Liquid Resource Types

TRISTAN KNOTH, University of California, San Diego, USA
DI WANG, Carnegie Mellon University, USA
ADAM REYNOLDS, University of California, San Diego, USA
JAN HOFFMANN, Carnegie Mellon University, USA
NADIA POLIKARPOVA, University of California, San Diego, USA

This article presents *liquid resource types*, a technique for automatically verifying the resource consumption of functional programs. Existing resource analysis techniques trade automation for flexibility – automated techniques are restricted to relatively constrained families of resource bounds, while more expressive proof techniques admitting value-dependent bounds rely on handwritten proofs. Liquid resource types combine the best of these approaches, using logical refinements to automatically prove precise bounds on a program's resource consumption. The type system augments refinement types with potential annotations to conduct an amortized resource analysis. Importantly, users can annotate data structure declarations to indicate how potential is allocated within the type, allowing the system to express bounds with polynomials and exponentials, as well as more precise expressions depending on program values. We prove the soundness of the type system, provide a library of flexible and reusable data structures for conducting resource analysis, and use our prototype implementation to automatically verify resource bounds that previously required a manual proof.

CCS Concepts: • **Software and its engineering** → **Functional languages**; • **Theory of computation** → **Program analysis**.

Additional Key Words and Phrases: Automated amortized resource analysis, Refinement types

ACM Reference Format:
Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann, and Nadia Polikarpova. 2020. Liquid Resource Types. *Proc. ACM Program. Lang.* 4, ICFP, Article 106 (August 2020), 29 pages. https://doi.org/10.1145/3408988

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.

Authors' addresses: Tristan Knoth, University of California, San Diego, USA, tknoth@ucsd.edu; Di Wang, Carnegie Mellon University, USA, diw3@cs.cmu.edu; Adam Reynolds, University of California, San Diego, USA, acreynol@ucsd.edu; Jan Hoffmann, Carnegie Mellon University, USA, jhoffmann@cmu.edu; Nadia Polikarpova, University of California, San Diego, USA, npolikarpova@ucsd.edu.

2475-1421/2020/8-ART106
https://doi.org/10.1145/3408988

```
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))
```

Fig. 1. Insertion sort

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) [Hoffmann and Hofmann 2010b] 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 [Radicek et al. 2018a] 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.

For example, consider insertion sort: Fig. 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 [Knoth et al. 2019] takes a first step in this direction by extending a *liquid type system* with resource analysis. Liquid types [Rondon et al. 2008] support automatic verification of nontrivial functional properties with an SMT solver. ReSyn augments an existing liquid type system [Polikarpova et al. 2016] 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 List a with a $\mapsto \mathsf{Int}^1$ yields List 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 Fig. 1 against the (polymorphic) type $x : a \to xs : \mathsf{List}\ \mathsf{a}^1 \to \mathsf{List\ a}$ to verify that the function makes one recursive call per element in the input xs. Here 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 $x : a \to xs : \mathsf{List}\ \mathsf{a}^{\mathsf{ite}(x>v,1,0)} \to \mathsf{List\ a}$, which states that insert

```
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>ν,1,0) →ISList a
```

Fig. 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.

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 [Alur et al. 2013].

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 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 paper, 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.

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 Fig. 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 Fig. 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

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

QList and ISList separately, we can define a more general type 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 paper 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 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.

2.1 Background: ReSyn

Liquid Types. In a refinement type system [Denney 1999; Swamy et al. 2016], 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** Nat = {Int $|\nu \geq 0$}, where the special variable ν, as before, denotes an inhabitant of the type. Liquid types [Rondon et al. 2008; Vazou et al. 2013] 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 [Knoth et al. 2019] 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 : 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 List a with a $\mapsto \mathsf{Nat}^1$ yields the type List Nat^1 of natural-number lists that carry one unit of potential per element. Here and throughout the paper, a missing potential

1 [insert: x : a → xs : List a^P → 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 →Cons x Nil 4
5 [insert: . . ., x: a, xs: List a, hd: a^P, tl: List a^P]	Cons hd tl → 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

Fig. 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.

annotation defaults to zero, so the type above stands for (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 List a^m with a ↦ Nat^n yields the type 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 : 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 Cons hd tl extends the context with new bindings hd :: Nat^1 and tl :: 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 Fig. 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:

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

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 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.

