TiML: A Functional Language for Practical Complexity Analysis with Invariants

PENG WANG, MIT CSAIL, USA
DI WANG, Peking University, China
ADAM CHLIPALA, MIT CSAIL, USA

We present TiML (Timed ML), an ML-like functional language with time-complexity annotations in types. It uses indexed types to express sizes of data structures and upper bounds on running time of functions; and refinement kinds to constrain these indices, expressing data-structure invariants and pre/post-conditions. Indexed types are flexible enough that TiML avoids a built-in notion of "size," and the programmer can choose to index user-defined datatypes in any way that helps her analysis. TiML's distinguishing characteristic is supporting highly automated time-bound verification applicable to data structures with nontrivial invariants. The programmer provides type annotations, and the typechecker generates verification conditions that are discharged by an SMT solver. Type and index inference are supported to lower annotation burden, and, furthermore, big-O complexity can be inferred from recurrences generated during typechecking by a recurrence solver based on heuristic pattern matching (e.g. using the Master Theorem to handle divide-and-conquer-like recurrences). We have evaluated TiML's usability by implementing a broad suite of case-study modules, demonstrating that TiML, though lacking full automation and theoretical completeness, is versatile enough to verify worst-case and/or amortized complexities for algorithms and data structures like classic list operations, merge sort, Dijkstra's shortest-path algorithm, red-black trees, Braun trees, functional queues, and dynamic tables with bounds like $mn \log n$. The learning curve and annotation burden are reasonable, as we argue with empirical results on our case studies. We formalized TiML's type-soundness proof in Coq.

CCS Concepts: • **Theory of computation → Program verification**; *Program analysis*; • **Software and its engineering → Software performance**; **Functional languages**; *Formal software verification*;

Additional Key Words and Phrases: refinement types, resource-aware type systems, asymptotic complexity

ACM Reference Format:
Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 79 (October 2017), 25 pages. https://doi.org/10.1145/3133903

1 INTRODUCTION

Static complexity analysis of computer programs has been under intensive study for a long time and recently has gained even more attention because of both technical breakthroughs (see Section 7 for a survey of related work) and its potential value in software quality assurance and security [Crosby and Wallach 2003; Kocher 1996]. Unlike functional correctness, static analysis of time complexity cannot be easily replaced by testing, because complexity bugs usually manifest

Authors' addresses: Peng Wang, Department of Electrical Engineering and Computer Science, MIT CSAIL, 32 Vassar Street, Cambridge, Massachusetts, 02139, USA, wangpeng@csail.mit.edu; Di Wang, School of Electronics Engineering and Computer Science, Peking University, No. 5 Yiheyuan Road, Haidian District, Beijing, Beijing, 100871, China, wayne.wangdi@pku.edu.cn; Adam Chlipala, Department of Electrical Engineering and Computer Science, MIT CSAIL, 32 Vassar Street, Cambridge, Massachusetts, 02139, USA, adamc@csail.mit.edu.

2475-1421/2017/10-ART79
https://doi.org/10.1145/3133903

themselves only under large inputs, which makes testing time-consuming. Profiling is a manual debugging tool that is not suited for automatic quality control. Static complexity analysis also has the potential to become a bridge between the programming-languages and algorithms research communities, transporting insights and tools between them [Harper 2014].

Most work on static complexity analysis falls into one of two camps, the first of which aims at full automation, while the second aims at expressiveness (much like the split in the broader software-verification literature). When it comes to ease of use, nothing beats push-button systems, though at the cost of restricting domains to e.g. polynomial bounds [Hoffmann et al. 2011] or first-order imperative programs [Gulwani et al. 2009]. Tools in this category also disallow user-provided hints when automation fails. The second camp aspires to verify hard programs against rich specifications [Charguéraud and Pottier 2015], using techniques like program logics and tools like proof assistants, at the cost of writing proofs manually. A class of middle-way approaches recently gained popularity in software verification, pioneered by DML [Xi and Pfenning 1999] and popularized by liquid types [Rondon et al. 2008] and Dafny [Leino 2010], whose central theme is to restrict the power of dependent types or program logics in exchange for some degree of automation. Our work is in the same spirit, where we ask the programmer to help by providing annotations, and we then try to make the typechecking experience as smooth as possible and, in case of failure, give useful feedback to help the programmer tweak annotations.

