by Hofmann and Jost [69] to automatically derive linear bounds on the heap-space consumption of first-order programs. It has then been extended to higher-order programs [76],

polynomial bounds [61, 59] and user-defined types [76, 60]. Most recently, AARA has been combined with refinement types [**?**] in the Re^2 type system [84] behind RESYN, a resource-aware program synthesizer. None of these works support user-defined potential functions. As discussed in Section 3.1, this chapter extends Re^2 with inductive datatypes that can be annotated with custom potential functions. The introduction of abstract potential functions allows this work to reuse RESYN's constraint solving infrastructure when reasoning about richer resource bounds. This work also formalizes the technique for user-defined inductive datatypes, while the Re^2 formalism admitted only reasoning about lists.

Several other works have used refinement types and dependent types for resource bound analysis. Danielsson [31] presented a dependent cost monad that has been integrated in the proof assistant Agda. dℓPCF [89] introduced linear dependent types to reason about the worst-case cost of PCF terms. Granule [100] introduces graded modal types, combining the indexed types of dℓPCF with bounded linear logic [47] and other modal type systems [45, 23]. While useful for a variety of applications, such as enforcing stateful protocols, reasoning about privacy, and bounding variable reuse, these techniques do not allow an amortized resource analysis. Çiçek et al. [28, 30] have pioneered the use of relational refinement type systems for verifying the bounds on the difference of the cost of two programs. It has been shown that linear AARA can be embedded in a generalized relational type systems for monadic refinements [111]. While this article does not consider relational verification, the presented type system allows for decidable type checking and is a conservative extension of AARA instead of an embedding.

Similarly, TiML [139] implements (non-relational) refinement types in the proof assistant Coq to aid verification of resource usage. A recent article also studied refinement types for a language with lazy evaluation [57]. However, these works do not directly support amortized analysis and do not reduce type checking of non-linear bounds to linear constraints.

Chapter 3

Type-directed Program Synthesis

3.1 Introduction

Chapter 1 showed one application of rich types as a synthesis specification: a type signature can describe functional properties as well as bounds on resource usage. While we have already discussed synthesis from formal logic [107], other works have used different type systems to implement synthesis specifications. For example, Myth and its follow-up Myth2 interpret input-output examples as types [101, 43], allowing users to specify programs via concrete test cases. The Granule project provides a synthesizer for graded modal types, encapsulating even more abstract notions of resource usage [72]. All of these tools, called *type-directed* synthesizers, use type information to specify the target program and to guide the search for a satisfying term.

Despite the variety of tools in the space, there are still two primary problems in type-directed program synthesis. First, innovations in search are not transferrable. Synthesis tools tend to propose an approach to search that's optimal for their particular setting. They may present a novel technique, or just hone in on the ideal algorithm for their particular context, but in either case the findings and engineering efforts are not easily shared between synthesizers. Second, providing strong specifications is still very difficult. Each tool mentioned above offers a different way to specify synthesis problems, but each approach has its own shortcomings. Input-output examples are often verbose and unwieldy, for example, and refinement types can take significant expertise to write.

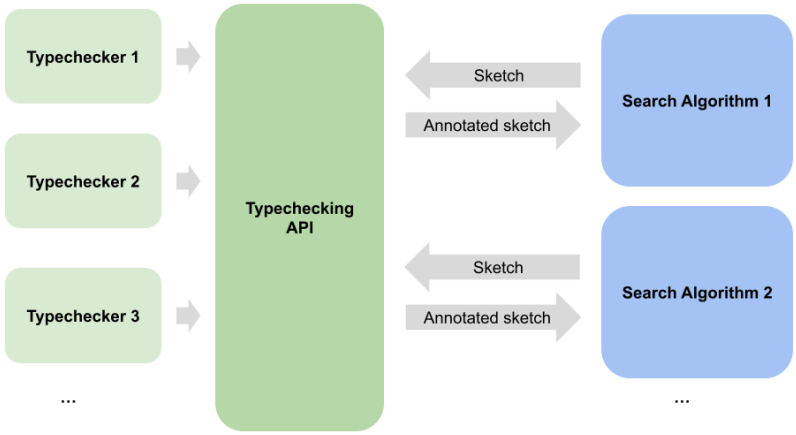

Figure 3.1. Framework architecture

Our observation is that despite the seemingly-fragmented state of research into type-directed synthesis, all the aforementioned tools share a common structure. They all propagate type information *top-down* while generating programs in order to shrink the search space. Whenever a synthesizer guesses some syntax node, it produces a number of new synthesis problems. Propagating the goal type top-down into these subproblems corresponds directly to pruning of the search space, as it allows the synthesizer to reject candidate programs without generating a complete term.

Our framework reifies this structure: we use the notion of a *sketch* – an incomplete term with holes – in order to cleanly separate search from verification. Figure 3.1 outlines the framework architecture. The framework provides a set of search algorithms, called generators, that enumerate program sketches from a core functional language. Then, the typechecker either *rejects* invalid sketches or *annotates* them with new specifications, thus guiding the generator towards a solution to the synthesis problem. We provide a set of APIs mediating this interaction: any typechecker that implements them can become a synthesizer. In this architecture, search techniques are modular and reusable, freeing researchers to address the problems of search and specification independently. We make the following contributions:

- A unified approach to type-directed program synthesis. A synthesizer combines a type-

checker that is capable of reasoning about incomplete terms with a generator that proposes program sketches.

- A guide to modifying a traditional bidirectional typechecker into one that can reason about incomplete programs, examples of how to do so for several real type systems, and formal statements of correctness.
- A Haskell framework, `tds`, for building type-directed synthesizers: specifically, a set of interfaces allowing a clean separation of search and verification.
- Implementations of several search algorithms for use with any compatible typechecker, collecting and unifying best practices for structuring search.
- Three separate instantiations of our framework cloning existing type-directed synthesizers. The clones are competitive if not faster than the original implementations, and are free of soundness issues.
- An instantiation of our framework with a *compositional* typechecker admitting new styles of specification.
- An evaluation of our framework with respect to performance, ease of use, and expressiveness.

3.2 Background

Given a specification and a set of components, a synthesizer attempts to build a satisfying program from the provided components. In type-directed synthesis, the specification is a type signature. This section will provide an overview of the research area. We show what synthesis problems look like for three representative synthesizers: Myth, Synquid, and Granule. All of these tools generate functional programs – a domain where types are particularly well-suited for specification.

3.2.1 Myth: Synthesis from examples

The Myth synthesizer generates recursive functions from a type signature and a set of input-output examples [101]. We can interpret this as a typechecking problem: we want to check that a function computes the given outputs from the given inputs. Follow-up work to Myth [43] augments this interpretation of examples as types with set-theoretic operators. As an illustrative example, consider a function `stutter` that takes a list and returns a list where each element has been copied. We can specify this function with a type signature (in syntax adapted from the original):

$$\texttt{stutter} :: [\mathsf{Int}] \to [\mathsf{Int}] \triangleright \langle [] \rightsquigarrow [] \mid [0] \rightsquigarrow [0,0] \mid [1,0] \rightsquigarrow [1,1,0,0] \rangle$$

In the type above, we decorate the simple function type on with a set of three examples. Referred to as a "partial function", the examples are pairs of inputs to `stutter` and their corresponding outputs. An empty list maps to itself, a singleton list maps to a list with its contents doubled, and so on. Given these three examples, Myth can generate the implementation one would expect:

```
stutter = \xs.
  match xs with
    Nil -> Nil
    Cons h t -> Cons h (Cons h (stutter t))
```

A completely naive synthesizer would simply enumerate the space of possible programs, typechecking each one. Not only is the space of programs huge, but in this case typechecking requires actually evaluating the program on a set of examples, making the process particularly costly. Instead of blindly enumerating, Myth and other tools in the space use the type system

itself to guide the search for a well-typed term. The implementation attempts to decompose examples *during* synthesis. For example, as soon as Myth decides to pattern-match on `xs`, it generates two new synthesis goals: one for each case. Then, Myth can *partition* the examples into two sets: the examples where `xs` is `[]`, and the examples where `xs` is non-empty.

Generating specifications for new synthesis goals corresponds directly to pruning: rejecting solutions to a subproblem without needing to generate a complete program drastically reduces the search space. For example, once we partition the examples for `stutter` between the empty and non-empty input cases, there is only one relevant output when the input list is empty: `[]`. Thus, Myth can immediately generate the corresponding program `Nil`, as it will always satisfy the single example.

3.2.2 Synquid: Synthesis from refinement types

The Synquid synthesizer allows users to specify programs with a refinement type signature. Refinement types decorate polymorphic types with logical predicates. For example, we can define the type of natural numbers by constraining integers:

$$\texttt{type Nat} = \{\mathsf{Int} \mid \nu \geq 0\}$$

Throughout, ν refers to the value in question; the natural numbers are simply any non-negative integer. Refinement types admit the specification of a wide variety of synthesis problems. As a first example, the following type signature specifies `replicate`, a function that returns `n` copies of some value `x`:

$$\texttt{replicate} :: n : \mathsf{Nat} \rightarrow x : a \rightarrow \{[a] \mid \mathsf{len}\ \nu = n\}$$

The type for `replicate` above simply states that the output list contains `n` values. Polymorphism is sufficient to ensure that the `x` is the only element of the output list. Given the type above and a library of functions over the natural numbers, Synquid generates the following implementation:

```
replicate = \n. \x.
  if n <= 0
    then Nil
    else Cons x (replicate (dec n) x)
```

Synquid can use the refinements during search to prune the search space just like Myth did. Suppose that while generating `replicate` Synquid attempts to use the function `copy` $:: x : a \to \{[a] \mid \text{len } \nu = 2\}$. Synquid can immediately reject *any* application of `copy` at the top level of `replicate` (any program of the form `\xs. copy ...`), as `copy` will never produce a list of length `n`. It does not need to guess any function arguments to do so – Synquid can prune away an entire family of terms after only guessing the head of the application.

In general, refinement types are particularly good for synthesis because they provide unbounded guarantees about the resulting program. Logical refinements are a natural way to specify a variety of nontrivial data structures, such as binary search trees or AVL trees. Doing so then allows one to generate code manipulating these data structures and have confidence that the implementations maintain the relevant invariants.

3.2.3 Granule: Synthesis from Graded Modal Types

Our third and final example, the Granule language, also provides a synthesizer [72]. Granule implements Graded Modal Types – a highly expressive technique for quantitative reasoning about program resource consumption. Granule is very abstract and flexible; here we will only provide a brief overview of some simpler use cases.

Granule generalizes the notion of *linear types* [46, 136]. In a linear type system, variables must be used exactly once. Figure 3.2 shows two programs that are *ill-typed* in Granule: `copy` uses its argument twice, and `drop` never uses its argument.

Granule's graded modal type system makes a linear type system more flexible: it allows users to annotate types with richer constraints on variable usage. Figure 3.3 shows how Granule

```
copy :: a -> (a, a)          drop :: a -> ()
copy x = (x, x)              drop x = ()
```

Figure 3.2. Two examples of ill-typed programs under a linear type system

```
copy' :: a [2] -> (a, a)     drop' :: a [0] -> ()
copy' x = (x, x)             drop' x = ()
```

Figure 3.3. Non-linear variable usage in Granule via grading

allows us to re-implement well-typed versions of `copy` and `drop`: we annotate the function arrows with natural numbers specifying how many times we will use each argument.

Counting variable usages allows programmers to reason about resource consumption. Granule, for example, uses linear types to offer a file-handling interface that guarantees that an open handle is always closed (and cannot be used after closure) [100].

Granule can also be used as a synthesizer. For example, Granule can synthesize the `vec3` program in Figure 3.4. `vec3` maps a function over a vector of length three (represented with nested tuples). Doing so requires using the higher order argument three times, but the vector only once. The argument types are graded with integers specifying usage: the lack of annotation on the vector itself means that it is linear and can only be used once. Just like Myth and Synquid, Granule propagates resource constraints top-down during synthesis. Attempting to use any of the vector elements more than once will fail immediately, regardless of whether Granule has generated a complete program.

3.3 Overview

Section 3.2 gave an overview of the range of type systems amenable to synthesis, as well as how tools use type information to guide the search process. All three tools mentioned

```
vec3 :: (a -> b) [3] -> ((a, a), a) -> ((b, b), b)
vec3 f ((x, y), z) = ((f x, f y), f z)
```

Figure 3.4. A Granule program mapping a function over a vector of length 3

in Section 3.2 rely on the same general approach to search: enumerate programs, using type information to avoid ill-typed candidate terms. This approach works well, as evidenced by the wide variety of programs the tools can produce.

Even though all of these tools use broadly similar search procedures, they are not formalized or implemented in a unified way. Each of these tools uses a custom synthesis calculus, combining typing and search logic into a monolithic implementation. This approach is clearly suboptimal. For one thing, the monolithic structure means that search and verification are intertwined and therefore brittle. Moreover, it means that innovations in search are not portable between synthesizers, despite the commonalities amongst the tools. Developing a new synthesizer entails either attempting to adapt an existing and convoluted synthesis calculus, or starting from scratch.

Our framework, called `tds`, solves these problems by articulating the common structure amongst the various type-directed synthesis tools. We enforce a clean separation between search and verification, meaning that developing a synthesizer amounts to little more than writing a typechecker compatible with our framework. The framework provides a set of search backends compatible with any satisfying typechecker, making engineering efforts reusable and portable.

3.3.1 Framework design

The primary goal of our framework is to ensure that the search component of a synthesizer is reusable. To do so, we need to separate search from verification. It should be possible to build a synthesizer by combining two components: a *typechecker* to verify candidate programs and a *generator* responsible for search itself. We also want to ensure that we can continue to prune the search space *while* generating programs – i.e, the typechecker needs to be able to guide the generator to a well-typed solution.

We use the notion of a "sketch" to mediate the interaction between generator and verifier in the tradition of tools like Morpheus [39]. A sketch is a partial program, possibly containing holes: AST nodes representing unsolved synthesis goals, written □. The typechecker and

generator communicate via sketches: the generator enumerates sketches, and the typechecker either annotates the sketches with new specifications for its holes or rejects them outright.

Figure 3.1 shows the general structure of our synthesis framework. The framework itself provides a set of search backends, which are responsible for generating sketches. Every time the generator guesses a sketch it produces new search goals. For example, guessing that a term might be a function application produces two new goals: the function and its argument (represented as the sketch $\square\ \square$). As it enumerates AST nodes, the generator can call into any compatible typechecker in order to produce specifications for each new subgoal. The typechecker *annotates* holes with these specifications in order to pass information back to the generator.

In the rest of this section we will gradually concretize the framework design. First, in Section 3.3.2 we will step through an example of how Myth-tds, a clone of Myth implemented with `tds`, generates an example program. Next, in Section 3.3.3 we will provide an overview of the actual APIs at the heart of the synthesizer. Finally, in Section 3.3.4 we will show how to leverage the compositional nature of the synthesis framework to experiment with new ways to specify programs.

3.3.2 Synthesis from input-output examples with `tds`

To build some intuition behind the interaction between the typechecker and the generator we will walk through the process of synthesizing `stutter` with Myth-tds. Recall from Section 3.2.1 that the following type signature specifies the behavior of `stutter`:

$$\texttt{stutter} :: [\mathsf{Int}] \to [\mathsf{Int}] \triangleright \langle [] \rightsquigarrow [] \mid [0] \rightsquigarrow [0,0] \mid [1,0] \rightsquigarrow [1,1,0,0] \rangle$$

Figure 3.5 outlines the process of searching for a satisfying implementation. Given the initial sketch $\square$, the generator calls into the typechecker for a specification – a typing context and goal type – in order to begin searching for a solution. Then it either guesses a variable from

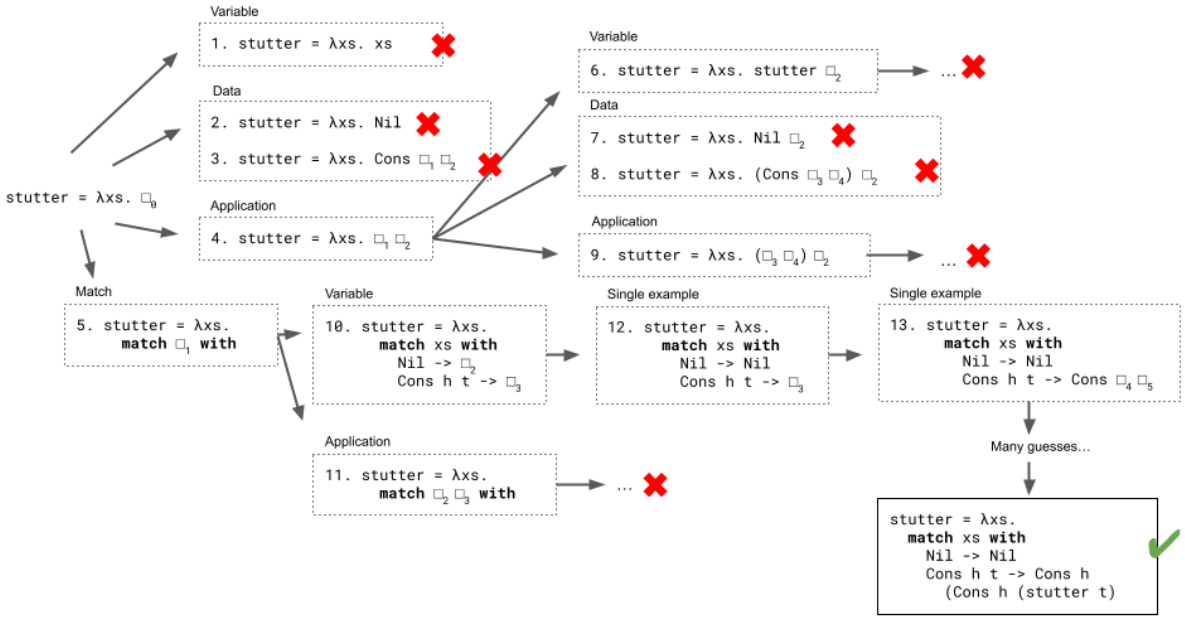

Figure 3.5. The search tree for synthesizing `stutter`. Sketches and holes are numbered for reference in the text. Boxes are labelled with a description of the search decision that produces the sketches within. Note how sketch 3 can be discarded before being fully solved, and how sketches 12 and 13 are produced deterministically.

the typing context or expands the goal by picking some AST node. Guessing an AST node can produce more goals in turn. Synthesizing `stutter` will proceed in this fashion: we enumerate sketches, using the typechecker to reject infeasible sketches and refine specifications for new goals.

Since we are synthesizing a function, the only possible choice for the top-level term is a lambda abstraction[1] This gives us the initial goal in Figure 3.5:

$$\begin{aligned}\square_0 :: [\mathsf{Int}] \triangleright \langle [xs \mapsto []] &\vdash [];\\ [xs \mapsto [0]] &\vdash [0,0];\\ [xs \mapsto [1,0]] &\vdash [1,1,0,0]\rangle\end{aligned}$$

The typechecker, upon encountering a lambda abstraction, decomposes the partial function into a set of possible *worlds*. A world represents one possible evaluation of the term that will eventually fill $\square_0$. Each world consists of a value and an environment binding variables to their values. In this case, in the world where `xs` is empty `stutter` should return the empty list. In the world where `xs` is the singleton list `[0]`, `stutter` should evaluate to `[0,0]`, and so on. Any solution to $\square_0$ should, given the variable bindings in a given world, evaluate to the corresponding output value.