Fig. 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 List a^1 in the context of the match scrutinee and List a^0 in the context of the branches. Matching the scrutinee type List a^1 against the type of the Cons constructor introduces new bindings hd :: a^1 and tl :: List 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 : a^{p_1} and hd : 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 : List a^{q_1} and tl : List 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 : 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: $\Gamma \vdash$ List a^{q_2} <: List 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:

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

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 $P = \mathsf{ite}(\mathsf{x} > \nu, 1, 0)$ 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, Fig. 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 ν = hd.

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

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

ReSyn satisfies these constraints by setting $p_2, q_2 = \lambda(\mathsf{x}, \nu).\mathsf{ite}(\mathsf{x} > \nu, 1, 0)$, and the rest of the unknowns to to $\lambda(\mathsf{x}, \nu).0$. Synthesis of CLIA expressions is a well-studied problem [Alur et al. 2013; Reynolds et al. 2019], and ReSyn uses counterexample-guided inductive synthesis (CEGIS) [Solar-Lezama et al. 2006] 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 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. Fig. 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:

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

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 Fig. 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 ISList a^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

```
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>
```

Fig. 4. A list datatype parameterized by a value-dependent, quadratic abstract potential.

```
1 [insert: ∀b.x : b → xs : List b^1 → List b, sort: ∀c.xs : List c^1 ⟨Q⟩ → List c]
2 [insert, sort: . . ., xs: List a^1 ⟨Q⟩]
3 [insert, sort: . . ., xs: List a^1 ⟨Q⟩]
4 [insert, sort: . . ., xs: List a]
5 [insert, sort: . . ., xs: List a, hd: a^1, tl: List a^(1+Q(hd,v)) ⟨Q⟩]
6 [insert, sort: . . ., xs: List a, hd: a^p1, tl: List a^(q1(hd,v)) ⟨q1⟩]
7 [insert, sort: . . ., xs: List a, hd: a^p2, tl: List a^(q2(hd,v)) ⟨q2⟩]
8 [insert, sort: . . ., xs: List a, hd: a^(p2−1), tl: List a^(q2(hd,v)) ⟨q2⟩]
```

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

Fig. 5. Similar to Figure Fig. 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.

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, 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 [Vazou et al. 2013], which parameterize datatypes by a refinement predicate; similarly, LRT allows parameterizing a datatype a potential function. Consider the definition of the List datatype in Fig. 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 the type of the elements).

Type checking. With abstract potentials, we can use the same List datatype from Fig. 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:

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

As before, omitted potential annotations are zero by default, so the return type List a is short for $(\mathsf{List}\ \mathsf{a}^0\ \langle \lambda(\_,\_).0 \rangle)^0$ The type checking process is illustrated in Fig. 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 Fig. 4, yields the substitution $\mathsf{t} \mapsto \mathsf{a}^1, q \mapsto 1$, adding the following two new bindings to the context of the Cons branch: $\mathsf{hd} : \mathsf{a}^1$ 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 Fig. 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:

$$\begin{aligned} \exists p_1, p_2, q_1, q_2, s \in \mathbb{N}.\ & p_1 + p_2 = 1 \land p_2 - 1 \geq 0 \\ & \land\ q_1 + q_2 = 2 \land q_2 \geq s + 1 \land s \geq 1 \end{aligned}$$

which is satisfiable with $p_2, q_2, s = 1$ and the rest of unknowns set to 0. Note that while the annotations in Fig. 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:

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

Type checking still proceeds as illustrated in Fig. 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 to yield $\mathsf{tl} : \mathsf{List}\ \mathsf{a}^{1+\mathsf{ite}(\mathsf{x} > \nu, 1, 0)}\ \langle \lambda(x_1, x_2).\ \mathsf{ite}(x_1 > x_2, 1, 0) \rangle$, in the 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, v and parameters x_1, x_2 of abstract potentials:

$$
\begin{aligned}
&\exists p_1, p_2, q_1, q_2, s \in \mathbb{N} \times \mathbb{N} \to \mathbb{N}.\ \forall \mathsf{hd}, v, x_1, x_2 \in \mathbb{N}. \\
&\quad p_1(\mathsf{hd}, v) + p_2(\mathsf{hd}, v) = 1 && \text{Sharing hd (line 5)} \\
&\quad \wedge\ p_2(\mathsf{hd}, v) - 1 \geq 0 && \text{Well-formedness of hd (line 8)} \\
&\quad \wedge\ q_1(\mathsf{hd}, v) + q_2(\mathsf{hd}, v) = 1 + \mathsf{ite}(\mathsf{hd} > v, 1, 0) && \text{Sharing tl (line 5)} \\
&\quad \wedge\ q_2(\mathsf{hd}, v) \geq s(\mathsf{hd}, v) + 1 && \text{Subtyping from the call to sort} \\
&\quad \wedge\ s(\mathsf{hd}, v) \geq \mathsf{ite}(\mathsf{hd} > v, 1, 0) && \text{Subtyping from the call to insert}
\end{aligned}
$$

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 [Handley et al. 2020; Radicek et al. 2018b; Wang et al. 2017]: 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 [Polikarpova et al. 2016; Vazou et al. 2013]. Additionally, while this section only shows the use of inductive and abstract potentials for expressing quadratic potentials on lists, Sec. 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.