With a pragmatist's mindset, we choose to start from a real programming language that is used every day and liked by practitioners. Standard ML (SML) is an obvious choice, because of both its elegant design and the attention its variants (OCaml and F#) are gaining in the finance and data-analysis industries. Our design philosophy is to add features regarding time complexity to SML without hindering its strengths like modularity and type inference.

We started the design of our language, TiML, by putting a number above the "arrow" of each function type, representing an upper bound on its running time (similar to many type-and-effect systems). This number is called an index of the function type. Since a function's running time can depend on the size of its input, we also put indices in datatypes, representing their sizes. A function can be parametric on indices to accept inputs of any sizes. Inspired by DML, we do not fix an indexing scheme for datatypes (like length-indexed lists) but instead let the programmer provide an indexing scheme in the definition of a datatype, doing away with any built-in notion of "size." This flexibility allows the programmer to choose size notions like depth of a tree, black-depth of a red-black-tree, largest element in a list, etc. Aside from this custom-size flexibility, indexed types are comparable to the mechanism of sized types [Hughes et al. 1996].

Many data structures (e.g. balanced search trees) have invariants involving their sizes, and many functions require/guarantee constraints on their input/output sizes. To make these requirements formal, we classify indices with sorts and introduce a special form of sorts called refinement sorts. Sorts are to indices what types are to terms, and refinement sorts are like refinement types [Rondon et al. 2008] but on the index/sort level. A refinement sort denotes a subset of indices of the base sort satisfying a predicate. Since this predicate may refer to other indices in the context to express relational constraints, the index/sort subsystem is a dependent type system. Usual refinement type systems, unlike ours, do not have this index/sort layer and directly make the term/type system dependent. We choose to keep the term/type system as conventional System F, suspecting that making it dependent would add significant technical difficulty associated with concepts like pure vs. impure terms.

We have a syntax-directed algorithmic version of typing rules for typechecking, and the refinement predicates in sorts will cause the typechecker to generate verification conditions (VCs) that are discharged by an SMT solver. Note that with full annotations of the running time of recursive functions, the VCs are not recurrences (like $T(n-1)+3 \leq T(n)$ with an unknown $T(\cdot)$) but regular inequality formulas (like $3(n-1)+3 \leq 3n$), so we do not face the difficult problem of recurrence solving. However, if the programmer decides that coming up with a time-complexity annotation like $3n$ is too burdensome, she can choose to omit this annotation, and our inference-enabled typechecker will generate VCs like $T(n-1)+3 \leq T(n)$ with an unknown $T(\cdot)$. In this situation we do face the problem of recurrence solving, which we handle in an incomplete way by using heuristic pattern-matching-based big-O complexity inference. For example, seeing a pattern $T(n-1)+3 \leq T(n)$, our solver infers that the function's time complexity is $O(n)$. Big-O notations are expressed in TiML as sorts refined by the big-O predicate, which is a binary relation between two indices of a function sort (sorts include not only natural and real numbers but also functions from sorts to sorts).

We want to emphasize that our design of TiML is not based on one or two deep technical epiphanies, yielding fundamentally new language elements. Instead, we have spent some time searching a particular design space, applying ideas from the programming-languages research literature, and we want to argue in this paper that with TiML we have found a very appealing point, combining expressiveness with a relatively streamlined programmer experience. Our evidence is mainly empirical. Specifically, our main contributions are:

- A novel use of refinement sorts for complexity analysis with invariants
- A sizable set of empirical studies on the practicality of a functional language with indexed types and refinement sorts for complexity specification and verification, including its applicability, learning curve, and annotation burden
- A rigorous type-soundness proof formalized in Coq

We give a TiML tutorial with example code in Section 2. In Section 3, we formally define the language, including its dynamic/static semantics, and give the statement of the main soundness theorem. In Section 4, we detail the heuristic pattern-matching-based recurrence solver, which is the biggest piece of unusual functionality in our typechecker implementation. Section 5 dives into the soundness proof, listing the key lemmas and reporting our experience formalizing it in Coq. In Section 6, we report our empirical evaluation of TiML on 17 benchmark examples, listing the verified bounds, lines of code, and processing time, and discuss annotation burden and the learning time required to get started using TiML. Section 7 compares TiML to related work. In Sections 8 and 9, we discuss TiML's limitations as well as our future-work plans and conclude. The source code of TiML is available at https://github.com/mit-plv/timl; a Web interface to experiment with TiML is available at http://timl.csail.mit.edu/.