Now, the generator starts guessing program candidates. The generator can enumerate variables, constructors, or applications. Guessing constructors allows the typechecker to reject sketches: Myth-tds can reject `Cons` $\square_1$ $\square_2$ (sketch 3) without solving its subgoals, as the synthesis goal contains empty list examples.

Producing a match node is more interesting. Pattern matching introduces new synthesis goals, and for each one the typechecker extends the context with new binders as necessary. Consider sketch 10, generated by first guessing that we need a `match` node, and then generating

[1]Myth-tds generates terms in β-normal η-long form. This is a common technique in synthesis to reduce the search space [56]. We will later show how to use `tds` to enforce normal forms in general.

a scrutinee:

```
stutter = \xs.
  match xs with
    Nil -> □2
    Cons h t -> □3
```

The typechecker can now produce stronger specifications for each of the two subgoals. Pattern matching means *partitioning* the provided examples by constructor. In the first branch, when input `xs` is [], the typechecker annotates $\square_2$ with a goal type featuring a single example: $\square_2 :: [\mathsf{Int}] \triangleright \langle [xs \mapsto []] \vdash [] \rangle$. The generator guesses the solution `Nil`, which immediately verifies (shown in sketch 12). Later, we will discuss an optimization allowing the generator to deterministically produce the appropriate term whenever all examples share a constructor.

In the second branch, we are concerned only about the worlds where the input list is non-empty. The typechecker again refines the specification for the synthesis goal, this time filtering away the worlds that correspond to empty inputs:

$$\square_3 :: [\mathsf{Int}] \triangleright \langle [xs \mapsto [0]] \vdash [0,0]; [xs \mapsto [1,0]] \vdash [1,1,0,0] \rangle$$

In general, the typechecker is propagating the original specification *top-down* through program sketches. Solving goal $\square_3$ is not as easy as the first was, but the type information still informs the generator. Since both output examples are non-empty lists (i.e, Cons nodes), the generator immediately reads off the following term for use in sketch 13: `Cons` $\square_4$ $\square_5$. Eventually, this process of alternately refining the specification and guessing AST nodes produces a well-typed term.

We can think of both Synquid and Granule as operating within the same synthesis paradigm. A refinement type checker can produce specifications for new goals just like how Myth-tds propagates concrete values through program sketches. Granule-tds, our clone of

Granule, achieves something similar by generating constraints that reflect a sketch's resource consumption, allowing it to reject sketches as soon as they overuse some resource.

3.3.3 Framework implementation

The `stutter` example from Section 3.3.2 demonstrates our interaction model. The typechecker is responsible for annotating holes with "specifications", necessary to ensure that a completion of the sketch can solve the goal. In this section we will give an overview of the actual framework implementation. Specifically, we discuss the actual term language, the interfaces by which a typechecker annotates sketches, and the implementation of a compatible generator. We will specialize our examples for Myth-tds throughout; Section 3.5 shows the full polymorphic interfaces.

Language

The synthesis framework provides a core functional language extended with first-class holes. A compatible typechecker must annotate holes with the information necessary to solve the corresponding synthesis problem. Thus, holes are either bare or decorated with a specification:

```
data Hole = Unspecified | Annotated Spec -- An unsolved synthesis problem
data Spec = Spec Context Type            -- A synthesis specification
```

In order to enumerate sketches, the generator needs a goal type and a set of components. A `Spec` provides this information: the generator will enumerate programs by guessing variables and constructors from the typing context.

Typechecker

The typechecker must be able to reject infeasible sketches and annotate feasible sketches with new specifications. To this end, the typechecker implements `refine`:

```
refine :: Env -> MType -> Term -> Maybe Term
```

Given a refinement type, context, and a term, `refine` either rejects the sketch or refines the specification [2]. Implicitly, we assume that `refine` *annotates* any synthesis goals with a `Spec`.

For example, consider calling `refine` to attempt to annotate `\xs. Cons` $\square_1$ $\square_2$ (sketch 3 in Figure 3.5):

```
refine env ([Int] → [Int] ▷ ⟨[] ⇝ [] | [0] ⇝ [0,0] | [1,0] ⇝ [1,1,0,0]⟩) (\xs. Cons □ □) =
  Nothing
```

Since the type for `stutter` contains one case with a non-empty output list, the typechecker rejects the sketch and returns `Nothing`. Imagine instead calling `refine` on the following sketch:

```
refine env ([Int] → [Int] ▷ ⟨[] ⇝ [] | [0] ⇝ [0,0] | [1,0] ⇝ [1,1,0,0]⟩) (
  \xs.
    match xs with
      Nil -> □2
      Cons h t -> □3)
```

The sketch is feasible, so we will get back the following *annotated* sketch:

```
stutter = \xs.
  match xs with
    Nil -> □2 : 𝒮2
    Cons h t -> □3 : 𝒮3
```

[2] In Section 3.5 we show the actual API which embeds failure to typecheck in the monadic type, enabling monadic backtracking

The holes are now annotated with specifications consisting of a context and goal type:

$$\mathscr{S}_2 = \langle \{\texttt{xs} : [\mathsf{Int}], \texttt{stutter} : \ldots\},$$
$$[\mathsf{Int}] \triangleright \langle [xs \mapsto []] \vdash [] \rangle \rangle$$
$$\mathscr{S}_3 = \langle \{\texttt{h} : \mathsf{Int}, \texttt{t} : [\mathsf{Int}], \texttt{xs} : [\mathsf{Int}], \texttt{stutter} : \ldots\}$$
$$[\mathsf{Int}] \triangleright \langle [xs \mapsto [0]][h \mapsto 0][t \mapsto []] \vdash [0,0];\ [xs \mapsto [1,0]][h \mapsto 1][t \mapsto [0]] \vdash [1,1,0,0] \rangle \rangle$$

In specification $\mathscr{S}_2$ we need only solve the $\square_2$ in the world where `xs` is empty: the typing context contains only `xs` and the recursive call, and the goal type has only a single example value. In specification $\mathscr{S}_3$ we have a new typing context and more complex goal type to specify $\square_3$. The environment has been extended with `h` and `t`, and the goal type specifies two input-output examples. When `xs` is `[0]`, `h` is `0` and `t` is `[]` the entire term should evaluate to `[0,0]`. Similarly, in the world where `xs` is `[1,0]`, `h` is `1` and `t` is `[0]` the entire term should evaluate to `[1,1,0,0]`.

In general, it should be fairly straightforward to implement `refine` via a traditional typechecker. At a high level there are only two requirements. First, the typechecker must overapproximate the semantics of holes: it should not reject program sketches that might be successfully solved. Second, when the typechecker encounters a hole, it should annotate such the hole with a `Spec` object. We will further discuss the details of implementing `refine` in Section 3.4 and Section 3.5.

Generator

Our interaction model, where the synthesizer simply has access to the `refine` interface, is very flexible. One can use `refine` to implement a number of search techniques. For example, `tds` provides generators that produce terms via depth-first search or iterative deepening, optionally memoizing solutions.

Perhaps most importantly, the generator can inspect annotated holes to guide the search process. For example, a generator can use $\mathscr{S}_2$ and $\mathscr{S}_3$ to inform its decisions when solving

$\square_2$ and $\square_3$. In both cases, all examples for each of the holes use the same constructor. Thus, the generator can deterministically produce the solution: `Nil` for $\square_2$ and a `Cons` node for $\square_3$. This is an instance of focusing, a more general technique from the proof search literature [12]. In Section 3.5 we discuss how our framework admits domain-specific focusing rules, key to efficiently synthesizing programs.

3.3.4 Combining specifications

A unified framework for type-directed synthesizers simplifies the task of developing new synthesis tools, as it reduces the problem to that of adapting a typechecker to implement `refine`. However, the framework also alleviates the need for new tools, as it allows us to combine the expressiveness of different type languages that operate on the core functional language.

Despite the variety of type-directed synthesis tools available, writing specifications that are strong enough for synthesis remains challenging. Consider as an example a standard function that filters a list according to some predicate:

```
filter :: (a -> Bool) -> [a] -> [a]
```

Myth can synthesize such a function, but requires 10 examples to do so, including two instantiations of the predicate in question. Polymorphic refinement types, as used by Synquid, provide an alternative and perhaps more ergonomic specification style. We can give `filter` the following refinement type:

```
filter :: <forall p :: a -> Bool>. (x:a -> {Bool| ν = p x}) -> [a] -> [{a|p ν}]
```

The type above uses a yet-unmentioned feature of refinement types: it is parameterized by a predicate `p`, providing a natural way to describe the functionality of `filter`. The higher-order argument to `filter` returns a boolean whose value is given by the parameter `p`. Then, the list returned by `filter` only contains elements that also satisfy the predicate `p`. Thus, this type is sufficient to verify that `filter` does in fact return a list containing only elements satisfying `p`.

$$\texttt{filter} :: (\forall\, p :: a \to \mathsf{Bool}).\ f : (x : a \to \{\mathsf{Bool} \mid \nu = p\ x\}) \to xs : [a] \to [\{a \mid p\ \nu\}]$$

$$\begin{aligned}\texttt{filter} :: (\mathsf{Int} \to \mathsf{Bool}) \to [\mathsf{Int}] \to [\mathsf{Int}] \triangleright \langle \mathsf{even} \rightsquigarrow ([] \rightsquigarrow [] \mid \\ [0] \rightsquigarrow [0] \mid \\ [0,0] \rightsquigarrow [0,0] \mid \\ [1,0] \rightsquigarrow [0]) \rangle\end{aligned}$$

Figure 3.6. A compositional specification for `filter`. The first type is a refinement type, the second is a set of examples. Note that the first is polymorphic and the second is not.

However, the type is not strong enough for synthesis. The term `filter = \p.\xs.[]` satisfies the provided specification, but is clearly not what we want. Strengthening the specification to indicate that *all* elements satisfying `p` must be in the output lists is not easy; the authors do not know how to do so with the particular refinement type system supported by Synquid.

Clearly, neither refinement types nor input-output examples offer an ideal way to specify `filter`. However, if we could *augment* the refinement type signature with a few simple examples, we could easily eliminate the degenerate case.

In fact, given two instantiations of the synthesis framework, we can do just that. We can provide two separate type signatures in two different type languages, and use the two typecheckers to prune a single search tree. Thus, we can synthesize a term that is well-typed in both systems, even if neither specification is complete on its own.

Figure 3.6 shows a pair of types that allow our compositional synthesizer to generate `filter`. Now, we only need to provide four examples instead of ten. All the four examples are small, and we only provide one instantiation of the higher-order argument instead of two. Moreover, the refinement type is much simpler than before; it is no longer reliant on abstract refinements.

3.4 The essence of type-directed synthesis

Thus far we have framed type-directed synthesis as the process of filling in holes in a term. Implicitly, using `tds` requires turning a typechecker for *complete* terms without holes into a typechecker for sketches that may contain holes. In this section, we present a unified formalism of the type systems corresponding to the Synquid, Myth, and Granule synthesizers, and in doing so attempt to generalize the process of adding support for holes. In all cases, we re-purpose existing formalisms as much as possible in order to make compatibility with our framework as painless as possible. For now we are concerned only with verifying sketches at some goal type: if we can typecheck sketches, we can implement a synthesizer by enumerating sketches, pruning any infeasible ones.

A naive approach to typechecking sketches would be to simply accept all incomplete terms. This will lead to a complete, but inefficient, synthesizer. Instead, we need to be able to reject as many incomplete terms as possible. To reason about terms with holes we can draw inspiration from tools like Hazelnut [99]: an editor that reasons about incomplete terms. Hazelnut in turn uses ideas from the gradual typing literature [121], which studies programs where terms might have an unknown type.

Our problem, however, turns out to be simpler than work on gradually typed programs. In particular, gradually typed languages might evaluate untyped code. This requirement leads to complex type systems that are difficult to extend with rich language features – in particular, generalizing notions of subtyping to the gradual setting remains challenging [142, 144].

Instead of dealing with the baggage involved in gradual type systems, we will reduce the problem of reasoning about holes to the problem of solving for unknown types. Doing so allows us to reject sketches when a solution does not exist, and it provides a concrete goal type when a solution does. In practice, we can solve for unknown types via unification, a well-studied technique. Some of our case studies will require additional machinery when we reason about the semantics of holes more precisely, but in every case we can use unification to simplify the task at

hand.

In this section we will show how to turn a typechecker for a standard term language into a synthesizer. Doing so requires reasoning about holes. In Section 3.4.1 we will introduce the concept by extending the simply-typed lambda calculus with holes. Then, we will go on to show how to repeat the same process the richer type systems of each case study.

Every example will feature a bidirectional presentation: bidirectional type checkers focus on deriving specifications for sub-goals in order to localize error detection. As shown via examples in Section 3.3, local error detection corresponds to pruning the search space, so such a formulation is ideal for synthesis.

3.4.1 Simple types

Type system

We begin by extending the simply-typed lambda calculus to support holes. We consider only a subset of the term language; Appendix A contains the full version of each type system shown in this section.

In Figure 3.7 we show the syntax for terms and types for this language, called $\lambda_{\square}$. Both are standard save for the inclusion of holes in the term language. We will use $\hat{e}$ to refer to terms from $\lambda_{\square}$, while e denotes a term from the simply-typed lambda calculus *without* holes (written λ). Note that a complete expression e is a well-formed term in $\lambda_{\square}$, while a sketch $\hat{e}$ is not necessarily well-formed in λ.

Figure 3.9 shows *bidirectional* typing rules for $\lambda_{\square}$. The bidirectional presentation uses two judgments. The checking judgment $\Gamma \vdash e \Leftarrow t$ checks some term e against goal type t. The inference judgment $\Gamma \vdash e \Rightarrow t$ infers the type t of some term e in context Γ. We use such a bidirectional presentation because it naturally lends itself to synthesis: when searching for a program we want to check the term against some goal type. The checking judgment corresponds to the notion of propagating a specification top-down through a sketch, allowing us to verify subgoals independently.

$$
\begin{aligned}
\hat{e} &::= \lambda\, x.\, \hat{e} \mid x \mid \hat{e}_1\, \hat{e}_2 \mid \square \\
B &::= \mathsf{Int} \mid \mathsf{Bool} \\
t &::= B \mid t_1 \to t_2
\end{aligned}
$$

Figure 3.7. Syntax and types of $\lambda_{\square}$

Our approach to reasoning about holes is straightforward: holes have any type. Holes nondeterministically produce *any* type t, implying in turn that they also check at any type. A concrete implementation will solve for the type of some hole via unification. For simplicity, we elide explicit type variables and everything else related to the constraint solver, as solving for unknown types is a well-studied problem. In fact, this is the key observation behind the usability of our framework: unification can do most of the heavy lifting involved in reasoning about incomplete terms. In sections Section 3.4.2 and Section 3.4.3, we will need to develop additional machinery to reason more precisely about the semantics of holes, but for now treating them as an unknown type is sufficient.

Type Safety

We need to ensure that a synthesizer for $\lambda_{\square}$ is sound and complete. The soundness of the synthesizer follows from the soundness of the original typechecker $\Gamma \vdash e \Leftarrow t$, as we will typecheck the final term. The completeness of a synthesizer follows from two properties: the completeness of the generator and the fact that our type system overapproximates the semantics of holes. We must not reject sketches that might have a well-typed solution. We formalize the second property, the safety of typing sketches, in Theorem 8. Theorem 8 states that if term e checks at type t, then any sketch $\hat{e}'$ that is *less precise* checks at the same type. The precision relation, defined in Figure 3.8, is a partial order on terms, where a term is more precise than another when it contains more information. The presence of holes makes a term less precise. Thus, Theorem 8 implies the completeness of a synthesizer for $\lambda_{\square}$, as it ensure that the typechecker will not reject any sketches that might be safely filled.

$\boxed{\hat{e} \sqsubseteq \hat{e}'}$

$$\frac{}{\hat{e} \sqsubseteq \square} \qquad \frac{}{x \sqsubseteq x} \qquad \frac{\hat{e}_1 \sqsubseteq \hat{e}_2}{\lambda\ x.\ \hat{e}_1 \sqsubseteq \lambda\ x.\ \hat{e}_2} \qquad \frac{\hat{e}_1 \sqsubseteq \hat{e}_2 \quad \hat{e}'_1 \sqsubseteq \hat{e}'_2}{\hat{e}_1\ \hat{e}'_1 \sqsubseteq \hat{e}_2\ \hat{e}'_2}$$

Figure 3.8. Term precision for $\lambda_{\square}$

Proposition 8 (Overapproximation)**.** *If $e \sqsubseteq \hat{e}'$ and $\Gamma \vdash e \Leftarrow t$, then $\Gamma \vdash \hat{e}' \Leftarrow t$.*

Proof. By induction on the derivation of $\Gamma \vdash e \Leftarrow t$. If $\hat{e}'$ is $\square$ we infer type t; otherwise we invoke the inductive hypothesis. When we encounter the IE rule we turn to 6. $\square$

Lemma 6 (Overapproximation of inference)**.** If $e \sqsubseteq \hat{e}'$ and $\Gamma \vdash e \Rightarrow t$, then $\Gamma \vdash \hat{e}' \Rightarrow t$.

Proof. By induction on the derivation of $\Gamma \vdash e \Rightarrow t$. If e is a variable, then $\hat{e}'$ is either the same variable or a hole. Both cases are trivial. If e is an application, then $\hat{e}'$ is either a hole or an application of weaker expressions, where we can apply the inductive hypothesis. $\square$

In the simply-typed setting these proofs are uninteresting. However, extending the same reasoning to richer type systems will isolate the additional work we need to do when attempting to reason more precisely about the semantics of holes.

One might notice the similarity between Theorem 8 and the static gradual guarantee [122]. The properties are in fact very similar; differing only in the fact that gradual languages want to ensure that less precise terms get less precise types. In the context of synthesis, however, we are always given a top-level goal type, and we are primarily interested in ensuring that our type system will not reject feasible sketches. Thus, replacing concrete sub-expressions with holes (making a term less precise) should not prevent verification at the original goal type.

Pruning

$\lambda_{\square}$ should provide a safe mechanism for rejecting infeasible sketches. As a simple example, checking $\cdot \vdash \lambda\ x.\ x\ \square \Leftarrow \mathsf{Int} \to \mathsf{Int}$ fails, as x does not unify with an arrow type.