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.

3.1 Setting the Stage: A Resource-Aware Core Language

Syntax. Fig. 6 presents the grammar of terms in the core calculus via abstract binding trees [Harper 2016]. We extend the core language of Re^2 [Knoth et al. 2019] with natural numbers, null tuples, ordered pairs, and replace lists with general inductive data structures. Expressions are in *a-normal-form* [Sabry and Felleisen 1992], which means that syntactic forms for non-tail positions allow only *atoms* $\hat{a} \in \mathrm{Atom}$, 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 Sec. 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 $a \in \mathrm{SimpAtom}$ 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* $v \in \mathrm{Val}$ 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,

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

Fig. 6. Syntax of the core calculus

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

$$
\frac{}{\langle \mathsf{if}(\mathsf{true}, e_1, e_2), q \rangle \mapsto \langle e_1, q \rangle}\ (\textsc{E-Cond-True})
\qquad
\frac{}{\langle \mathsf{if}(\mathsf{false}, e_1, e_2), q \rangle \mapsto \langle e_2, q \rangle}\ (\textsc{E-Cond-False})
$$

$$
\frac{v_1 \in \mathsf{Val}}{\langle \mathsf{let}(v_1, x.e_2), q \rangle \mapsto \langle [v_1/x]e_2, q \rangle}\ (\textsc{E-Let-Val})
$$

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

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

$$
\frac{v_2 \in \mathsf{Val}}{\langle \mathsf{app}(\lambda(x.e_0), v_2), q \rangle \mapsto \langle [v_2/x]e_0, q \rangle}\ (\textsc{E-App-Abs})
\qquad
\frac{v_2 \in \mathsf{Val}}{\langle \mathsf{app}(\mathsf{fix}(f.x.e_0), v_2), q \rangle \mapsto \langle [\mathsf{fix}(f.x.e_0), v_2/f, x]e_0, q \rangle}\ (\textsc{E-App-Fix})
$$

Fig. 7. Selected rules of the small-step operational cost semantics

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*, 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. Fig. 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.

Refinement

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

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

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

Fig. 8. Syntax of the core type system

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 $\mapsto$.

3.2 Types and Refinements

Refinements. We follow the approach of liquid types [Knoth et al. 2019; Polikarpova et al. 2016; Rondon et al. 2012] and develop a refinement language that is distinct from the term language. Fig. 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 $\mathcal{I}_D$: (values of datatype D) $\to$ (refinements of sort Δ_D). The function $\mathcal{I}_D$ 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 [Polikarpova et al. 2016]), they are orthogonal to resource analysis. We include the full development with measures in the technical report [Knoth et al. 2020]

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

$$\mathcal{I}(x) = x \qquad \mathcal{I}(\overline{n}) = n \qquad \mathcal{I}(\mathsf{triv}) = \star \qquad \mathcal{I}(\mathsf{true}) = \top \qquad \mathcal{I}(\mathsf{false}) = \bot$$
$$\mathcal{I}(\mathsf{pair}(a_1, a_2)) = (\mathcal{I}(a_1), \mathcal{I}(a_2)) \qquad \mathcal{I}(C(a_0, \langle a_1, \cdots, a_m\rangle)) = \mathcal{I}_D(C(a_0, \langle a_1, \cdots, a_m\rangle))$$

Example 3.1 (Interpretations of datatypes). Consider a natural-number list type NatList with constructors Nil and Cons. In the core language, an empty list is encoded as Nil(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 $\mathcal{I}_{\mathsf{NatList}}$: (values of NatList) $\to$ (refinements of sort $\mathbb{N}$) that computes the length of a list:

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

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

We will use the abbreviations $\bot, \vee, \implies, \geq, <, >$, **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 [Knoth et al. 2019] 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 NatList in Example 3.1 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 Sec. 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 [Tarjan 1985]; 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 $2 \cdot \mathcal{I}_{\mathsf{NatList}}(\ell) = 2 \cdot |\ell|$ units of potential where $|\ell|$ is the length of ℓ. As we will show in Sec. 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 [Knoth et al. 2019], 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.