2 TIML EXAMPLES

```
datatype list α : {ℕ} = Nil of list α {0}
                      | Cons {n : ℕ} of α * list α {n}⟶list α {n + 1}

fun foldl [α β] {m n : ℕ} (f : α * β -$m→ β) acc (l : list α {n})
    return β using $(m + 4) * $n =
  case l of
      [] ⇒ acc
    | x :: xs ⇒ foldl f (f (x, acc)) xs
```

Fig. 1. Definition of list and fold-left

In this section, we give a tutorial introduction to TiML programming. Our first example is the fold-left function on lists, through which we introduce the basics of TiML. Figure 1 gives the definition of list and fold-left. TiML mimics SML's syntax, on top of which we add indices and sorts. Datatype `list` is parameterized not only on a type variable α but also on an index of sort $\mathbb{N}$ (natural numbers), expressed by the "`: {ℕ}`" part in the header. This index argument varies in different constructors. In the `Nil` case, it is fixed to 0, while for `Cons`, it is `n+1`, where `n` is the index of the tail list. Obviously this index represents the length of a list. Note that the TiML language itself has no built-in knowledge of "sizes"; they are just a datatype's index arguments, which may happen to correspond to some human intuition about how big a chunk of data is. The meta-theoretic treatment does not talk about "sizes," and the user does not get formal guarantees about "sizes" from the soundness theorem in Section 5, so "sizes" can be regarded as a proof intermediary for "time."

The fold-left function `foldl` takes two type arguments (unlike in SML, they are explicit in TiML) α and β, two index arguments `m` and `n`, an operation `f` to be performed on the list, an accumulator `acc`, and an input list `l`, and returns a result of type β. The two indices stand for the time bound of operation `f` and the length of the input list. The sort `Time` is defined as nonnegative real numbers (real numbers because we allow logarithms); the `$` sign is used to convert a natural number to a time. Arrows in function types are extended to "long arrows" (e.g. $-\boxed{\texttt{\$}m}\!\rightarrow$) carrying time specifications on their shoulders. Recursive functions need to be annotated with return types and time bounds via the `return` and `using` keywords. We only count the number of function calls (beta reductions), each beta reduction counting as one step. Here, `foldl`'s running time is bounded by `$(m + 4) * $n` ("4" comes from these four function applications illustrated by the dots: `foldl . f . (f . (x, acc)) . xs`).

Typechecking `foldl` will generate multiple VCs, among which the only nontrivial one is

$$\forall m\ n\ n' : \text{Nat}.\ n' + 1 = n \longrightarrow m + 4 + (m + 4)n' \le (m + 4)n.$$

n' is introduced by constructor `Cons` to represent the length of the tail list, and the premise $n' + 1 = n$ is introduced by typechecking the pattern-matching, which connects the inner index n' to the outer index argument n. The inequality $m + 4 + (m + 4)n' \le (m + 4)n$ dictates that the actual running time of this branch, $m+4+(m+4)n'$, should be bounded by the specified bound $(m+4)n$.

The next example is merge sort (Figure 2), in which we show the use of the big-O notation to reduce annotation burden. Let us first look at the function `msort`. Instead of using a concrete time bound such as `$(m + 4) * $n`, we bound `msort`'s running time by `T_msort m n`. `T_msort` is an index of sort `BigO (λ m n ⇒ $m * $n * log₂ $n)`, which is the sort of *time functions* (functions from multiple natural numbers to time) that are in the big-O class $O(mn \log_2 n)$. Under the hood, `BigO f` is syntax sugar for { `g | g ≦ f` }, the sort of time functions refined by the big-O binary relation $\leqq$.