$\boxed{\Gamma \vdash \hat{e} \Leftarrow t \text{ and } \Gamma \vdash \hat{e} \Rightarrow t}$

$$\text{Var}\frac{\Gamma(x) = t}{\Gamma \vdash x \Rightarrow t} \quad \text{HoleInfer}\frac{}{\Gamma \vdash \square \Rightarrow t} \quad \text{IE}\frac{\Gamma \vdash \hat{e} \Rightarrow t}{\Gamma \vdash \hat{e} \Leftarrow t}$$

$$\text{App}\frac{\Gamma \vdash \hat{e}_1 \Rightarrow t_1 \to t_2 \quad \Gamma \vdash \hat{e}_2 \Leftarrow t_1}{\Gamma \vdash \hat{e}_1\ \hat{e}_2 \Rightarrow t_2} \quad \text{Abs}\frac{\Gamma; x : t_1 \vdash \hat{e} \Leftarrow t_2}{\Gamma \vdash \lambda\ x.\ \hat{e} \Leftarrow t_1 \to t_2}$$

Figure 3.9. Bidirectional typing rules for $\lambda_\square$

However, checking $\cdot \vdash \lambda\ x.\ \square\ x \Leftarrow \mathsf{Int} \to \mathsf{Int}$ succeeds, as the hole emits an unknown type, which does unify with an arrow type.

3.4.2 Refinement types

So far we have shown how to add holes to the simply-typed lambda calculus. To implement Synquid-tds we repeat the process with refinement types: polymorphic types decorated with logical predicates [115].

Type system

Figure 3.10 shows the type language for a simple polymorphic refinement type system. Types are either dependent function types, base types annotated with a logical predicate, or "contextual" types that include a set of variable bindings. For example, $\{\mathsf{Int} \mid \nu \geq 0\}$ is the type of natural numbers: ν always refers to the value itself. Importantly, the system is also polymorphic; the typing context can contain polymorphic type schemata. As far as terms go, we are still considering only the lambda calculus discussed in Section 3.4.1. As such, we do not include datatypes or path conditions, focusing instead on a very minimal refinement type system.

Framing the synthesis of refinement-typed programs in terms of our framework means as usual first adding holes to the language. Figure 3.11 shows a very simplified form of the rules necessary to check function applications against a refinement type. We include the AppFO rule handling first-order applications; higher-order applications and the rest of the language are in A.0.2. Traditional presentations of a refinement type system force terms into A-normal form in

order to ensure that all function arguments are in the typing context [115]. This allows them to use function arguments directly in logic terms. In our case, adding holes to the logic is nontrivial, so instead we use a Synquid-style presentation featuring contextual types. When inferring the type of a function application we include a typing context recording the type of each function argument.

We also include the subtyping rules: subtyping between scalar types reduces to implication between their refinements – $[\![\Gamma]\!]$ embeds the assumptions in context in the logic for validation via SMT. Subtyping involving contextual types requires additionally including any assumptions related to the extended context.

This time our rule for typing holes is nontrivial. We include a premise stating that the hole type must be "consistent" with the typing context. Consistency, defined in Figure 3.12 essentially means that the type *might* be inhabitable. For refined base types this means that the refinement is simultaneously satisfiable with the context. For dependent function types this means that the return type is consistent with the context. This premise prevents the type constraint solver from simply instantiating holes with a $\bot$ refinement in order to trivially satisfy logical constraints. Doing so would be safe, but would forego many opportunities for pruning.

Type Safety

To prove that our system reasons about holes appropriately we need to adapt our formal analysis of the type inference judgment from Section 3.4.1. While we can re-use Theorem 8 for reasoning about the checking judgment in $\lambda_{\square}^{\Psi}$, we need a stronger lemma when considering the inference case: we need to ensure that adding holes to a term still produces a type that is a subtype of our goal. We also assume that our inferred type is consistent with the typing context – doing so ensures that we will not generate uninhabitable types. This condition will hold as long as there are no inconsistent types in context when computing $\Gamma \vdash e \Rightarrow T$ and our original goal type is consistent. Both properties are easily enforceable in practice. 7 states the overapproximation property formally.

$$
\begin{array}{rll}
B ::= & \mathsf{Int} \mid \mathsf{Bool} & \text{Base types} \\
T ::= & \{B \mid \psi\} \mid x : T \to T \mid \texttt{let}\ C\ \texttt{in}\ T & \text{Types} \\
S ::= & \forall \alpha_i . T \mid T & \text{Type schema} \\
C ::= & \cdot \mid x : T ; C & \text{Context} \\
\Gamma ::= & \cdot \mid x : S ; \Gamma & \text{Environment}
\end{array}
$$

Figure 3.10. A minimal refinement type system.

Lemma 7 (Overapproximation of inference: Refinement types)**.** If $e \sqsubseteq \hat{e}'$, and $\Gamma \vdash e \Rightarrow T$, then $\Gamma \vdash \hat{e}' \Rightarrow T'$ where $\Gamma \vdash T' <: T$.

Proof. We proceed by induction on the derivation of $\Gamma \vdash e \Rightarrow T$. If e is a variable, the non-trivial case occurs when $\hat{e}'$ is a hole. Here, the hole emits type T. We assume that the context Γ does not contain inconsistent types, in which case complete terms e must emit a consistent type – discussed in more detail in *Section A*.0.2. Thus, T must be consistent.

Now consider the case where e is some application $e_1\ e_2$. If $\hat{e}'$ is a hole, we simply infer type T. If not, then $\hat{e}' = \hat{e}'_1\ \hat{e}'_2$ for some $\hat{e}_1 \sqsubseteq \hat{e}'_1$ and $\hat{e}_2 \sqsubseteq \hat{e}'_2$. We apply the inductive hypothesis to each term and our result follows by the transitivity of subtyping. □

Pruning

As usual, $\lambda^{\psi}_{\square}$ is only useful if we can reject non-trivial sketches. Consider the following typechecking query that could arise during a synthesis problem:

$$\texttt{inc} : x : \mathsf{Nat} \to \{\mathsf{Nat} \mid \nu = x + 1\}, \ldots \vdash \texttt{inc}\ \square \Leftarrow \{\mathsf{Int} \mid \nu < 0\}$$

We will step through the process of inferring the type of `inc` □, and in doing so show how Synquid-tds can reject an incomplete function application based on its refinements. First, we infer the type of `inc`: $x : \mathsf{Nat} \to \{\mathsf{Nat} \mid \nu = x + 1\}$. Then we instantiate a fresh type variable β for the type of the hole. Applying the function yields the constraint $\Gamma \vdash \beta <: \mathsf{Nat}$, where Γ contains `inc`.

$\boxed{\Gamma \vdash \hat{e} \Leftarrow T \text{ and } \Gamma \vdash \hat{e} \Rightarrow T}$

$$\text{AppFO}\frac{\Gamma \vdash \hat{e}_1 \Rightarrow \texttt{let}\ C_1\ \texttt{in}\ x : \{B \mid \psi\} \to T_2 \qquad \Gamma; C_1 \vdash \hat{e}_2 \Rightarrow \texttt{let}\ C_2\ \texttt{in}\ T_1 \qquad \Gamma; C_1; C_2 \vdash T_1 <: \{B \mid \psi\}}{\Gamma \vdash \hat{e}_1\ \hat{e}_2 \Rightarrow \texttt{let}\ C_1; C_2; x : T_1\ \texttt{in}\ T_2}$$

$$\text{IE}\frac{\Gamma \vdash \hat{e} \Rightarrow T' \qquad \Gamma \vdash T' <: T}{\Gamma \vdash \hat{e} \Leftarrow T} \qquad \text{HoleInfer}\frac{\Gamma \Vdash T}{\Gamma \vdash \square \Rightarrow T}$$

$\boxed{\Gamma \vdash T <: T'}$

$$<:\text{-Scalar}\frac{\Gamma \vdash B <: B' \qquad \mathsf{Valid}(\llbracket \Gamma \rrbracket \wedge \llbracket \psi \rrbracket \implies \llbracket \psi' \rrbracket)}{\Gamma \vdash \{B \mid \psi\} <: \{B' \mid \psi'\}}$$

$$<:\text{-Arrow}\frac{\Gamma \vdash T_y <: T_x \qquad \Gamma; y : T_y \vdash [y/x]T <: T'}{\Gamma \vdash x : T_x \to T <: y : T_y \to T'}$$

$$<:\text{-CtxL}\frac{\Gamma; C \vdash T <: T'}{\Gamma \vdash \texttt{let}\ C\ \texttt{in}\ T <: T'} \qquad <:\text{-CtxR}\frac{\Gamma; C \vdash T <: T'}{\Gamma \vdash T <: \texttt{let}\ C\ \texttt{in}\ T'}$$

Figure 3.11. Selected typing rules of $\lambda_{\square}^{\psi}$

$\boxed{\Gamma \Vdash T}$

$$\text{Cons-Sc}\frac{\mathsf{SAT}\left(\llbracket \Gamma; v : B \rrbracket_{\psi} \wedge \psi\right)}{\Gamma \Vdash \{B \mid \psi\}} \qquad \text{Cons-Fun}\frac{\Gamma; x : T \vdash T' \Vdash}{\Gamma \Vdash x : T \to T'}$$

Figure 3.12. Type consistency in $\lambda_{\square}^{\psi}$

The entire application term emits the contextual type $\texttt{let}\ x : \beta\ \texttt{in}\ \{\mathsf{Nat} \mid \nu = x+1\}$, and checking it against the goal type produces the constraint $\Gamma \vdash \texttt{let}\ x : \beta\ \texttt{in}\ \{\mathsf{Nat} \mid \nu = x+1\} <: \{\mathsf{Int} \mid \nu < 0\}$. This leaves us with the following set of constraints:

$$\Gamma \Vdash \beta$$

$$\Gamma \vdash \beta <: \mathsf{Nat}$$

$$\Gamma \vdash \texttt{let}\ x : \beta\ \texttt{in}\ \{\mathsf{Nat} \mid \nu = x+1\} <: \{\mathsf{Int} \mid \nu < 0\}$$

The type constraint solver attempts to find a valuation for β, but fails to do so. Essentially, it must find some assignment $\{B \mid \psi\}$ for β such that the following logical constraints hold:

$$\mathsf{SAT}(\psi)$$

$$\mathsf{Valid}(\psi \implies \nu \geq 0)$$

$$\mathsf{Valid}([x/\nu]\psi \wedge \nu \geq 0 \wedge \nu = x+1 \implies \nu < 0)$$

Note that without the consistency constraint we could safely instantiate β to $\{\mathsf{Nat} \mid \bot\}$. While the resulting synthesizer would still be complete it would be much less efficient. As another example, consider the following typechecking problem (note the more general type of `inc`):

$$\texttt{inc} : x : \mathsf{Int} \to \{\mathsf{Int} \mid \nu = x+1\} \vdash \texttt{inc}\ \square \Leftarrow \{\mathsf{Int} \mid \nu < 0\}$$

Inferring the type of the application $\texttt{inc}\ \square$ yields the type $\texttt{let}\ x : \beta\ \texttt{in}\ \{\mathsf{Int} \mid \nu = x+1\}$. Checking said type against our goal provides a constraint: $\texttt{let}\ x : \beta\ \texttt{in}\ \{\mathsf{Int} \mid \nu = x+1\} <: \{\mathsf{Int} \mid \nu < 0\}$, providing us with a similar set of constraints as the previous example:

$$\mathsf{SAT}(\psi)$$

$$\mathsf{Valid}([x/\nu]\psi \wedge \nu = x+1 \implies \nu < 0)$$

There are a number of possible solutions for ψ; for example $v = -2$. Thus, the sketch checks, as we safely overapproximate the semantics of the hole.

3.4.3 Input-output examples

We can apply the same technique to build Myth-tds by interpreting input-output examples as type annotations akin to singleton refinements. As before, we add holes to a core calculus and show how this simple extension allows us to prune program sketches.

Adding holes is not as easy as it was when the static semantics were more abstract. In Myth, we check that a program term satisfies a set of examples by evaluating it. That means we now need a form of dynamic semantics for incomplete terms. However, we also cannot blindly reuse work from the gradual typing community, as we can have "partial" specifications for certain functions.

Type system

Figure 3.13 shows the syntax and types of $\lambda^{\mathsf{IO}}_{\square}$: the lambda calculus extended with sum types. As usual, we consider an extremely minimal language: we include integers, functions, and sum types, but omit everything else. We will show how $\lambda^{\mathsf{IO}}_{\square}$ still allows us to typecheck sketches based on the structure of terms and types The implementation, as usual, supports the full language from the original paper [101], including product and recursive types.

In a way, the type system looks very similar to refinement types, except now types are annotated with a set of possible values, denoted $T \triangleright \langle X \rangle$. The set of examples X consists of mappings from environments to values. We must track the value of each variable in a given "world". Values can be integers, unknown, constructor applications, partial functions (sets of possible input-output pairs), or sets of possible values, written $\|\overline{v_i}^i\|$.

Before discussing typechecking we consider the semantics of values in $\lambda^{\mathsf{IO}}_{\square}$. Figure 3.14 shows some rules for *evaluating* partial functions. Since typechecking means confirming that the program is consistent with the given set of input-output examples, we need dynamic semantics

$$
\begin{array}{rll}
e ::= & \Box \mid \mathbb{Z} \mid x \mid \mathsf{Left}\ e \mid \mathsf{Right}\ e \mid \lambda\, x.\, e & \text{Terms} \\
& e_1\ e_2 \mid \mathtt{match}\ e_1\ \mathtt{with}\ \mathsf{Left}\ x_1 \to e_2 \mid \mathsf{Right}\ x_2 \to e_3 & \\
T ::= & \mathsf{Int} \mid T \to T \mid T \oplus T \mid T \triangleright \langle X \rangle & \text{Types} \\
\sigma ::= & \cdot \mid [x \mapsto v]\sigma & \text{Bindings} \\
v ::= & \Box \mid \mathbb{Z} \mid \mathsf{Left}\ v \mid \mathsf{Right}\ v \mid \|\overline{v_i}^i\| \mid pf & \text{Values} \\
pf ::= & v_1 \rightsquigarrow v_1; \ldots; v_n \rightsquigarrow v_n & \text{Partial function} \\
X ::= & \cdot \mid \sigma \vdash v; X & \text{Example set}
\end{array}
$$

Figure 3.13. Syntax and types of $\lambda_{\Box}^{\mathsf{IO}}$

$\boxed{v \longrightarrow v'}$

$$
\mathrm{PFApp}\frac{v \neq \Box \qquad v_j \in \overline{v_i}^i \qquad v_j \sim v}{\overline{v_i \rightsquigarrow v_i'}^i\ v \longrightarrow v_j'}
$$

$$
\mathrm{PFAppAnyL}\frac{}{\Box\ v \longrightarrow \Box} \qquad \mathrm{PFAppAnyR}\frac{}{\overline{v_i \rightsquigarrow v_i'}^i\ \Box \longrightarrow \|\overline{v_i'}^i\|}
$$

$$
\mathrm{OneOfL}\frac{\overline{w_j}^j = \cup_i \{v' \mid v_i\ v \longrightarrow^* v'\}}{\|\overline{v_i}^i\|\ v \longrightarrow \|w_j\|^j} \qquad \mathrm{OneOfR}\frac{\overline{w_j}^j = \cup \{v' \mid v\ v_i \longrightarrow^* v'\}}{v\ \|\overline{v_i'}^i\| \longrightarrow \overline{w_j}^j}
$$

Figure 3.14. Selected evaluation rules for examples in $\lambda_{\Box}^{\mathsf{IO}}$

$\boxed{v \sim v'}$

$$
\mathrm{Refl}\frac{}{v \sim v} \qquad \mathrm{AnyL}\frac{}{\Box \sim v} \qquad \mathrm{AnyR}\frac{}{v \sim \Box}
$$

$$
\mathrm{PF}\frac{\forall i.\exists j.v_i \sim w_j \wedge v_i' \sim w_j' \qquad \forall j.\exists i.v_i \sim w_j \wedge v_i' \sim w_j'}{\overline{v_i \rightsquigarrow v_i'}^i \sim \overline{w_j \rightsquigarrow w_j'}^j} \qquad \mathrm{OneOf}\frac{\exists j.(v_j \sim v \wedge v_j \in \overline{v_i}^i)}{\|\overline{v_i}^i\| \sim v}
$$

Figure 3.15. Selected value consistency rules for $\lambda_{\Box}^{\mathsf{IO}}$

for sketches. Rule PFApp shows what happens when we apply a partial function to a concrete value v: if v is consistent with an input v_i, we return the corresponding output value. PFAppAnyL handles unknown functions: we simply return an unknown value. PFAppAnyR applies when the function argument is instead unknown: in this case, we return a set of possible values. The application can evaluate to any of the partial function outputs. Finally, the last two rules apply when the function or its argument is a set: we return the union of all possible evaluations given the set of possible values in the original term.

It's worth briefly discussing value consistency, shown in Figure 3.15. Consistency weakens equality to account for unknown values. A set of values is consistent with a single value v when v appears in the set, corresponding to our interpretation of values $\|\overline{v_i}^i\|$ as an overapproximation of a term's semantics.

Finally, Figure 3.16 shows a few typing rules for $\lambda_{\square}^{\mathsf{IO}}$. The first two rules, for introducing sum constructors, match all examples to the relevant constructor. In both rules, the first premise demands that all examples be applications of said constructor. Then, we extract the examples and proceed to check the argument. For example, the sketch $\mathsf{Left}\ \square$ is well-typed at $\mathsf{Int} \oplus \mathsf{Int} \triangleright \langle \cdot \vdash \mathsf{Left}\ 2 \rangle$, as the single example uses the correct constructor. As usual, holes check against any goal type, and emit an unknown type. Like before, we implicitly solve for unknown types via unification.

To eliminate sums, we partition the provided examples between the two cases. As an example, consider typing $\texttt{match}\ x\ \texttt{with}\ \mathsf{Left}\ x_1 \to x_1 \mid \mathsf{Right}\ x_2 \to x_2$ at goal type $\mathsf{Int} \triangleright \langle \langle [x \mapsto \mathsf{Left}\ 1] \vdash 1, [x \mapsto \mathsf{Right}\ 2] \vdash 2 \rangle \rangle$. The term x_1 must be consistent with each of the example worlds where x evaluates to an application of the left constructor. In the world where x evaluates to $\mathsf{Left}\ 1$, for example, we extend the substitution with $[x_1 \mapsto 1]$ and check the examples. The same process applies to the other branch, so typing succeeds.

The IE rule transitions between type checking and inference. To do so, we simply ensure that the inferred type unifies with the goal type, and we check that the actual term $\hat{e}$ is *consistent* with the examples X. This means *evaluating* $\hat{e}$ in every world in X via the relation in Figure 3.14.