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* [Hoffmann and Hofmann 2010b] 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 [Hoffmann et al. 2011a; Hoffmann and Hofmann 2010b; Knoth et al. 2019].

Simple numeric annotations. We start by adding numeric annotations to datatypes, following the approach of univariate polynomial potentials [Hoffmann and Hofmann 2010b]. Recall the type NatList introduced in Example 3.1. 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 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:

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

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 (1) is that the potential function $\Phi(\cdot)$ can be defined *inductively* on the data structure, and be expressed using only linear arithmetic.

Proposition 3.2. *Define the potential function* $\Phi(\cdot)$ *for type* $\mathsf{NatList}^{\vec{q}}$ *as follows:*

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

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

Based on the observation presented above, prior work [Hoffmann et al. 2011a; Hoffmann and Hofmann 2010b] 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

$$\Phi(\ell : \mathsf{NatList}^\theta) = \sum_{i=1}^{k} \sum_{1 \le j_1 < \cdots < j_i \le |\ell|} \theta_i(v_{j_1}, \cdots, v_{j_i}). \tag{2}$$

Example 3.3 (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), a feasible $\theta = (\theta_1, \theta_2)$ can be defined as follows:

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

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 Sec. 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 Proposition 3.2.

Proposition 3.4. *Define the potential function $\Phi(\cdot)$ for type $\mathsf{NatList}^\theta$ as follows:*

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

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 $\theta'_1 \stackrel{\text{def}}{=} \lambda x : \mathbb{N}.(\theta_1(x) + \theta_2(y, x))$, $\theta'_2 \stackrel{\text{def}}{=} \lambda x : \mathbb{N}^2.(\theta_2(x) + \theta_3(y, x))$, …, $\theta'_{k-1} \stackrel{\text{def}}{=} \lambda x : \mathbb{N}^{k-1}.(\theta_{k-1}(x) + \theta_k(y, x))$, and $\theta'_k \stackrel{\text{def}}{=} \theta_k$. Then (2) 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.*, Proposition 3.4). This form of generic annotations formulates the notion of *abstract potentials* (introduced in Sec. 2.2), which is one major contribution of this paper.

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:

$$\Phi(\mathsf{Nil}(\mathsf{triv}, \langle\rangle) : \mathsf{NatList}^\theta) \stackrel{\text{def}}{=} 0,$$

$$\Phi(\mathsf{Cons}(v_h, \langle v_t \rangle) : \mathsf{NatList}^\theta) \stackrel{\text{def}}{=} \pi(v_h)(\theta) + \Phi(v_t : \mathsf{NatList}^{\lhd(v_h)(\theta)}).$$

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 $\triangleleft, \pi$ and the potential annotation θ, we can now define the potential function $\Phi(\cdot)$ for general inductive datatypes as an inductive function:

$$\begin{aligned}\Phi(C_j(v_0, \langle v_1, \cdots, v_{m_j}\rangle) : \mathsf{ind}^{\theta}_{\triangleleft,\pi}(\overrightarrow{C : (T, m)})) \stackrel{\text{def}}{=}\ & \Phi(v_0 : T_j) \\ & + \pi.\mathbf{j}(\mathcal{I}(v_0))(\theta) \\ & + \sum_{i=1}^{m_j} \Phi(v_i : \mathsf{ind}^{\triangleleft.\mathbf{j}(\mathcal{I}(v_0))(\theta).\mathbf{i}}_{\triangleleft,\pi}(\overrightarrow{C : (T, m)})).\end{aligned} \tag{3}$$

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 $\mathcal{I}(\cdot)$ defined in Sec. 3.2 to interpret values as their logical refinements.

Example 3.5 (Generic potential annotations). Recall the dependently annotated list type $\mathsf{NatList}^{(\theta_1,\theta_2)}$ in Example 3.3. We can now formalize it in the core type system. Let

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

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

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

Different instantiations of θ_1, θ_2 lead to different potential functions. Example 3.3 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.

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 [Knoth et al. 2020] 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. Fig. 9 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 $\Gamma \vdash a : B$ assigns base types to interpretable atoms $a \in \mathsf{SimpAtom}$. Atomic typing is useful in the rule (T-SimpAtom), which uses the interpretation $\mathcal{I}(\cdot)$ 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 NatList admits a length interpretation).