$\boxed{\Gamma \vdash \hat{e} \Leftarrow T \triangleright \langle X \rangle, \Gamma \vdash \hat{e} \Rightarrow T, \text{ and } \hat{e} \models X}$

$$\text{Left}\frac{X = \overline{\sigma_i \vdash \mathsf{Left}\ v_i}^{\,i} \qquad \Gamma \vdash \hat{e} \Leftarrow T_1 \triangleright \langle \overline{\sigma_i \vdash v_i} \rangle}{\Gamma \vdash \mathsf{Left}\ \hat{e} \Leftarrow T_1 \oplus T_2 \triangleright \langle X \rangle}$$

$$\text{Right}\frac{X = \overline{\sigma_i \vdash \mathsf{Right}\ v_i}^{\,i} \qquad \Gamma \vdash \hat{e} \Leftarrow T_2 \triangleright \langle \overline{\sigma_i \vdash v_i} \rangle}{\Gamma \vdash \mathsf{Right}\ \hat{e} \Leftarrow T_1 \oplus T_2 \triangleright \langle X \rangle}$$

$$\text{SumElim}\frac{\begin{array}{c} X_1 = \{[v' \mapsto x_1]\sigma \vdash v \mid \sigma \vdash v \in X, \sigma(\hat{e}) \longrightarrow^* \mathsf{Left}\ v'\} \\ X_2 = \{[v' \mapsto x_2]\sigma \vdash v \mid \sigma \vdash v \in X, \sigma(\hat{e}) \longrightarrow^* \mathsf{Right}\ v'\} \\ \Gamma \vdash \hat{e} \Rightarrow T_1 \oplus T_2 \qquad \Gamma; x_1 : \alpha_1 \vdash \hat{e}_1 \Leftarrow T_1 \triangleright \langle X_1 \rangle \qquad \Gamma; x_2 : \alpha_2 \vdash \hat{e}_2 \Leftarrow T_2 \triangleright \langle X_2 \rangle \end{array}}{\Gamma \vdash \mathtt{match}\ \hat{e}\ \mathtt{with}\ \mathsf{Left}\ x_1 \to \hat{e}_1 \mid \mathsf{Right}\ x_2 \to \hat{e}_2 \Leftarrow T \triangleright \langle X \rangle}$$

$$\text{IE}\frac{\Gamma \vdash \hat{e} \Rightarrow T \qquad \hat{e} \models X}{\Gamma \vdash \hat{e} \Leftarrow T \triangleright \langle X \rangle} \qquad \text{HoleInfer}\frac{}{\Gamma \vdash \square \Rightarrow T}$$

$$\text{Var}\frac{\Gamma(x) = T}{\Gamma \vdash x \Rightarrow T} \qquad \text{App}\frac{\Gamma \vdash \hat{f} \Rightarrow T \to T' \qquad \Gamma \vdash \hat{x} \Leftarrow T \triangleright \langle \cdot \rangle}{\Gamma \vdash \hat{f}\ \hat{x} \Rightarrow T'}$$

$$\text{Satisfies}\frac{\forall \sigma \vdash ex \in X . \sigma(\hat{e}) \longrightarrow^* v \wedge v \sim ex}{\hat{e} \models X}$$

Figure 3.16. Selected bidirectional typing rules for $\lambda^{\mathsf{IO}}_{\square}$

As an example, consider the following typechecking problem that might arise when generating `stutter` from Section 3.2.1:

$$\Gamma \vdash \mathtt{stutter}\ \square \Leftarrow [\mathsf{Int}] \triangleright \langle [\mathtt{stutter} \mapsto ([] \rightsquigarrow []; [0] \rightsquigarrow [0,0]; [1,0] \rightsquigarrow [1,1,0,0])] \vdash [] \rangle$$

Since the term is a function application, we must evaluate it and confirm that the resulting value is consistent with the goal values (in this case, just []). Concretely, that means we check that the application term is satisfactory in *every world* by applying the relevant substitutions. In this case, we are considering only a single world, so we substitute the partial function for `stutter`, and evaluate the resulting function application. Applying the partial function $([] \rightsquigarrow []; [0] \rightsquigarrow [0,0]; [1,0] \rightsquigarrow [1,1,0,0])$ to the unknown value $\square$ yields a set of possibilities: $\|[], [0,0], [1,1,0,0]\|$. In turn, the set $\|[], [0,0], [1,1,0,0]\|$ is consistent with the goal value [], as [] is an element of the set, so typechecking succeeds.

Type Safety

Once again we need to adapt our formal analysis of the type inference judgment. The IE rule checks an inferred type by first ensuring that it unifies with the goal type, then ensuring that the term itself evaluates to a value consistent with the example set. In order to prove Theorem 8 in the context of $\lambda_{\square}^{\mathsf{IO}}$ we need an additional property: a weakened term should evaluate to a value that is consistent with the original value. 8 will allow us to show that weakening a term will not affect its consistency with some example set X.

Lemma 8 (Overapproximation of evaluation: Input-output examples)**.** If $e \sqsubseteq \hat{e}'$ and $e \longrightarrow^* v$, then $\hat{e}' \longrightarrow^* v'$ where $v \sim v'$

Proof. By induction on the derivation of $e \longrightarrow^* v$, considering the final step according to the structure of e. In general, when $\hat{e}'$ is a hole, the conclusion is trivial. In most cases, when $\hat{e}'$ is a weakened expression applying the inductive hypothesis is straightforward since evaluation is mostly structural. Applications are less obvious due to the introduction of the $\|\overline{v_i}^i\|$ constructor. However, it turns out that the case for $\|\overline{v_i}^i\|$ reduces to the case where $\hat{f}'$ evaluates to any other value, as there must be some v_i that is consistent with the original. We do not attempt to reason about the application of library functions to holes; such a term always evaluates to $\square$. Nor do we reason formally about termination. In practice, however, the typechecker includes a structural termination checker and positivity restriction on datatypes in order to ensure that the evaluation relation does not diverge. $\square$

Pruning

Even the language fragment shown in Figure 3.16 is sufficient to prune a variety of program sketches. Consider the following program sketch and associated goal type:

$$\mathsf{Left}\ \square \Leftarrow \mathsf{Int} \oplus \mathsf{Int} \triangleright \langle [x \mapsto \mathsf{Left}\ 0] \vdash \mathsf{Left}\ 0; [x \mapsto \mathsf{Right}\ 1] \vdash \mathsf{Right}\ 1 \rangle$$

$\Gamma \vdash \texttt{fix}\ f.\ \lambda\ x.\ zero\ (f\ \square) \Leftarrow \mathsf{Int} \to \mathsf{Int} \triangleright \langle 0 \rightsquigarrow 0; 1 \rightsquigarrow 1; 2 \rightsquigarrow 3 \rangle$
where
$zero = \lambda\ x.\ 0 \in \Gamma$

Figure 3.17. Pruning incomplete application terms in $\lambda^{\mathsf{IO}}_{\square}$

It turns out that we can immediately reject this term. The sketch includes the Left constructor, but one of the examples uses the Right constructor. Thus, the Left rule does not apply as the sketch will never satisfy the entire example set, so we reject it.

In fact, we can actually go even further. The original Myth implementation does not reason about incomplete application terms whatsoever. However, by extending the dynamic semantics to account for holes, our version actually allows us to reject incomplete application sketches. The introduction of the "one of" value constructor allows us to check the compatibility of an incomplete application term with desired examples.

Figure 3.17 shows such an example. We are synthesizing a recursive function f, defined by three input-output pairs. The typing context contains a constant function *zero*. It turns out we can already reject the sketch shown. Briefly, the recursive call $f\ \square$ evaluates to a set of possible values: $\|0,1,3\|$. Since *zero* comes with a concrete implementation, we evaluate it at each of the possibilities, which yields a single value 0. However, checking the entire term means checking it in every possible world (i.e, ensuring that *every* one of the partial function outputs is viable). Doing so fails: we know that the application term always evaluates to 0.

3.4.4 Graded Modal Types

Finally, we discuss synthesis from graded modal types [100], a substructural type system that abstracts over various notions of resource usage. The most important difference between the Granule type system and the ones we have discussed thus far is linearity: in Granule, linear variables must be used exactly once. The main contribution of this section is therefore showing that once again, solving for unknown types is sufficient to reject infeasible sketches.

$$
\begin{aligned}
e &::= \mathbb{Z} \mid x \mid \lambda\, x.\, e \mid e_1\, e_2 \mid \Box \\
\hat{T} &::= \mathsf{Int} \mid \hat{T}_1 \multimap \hat{T}_2 \\
\Gamma &::= \cdot \mid x : \hat{T}, \Gamma
\end{aligned}
$$

Figure 3.18. Syntax and types of $\lambda_{\Box}^{\multimap}$

Type system

In Figure 3.18 we present the syntax and types of $\lambda_{\Box}^{\multimap}$, a subset of the Granule language [100]. As usual, we consider a minimal language with lambdas, variables, applications, and holes. The type system includes integers and linear arrow types. Note that we actually omit graded modal types for now. Graded modal types are eliminated by conversion to linear types, so the linear system should be sufficient to convey the main ideas involved in adding holes to Granule.

Figure 3.19 shows the bidirectional typing rules. Since the system is linear, variables type only in a singleton context. Holes, as usual, can have any type. In the context of $\lambda_{\Box}^{\multimap}$, where typing contexts correspond to resources, we cannot make any claims about the resource usage of the unknown term, so holes can produce any type in any context. The application rule also uses a new notion of *context addition*. In our limited type system, this essentially amounts to partitioning the variables in context in order to typecheck the function and its argument in disjoint contexts. Context addition is more subtle in the full type system supporting graded types. However, thinking of context addition as a partitioning of available resources suffices here. For example, we have $x : A; y : B = x : A + y : B$. The rules for typing abstractions, as well as that for switching between checking and inference are again standard.

Type Safety

Since we do not attempt to reason about the semantics of holes except with respect to their simple types, we can reuse Theorem 8 and 6. The proofs are straightforward by induction on the typing derivations; as typing holes does not constrain the typing context at all.

$\boxed{\Gamma \vdash \hat{e} \Leftarrow T \text{ and } \Gamma \vdash \hat{e} \Rightarrow T}$

$$\text{Var}\frac{}{x : T \vdash x \Rightarrow T} \quad \text{HoleInfer}\frac{}{\Gamma \vdash \square \Rightarrow T} \quad \text{App}\frac{\Gamma_1 \vdash \hat{e}_1 \Rightarrow T_1 \multimap T_2 \qquad \Gamma_2 \vdash \hat{e}_2 \Leftarrow T_1}{\Gamma_1 + \Gamma_2 \vdash \hat{e}_1\ \hat{e}_2 \Rightarrow T_2}$$

$$\text{Abs}\frac{\Gamma, x : T \vdash \hat{e} \Leftarrow T_2}{\Gamma \vdash \lambda\ x.\ \hat{e} \Leftarrow T_1 \multimap T_2} \quad \text{IE}\frac{\Gamma \vdash \hat{e} \Rightarrow T}{\Gamma \vdash \hat{e} \Leftarrow T}$$

Figure 3.19. Selected bidirectional typing rules of $\lambda_{\square}^{\multimap}$

Pruning

Consider the following goal type and associated program sketch:

$$\texttt{foo} :: (\mathsf{Int} \multimap \mathsf{Int} \multimap \mathsf{Int}) \multimap \mathsf{Int} \multimap \mathsf{Int} \multimap \mathsf{Int}$$

$$\mathsf{foo} = \lambda\ f.\ \lambda\ x_1.\ \lambda\ x_2.\ f\ x_1\ \square$$

Typing the sketch essentially amounts to partitioning a context Γ that contains f, x_1, x_2 into three disjoint contexts Γ_i that can type each sub-term. Doing so highlights the power of linear types: the *only* possible way to fill the hole is with x_2, as x_1 has already been used. A non-linear type system would admit several solutions to this synthesis problem, but the linear system does not. This idea generalizes and allows us to prune sketches that violate resource constraints: over-using a variable will immediately lead to unsatisfiable SMT constraints.

3.5 Implementing and optimizing synthesis

Section 3.4 showed how to extend a typechecker to reason about program sketches. To turn the typechecker into a synthesizer we need only implement a generator. We can do so via backtracking search: we enumerate sketches, typechecking each one, only solving the subgoals in feasible sketches. This structure improves upon blind enumeration, as we can stop attempting to fill a given sketch as soon as it becomes inconsistent with the goal type. In this section we discuss what it takes to implement a more efficient synthesizer. Instead of using the typechecker

to simply reject sketches, we will now *annotate* the holes with information the generator can use to guide its search. This architecture, originally presented in Figure 3.1, addresses the two primary sources of inefficiency in a synthesizer built around simply rejecting sketches:

- When typechecking a sketch, we might re-check already-generated terms
- When enumerating sketches, we can guess AST nodes that fail immediately. For example, symbols of incorrect arity, functions with the wrong return type, or nodes that violate the term's normal form.

This section will show how to use `tds` to implement an efficient synthesizer that addresses these two issues. Figure 3.20 (based on the architecture in Figure 3.1) outlines the separation of implementation responsibilities. We begin in Section 3.5.1 by outlining the requirements on a compatible typechecker, with the goal of providing a high-level guide to interacting with `tds`. Specifically, we discuss the Haskell interfaces that mediate the interaction between typechecker and generator, and we introduce a `Spec` data structure that enriches specifications (typing contexts and goal types) with additional information. For example, the typechecker can indicate which AST nodes might be feasible for a given synthesis goal, thus preventing the synthesizer from guessing terms guaranteed to fail.

In Section 3.5.2 we discuss the implementation of a generator, describing how we implemented the backends available in `tds` and providing a basis for further experimentation and development. The remainder of the section discusses some further optimizations: at times the typechecker can provide even more information in order to improve synthesis performance. In general, this section describes how `tds` unifies best practices from each of the tools.

3.5.1 Implementing a typechecker

The typechecker is responsible for rejecting infeasible sketches, and annotating feasible sketches with specifications for their subgoals. In this section, we concretize the type checker, first discussing the concrete term language before providing an overview of the relevant APIs.

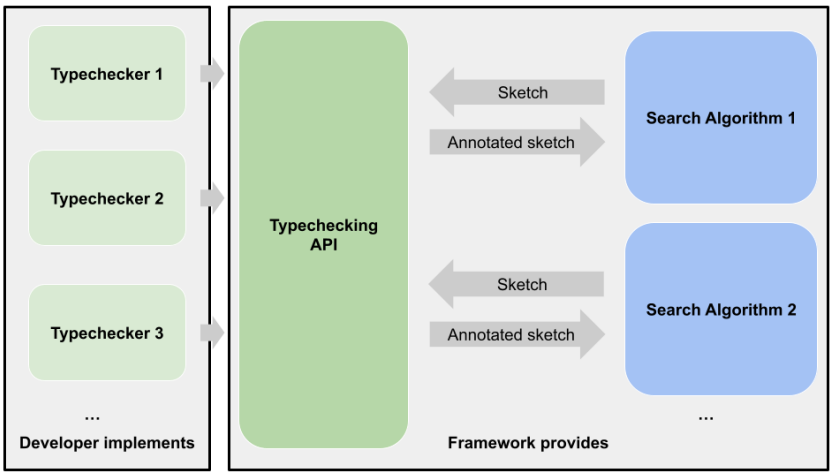

Figure 3.20. Separation of implementation responsibilities: Synthesizer developer implements a compatible typechecker; `tds` provides a suite of search backends.

Language

The typechecker must operate on our term language `PTerm s t`, parameterized by specifications `s` and types `t`. The term language is a standard functional language with first-class data constructors, shown in Figure 3.21. There are three kinds of holes: unspecified holes $\square$, holes annotated with a specification $\mathscr{S}$, or "solved" holes containing a term e. In the `tds` implementation, all terms carry an annotation of type `t` so that typecheckers can annotate them with type information. We include the special "solved" constructor so that the typechecker can avoid rechecking complete terms when possible: like other terms, solved goals can be annotated with their type in order to mitigate the need to traverse the term.

Holes, and therefore terms, are also parameterized by a specification `s`. So far, the term "specification" has referred to nothing more than a goal type and typing context. However, the interface is polymorphic with respect to the specification – eventually we will add additional information.

Interfaces

A compatible typechecker must implement several Haskell typeclasses, shown in Figure 3.22. The main typeclass, `Typechecker`, is parameterized by a monad `m`, typing context `e`,

$$
\begin{aligned}
e &::= \lambda\, x.\, e \mid \texttt{fix}\, f.\, e \mid x \mid e_2\, e_2 \mid \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \\
&\quad \mid \texttt{match}\, e_1\, \texttt{with}\, \mid_i C_i(\overline{x_j}^j) \mapsto e_i \mid C(e_1, e_2, \ldots, e_n) \mid h \mid \texttt{error} \\
h &::= \square \mid \square : \mathscr{S} \mid \square : e
\end{aligned}
$$

Figure 3.21. The term language provided by `tds`

type language `t`, and specification `s`. For now we will focus on a single method, `refine`. In practice, the typeclass includes some additional methods used to enable various search optimizations. However, these are always optional, and come with default implementations.

The actual typechecking monad `m` must be an instance of `MonadLogic`, a typeclass for performing backtracking computations. The library comes with instances for some common monads, and `tds` provides even more. `MonadLogic` comes with a single method, `msplit`, used to implement richer search primitives than standard monadic approaches to search. We will use the `MonadPlus` terminology, `mplus` and `mzero`, to denote nondeterministic choice and failure respectively. After some experimentation, our implementation exclusively uses the `logict` [81] package to handle nondeterminism, but there is certainly room for further research on this front.

The type language `t` must implement the `IsType` typeclass, which contains a single method that extracts the "shape" of a type. The shape of a type is simply its arrow structure; during synthesis it is often useful to know the arity of a type. Otherwise, the synthesizer is ambivalent to the actual type language. Typing contexts must implement the `Env` typeclass, which contains two methods. The first, `allVars`, allows one to extract all the symbols in the typing context. The second, `allDatatypes`, returns all the possible datatypes, where a `Datatype t` value contains the type name and all of its constructors. These methods are useful during synthesis when guessing symbols or constructors.

Implementing `refine`

Essentially, `refine` provides an interface to the actual functionality of the typechecker. At a high level, `refine` checks some term against a given goal type in some context.

```
1  class (Env e t, MonadLogic m) => Typechecker m e t s | m ->  e t s where
2    refine :: e -> t -> PTerm s t -> m (PTerm s t)
3    ...
4
5  class IsType t where
6    shapeOf :: t -> Shape
7
8  class IsType t => Env e t where
9    allVars :: e -> [Id]
10   allDatatypes :: e -> [Datatype t]
```

Figure 3.22. Interface to typechecker

However, the most important properties of `refine` are not obvious from the type signature. In order to implement an efficient synthesizer, `refine e t term` should:

1. Annotate any holes with a synthesis *specifications* `s`.
2. Annotate a complete term with its final type `t`.
3. Call `mzero` upon failure to initiate backtracking.

The first two requirements ensure that the synthesizer can be efficient. Local specifications enable pruning. More practically, we should avoid re-checking complete terms every time we call `refine`. As with many framework features the synthesizer will still work, albeit more slowly, if the typechecker rechecks synthesized terms. However, term annotations offer a way around the issue. The third requirement is mandatory, as it allows the generator to implement backtracking.

As discussed in Section 3.4, propagating specifications top-down corresponds to pruning during synthesis. We do not *require* that a typechecker be bidirectional; but the more effective it is at producing strong specifications for subgoals, the faster the resulting synthesizer will be.

Figure 3.23 outlines a very simple `refine` implementation for the simply-typed lambda calculus. It satisfies our three requirements. When it encounters an un-annotated hole on lines 3 and 13, it creates a `Spec` value with the appropriate type and context. In checking mode, the goal type goes into the `Spec` object. In inference mode, a fresh type `t` instead goes into the `Spec`. The

```
1  -- Type checking
2  refine :: Env -> Type -> Checker m (PTerm Spec Type)
3  refine env t (Hole Unspecified) = do
4    return (annotate (specifiedHole (Spec env t)) t)  -- Annotate hole
5  refine env (Arrow a b) (Lambda x e) = do
6    e' <- refine (extend env x a) b e                 -- Typecheck body
7    return (annotate (Lambda x e') (Arrow a b))       -- Annotate with final type
8  refine env t e = do
9    e' <- refineInfer env e                           -- Switch to type inference
10   equal t (annotation e')                           -- Assert that the types are equal
11
12 -- Type inference
13 refineInfer env (Hole Unspecified) = do
14   t <- freshType                    -- Instantiate an unknown type for the hole
15   return (annotate (specifiedHole (Spec env t))) t)
16 refineInfer env (Symbol x) = lookup x env
17 refineInfer env (App f x) = do
18   f' <- refineInfer env f
19   (Arrow a b) <- arrowType f'       -- Assert that f' must have an arrow type
20   x' <- refine env a x
21   return (annotate (App f' x') b)   -- Annotate with final type
```

Figure 3.23. Pseudocode for part of the bidirectional `refine` implementation for the simply-typed lambda calculus. The functions `arrowType` and `equal` that constrain types call `mzero` if the constraints are unsatisfiable.

unknown hole type is later constrained by assertions on line 10 or 19. Implicitly, the functions `arrowType` and `equal` must call `mzero` if the type constraints are unsatisfiable. Throughout, all terms are annotated with their types; the typechecker uses the annotations to implement type inference. A more realistic typechecker would presumably include some more sophisticated error handling code; a developer must simply add a call to `mzero` before the computation is actually interrupted. We also omit for simplicity the other two kinds of hole (specified or solved). Solved holes should already have a final type, and already specified holes can be overwritten with an updated specification.

3.5.2 Implementing a generator

The synthesis framework provides a variety of generators written against the `refine` interface. Figure 3.24 outlines a simple generator for $\lambda_{\square}$ terms. This presentation simplifies the types involved, monomorphizing the term, type, and typing contexts. The generator consists of two functions. `generate` takes an incomplete term, calls `refine` to specify the syntheis goals, and calls `fill` to solve them. In turn, `fill` simply guesses one of the possible AST nodes that might fill the now-specified hole. Guessing every possible node is clearly not optimal; in Section 3.5.3 we will show how to improve upon this.

The most important piece that's missing from this simplified generator is a notion of term cost. In general, search needs to be ordered with respect to some notion of cost: size, depth, or some richer metric. The `tds` implementation provides several generators, all of which either generate terms via depth-first search (to some fixed depth) or via iterative deepening. Within these two paradigms, we provide several other options. Generators can produce terms in order of "cost" (size or something richer), or in order of depth.

We also provide several implementations of memoization, as well as an implementation of *relevant* enumeration – a technique used in Myth [101] inspired by relevance logic [11]. The idea is to keep the typing context in order, and then define the synthesis function in terms of a "relevant variable"– $\mathsf{synth}(x;\Gamma,t) = \mathsf{synth}_x(\Gamma,t) \cup \mathsf{synth}(\Gamma,t)$, where the function synth_x enumerates terms that mention x. Relevant enumeration should lead to more cache hits: as the context is extended, we only need to enumerate terms that contain new symbols. In order to support relevant enumeration we need to extend the `Env` typeclass: we add a function `orderedVars` that returns the typing context ordered by when the variable was added. Failing to implemented `orderedVars` only means that relevant enumeration is not available as an option.

```haskell
-- Solve subgoals in an AST node by refining specifications
generate :: Environment -> Type -> Term -> Generator Term
generate env typ (Lambda x e) = do
  (Lambda x e') <- refine env typ (Lambda x e)
  body <- fill e'
  return (Lambda x body)
generate env typ (App f x) = do
  (App f' _) <- refine env typ (App f x)
  fun <- fill f'
  (App _ x') <- refine env typ (App fun x)
  arg <- fill x'
  return (App fun arg)
generate env typ (Symbol x) = do
  refine env typ (Symbol x)

-- Fill an annotated hole by guessing instantiations
fill :: Term -> Generator Term
fill (Hole (Spec env typ terms)) = do
  terms <- allTerms env        -- Generate all possible subterms
  t <- msum (map return terms) -- Pick one of the available terms
  generate env typ t

-- Every possible AST node from a typing context
allTerms :: Environment -> Generator [Term]
allTerms env = do
  x <- freshVar env   -- Generate a unique binder for a possible lambda
  return (allVars env ++ [App hole hole] ++ [Lambda x hole])
```

Figure 3.24. Pseudocode outlining a simple generator for $\lambda_{\square}$. `Generator` is a monad for synthesis providing nondeterministic choice as well as the infrastructure for generating fresh names. In the full implementation, the synthesis monad is parameterized by the typechecking monad, type language, and other framework parameters.

3.5.3 Optimizations: Focusing

Clearly, `refine` allows us to write a straightforward generator that typechecks AST nodes immediately upon guessing them, allowing early error detection. However, this is still not enough for a performant synthesizer. Existing synthesizers often use focusing, a key technique from the proof search community [12]. Focusing includes deterministically applying invertible proof rules, i.e. rules whose premises are derivable whenever the conclusion is derivable. For example, Synquid always generates lambda abstractions at the top level of an arrow-typed program. Similarly, when Myth encounters a type decorated with a single example, it can deterministically generate the program term associated with said example. Granule utilizes focusing even more heavily.

To support focusing we must augment the synthesis specifications. For the following examples we will add an optional list of possible terms to the specification data structure (recall that terms are annotated with types and specifications):

```
data Spec e t = Spec e t (Maybe [PTerm (Spec e t) t])
```

Now, the typechecker can provide a set of possible terms. This allows it to avoid guessing terms that will fail immediately, or to deterministically produce well-typed expressions when possible. As a first example, Figure 3.25 updates our implementation of `refine` for the simply-typed lambda calculus. Now, when encountering a top-level arrow type we immediately generate a lambda abstraction. Otherwise we do not provide any guidance to the generator.

This same technique admits more precise applications of focusing, a key contributor to the efficiency of Myth and Granule. Myth, for example, picks a concrete constructor based on the structure of the examples. Granule utilizes focusing even more aggressively, greedily matching on terms in context whenever possible.

```haskell
-- Type checking
refine :: Env -> Type -> Checker m (PTerm Spec Type)
refine env t (Hole Unspecified) = do
  ts <- case t of
          Arrow _ _ -> do -- Generate a lambda for a top-level arrow type
            x <- freshVar
            return (Just [Lambda x hole])
          _ -> return Nothing
  return (annotate (specifiedHole (Spec env t (Just ts))) t)
refine env (Arrow a b) (Lambda x e) = ...
refine env t e = ...

-- Type inference
refineInfer env (Hole Unspecified) = do
  t <- freshType              -- Instantiate an unknown type for the hole
  return (annotate (specifiedHole (Spec env t Nothing))) t)
refineInfer env (Symbol x) = ...
refineInfer env (App f x) = ...
```

Figure 3.25. Revisions to our implementation of `refine` for the simply-typed lambda calculus, now generating terms in β-normal η-long form. Unchanged code is omitted.

Focusing in Myth

As alluded to earlier, Myth can use its input-output examples to focus on terms. This happens in two concrete ways. If the goal type has only a singleton refinement, and said refinement translates directly to a program term (like our example in Section 3.2.1), Myth can simply produce the corresponding program term. This idea generalizes: when all examples utilize the same constructor, Myth immediately generates said constructor. For example, if the goal type is a list, and all output examples are non-empty lists, Myth promptly produces a `Cons` node. Our framework admits the same optimizations.

Figure 3.26 outlines the implementation of `refine` in Myth-tds. We implement focusing by inspecting the goal type for an unspecified hole. If all examples associated with the goal type use the same constructor we generate the corresponding sketch. This property generalizes the intuition that whenever the type has only a single example Myth-tds can simply read off said value. The function `generateConstructor` simply produces a sketch corresponding to the

```haskell
-- MType: Type annotated with examples
refine :: MEnv -> MType -> PTerm MSpec MType -> Checker (PTerm MSpec MType)
refine env typ (Hole Unspecified)
  -- Predicate for types where all examples use the same constructor:
  | usesSingleConstructor typ =
      specHole env typ (generateConstructor env typ c)
  | otherwise = specHole typ Nothing  -- Default case: just enumerate
```

Figure 3.26. Implementing focusing in Myth-tds

structure of the example set. If all examples share a constructor, say `Cons`, but not its arguments, then `generateConstructor` will produce the sketch `Cons` □ □. If the examples are all the same value, then `generateConstructor` will convert it to a program term.

Focusing in Granule

Similarly, Granule relies heavily on focusing. Granule deterministically eliminates graded values whenever possible, eliminates datatypes whenever possible, and introduces datatypes whenever the goal type indicates. We can implement this behavior with our extension to the specifications as well. Given a sum or product type in context, we immediately eliminate it. If the goal type is product, we immediately introduce the appropriate constructor (we cannot introduce sums in the same way). It is worth noting that such aggressive focusing is possible only because Granule does not synthesize programs over recursive data structures.

Figure 3.27 outlines the implementation of this logic in the context of graded modal types via pseudocode for a partial implementation of `refine`. If there is anything we can match on in context – i.e. a graded type, or a sum or product type – we immediately do so. We can also determinstically generate constructors as we did when implementing Myth-tds.

3.5.4 Additional Optimizations

Finally, we will briefly mention two additional optimizations: Prolog-style cuts and condition abduction. If the solution to a given Synthesis goal does not affect any adjacent goals, we can "cut" the solution and prevent backtracking. Implementing such a feature requires

```
-- GType: Graded modal type
refine :: GEnv -> GType -> PTerm GSpec GType -> Checker (PTerm GSpec GType)
refine env typ (Hole Unspecified)
  -- Eliminate any datatypes in context
  | hasMatchable env = return (specHole env typ (generateMatches env))
refine env t@Box{} (Hole Unspecified)
  = return (box hole)              -- Introduce graded value
refine env t@TyApp{} (Hole Unspecified) =
  let ts =
    if isTuple t                     -- Check if t is a product type
      then return (pair hole hole)   -- If so, use smart constructor for tuples
      else allNodes env t            -- Otherwise, guess a node
   in return (specHole env typ (Just ts))
refine env t@TyCon{} (Hole Unspecified) =
  let ts =
    if isUnit t                      -- Check if t is unit
      then return unit               -- If so, return unit
      else allNodes env t            -- Otherwise, guess a node
   in return (specHole env typ (Just ts))
refine env t (Hole Unspecified) =
  specHole env typ Nothing  -- Default case: just enumerate
```

Figure 3.27. Implementing focusing in Granule-tds. Our implementation imports the type and context representations from the original Granule library.

additional information about from the typechecker, as goal-independence is a property of both a specific term and a specific type system. In Synquid, for example, the different cases of a branching statement are all independent. Once we've solved a given branch we will never need to find a different solution. Thus, we extend program specifications to allow the typechecker to mark a given goal as "final". As always, a typechecker does not need to support cuts; failing to mark goals as "final" will not affect completeness. However, doing so can yield performance benefits if the generator reasons about the additional annotations.

Condition abduction is the process of generating a term, then generating a condition under which said term satisfies the synthesis goal [50]. It's vital to the original Synquid implementation, as well as a variety of synthesis tools in different domains [2, 82, 10]. Our framework supports condition abduction; again we must extend the `Typechecker` interface to do so. Abduction requires a typechecker capable of generating conditions that would allow the verification of a given term, so we need to add a method to the typeclass. As usual, it comes with a default implementation and thus does not impact type systems for which abduction does not make sense.

3.6 Qualitative evaluation: Case studies

We begin evaluating `tds` by considering two qualitative criteria:

Ease and flexibility of use: Does using the framework make it easier to implement synthesizers? Is the synthesis paradigm compatible with a variety of type systems?

Expressiveness of combining specifications: Do compositional synthesizers admit more expressive specifications?

To address these questions we discuss four case studies on synthesizers implemented with `tds`, outlined in Table 3.1. We have already discussed the first three at length: we cloned three existing synthesis tools. In Section 3.6.1 we will discuss each implementation in order to

Case Study	Type system	LOC (orig.)	LOC (`tds`)
Synquid-tds	Refinement types	5516	4610
Myth-tds	Input-output examples	7928	1327
Granule-tds	Graded modal types	18024	1195
Mynquid	Refinement types *and* input-output examples	N/A	198

Table 3.1. Case studies and the sizes of their implementations

provide a sense of the development effort involved in each. Each of the three clones presents different implementation challenges, providing a sense of the framework's flexibility.

Our fourth and final case study, Mynquid is a new synthesizer we implemented by *combining* refinement types with input-output examples. In Section 3.6.2 we will show how `tds` allows us to easily experiment with new styles of specification, providing a proof of concept for one way to address some issues with both refinement types and input-output examples.

3.6.1 Implementing real synthesizers with `tds`

The three case studies in Table 3.1 are themselves evidence for the expressiveness of `tds`. Clearly, the framework accommodates three fairly disparate specification styles. Granule-tds in particular imposes some unique requirements. While it does not generate recursive programs, it must be able to quickly generate very large terms via focusing. Moreover, resource constraints are inherently less conducive to local verification, as they introduce dependencies across the entire program. Despite these challenges, our implementation is actually slightly faster of the original tool on benchmarks involving large application terms.

It remains to discuss the process of actually implementing a synthesizer with `tds`. Table 3.1 shows the lines of code we wrote for each of the case studies as compared to the original implementation. This is certainly an incomplete and flawed way to address the question: we are comparing codebases of different languages with different intents. Granule in particular provides a well-developed and uniquely well fleshed out language implementation; the others are single-purpose synthesizers. In order to give a better sense of the effort required to implement a synthesizer with `tds` we will instead qualitatively discuss the development process.

Cloning Synquid

Implmenting Synquid-tds was the easiest development task. The original Synquid implementation is already written in Haskell, and its AST looks much like ours. All we had to do was plug the explicitly bidirectional typechecker into our suite of interfaces – the typechecker already supports polymorphism reusable for reasoning about holes.

Cloning Myth

Implementing Myth-tds was significantly more involved than the previous case study. We had to reimplement the typechecker following the paper and the original OCaml implementation. However, `tds` dramatically simplifies this process. All we had to do was implement a typechecker, we could ignore the more parts of the paper describing all the work the authors put into structuring and optimizing search. Moreover, *testing* and *debugging* the typechecker is much easier than testing a synthesizer. Dealing only with concrete programs during the process of validating the typechecker dramatically simplifies the process.

We did put some effort into our dynamic semantics for holes from Section 3.4.3. However, we could have taken a less precise approach to the semantics of partial functions applied to holes to simplify the task at hand.

Cloning Granule

We also had to implement our own typechecker for the clone of Granule. However, we were able to import all the constraint generation and solving machinery directly from the main Granule repository, also implemented in Haskell. All we had to do was take care of traversing our AST. Moreover, the Granule paper presents a bidirectional algorithm, further simplifying our task [100].

3.6.2 Case Study: Mynquid

We have not yet discussed the final benefit of our synthesis framework. Given two typecheckers for the same term language, we can *combine* them into a new, compositional

typechecker, where terms must well typed in both systems.

Composing type systems in this way enables easy experimentation with new styles of specification. To demonstrate this, we build a synthesizer "Mynquid" that takes a refinement type and a set of concrete input-output examples. This alleviates some issues related to both styles of specification: logical refinements can help make examples less unwieldy, and examples can help strengthen specifications when refinements are insufficient. Most importantly, implementation is nearly trivial, we had only to write a simple frontend to get a usable synthesizer.

As an example, consider a function `reverse` that reverses a list via tail calls. Myth does not support polymorphic types, so the function has type $[\mathsf{Int}] \to [\mathsf{Int}] \to [\mathsf{Int}]$; Myth requires an additional fourteen examples to actually specify the synthesis problem. By adding the following refinement type, we can reduce that to only six examples:

$$\texttt{reverse} \ :: \ xs : [a] \to ys : [a] \to \{[a] \mid \mathsf{elems}\ \nu = \mathsf{elems}\ xs \cup \mathsf{elems}\ ys\}$$

Moreover, using refinement types alone to specify `reverse` is nontrivial. The Synquid specification relies on `snoc` and a complicated specification involving an abstract refinement over an arbitrary predicate between adjacent list elements. It is not clear how to extend the same idea to specifying the tail call version instead – one would have to relate the elements of the two inputs, as well as their relative order.

Table 3.2 lists a few more representative examples of compositional specifications solvable with Mynquid. We do not even know how to specify four of the example functions with a refinement type alone. In other cases the refinement type is very simple, often not even needing any logical annotations. These cases benefit from the additional constraints imposed by polymorphic types – by offloading reasoning about polymorphic values to the Synquid-tds typechecker we eliminate implementatin effort involved in adding polymorphism and polymorphic examples directly to Myth-tds. Additionally, we generally need fewer examples than we do with Myth alone, and the examples we do need are significantly smaller.

Table 3.2. A few examples of Mynquid specifications. For each, we show the refinement type used with Mynquid, as well as the number (N) and size in AST Nodes (S) of the examples sets for both Mynquidand Myth (marked with M). Benchmarks marked with * we do *not* know how to specify with refinement types alone.

Function	Description	Refinement type	N	S	N(M)	S(M)
`reverse*`	Reverse with tail call	$xs : [a] \to ys : [a] \to \{[a] \mid \mathsf{elems}\ \nu = \mathsf{elems}\ xs \cup \mathsf{elems}\ ys\}$	6	42	14	122
`concat`	List concat	$[[a]] \to [a]$	3	20	6	58
`last`	Last element of list	$xs : [a] \to \{\mathsf{Maybe}\ a \mid \mathsf{len}\ xs > 0 \implies \mathsf{isJust}\ \nu\}$	4	19	6	33
`compress`	Remove adj. duplicates	$xs : [a] \to \{[a] \mid \mathsf{elems}\ \nu = \mathsf{elems}\ xs\}$	10	90	13	142
`pairs*`	All ordered pairs	$[a] \to [(a,a)]$	4	36	5	56
`ith`	ith element of list	$[\mathsf{Nat}] \to \mathsf{Nat} \to \mathsf{Nat}$	9	27	24	48
`swap*`	Swap elements pairwise	$[a] \to [a]$	6	46	20	184
`nodes_at*`	Count tree nodes at a level	$\mathsf{Tree}\ a \to \mathsf{Nat} \to \mathsf{Nat}$	12	43	24	90

3.7 Quantitative evaluation: Performance

Experiment selection and setup

To evaluate the performance of our synthesizers, we compare the original implementation of a given synthesizer against a variety of search backends for its clone implemented with `tds`. For each of the three case studies, we use the set of benchmarks from the original paper. Our benchmark suite for Synquid-tds omits three of the original benchmarks, as we did not implement an expressive enough form of conditional abduction to solve them. However, we include four additional benchmarks that are actually harder than any present in the original paper. Many of the original benchmarks involve synthesizing helper functions – that is, they come with concrete refinement types for a useful helper. We took four of the hardest benchmarks and removed the specifications for the helper functions, instead asking the synthesizer to find their definitions *inline*. Doing so makes the benchmarks much more challenging, as it greatly increases the sizes of the solutions. We also added four additional Granule benchmarks: Granule-tds performs very similarly to the original tool on the original benchmark suite, so we wrote four vector benchmarks featuring larger application terms than the originals in order to better separate the tools' performance. We ran all experiments with a 60-second timeout.