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. Fig. 10 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 $\theta_1 \le \theta_2$ 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 [Knoth et al. 2020]

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 $\mathcal{I}(\cdot)$, we can derive

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

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 $\mathcal{I}(a_0) = 0$ and $\mathcal{I}(a_0) = x_t + 1$, respectively.

The type system has two rules for function applications: (T-App) and (T-App-SimpAtom). In 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 $\mathcal{I}(a)$. 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.

$\boxed{\Gamma \vdash a : B}$

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

(SimpAtom-Pair) $\dfrac{\vdash \Gamma \between \Gamma_1 \mid \Gamma_2 \quad \Gamma_1 \vdash a_1 : B_1 \quad \Gamma_2 \vdash a_2 : B_2}{\Gamma \vdash \text{pair}(a_1, a_2) : B_1 \times B_2}$

(SimpAtom-ConsD) $\dfrac{\vdash \Gamma \between \Gamma_1 \mid \Gamma_2 \quad \Gamma_1 \vdash a_0 :: T_j \quad \Gamma_2 \vdash \langle a_1, \cdots, a_{m_j} \rangle : \prod_{i=1}^{m_j} \text{ind}_{\lhd,\pi}^{\lhd.\mathbf{j}(\mathcal{I}(a_0))(\theta).\mathbf{i}}(\overrightarrow{C : (T, m)})}{\Gamma, \pi.\mathbf{j}(\mathcal{I}(a_0))(\theta) \vdash C_j(a_0, \langle a_1, \cdots, a_{m_j} \rangle) : \text{ind}_{\lhd,\pi}^{\theta}(\overrightarrow{C : (T, m)})}$

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

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

(T-Tick-N) $\dfrac{c < 0 \quad \Gamma, -c \vdash e_0 :: T}{\Gamma \vdash \text{tick}(c, e_0) :: T}$

(T-Cond) $\dfrac{\Gamma \vdash a_0 : \text{bool} \quad \Gamma, \mathcal{I}(a_0) \vdash e_1 :: T \quad \Gamma, \neg\mathcal{I}(a_0) \vdash e_2 :: T}{\Gamma \vdash \text{if}(a_0, e_1, e_2) :: T}$

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

(T-MatD)

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

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

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

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

(T-Abs) $\dfrac{\Gamma \vdash T_x \text{ type} \quad \Gamma, x : T_x \vdash e_0 :: T \quad \vdash \Gamma \between \Gamma \mid \Gamma}{\Gamma \vdash \lambda(x.e_0) :: x : T_x \to T}$

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

(T-Fix) $\dfrac{S = \forall\overrightarrow{\alpha}.x : T_x \to T \quad \Gamma \vdash S \text{ type} \quad \Gamma, f : S, \overrightarrow{\alpha}, x : T_x \vdash e_0 :: T \quad \vdash \Gamma \between \Gamma \mid \Gamma}{\Gamma \vdash \text{fix}(f.x.e_0) :: S}$

(S-Gen) $\dfrac{v \in \text{Val} \quad \Gamma, \alpha \vdash v :: S}{\Gamma, \alpha \vdash S \between S \mid S \quad \Gamma \vdash v :: \forall\alpha.S}$

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

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

(S-Transfer) $\dfrac{\Gamma' \vdash e :: S \quad \Gamma \models \Phi(\Gamma) = \Phi(\Gamma') \quad |\Gamma| = |\Gamma'|}{\Gamma \vdash e :: S}$

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

Fig. 9. Typing rules

To carry out amortized resource analysis [Tarjan 1985], 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 [Walker 2002]. As in Re^2 [Knoth et al. 2019], we have to 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. Fig. 10 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 (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 $x_i : \mathsf{ind}^{\lhd.\mathbf{j}(x_0)(\theta).\mathbf{i}}_{\lhd,\pi}(\overrightarrow{C : (T, m)})$, 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 $\mathsf{ind}^{\theta}_{\lhd,\pi}(\overrightarrow{C : (T, m)})$, the rule requires $\pi.\mathbf{j}(\mathcal{I}(a_0))(\theta)$ 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 $\mathsf{ind}^{\lhd.\mathbf{j}(\mathcal{I}(a_0))(\theta).\mathbf{i}}_{\lhd,\pi}(\overrightarrow{C : (T, m)})$.