When running each of the original tools we used the configuration from their original results. The Granule paper compares several approaches to pruning: additive, subtractive, and "alternate". Rather than re-comparing the three approaches we exclusively ran Granule in alternate mode, as it provides a balanced approach to pruning. Granule-tds supports the same three constraint generation modes; we used the same approach when running our clone in order to ensure a fair comparison.

For each case study we include a plot (Figure 3.28, Figure 3.30, and Figure 3.29) comparing the original tool's performance to that of our clone. Each plot shows time on the x-axis and total number of benchmarks solved on the y-axis. We evaluate a variety of generators; Table 3.3 elaborates on the legend for each plot, providing a brief description of each search

strategy.

Results: Synquid-tds

The depth-first variant of Synquid-tds performs best, though generally not quite as well as the original tool. However, the new benchmarks exposed a soundness bug in Synquid. This highlights the value of approaching synthesis as a typechecking problem: since Synquid-tds typechecks all complete terms it is easy to have confidence in the soundness of the implementation. While this approach has some overhead, it ensures that our tool finds correct programs.

Results: Myth-tds

The depth-first variant of Myth-tds with memoization performs best. Again, it is slightly slower than the original tool, as expected due to the additional typechecking overhead. It is worth pointing out that the original tool assumes that programs recurse on their last argument. We make no such assumption, leading to a larger search space, but ensuring that the tool is forwards-compatible with richer termination metrics.

Results: Granule-tds

We compare Granule-tds against two versions of the original tool: v0.9.0.0 and v0.7.8.0. Here, Granule-tds actually performs slightly better than both. It outperforms them on the new benchmarks that require additional enumeration of application terms. We showed the performance of two versions of Granule in order to highlight another downside of an implementation that integrates synthesis and typechecking. Changes to the typechecker have significant impacts on synthesis performance.

General conclusions

The most apparent takeaway is that iterative deepening trades performance for completeness. It does have utility; it is complete regardless of term depth bounds. However, it is much slower than incomplete alternatives on larger benchmarks.

In general, memoization is only of marginal value, helping on some of the easier Myth

benchmarks. This is a surprising result: memoization is the most common strategy for saving time when solving combinatorial problems. However, our use of `logict` as a basis for implementing nondeterministic search primitives hurts our memo implementation. It is well known that the performance of `logict` deteriorates when one inspects intermediate results with `msplit` [106]. This leads to a dilemma. If we want to generate *all* solutions to a synthesis query we need to use `msplit` to inspect computations. We can only avoid doing so by ensuring that memo keys are strong enough to precisely specify a synthesis problem. Doing so, however, means memoizing against precise types and thus *minimizing* cache hits. Moreover, it is not always straightforward to make memo keys strong enough: Granule-tds stores resource constraints in the state of the typechecker, not the typing context, meaning we would have to include the explicit state of the monadic computation in the memo keys to avoid slowdowns from `msplit`.

The asymptotic costs of inspecting search results hurt our implementation of relevant enumeration in particular. While relevant enumeration increases the rate of cache hits, it does so by making more synthesis queries, thus amplifying inefficiencies associated with monadic backtracking. Further research into monadic backtracking might prove particularly useful here: currently we must choose between a useful memo and efficient backtracking.

We implemented a few other approaches to search as well, but the results were strictly worse than those shown. We compared size and depth as orders for term enumeration: depth always performed better. This would probably no longer be the case on some larger benchmarks; empirical results from other contexts have shown that size tends to scale better [16, 118]. Memoization is available in all modes, but we only include the results for depth-first search as that is the condition under which it ever proved beneficial. We also experimented with some non-trivial cost models to guide search, but found the results too brittle. Learning-based approaches present a clear opportunity for improvement on that front.

Table 3.3. Search backends for `tds`.

Label	Search strategy
ID	Iterative deepening
DF	Depth-first enumeration
DF+M	Depth first enumeration, with memoization
DF+RM	Depth first and *relevant* enumeration with memoization

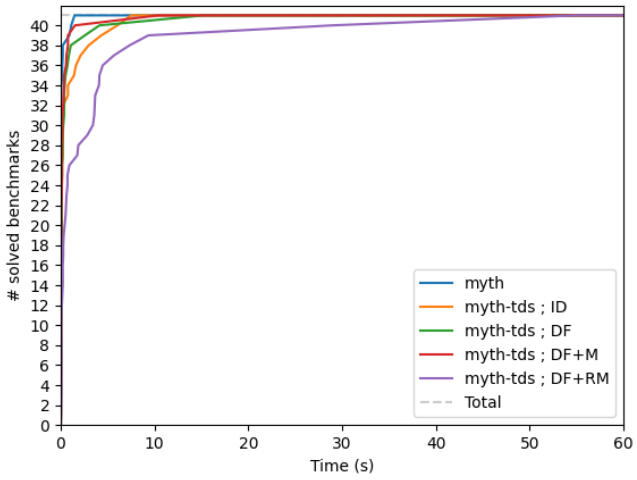

Figure 3.28. Plots of benchmarks solved over time for Myth-tds and the original tool. Search techniques in the legend are defined in Table 3.3.

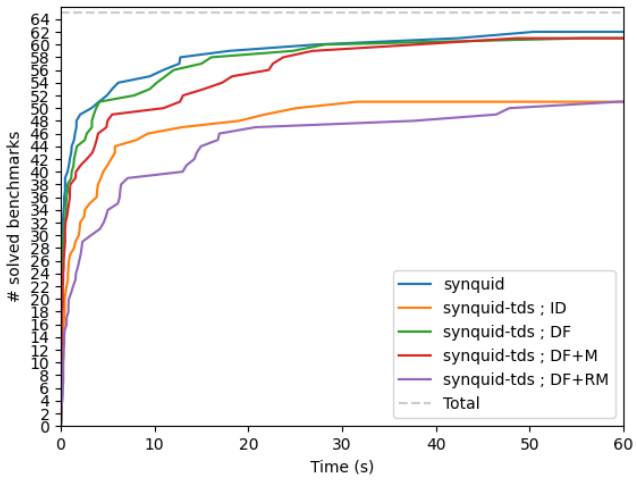

Figure 3.29. Plots of benchmarks solved over time for Synquid-tds and the original tool. Synquid results include one unsound program. Search techniques in the legend are defined in Table 3.3.

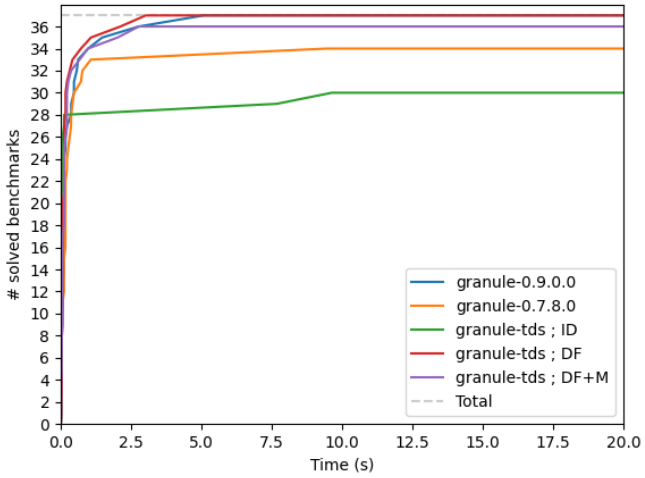

Figure 3.30. Plots of benchmarks solved over time for Granule-tds and the original tool. Search techniques in the legend are defined in Table 3.3. Experiments were run with a 60-second timeout; the graph only shows the first 20 seconds in order to better highlight the differences in performance.

3.8 Related Work

Synthesis frameworks

Perhaps the closest analogue to this work is FlashMeta [108]. FlashMeta also provides a framework, but for building inductive synthesizers in the style of FlashFill [52]. While FlashMeta is capable of generating an industrial-strength synthesizer for a domain-specific language, it is limited to example-based specifications and does not generate looping or recursive programs. Another line of work proposes semantics-guided synthesis: a way to define syntax and semantics within a constraint solver, and get a synthesizer for free [34]. This is in fact even more general – synthesizers are parameterized by a language, semantics, and a solver. Hence there are fewer opportunities to use domain knowledge to implement an efficient solver. We focus instead on rich type systems, as they provide opportunities for pruning while still describing a wide variety of programs.

There is also a variety of work on generic approaches to the *search* part of synthesis. The proof search community, for example, has long been interested in the problem of searching for

proofs in an arbitrary logical context. There are tools for proof search in logical frameworks like λprolog [87, 135]. These approaches are completely generic and work for any calculus expressible in the framework. However, the space of compatible search techniques is very constrained, and it is not clear how to provide domain-specific insight when possible.

Hole-driven design (and synthesis)

Holes are not our invention. A variety of tools use holes in program terms to guide program development, or to take the next step in automation and synthesize terms. GHC, for example, provides typed holes as a development tool, and another project extends this idea to refinement types [1, 112]. Then, the Haskell Wingman tool attempts to fill in Haskell sketches [94], but is specialized to GHC and the Haskell type system. The Hazel programming environment, formalizes the notion of holes via static and dynamic semantics in order to help developers reason about their work-in-progress [99]. A follow-up work uses these sketches to improve upon Myth's need for trace complete example sets [93]. While the techniques for reasoning about holes were a source of inspiration for our own approach in Section 3.4, the synthesis techniques do not generalize to other specification styles. All of these tools do, however, need to typecheck programs with holes.

Using sketches in the synthesis specification process goes back much further – sketching can allow users to provide some insight into program structure [126, 20, 127, 48]. Another project takes the next step and uses a neural network to generate plausible sketches [96]. Finally, other tools even frame synthesis as the problem of generating and rejecting sketches, as we do [38, 95, 102]. All of these techniques inspire our approach to the more general problem of synthesizing programs from a broad spectrum of specifications.

Gradual types

Gradual typing considers the problem of combining static and dynamic type systems [121]. It turns out that this resembles the question of reasoning statically about program sketches: gradually typed programs assign the dynamically typed parts an unknown type, then attempt to

continue with their static reasoning. In fact, research on statically reasoning about programs with holes in the context of an editor relies directly on a gradual types-style formalism [99]. Gradual languages also insert casts in order to ensure that dynamically typed parts of the program throw cast errors at runtime. This part of the process, however, is not as relevant to synthesis.

Our theoretical presentation shares a common goal with gradual type systems: we would like to be able to statically reject ill-typed programs without needing complete type information. Our solution, however differs. Rather than introduce a new type we simply state that holes can have any type, and solve for it via unification. Gradual languages, on the other hand, introduce an unknown type, and then weaken their type-level relations to account for this unknown type. Type equality becomes type consistency [121], and subtyping becomes consistent subtyping [120]. However, defining consistent subtyping in the presence of polymoprphism turns out to be nontrivial [144, 73, 44, 142]. Thus, the gradual-style formalism relying on type consistency is difficult to adapt to new type systems.

Our approach, on the other hand, relies only on well-understood mechanization (that probably already exists in the presence of polymorphism). The problem of rejecting program sketches is simpler than the problem of combining static and dynamic types: we will never execute an incomplete program. Thus, we can use a simpler theoretical approach as well.

3.9 Conclusion

This work provides a foundation for future investigations into type-directed synthesis. Despite the relative simplicity of the generators, our clones of existing synthesizers offer competitive performance and increased confidence in the safety of their outputs. There is clearly room for engineering effort on the search side of `tds`. Our generators are all enumerative: learning-based approaches could dramatically improve performance. Similarly, work on monadic search or a search backend that backtracks explicitly could allow these tools to benefit from more efficient memoization implementations. Finally, our brief experiments with Mynquid show how composi-

tional specifications can alleviate the pain points involved with monolithic specifications. There are numerous opportunities for future work on this front as well: compositional synthesizers currently forgo most opportunities for optimization, and ensuring that the syntax trees match can be nontrivial. Nonetheless, Mynquid clearly opens the door to new interaction styles with a type-directed synthesizer.

Appendix A

Detailed presentation of type systems for `tds` case studies

A.0.1 Language

The full term language is in Figure A.1.

A.0.2 Refinement Types

The type language is in Figure A.2 and the full set of typing rules is in Figure A.4. Any omitted definitions (such as $[\![\Gamma]\!]$) are as in Section 3.4 and [107].

Proposition 9 (Overapproximation)**.** *If* $e \sqsubseteq \hat{e}'$*, and* $\Gamma \vdash e \Leftarrow T$*, then* $\Gamma \vdash \hat{e}' \Leftarrow T$.

Proof. By induction on the derivation of $\Gamma \vdash e \Leftarrow T$.

Case $e = \lambda\ x.\ e_1$**:** If $\hat{e}'$ is a hole, then the result is trivial. Otherwise, $\hat{e}' = \lambda\ x.\ \hat{e}'_1$, so we apply the inductive hypothesis.

Case $e = \texttt{fix}\ x.\ e_1$**:** As above.

$$
\begin{aligned}
e ::= {} & \lambda\ x.\ e \mid \texttt{fix}\ f.\ e \mid x \mid e_1\ e_2 \mid C(e_1,\ldots,e_n) \\
& \mid \texttt{match}\ e\ \texttt{with}\ \mid_i C_i(x_1,\ldots,x_n) \mapsto e_i \\
& \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \Box
\end{aligned}
$$

Figure A.1. Full term language

Case $e = \texttt{match}\ \hat{e}\ \texttt{with}\ \mid_i C_i(\overline{x_i j}^{j}) \mapsto \hat{e}_i$**:** We invoke 9 on $\hat{e}$, and the inductive hypothesis on each case i (when they are not trivial).

Otherwise: If $\hat{e}'$ is not a hole, invoke 9.

□

Lemma 9 (Overapproximation of inference)**.** If $e \sqsubseteq \hat{e}'$, and $\Gamma \vdash e \Rightarrow T$, then $\Gamma \vdash \hat{e}' \Rightarrow T'$ with $\Gamma \vdash T' <: T$.

Proof. By induction on the derivation of $\Gamma \vdash e \Rightarrow T$. Note that 10 we know that $\Gamma \Vdash T$ as long as Γ only contains consistent type (easily enforced in practice).

Case $e = x$**: Case VarSc:** Either $\hat{e}' = \square$ or $\hat{e}' = e$. In the first case we emit type T, known to be consistent.

Case Var$\forall$**:** As above.

Case $e = f\ x$**: Case AppFO:** Either $\hat{e}' = \square$ and we emit type T, or we apply the inductive hypothesis.

Case AppHO: Either $\hat{e}' = \square$ and we emit type T, or we apply Theorem 9.

□

Lemma 10 (Consistency of inferred types)**.** If $\forall x : T \in \Gamma.\Gamma \Vdash T$, then for any e with $\Gamma \vdash e \Rightarrow T$ where $\Gamma \Vdash T$

Proof. By induction on the derivation of $\Gamma \vdash e \Rightarrow T$

Case $e = x$**: Case Var**$\forall$**:** We assume the type constraint solver only produces consistent types T_i – this will be true as long as all types in Γ are consistent.

Case VarSc: Follows from our assumption that all types in Γ are consistent.

Case $e = e_1\ e_2$**: Case AppHO:** By the inductive hypothesis, $\texttt{let}\ C\ \texttt{in}\ T_x \to T$ is consistent, meaning $\texttt{let}\ C\ \texttt{in}\ T$ is consistent by definition.

$$
\begin{aligned}
S &::= \forall \alpha_i.T \mid T \\
T &::= \{B \mid \psi\} \mid x : T_x \to T \mid \texttt{let}\ C\ \texttt{in}\ T \\
B &::= D\ T_i \mid \mathsf{Bool} \mid \mathsf{Int} \\
C &::= \cdot \mid x : T; C \\
\Gamma &::= \cdot \mid x : S; \Gamma \mid \psi; \Gamma
\end{aligned}
$$

Figure A.2. Type language for $\lambda_{\square}^{\psi}$

Case AppFO: By the inductive hypothesis, $\texttt{let}\ C_1\ \texttt{in}\ x : \{B \mid \psi\} \to T$ is consistent, ie T is consistent with C_1 and x in context. Also by the inductive hypothesis, $\texttt{let}\ C_2\ \texttt{in}\ T_x$ is consistent with C_1 in context. Since T_x is *stronger* than $\{B \mid \psi\}$ with context $C_1; C_2$, we see that T is consistent with additional context $C_1; C_2; x : T_x$.

□

A.0.3 Input-Output Examples

The type language for $\lambda_{\square}^{\mathsf{IO}}$ is in Figure A.5, the typing rules are in Figure A.6, some helper functions are in Figure A.7, and some value semantics are in Figure A.8.

Lemma 11 (Overapproximation of evaluation: Input-output examples)**.** If $e \sqsubseteq \hat{e}'$ and $e \longrightarrow^* v$, then $\hat{e}' \longrightarrow^* v'$ where $v \sim v'$

Proof. Let e, $\hat{e}'$ be arbitrary with $e \sqsubseteq \hat{e}'$ and $e \longrightarrow^* v$. We proceed by induction on the derivation of $e \longrightarrow^* v$, considering the final step according to the structure of e. If $\hat{e}' = \square$, then $\hat{e}' \longrightarrow^* \square$, which is consistent with any value v. We consider the other cases below. Note that we do not reason formally about termination, but the implementation includes a structural termination checker.

Case $e = \lambda\ x.\ e_1$**:** Then $\hat{e}' = \lambda\ x.\ \hat{e}'_1$ with $e_1 \sqsubseteq \hat{e}'_1$. Since $e \longrightarrow^* \lambda\ x.\ e_1$, we have $\hat{e}' \longrightarrow^* \lambda\ x.\ \hat{e}'_1$ with a consistent closure body by definition.

$$\boxed{\text{Well-formedness}}$$

$$\text{WF-Scalar}\frac{\Gamma; v : B \vdash \psi}{\Gamma \vdash \{B \mid \psi\}} \qquad \text{WF-Ctx}\frac{\Gamma; C \vdash T}{\Gamma \vdash \texttt{let}\ C\ \texttt{in}\ T}$$

$$\text{WF-FO}\frac{\Gamma \vdash \{B \mid \psi\} \qquad \Gamma; x : \{B \mid \psi\} \vdash T}{\Gamma \vdash x : \{B \mid \psi\} \to T}$$

$$\text{WF-HO}\frac{T_x \text{non-scalar} \qquad \Gamma \vdash T_x \qquad \Gamma \vdash T}{\Gamma \vdash x : T_x \to T}$$

$$\boxed{\text{Subtyping}}$$

$$<:\text{-Refl}\frac{}{\Gamma \vdash B <: B}$$

$$<:\text{-Scalar}\frac{\Gamma \vdash B <: B' \qquad \mathsf{Valid}(\llbracket\Gamma\rrbracket_{\psi \Longrightarrow \psi'} \wedge \psi \implies \psi')}{\Gamma \vdash \{B \mid \psi\} <: \{B' \mid \psi'\}}$$

$$<:\text{-Fun}\frac{\Gamma \vdash T_y <: T_x \qquad \Gamma; y : T_y \vdash [y/x]T <: T'}{\Gamma \vdash x : T_x \to T <: y : T_y \to T'}$$

$$<:\text{-DT}\frac{\Gamma \vdash T_i <: T_i'}{\Gamma \vdash D\ T_i <: D\ T_i'}$$

$$<:\text{-CtxL}\frac{\Gamma; C \vdash T <: T'}{\Gamma \vdash \texttt{let}\ C\ \texttt{in}\ T <: T'} \qquad <:\text{-CtxR}\frac{\Gamma; C \vdash T <: T'}{\Gamma \vdash T <: \texttt{let}\ C\ \texttt{in}\ T'}$$

Figure A.3. Well-formedness and subtyping for refinement types

$$\text{HInfer}\frac{\alpha \text{ fresh}}{\Gamma \vdash \square \Rightarrow \alpha} \quad \text{VarSc}\frac{\Gamma(x) = \{B \mid \psi\}}{\Gamma \vdash x \Rightarrow \{B \mid \psi\}} \quad \text{Var}\forall\frac{\Gamma(x) = \forall\alpha.T \qquad \Gamma \vdash T_i}{\Gamma \vdash x \Rightarrow [T_i/\alpha]T}$$

$$\text{AppFO}\frac{\begin{array}{c}\Gamma \vdash f \Rightarrow \texttt{let}\ C_1\ \texttt{in}\ x : \{B \mid \psi\} \to T \\ \Gamma; C_1 \vdash x \Rightarrow \texttt{let}\ C_2\ \texttt{in}\ T_x \\ \Gamma; C_1; C_2 \vdash T_x <: \{B \mid \psi\}\end{array}}{\Gamma \vdash f\ x \Rightarrow \texttt{let}\ C_1; C_2; x : T_x\ \texttt{in}\ T}$$

$$\text{AppHO}\frac{\begin{array}{c}\Gamma \vdash f \Rightarrow \texttt{let}\ C\ \texttt{in}\ x : T_x \to T \\ \Gamma; C \vdash x \Leftarrow T_x\end{array}}{\Gamma \vdash f\ x \Rightarrow \texttt{let}\ C\ \texttt{in}\ T}$$

$$\text{Fix}\frac{\Gamma; f : S^{\prec} \vdash e \Leftarrow S}{\Gamma \vdash \texttt{fix}\ f.\ e \Leftarrow S} \quad \text{Lam}\frac{\Gamma; y : T_x \vdash e \Leftarrow [y/x]T}{\Gamma \vdash \lambda\ y.\ e \Leftarrow x : T_x \to T} \quad \text{Abs}\frac{\Gamma \vdash e \Leftarrow T \qquad \alpha \text{ not free}}{\Gamma \vdash e \Leftarrow \forall\alpha.T}$$

$$\text{If}\frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow \texttt{let}\ C\ \texttt{in}\ \{\mathsf{Bool} \mid \psi\} \\ \Gamma; C; [\top/v]\psi \vdash e_2 \Leftarrow T \qquad \Gamma; C; [\bot/v]\psi \vdash e_3 \Leftarrow T\end{array}}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Leftarrow T}$$

$$\text{Match}\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow \texttt{let}\ C\ \texttt{in}\ \{D\ T_k \mid \psi\} \\ \Gamma(C_i) = x : T_i^{\,j} \to \{D\ T_k \mid \psi_i'\} \qquad \Gamma_i = \{x_j : T_i^{\,j}\}; [x'/v]\psi_i' \\ \Gamma; C; [x'/v]\psi; \Gamma_i \vdash e_i \Leftarrow T\end{array}}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \ |_i\, C_i(\overline{x_j}^{\,j}) \mapsto e_i \Leftarrow T}$$

Figure A.4. Typing rules for $\lambda_{\square}^{\psi}$

$$\begin{aligned}
T &::= D\ x \mid T \to T \\
v &::= \square \mid \|v\| \mid C(v_1, \ldots, v_n) \mid \lambda\ x.\ e \mid pf \\
ex &::= C(ex_1, \ldots, ex_n) \mid pf \\
pf &::= v_1 \rightsquigarrow ex_1; \ldots; v_n \rightsquigarrow ex_n \\
\sigma &::= [v \mapsto x]\sigma \\
X &::= \cdot \mid \sigma \vdash ex; X \\
\Gamma &::= \cdot \mid x : T; \Gamma
\end{aligned}$$

Figure A.5. Type language for $\lambda_{\square}^{\mathsf{IO}}$

$$\text{Satisfies}\ \frac{\forall \sigma \vdash ex \in X.\sigma(\hat{e}) \longrightarrow^* v \wedge v \sim ex}{\hat{e} \models X}$$

$$\text{Var}\ \frac{\Gamma(x) = T}{\Gamma \vdash x \Rightarrow T} \qquad \text{App}\ \frac{\Gamma \vdash \hat{f} \Rightarrow T \to T' \qquad \Gamma \vdash \hat{x} \Leftarrow T \triangleright \langle \cdot \rangle}{\Gamma \vdash \hat{f}\ \hat{x} \Rightarrow T'}$$

$$\text{IE}\ \frac{\Gamma \vdash \hat{e} \Rightarrow T \qquad \hat{e} \models X}{\Gamma \vdash \hat{e} \Leftarrow T \triangleright \langle X \rangle}$$

$$\text{Fix}\ \frac{\begin{array}{c} X = \sigma_1 \vdash pf_1; \ldots; \sigma_n \vdash pf_n \\ X' = \mathsf{recurse}(f, \sigma_1, pf_1), \ldots, \mathsf{recurse}(f, \sigma_n, pf_n) \\ \Gamma; f : T \vdash \hat{e} \Leftarrow T \triangleright \langle X' \rangle \end{array}}{\Gamma \vdash \texttt{fix}\ f.\ \hat{e} \Leftarrow T \triangleright \langle X \rangle}$$

$$\text{Lam}\ \frac{\begin{array}{c} X = \sigma_1 \vdash pf_1; \ldots; \sigma_n \vdash pf_n \qquad X' = \mathsf{apply}(x, \sigma_1, pf_1); \ldots; \mathsf{apply}(x, \sigma_n, pf_n) \\ \Gamma; x : T \vdash \hat{e} \Leftarrow T' \triangleright \langle X' \rangle \end{array}}{\Gamma \vdash \lambda\ x.\ \hat{e} \Leftarrow T \to T' \triangleright \langle X \rangle}$$

$$\text{Cons}\ \frac{\begin{array}{cc} X = \overline{\sigma_i \vdash C(ex_{1i}, \ldots, ex_{ki})}^{\,i} & \Gamma(C) = T_1 \to \ldots \to T_k \to D\ x \\ \mathsf{proj}(X) = X_1, \ldots, X_k & \Gamma \vdash \hat{e}_i \Leftarrow T_i \triangleright \langle X_i \rangle \end{array}}{\Gamma \vdash C((\hat{e}_1, \ldots, \hat{e}_n)) \Leftarrow D\ x \triangleright \langle X \rangle}$$

$$\text{Match}\ \frac{\begin{array}{c} \Gamma \vdash e \Rightarrow T \qquad \mathsf{distribute}(\Gamma, T, X, e) = \overline{(b_i, X_i')}^{\,i} \\ \overline{\mathsf{binders}(\Gamma, b_i) = \Gamma_i}^{\,i} \qquad \Gamma; \Gamma_i \vdash \hat{e}_i \Leftarrow T \triangleright \langle X_i' \rangle \end{array}}{\Gamma \vdash \texttt{match}\ \hat{e}\ \texttt{with}\ \ |_i\, C_i(x_{i1}, \ldots, x_{in}) \mapsto \hat{e}_i \Leftarrow \hat{T} \triangleright \langle X \rangle}$$

Figure A.6. Typing rules for $\lambda^{\mathsf{IO}}_{\square}$

$$\mathsf{recurse}(f,\sigma,pf) = [pf \mapsto f]\sigma \vdash pf$$

$$\mathsf{apply}(x,\sigma,pf) = \overline{[v_i \mapsto x]\sigma \vdash ex_i}^{\,i}$$
$$\text{where } pf = \overline{v_i \rightsquigarrow ex_i}^{\,i}$$

$$\mathsf{proj}(X) = \overline{\sigma_i \vdash ex_{1i}}^{\,i}, \ldots, \overline{\sigma_i \vdash ex_{ni}}$$
$$\text{where } X = \overline{\sigma_i \vdash C(ex_{1i},\ldots,ex_{ni})}^{\,i}$$

$$\mathsf{distribute}(\Gamma,T,X,\hat{e}) = (b_1,X_1'),\ldots,(b_n,X_n')$$
$$\text{where } \mathsf{ctors}(\Gamma,T) = C_1,\ldots,C_n$$
$$\forall i \in 1,\ldots,n.p_1 = \mathsf{pattern}(\Gamma,C_i)$$
$$\forall i \in 1,\ldots,n.X_i' = [\sigma'\sigma \mapsto ex \mid \sigma \mapsto ex \in X, \sigma(E) \longrightarrow^* C_i(\overline{ex}), \mathsf{vbinders}(p_i,\overline{ex}) = \sigma']$$

$$\mathsf{binders}(\Gamma,C(x_1,\ldots,x_n)) = x_1 : T_1,\ldots,x_n : T_n$$
$$\text{where } \Gamma(C) = T_1 \to \ldots \to T_n \to D\ x$$

$$\mathsf{ctors}(\Gamma,T) = C_1,\ldots,C_n$$
$$\text{where } \forall i \in [1,n].C_i : \ldots \to T \in \Gamma$$

$$\mathsf{pattern}(\Gamma,T) = C(x_1,\ldots,x_n)$$
$$\text{where } C :: T_1 \to \ldots \to T_n \to T \in \Gamma$$

$$\mathsf{vbinders}(C(x_1,\ldots,x_n),\hat{e}_1,\ldots,\hat{e}_n) = [\hat{e}_1 \mapsto x_1]\ldots[\hat{e}_n \mapsto x_n]$$

Figure A.7. Auxiliary functions for manipulating examples in $\lambda_{\square}^{\mathsf{IO}}$

$$\text{PFApp}\ \frac{v \neq \square \qquad v_j \sim v \qquad v_j \in \overline{v_i}^{\,i}}{\overline{v_i \rightsquigarrow ex_i}^{\,i} \longrightarrow vex_j}$$

$$\text{PFAppAnyR1}\ \frac{}{\overline{v_i \rightsquigarrow ex_i}^{\,i}\ \square \longrightarrow \|\overline{ex_i}^{\,i}\|} \qquad \text{PFAppAnyL}\ \frac{}{\square\ v \longrightarrow \square}$$

$$\text{OneOfApp}\ \frac{\overline{w_j}^{\,j} = \cup_i\{v' \mid v_i\ v \longrightarrow^* v'\}}{\|\overline{v_i}^{\,i}\|\ v \longrightarrow \|w_j\|^j}$$

$$\text{ClosAny}\ \frac{v \text{ contains } \square}{(\lambda\ x.\ e)\ v \longrightarrow \square} \qquad \text{ClosOneOf}\ \frac{(\lambda\ x.\ e)\ v_i \longrightarrow^* v_i'}{(\lambda\ x.\ e)\ \|\overline{v_i}\| \longrightarrow \|v_i'\|}$$

Figure A.8. Selected evaluation rules on values. We omit the case of applying a closure value to a complete value, in which case we rely on the semantics of the term language. We also omit the evaluation of constructors, which occurs component-wise.

$$\text{Refl}\frac{}{v \sim v} \quad \text{AnyL}\frac{}{\Box \sim v} \quad \text{AnyR}\frac{}{v \sim \Box}$$

$$\text{OneOf}\frac{\exists j.(v_j \sim v \wedge v_j \in \overline{v_i}^i)}{\|\overline{v_i}^i\| \sim v} \quad \text{Data}\frac{\forall i < k.v_i \sim v'_i}{C(\overline{v_i}^{i<k}) \sim C(\overline{v'_i}^{i<k})}$$

$$\text{PFPF}\frac{\forall i.\exists j.v_i \sim v'_j \wedge ex_i \sim ex'_j \qquad \forall j.\exists i.v_i \sim v'_j \wedge ex_i \sim ex'_j}{\overline{v_i \rightsquigarrow ex_i}^i \sim \overline{v'_j \rightsquigarrow ex'_j}^j}$$

$$\text{PFFix}\frac{\forall i.(\lambda\ x.\ e)\ v_i \longrightarrow^* v \wedge v \sim ex_i}{\overline{v_i \rightsquigarrow ex_i}^i \sim \lambda\ x.\ e}$$

Figure A.9. Value consistency rules in $\lambda_{\Box}^{\mathsf{IO}}$

Case $e = \texttt{match}\ e_s\ \texttt{with}\ \mid_i C_i(x_{i1},\ldots,x_{in}) \mapsto e_i$**:** Then $\hat{e}' = \texttt{match}\ \hat{e}'_s\ \texttt{with}\ \mid_i C_i(x_{i1},\ldots,x_{in}) \mapsto \hat{e}'_i$ with $e_s \sqsubseteq \hat{e}'_s$ and all $e_i \sqsubseteq \hat{e}'_i$. By the inductive hypothesis, $e_s \longrightarrow^* v_s$ and $\hat{e}'_s \longrightarrow^* v'_s$, where $v_s \sqsubseteq v'_s$. Let $S_{x_i,v}(e)$ be the expression resulting by substituting the proper value for each x_i in e when the match scrutinee evaluates to v. Then we have $S_{x_i,v_s} \sqsubseteq S_{x_i,v'_s}(e_i)$, so the conclusion follows by the inductive hypothesis.

Case $e = C(e_1,\ldots,e_n)$**:** Then $\hat{e}' = C(\hat{e}'_1,\ldots,\hat{e}'_n)$ with all $e_i \sqsubseteq \hat{e}_i{}'$. We have $C(e_1,\ldots,e_n) \longrightarrow^* C(v_1,\ldots,v_n)$, where $e_i \longrightarrow^* v_i$. We apply the inductive hypothesis on each of these e_i and v_i.

Case $e = e_1\ e_2$**:** Then $\hat{e}' = \hat{e}'_1\ \hat{e}'_2$ with $e_1 \sqsubseteq \hat{e}'$ and $e_2 \sqsubseteq \hat{e}'_2$.

Case $e_1 = \overline{v_i \rightsquigarrow ex_i}^i$**:** Then $e_2 \longrightarrow^* v_2$ where $v_2 \sim v_j$ for some j, and $v_j \rightsquigarrow v \in e_1$. First, note that $\hat{e}'_2 \longrightarrow^* v'_2$ where $v'_2 \sim v_2$ by the inductive hypothesis. There are two cases for $\hat{e}'_1$: either evaluates to a partial function or a set of partial functions. We consider only the former case: the latter reduces to the former by the definition of consistency on value sets. Then $\hat{e}'_1 \longrightarrow^* \overline{v_k \rightsquigarrow ex_k}^k$, where there exists some l with $v_l \rightsquigarrow ex_l \sim v_j \rightsquigarrow v$. Thus, $v'_2 \sim v_l$, and since $ex_l \sim v$ the entire value is consistent with v.

Case $e_1 = \lambda\ x.\ e'_1$**:** Then $\hat{e}'_1 = \lambda\ x.\ \hat{e}''_1$ with $e'_1 \sqsubseteq \hat{e}''_1$. If $\hat{e}'_2$ contains $\Box$, the entire term

$$
\begin{array}{l}
\hat{e} ::= \Box \mid 1 \mid \lambda\, x.\, e \mid [\hat{e}] \mid (\hat{e}, \hat{e}) \mid \mathsf{Left}\ \hat{e} \mid \mathsf{Right}\ \hat{e} \mid \hat{e}\ \hat{e} \mid \mathsf{let}\ 1 = \hat{e}\ \mathsf{in}\ \hat{e} \\
\quad \mid \mathsf{let}\ [x] = \hat{e}\ \mathsf{in}\ \hat{e} \mid \mathsf{let}\ (x_1, x_2) = \hat{e}\ \mathsf{in}\ \hat{e} \mid \mathsf{case}\ \hat{e}\ \mathsf{of}\ \mathsf{Left}\ x_1 \to \hat{e} \mid \mathsf{Right}\ x_2 \to \hat{e} \\
T ::= 1 \mid T \multimap T \mid T \oplus T \mid T \times T \mid \blacksquare_r T \\
\Gamma ::= \cdot \mid x : T; \Gamma \mid x : [T]_r; \Gamma
\end{array}
$$

Figure A.10. Graded modal types and terms

evaluates to $\Box$. Otherwise, $\hat{e}_2' = e_2$. In this case, we rely on the inductive hypothesis after substituting e_2 for x in $\hat{e}_1''$.