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 3.6 (Insertion sort). As shown in Sec. 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,

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

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

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

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

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

$$\frac{\alpha \in \Gamma \qquad m = m_1 + m_2}{\Gamma \vdash m \cdot \alpha \curlyvee m_1 \cdot \alpha \mid m_2 \cdot \alpha}\ (\textsc{Share-Tvar}) \qquad \frac{\Gamma \vdash B \curlyvee B_1 \mid B_2 \qquad \Gamma \vdash \{B \mid \psi\}\ \mathsf{type}}{\Gamma \vdash \{B \mid \psi\} \curlyvee \{B_1 \mid \psi\} \mid \{B_2 \mid \psi\}}\ (\textsc{Share-Subset})$$

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

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

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

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

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

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

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

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

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

Fig. 10. Sharing and subtyping

using the dependently annotated list type $\mathsf{NatList}^{(\theta_1,\theta_2)}$ from Example 3.3:

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

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 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 $T \stackrel{\text{def}}{=} \mathsf{NatList}^{(\lambda x:\mathbb{N}.0,\lambda x:\mathbb{N}\times\mathbb{N}.0)}$:

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

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:

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

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

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

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:

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

Recall that in Example 3.3, 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 Example 3.6, we will show how our system is able to type a polymorphic version of sort, which has been informally demonstrated in Sec. 2.2.

Example 3.7 (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:

$$\mathsf{insert} :: \forall\alpha.y : \alpha \to \ell : \mathsf{List}^{\lambda(x_1,x_2).0}(\alpha^{\mathbf{ite}(y>\nu,1,0)}) \to \mathsf{List}^{\lambda(x_1,x_2).0}(\alpha), \tag{4}$$

$$\mathsf{sort} :: \forall\alpha.\ell : \mathsf{List}^{\lambda(x_1,x_2).\mathbf{ite}(x_1>x_2,1,0)}(\alpha^{1}) \to \mathsf{List}^{\lambda(x_1,x_2).0}(\alpha) \tag{5}$$

Similar to the type derivation in Example 3.6, 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 (5). To type the function call $\mathsf{app}(f, t)$, we instantiate f with $\alpha^{\mathbf{ite}(h>\nu,1,0)}$, *i.e.*, f has type $\ell : \mathsf{List}^{\lambda(x_1,x_2).\mathbf{ite}(x_1>x_2,1,0)}(\alpha^{1+\mathbf{ite}(h>\nu,1,0)}) \to \mathsf{List}^{\lambda(x_1,x_2).0}(\alpha^{\mathbf{ite}(h>\nu,1,0)})$. Thus, the type of the return value t' of $\mathsf{app}(f, t)$ matches the argument type of insert, and we can derive the function application $\mathsf{let}(\mathsf{app}(\mathsf{insert}, h), \mathit{ins}.\mathsf{app}(\mathit{ins}, t'))$ has the desired return type $\mathsf{List}^{\lambda(x_1,x_2).0}(\alpha)$.

3.5 Soundness

We now extend Re^2's type soundness [Knoth et al. 2019] 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 3.8 (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 3.9 (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 3.10 (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 paper.

Theorem 3.11 (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 [Knoth et al. 2020]

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: How well 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?

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 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 [Knoth et al. 2020].

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 Sec. 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:

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

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 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 Sec. 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

Table 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 $a^{q\,x}$ <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>	$q \cdot \lvert\ell\rvert$

pathed potential tree. PTree is parameterized by a boolean-sorted *abstract refinement* [Vazou et al. 2013], 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 other resource bounds as needed.

4.2 Benchmark Programs

We evaluate the expressiveness of LRTCHECKER on a suite of 12 benchmark programs listed in Tab. 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 1, showing the flexibility and reusability of these datatype definitions. The cost model in all benchmarks is the number of recursive calls (as in Sec. 2).

Benchmarks 1-7 require only standard quadratic bounds. Benchmarks 2-7 are those programs from the original SYNQUID benchmark suite [Polikarpova et al. 2016] that RESYN could not handle, because they require non-linear bounds. Some of the analyses, such as merge sort, are overapproximate. 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 subsetSum :: EList Int ⟨2⟩ → Int → Bool, we can use the provided closed-form potential function 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,

Table 2. Functional benchmarks. For each benchmark, we list its type signature, verification time (t), and source for the benchmark – either RAML [Hoffmann and Hofmann 2010b], SYNQUID [Polikarpova et al. 2016], or RELCOST [Radicek et al. 2018a].

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

hence we specify this benchmark using LTree as an intermediate data structure. Prior work has shown [Augusteijn 1999] 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 LTree 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 Sec. 2. 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.