□

A.0.4 Graded Modal Types

Figure A.10 contains the full type language and corresponding term language (which we embed in the shared language). Figure A.11 contains the full set of typing rules. Other operations are as defined in [72]. Theorem 8 and 6 apply directly to $\lambda_{\Box}^{\multimap}$, so we omit their proofs as we rely exclusively on unification.

$$\text{Sym}\frac{}{x : T \vdash x \Rightarrow T} \quad \text{Hole}\frac{}{\Gamma \vdash \square \Rightarrow T} \quad \text{IE}\frac{\Gamma \vdash \hat{e} \Rightarrow T}{\Gamma \vdash \hat{e} \Leftarrow T}$$

$$\text{App}\frac{\Gamma_1 \vdash \hat{e}_1 \Rightarrow A \multimap B \qquad \Gamma_2 \vdash \hat{e}_2 \Leftarrow A}{\Gamma_1 + \Gamma_2 \vdash \hat{e}_1\ \hat{e}_2 \Rightarrow B} \quad \text{Lam}\frac{\Gamma; x : A \vdash \hat{e} \Leftarrow B}{\Gamma \vdash \lambda\ x.\ \hat{e} \Leftarrow A \multimap B}$$

$$\text{WeakS}\frac{\Gamma \vdash \hat{e} \Rightarrow B}{\Gamma; x : [A]_0 \vdash \hat{e} \Rightarrow B} \quad \text{DerS}\frac{\Gamma; x : A \vdash \hat{e} \Rightarrow B}{\Gamma; x : [A]_1 \vdash \hat{e} \Rightarrow B} \quad \text{PrS}\frac{[\Gamma] \vdash \hat{e} \Rightarrow B}{r \times [\Gamma] \vdash \hat{e} \Rightarrow \blacksquare_r B}$$

$$\text{WeakC}\frac{\Gamma \vdash \hat{e} \Leftarrow B}{\Gamma; x : [A]_0 \vdash \hat{e} \Leftarrow B} \quad \text{DerC}\frac{\Gamma; x : A \vdash \hat{e} \Leftarrow B}{\Gamma; x : [A]_1 \vdash \hat{e} \Leftarrow B} \quad \text{PrC}\frac{[\Gamma] \vdash \hat{e} \Leftarrow B}{r \times [\Gamma] \vdash \hat{e} \Leftarrow \blacksquare_r B}$$

$$\text{Let}\blacksquare\frac{\Gamma_1 \vdash \hat{e}_1 \Leftarrow [A]_r \qquad \Gamma_2; x : [A]_r \vdash \hat{e}_2 \Leftarrow B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ [x] = \hat{e}_1\ \mathsf{in}\ \hat{e}_2 \Leftarrow B} \quad 1\frac{}{\cdot \vdash 1 \Rightarrow 1}$$

$$\text{Let1}\frac{\Gamma_1 \vdash \hat{e}_1 \Leftarrow 1 \qquad \Gamma_2 \vdash \hat{e}_2 \Leftarrow B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ 1 = \hat{e}_1\ \mathsf{in}\ \hat{e}_2 \Leftarrow B}$$

$$\text{Pair}\frac{\Gamma_1 \vdash \hat{e}_1 \Leftarrow A \qquad \Gamma_2 \vdash \hat{e}_2 \Leftarrow B}{\Gamma_1 + \Gamma_2 \vdash (\hat{e}_1, \hat{e}_2) \Leftarrow A \times B} \quad \text{PairElim}\frac{\Gamma_1 \vdash \hat{e}_1 \Leftarrow A \times B \Gamma_2; x_1 : A; x_2 : B \vdash \hat{e}_2 \Leftarrow C}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ (x_1, x_2) = \hat{e}_1\ \mathsf{in}\ \hat{e}_2 \Leftarrow C}$$

$$\text{ApproxC}\frac{\Gamma; x : [A]_r; \Gamma' \vdash \hat{e} \Leftarrow A \qquad r \sqsubseteq s}{\Gamma; x : [A]_s; \Gamma' \vdash \hat{e} \Leftarrow A} \quad \text{ApproxS}\frac{\Gamma; x : [A]_r; \Gamma' \vdash \hat{e} \Rightarrow A \qquad r \sqsubseteq s}{\Gamma; x : [A]_s; \Gamma' \vdash \hat{e} \Rightarrow A}$$

$$\text{InL}\frac{\Gamma \vdash \hat{e} \Leftarrow A}{\Gamma \vdash \mathsf{Left}\ \hat{e} \Leftarrow A \oplus B} \quad \text{InR}\frac{\Gamma \vdash \hat{e} \Leftarrow B}{\Gamma \vdash \mathsf{Right}\ \hat{e} \Leftarrow A \oplus B}$$

$$\text{SumElim}\frac{\Gamma_1 \vdash \hat{e}_1 \Leftarrow A \oplus B \qquad \Gamma_2; x_1 : A \vdash \hat{e}_2 \Leftarrow C \qquad \Gamma_3; x_2 : B \vdash \hat{e}_3 \Leftarrow C}{\Gamma_1 + (\Gamma_2 \sqcup \Gamma_3) \vdash \mathsf{case}\ \hat{e}_1\ \mathsf{of}\ \mathsf{Left}\ x_1 \to \hat{e}_2\ \mid \mathsf{Right}\ x_2 \to \hat{e}_3 \Leftarrow C}$$

Figure A.11. Typing rules for $\lambda^{\multimap}_{\square}$

Bibliography

[1] Typed holes in ghc, 2014.

[2] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *CAV*, 2013.

[3] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *J. Automated Reasoning*, 46, February 2011.

[4] E. Albert, J. C. Fernández, and G. Román-Díez. Non-cumulative Resource Analysis. In *Tools and Algs. for the Construct. and Anal. of Syst. (TACAS'15)*, 2015.

[5] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. Automatic Inference of Resource Consumption Bounds. In *Logic for Programming, Artificial Intelligence, and Reasoning, 18th Conference (LPAR'12)*, pages 1–11, 2012.

[6] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, pages 161–203, 2011.

[7] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theor. Comput. Sci.*, 413(1):142 – 159, 2012.

[8] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *17th Int. Static Analysis Symposium (SAS'10)*, pages 117–133, 2010.

[9] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.

[10] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *TACAS (Part I)*, volume 10205 of *LNCS*, pages 319–336. Springer, 2017.

[11] ALAN ROSS Anderson, NUEL D. BELNAP, J. MICHAEL DUNN, Kit Fine, Alasdair Urquhart, Daniel Cohen, Steve Giambrone, Dorothy L. Grover, Anil Gupta, Glen Helman,

Errol P. Martin, Michael A. McRobbie, Stuart Shapiro, and Robert G. Wolf. *Entailment, Vol. II: The Logic of Relevance and Necessity*. Princeton University Press, 1992.

[12] Jan-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2:297–347, 1992.

[13] Lex Augusteijn. Sorting morphisms. In S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques, editors, *Advanced Functional Programming*, pages 1–27, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[14] M. Avanzini and G. Moser. A Combination Framework for Complexity. In *Int. Conf. on Rewriting Techniques and Applications (RTA'13)*, 2013.

[15] Martin Avanzini, Ugo Dal Lago, and Georg Moser. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *29th Int. Conf. on Functional Programming (ICFP'15)*, 2012.

[16] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[17] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation*, 2015.

[18] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10)*, pages 103–118, 2010.

[19] Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 140–156, 2009.

[20] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. *SIGPLAN Not.*, 51(1):775–788, January 2016.

[21] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Algs. for the Construct. and Anal. of Syst. (TACAS'14)*, 2014.

[22] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Alg. for the Constr. and Anal. of Systems - 20th Int. Conf. (TACAS'14)*, pages 140–155, 2014.

[23] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, page 351–370, Berlin, Heidelberg, 2014. Springer-Verlag.

[24] Pavol Cerný, Krishnendu Chatterjee, Thomas A. Henzinger, Arjun Radhakrishna, and Rohit Singh. Quantitative synthesis for concurrent programs. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 243–259, 2011.

[25] Pavol Cerný, Edmund M. Clarke, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, Roopsha Samanta, and Thorsten Tarrach. From non-preemptive to preemptive scheduling using synchronization synthesis. In *CAV*, 2015.

[26] Pavol Cerný, Thomas A. Henzinger, Laura Kovács, Arjun Radhakrishna, and Jakob Zwirchmayr. Segment Abstraction for Worst-Case Execution Time Analysis. In *24th European Symposium on Programming (ESOP'15)*, pages 105–131, 2015.

[27] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. *SIGPLAN Not.*, 48(6):3–14, June 2013.

[28] E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann. Relational Cost Analysis. In *Princ. of Prog. Lang. (POPL'17)*, 2017.

[29] Ezgi Cicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational Cost Analysis. In *44th Symposium on Principles of Programming Languages (POPL'17)*, 2017.

[30] Ezgi Çiçek, Weihao Qu, Gilles Barthe, Marco Gaboardi, and Deepak Garg. Bidirectional type checking for relational properties. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 533–547. ACM, 2019.

[31] Nils Anders Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *35th ACM Symp. on Principles Prog. Langs. (POPL'08)*, pages 133–144, 2008.

[32] N. Danner, D. R. Licata, and R. Ramyaa. Denotational Cost Semantics for Functional Languages with Inductive Types. In *Int. Conf. on Functional Programming (ICFP'15)*, 2015.

[33] Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational Cost Semantics for Functional Languages with Inductive Types. In *29th Int. Conf. on Functional Programming (ICFP'15)*, 2012.

[34] Loris D'Antoni, Qinheping Hu, Jinwoo Kim, and Thomas Reps. Programmable program synthesis. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 84–109, Cham, 2021. Springer International Publishing.

[35] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[36] Ewen Denney. *A theory of program refinement*. PhD thesis, University of Edinburgh, UK, 1999.

[37] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 345–356, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 420–435, 2018.

[39] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 422–436, 2017.

[40] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex apis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612, 2017.

[41] Kostas Ferles, Jacob Van Geffen, Isil Dillig, and Yannis Smaragdakis. Symbolic reasoning for automatic signal placement. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 120–134, 2018.

[42] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Programming Language Design and Implementation (PLDI)*, 2015.

[43] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *POPL*, 2016.

[44] Ronald Garcia and Matteo Cimini. Principal type schemes for gradual programs. *SIGPLAN Not.*, 50(1):303–315, jan 2015.

[45] Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, page 331–350, Berlin, Heidelberg, 2014. Springer-Verlag.

[46] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[47] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: A modular approach to polynomial-time computability. *Theor. Comput. Sci.*, 97(1):1–66, 1992.

[48] Matthías Páll Gissurarson. Suggesting valid hole fits for typed-holes (experience report). *SIGPLAN Not.*, 53(7):179–185, sep 2018.

[49] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Princ. of Prog. Lang. (POPL'09)*, 2009.

[50] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, August 2012.

[51] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-Flow Refinement and Progress Invariants for Bound Analysis. In *Conf. on Prog. Lang. Design and Impl. (PLDI'09)*, pages 375–385, 2009.

[52] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73, 2011.

[53] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, pages 127–139, 2009.

[54] Sumit Gulwani and Florian Zuleger. The Reachability-Bound Problem. In *Conf. on Prog. Lang. Design and Impl. (PLDI'10)*, pages 292–304, 2010.

[55] Ashutosh Gupta, Thomas A. Henzinger, Arjun Radhakrishna, Roopsha Samanta, and Thorsten Tarrach. Succinct representation of concurrent trace sets. In *POPL*, 2015.

[56] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, 2013.

[57] Martin A. T. Handley, Niki Vazou, and Graham Hutton. Liquidate your assets: reasoning about resource usage in liquid haskell. *PACMPL*, 4(POPL):24:1–24:27, 2020.

[58] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

[59] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *Princ. of Prog. Lang. (POPL'11)*, 2011.

[60] J. Hoffmann, A. Das, and S.-C. Weng. Towards Automatic Resource Bound Analysis for OCaml. In *Princ. of Prog. Lang. (POPL'17)*, 2017.

[61] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *European Symp. on Programming (ESOP'10)*, 2010.

[62] Jan Hoffmann. Finding a Tree Structure in a Resolution Proof is NP-Complete. *Theoretical Computer Science*, 410(21-23), 2009.

[63] Jan Hoffmann. RAML Web Site. http://raml.co/, 2018.

[64] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. In *38th Symposium on Principles of Programming Languages (POPL'11)*, 2011.

[65] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. In *38th Symp. on Principles of Prog. Langs. (POPL'11)*, pages 357–370, 2011.

[66] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource Aware ML. In *24rd International Conference on Computer Aided Verification (CAV'12)*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012.

[67] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards Automatic Resource Bound Analysis for OCaml. In *44th Symposium on Principles of Programming Languages (POPL'17)*, 2017.

[68] Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polynomial Potential - A Static Inference of Polynomial Bounds for Functional Programs. In *In Proceedings of the 19th European Symposium on Programming (ESOP'10)*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010.

[69] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Princ. of Prog. Lang. (POPL'03)*, 2003.

[70] M. Hofmann and G. Moser. Multivariate Amortised Resource Analysis for Term Rewrite Systems. In *Int. Conf. on Typed Lambda Calculi and Applications (TLCA'15)*, 2015.

[71] Martin Hofmann and Steffen Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, pages 185–197, 2003.

[72] Jack Hughes and Dominic Orchard. Resourceful Program Synthesis from Graded Linear Types (Appendix), December 2020. This work is supported by an EPSRC Doctoral Training Award and EPSRC grant EP/T013516/1 (Verifying Resource-like Data Use in Programs via Types).

[73] Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On polymorphic gradual typing. *Proc. ACM Program. Lang.*, 1(ICFP), aug 2017.

[74] Jeevana Priya Inala, Nadia Polikarpova, Xiaokang Qiu, Benjamin S. Lerner, and Armando Solar-Lezama. Synthesis of recursive ADT transformations from reusable templates. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 247–263, 2017.

[75] Jeevana Priya Inala, Rohit Singh, and Armando Solar-Lezama. Synthesis of domain specific CNF encoders for bit-vector solvers. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 302–320, 2016.

[76] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *Princ. of Prog. Lang. (POPL'10)*, 2010.

[77] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)*, pages 223–236, 2010.

[78] Ming Kawaguchi, Patrick Maxim Rondon, and Ranjit Jhala. Type-based data structure verification. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 304–315, 2009.

[79] Z. Kincaid, J. Breck, A. F. Boroujeni, and T. Reps. Compositional Recurrence Analysis Revisited. In *Prog. Lang. Design and Impl. (PLDI'17)*, 2017.

[80] Z. Kincaid, J. Cyphert, J. Breck, and T. Reps. Non-linear Reasoning for Invariant Synthesis. In *Princ. of Prog. Lang. (POPL'19)*, 2019.

[81] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). *SIGPLAN Not.*, 40(9):192–203, sep 2005.

[82] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.

[83] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. *CoRR*, abs/1904.07415, 2019.

[84] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 253–268, New York, NY, USA, 2019. Association for Computing Machinery.

[85] Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann, and Nadia Polikarpova. Liquid resource types (extended version). 2020.

[86] Kenneth Knowles and Cormac Flanagan. Compositional reasoning and decidable checking for dependent contract types. In *PLPV*, 2009.

[87] Michael Kohlhase, Florian Rabe, Claudio Sacerdoti Coen, and Jan Frederik Schaefer. Logic-independent proof search in logical frameworks: (short paper). In *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, pages 395–401, Berlin, Heidelberg, 2020. Springer-Verlag.

[88] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. In *Symposium on Principles of Programming Languages (POPL'15)*, 2015.

[89] U. D. Lago and M. Gaboardi. Linear Dependent Types and Relative Completeness. In *Logic in Computer Science (LICS'11)*, 2011.

[90] Ugo Dal Lago and Marco Gaboardi. Linear Dependent Types and Relative Completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS'11)*, pages 133–142, 2011.

[91] Calvin Loncaric, Michael D. Ernst, and Emina Torlak. Generalized data structure synthesis. In *ICSE*, 2018.

[92] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast synthesis of fast collections. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 355–368, New York, NY, USA, 2016. ACM.

[93] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. Program sketching with live bidirectional evaluation. *Proc. ACM Program. Lang.*, 4(ICFP):109:1–109:29, 2020.

[94] Sandy Maguire. Wingman for haskell, 2021.

[95] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. Bottom-up synthesis of recursive functional programs using angelic execution. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[96] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *ICLR*, 2018. To appear.

[97] V. C. Ngo, Mario Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. Verifying and Synthesizing Constant-Resource Implementations with Types. In *Symp. on Sec. and Privacy (SP'17)*, 2017.

[98] L. Noschinski, F. Emmes, and J. Giesl. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *J. Automated Reasoning*, 51, June 2013.

[99] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: A bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 86–99, New York, NY, USA, 2017. Association for Computing Machinery.

[100] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[101] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.

[102] Shankara Pailoor, Yuepeng Wang, Xinyu Wang, and Isil Dillig. Synthesizing data structure refinements from integrity constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 574–587, New York, NY, USA, 2021. Association for Computing Machinery.

[103] Adam Chlipala Peng Wang, Di Wang. Timl: A functional language for practical complexity analysis with invariants. In *OOPSLA*, 2017.

[104] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 396–407, New York, NY, USA, 2014. ACM.

[105] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, and Dinakar Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 297–310, 2016.

[106] Atze van der Ploeg and Oleg Kiselyov. Reflection without remorse: Revealing a hidden sequence to speed up monadic reflection. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, page 133–144, New York, NY, USA, 2014. Association for Computing Machinery.

[107] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Programming Language Design and Implementation (PLDI)*, pages 522–538, 2016.

[108] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 107–126, New York, NY, USA, 2015. Association for Computing Machinery.

[109] Xiaokang Qiu and Armando Solar-Lezama. Natural synthesis of provably-correct data-structure manipulations. *PACMPL*, 1(OOPSLA):65:1–65:28, 2017.

[110] Ivan Radicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. Monadic refinements for relational cost analysis. *PACMPL*, 2(POPL):36:1–36:32, 2018.

[111] Ivan Radicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. Monadic refinements for relational cost analysis. *PACMPL*, 2(POPL):36:1–36:32, 2018.

[112] Patrick Redmond, Gan Shen, and Lindsey Kuper. Toward hole-driven development with liquid haskell, 2021.

[113] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. Refutation-based synthesis in SMT. *Formal Methods Syst. Des.*, 55(2):73–102, 2019.

[114] Patrick Maxim Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala. Csolve: Verifying C with liquid types. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 744–750, 2012.

[115] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *PLDI*, 2008.

[116] A. Sabry and M. Felleisen. Reasoning about Programs in Continuation-Passing Style. In *LISP and Functional Programming (LFP'92)*, 1992.

[117] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 305–316, 2013.

[118] Rohin Shah, Sumith Kulal, and Rastislav Bodik. Scalable synthesis with symbolic syntax graphs.

[119] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. Conditionally correct superoptimization. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 147–162, 2015.

[120] Jeremy Siek and Walid Taha. Gradual typing for objects. pages 2–27, 08 2007.

[121] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, pages 81–92, 2006.

[122] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, pages 274–293, 2015.

[123] Hugo R. Simões, Pedro B. Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *17th Int. Conf. on Funct. Prog. (ICFP'12)*, pages 165–176, 2012.

[124] Moritz Sinn, Florian Zuleger, and Helmut Veith. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th Int. Conf. (CAV'14)*, page 743–759, 2014.

[125] Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. In *PLDI*, pages 326–340. ACM, 2016.

[126] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[127] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[128] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 313–326, 2010.

[129] Nikhil Swamy, Cătălin Hriundefinedcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in f*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 256–270, New York, NY, USA, 2016. Association for Computing Machinery.

[130] R. E. Tarjan. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods*, 6, August 1985.

[131] Robert Endre Tarjan. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods*, 6(2):306–318, 1985.

[132] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 54, 2014.

[133] Pedro Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St Andrews, 2008.

[134] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. Abstract refinement types. In *ESOP*, 2013.

[135] Dmitry Vlasov. Proof search algorithm in pure logical framework. *Sibirskie Elektronnye Matematicheskie Izvestiya*, 2017.

[136] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.

[137] D. Walker. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*. MIT Press, 2002.

[138] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 452–466, 2017.

[139] P. Wang, D. Wang, and A. Chlipala. TiML: A Functional Language for Practical Complexity Analysis with Invariants. In *Object-Oriented Prog., Syst., Lang., and Applications (OOPSLA'17)*, 2017.

[140] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *PACMPL*, 2(POPL):63:1–63:30, 2018.

[141] Ben Wegbreit. Mechanical Program Analysis. *Commun. ACM*, 18(9):528–539, 1975.

[142] Ningning Xie, Xuan Bi, Bruno C. D. S. Oliveira, and Tom Schrijvers. Consistent subtyping for all. *ACM Trans. Program. Lang. Syst.*, 42(1), nov 2019.

[143] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *PACMPL*, 1(OOPSLA):63:1–63:26, 2017.

[144] Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. A mechanical formalization of higher-ranked polymorphic type inference. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019.

[145] F. Zuleger, M. Sinn, S. Gulwani, and H. Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *Static Analysis Symp. (SAS'11)*, 2011.

[146] Florian Zuleger, Moritz Sinn, Sumit Gulwani, and Helmut Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th Int. Static Analysis Symp. (SAS'11)*, pages 280–297, 2011.