4.3 Discussion and Limitations

Tab. 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 Sec. 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 [Radicek et al. 2018a] allows one to manually verify all of the bounds in Sec. 4.2. [Çiçek et al. 2019] 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 Tab. 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 [Hoffmann and Hofmann 2010b], but this technique does not generalize to the dependent case.

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 [Wegbreit 1975] and has been later been studied for imperative programs [Albert et al. 2011, 2015] using techniques such as abstract interpretation and symbolic analysis [Kincaid et al. 2017, 2019]. The approach can also be used for higher-order functional programs by extracting higher-order recurrences [Danner et al. 2015]. Other resource analysis techniques are based on static analysis [Gulwani et al. 2009a,b; Sinn et al. 2014; Zuleger et al. 2011] and term rewriting [Avanzini and Moser 2013; Brockschmidt et al. 2014; Hofmann and Moser 2015; Noschinski et al. 2013].

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 [Hofmann and Jost 2003] to automatically derive linear bounds on the heap-space consumption of first-order programs. It has then been extended to higher-order programs [Jost et al. 2010], polynomial bounds [Hoffmann et al. 2011b; Hoffmann and Hofmann 2010a] and user-defined types [Hoffmann et al. 2017; Jost et al. 2010]. Most recently, AARA has been combined with refinement types [Freeman and Pfenning 1991] in the Re^2 type system [Knoth et al. 2019] behind ReSyn, a resource-aware program synthesizer. None of these works support user-defined potential functions. As discussed in Sec. 1, this paper 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 [Danielsson 2008] presented a dependent cost monad that has been integrated in the proof assistant Agda. dℓPCF [Lago and Gaboardi 2011] introduced linear dependent types to reason about the worst-case cost of PCF terms. Granule [Orchard et al. 2019] introduces graded modal types, combining the indexed types of dℓPCF with bounded linear logic [Girard et al. 1992] and other modal type systems [Brunel et al. 2014; Ghica and Smith 2014]. 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. [Çiçek et al. 2017, 2019] 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 [Radicek et al. 2018b]. 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 [Wang et al. 2017] 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 [Handley et al. 2020]. However, these works do not directly support amortized analysis and do not reduce type checking of non-linear bounds to linear constraints.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Richard Eisenberg, for their valuable and detailed feedback on earlier versions of this article.

This article is based on research supported by DARPA under AA Contract FA8750-18-C-0092 and by the National Science Foundation under SaTC Award 1801369, CAREER Award 1845514, and SHF Awards 1812876, 1814358, and 2007784. Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

REFERENCES

E. Albert, P. Arenas, S. Genaim, and G. Puebla. 2011. Closed-Form Upper Bounds in Static Cost Analysis. *J. Automated Reasoning* 46 (February 2011). Issue 2.

E. Albert, J. C. Fernández, and G. Román-Díez. 2015. Non-cumulative Resource Analysis. In *Tools and Algs. for the Construct. and Anal. of Syst. (TACAS'15)*.

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. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. 1–8. http://ieeexplore.ieee.org/document/6679385/

Lex Augusteijn. 1999. Sorting Morphisms. In *Advanced Functional Programming*, S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–27.

M. Avanzini and G. Moser. 2013. A Combination Framework for Complexity. In *Int. Conf. on Rewriting Techniques and Applications (RTA'13)*.

M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. 2014. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Algs. for the Construct. and Anal. of Syst. (TACAS'14)*.

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 351–370. https://doi.org/10.1007/978-3-642-54833-8_19

E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann. 2017. Relational Cost Analysis. In *Princ. of Prog. Lang. (POPL'17)*.

Ezgi Çiçek, Weihao Qu, Gilles Barthe, Marco Gaboardi, and Deepak Garg. 2019. Bidirectional type checking for relational properties. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation,*

PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 533–547. https://doi.org/10.1145/3314221.3314603

Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *35th ACM Symp. on Principles Prog. Langs. (POPL'08)*. 133–144.

N. Danner, D. R. Licata, and R. Ramyaa. 2015. Denotational Cost Semantics for Functional Languages with Inductive Types. In *Int. Conf. on Functional Programming (ICFP'15)*.

Ewen Denney. 1999. *A theory of program refinement.* Ph.D. Dissertation. University of Edinburgh, UK. http://hdl.handle.net/1842/381

T. Freeman and F. Pfenning. 1991. Refinement Types for ML. In *Prog. Lang. Design and Impl. (PLDI'91)*.

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 331–350. https://doi.org/10.1007/978-3-642-54833-8_18

Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. 1992. Bounded Linear Logic: A Modular Approach to Polynomial-Time Computability. *Theor. Comput. Sci.* 97, 1 (1992), 1–66.

Sumit Gulwani, Sagar Jain, and Eric Koskinen. 2009a. Control-Flow Refinement and Progress Invariants for Bound Analysis. In *Conf. on Prog. Lang. Design and Impl. (PLDI'09)*. 375–385.

S. Gulwani, K. K. Mehra, and T. M. Chilimbi. 2009b. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Princ. of Prog. Lang. (POPL'09)*.

Martin A. T. Handley, Niki Vazou, and Graham Hutton. 2020. Liquidate your assets: reasoning about resource usage in liquid Haskell. *PACMPL* 4, POPL (2020), 24:1–24:27. https://doi.org/10.1145/3371092

R. Harper. 2016. *Practical Foundations for Programming Languages.* Cambridge University Press.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011a. Multivariate Amortized Resource Analysis. In *38th Symp. on Principles of Prog. Langs. (POPL'11)*. 357–370.

J. Hoffmann, K. Aehlig, and M. Hofmann. 2011b. Multivariate Amortized Resource Analysis. In *Princ. of Prog. Lang. (POPL'11)*.

J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Princ. of Prog. Lang. (POPL'17)*.

J. Hoffmann and M. Hofmann. 2010a. Amortized Resource Analysis with Polynomial Potential. In *European Symp. on Programming (ESOP'10)*.

Jan Hoffmann and Martin Hofmann. 2010b. 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) (Lecture Notes in Computer Science)*, Vol. 6012. Springer, 287–306.

M. Hofmann and S. Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Princ. of Prog. Lang. (POPL'03)*.

M. Hofmann and G. Moser. 2015. Multivariate Amortised Resource Analysis for Term Rewrite Systems. In *Int. Conf. on Typed Lambda Calculi and Applications (TLCA'15)*.

S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *Princ. of Prog. Lang. (POPL'10)*.

Z. Kincaid, J. Breck, A. F. Boroujeni, and T. Reps. 2017. Compositional Recurrence Analysis Revisited. In *Prog. Lang. Design and Impl. (PLDI'17)*.

Z. Kincaid, J. Cyphert, J. Breck, and T. Reps. 2019. Non-linear Reasoning for Invariant Synthesis. In *Princ. of Prog. Lang. (POPL'19)*.

Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-Guided Program Synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 253–268. https://doi.org/10.1145/3314221.3314602

Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann, and Nadia Polikarpova. 2020. Liquid Resource Types (Extended Version). (2020). arXiv:cs.PL/2006.16233

U. D. Lago and M. Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *Logic in Computer Science (LICS'11)*.

L. Noschinski, F. Emmes, and J. Giesl. 2013. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *J. Automated Reasoning* 51 (June 2013). Issue 1.

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (July 2019), 30 pages. https://doi.org/10.1145/3341714

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Programming Language Design and Implementation (PLDI)*. 522–538.

Ivan Radicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2018a. Monadic refinements for relational cost analysis. *PACMPL* 2, POPL (2018), 36:1–36:32. https://doi.org/10.1145/3158124

Ivan Radicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2018b. Monadic refinements for relational cost analysis. *PACMPL* 2, POPL (2018), 36:1–36:32. https://doi.org/10.1145/3158124

Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. 2019. Refutation-based synthesis in SMT. *Formal Methods Syst. Des.* 55, 2 (2019), 73–102.

Patrick Maxim Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala. 2012. CSolve: Verifying C with Liquid Types. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 744–750. https://doi.org/10.1007/978-3-642-31424-7_59

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *PLDI*.

A. Sabry and M. Felleisen. 1992. Reasoning about Programs in Continuation-Passing Style. In *LISP and Functional Programming (LFP'92)*.

Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th Int. Conf. (CAV'14)*. 743–759.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *ASPLOS*.

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. 2016. 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)*. Association for Computing Machinery, New York, NY, USA, 256–270. https://doi.org/10.1145/2837614.2837655

R. E. Tarjan. 1985. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods* 6 (August 1985). Issue 2.

Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *ESOP*.

D. Walker. 2002. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*. MIT Press.

P. Wang, D. Wang, and A. Chlipala. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. In *Object-Oriented Prog., Syst., Lang., and Applications (OOPSLA'17)*.

Ben Wegbreit. 1975. Mechanical Program Analysis. *Commun. ACM* 18, 9 (1975), 528–539.

F. Zuleger, M. Sinn, S. Gulwani, and H. Veith. 2011. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *Static Analysis Symp. (SAS'11)*.