The point is that only the sort of `T_msort` is needed, not the definition (written as an underscore). The typechecker generates VC

$$\exists T.\ T \leqq (\lambda m\, n.\ mn \log_2 n) \wedge \forall m\ n.\ T(m, \lceil n/2 \rceil) + T(m, \lfloor n/2 \rfloor) + 7 + T_{\text{split}} + T_{\text{merge}} \le T(m, n), \quad (1)$$

for which $T_{\text{split}} \leqq (\lambda n.\ n)$ and $T_{\text{merge}} \leqq (\lambda m\ n.\ m \times n)$ are available premises in the context. This VC is discharged by our heuristic pattern-matching-based recurrence solver. The solver will not give a solution for T; instead, it sees that this VC matches the pattern of one of the cases in the Master Theorem [Cormen et al. 2009], therefore concluding that this VC is true. The absence of a concrete definition of T (i.e. `T_msort`) is OK because `T_msort` is declared as an *abstract index* by `absidx`. Outside the current module, its definition is not visible. If the module scope is still too

```
absidx T_split: BigO _ (* (fn n => $n) *) = _
fun split [α] {n: ℕ} (l: list α {n})
    return list α {ceil ($n/2)} * list α {floor ($n/2)} using T_split n =
  case l of
      [] ⇒ ([], [])
    | [_] ⇒ (l, [])
    | x1 :: x2 :: xs ⇒
      let val (xs1, xs2) = split xs in
        (x1 :: xs1, x2 :: xs2) end

absidx T_merge: BigO (λ m n ⇒ $m * $n) = _
fun merge [α] {m n1 n2: ℕ} (le: α * α -$m→ bool)
      (xs: list α {n1}, ys: list α {n2})
    return list α {n1 + n2} using T_merge m (n1 + n2) =
  case (xs, ys) of
      ([], _) ⇒ ys
    | (_, []) ⇒ xs
    | (x :: xs', y :: ys') ⇒
      if le (x, y) then x :: merge le (xs', ys)
      else y :: merge le (xs, ys')

absidx T_msort: BigO (λ m n ⇒ $m * $n * log₂ $n) = _
fun msort [α] {m n: ℕ} (le: α * α -$m→ bool) (xs: list α {n})
    return list α {n} using T_msort m n =
  case xs of
    [] ⇒ xs
  | [_] ⇒ xs
  | _ :: _ :: _ ⇒
    let val (xs1, xs2) = split xs in
    merge le (msort le xs1, msort le xs2) end
```

Fig. 2. Merge sort

large, we have another variant `absidx ... end` that makes an index abstract outside a local block, similar to SML's `abstype ... end` facility.

Since the Master Theorem can decide the solution's big-O class from the recurrence, the big-O class in a big-O sort can also be omitted and inferred, as shown by function `split`. Notice that its big-O class `(λ n ⇒ $n)` is commented out.

Our last example is the definition of red-black trees (Figure 3), where we need to encode the invariants of the data structure. A red-black tree `rbt` is indexed by three indices: the size, the root color, and the black-height. We use { `P` } as syntax sugar for { `_ :` { `_ : unit | P` } } where the index itself is not interesting. We are essentially abusing one sort's refinement to put constraints on other indices. `Leaf` is black with zero size and zero black-height. In the case of `Node`, the children must have the same black-height, and when the root color is red, the children's root colors must both be black. `b2n` does conversion from Booleans `true` and `false` to natural numbers `1` and `0` respectively.

```
datatype color : {𝔹} =
  Black of color {true}
| Red of color {false}

datatype rbt a : {ℕ} {𝔹} {ℕ} =
  Leaf of rbt a {0} {true} {0}
| Node {lcolor color rcolor : 𝔹}
       {lsize rsize bh (*black-height*) : ℕ}
       {color = false → lcolor = true ∧ rcolor = true}
       { ... (*other invariants*)}
  of color {color} * rbt a {lsize} {lcolor} {bh} * (key * a)
       * rbt a {rsize} {rcolor} {bh}
      ⟶rbt a {lsize + 1 + rsize} {color} {bh + b2n color}
```

Fig. 3. Red-black trees

3 LANGUAGE DEFINITION

The TiML code in Section 2 uses the surface syntax understood by the parser. In this section we define TiML as a formal calculus whose soundness we prove. The gap from the surface language to the formal calculus involves packaging recursive types as algebraic datatypes and inlining rules Relax and TyEq to make typing rules syntax-directed, both of which are standard practice.

3.1 Syntax

$$
\begin{array}{lcl}
\text{Base Sort} & & \\
\quad \underline{s} & ::= & \text{Nat} \mid \text{Time} \mid 1 \mid 2 \mid \underline{s} \Rightarrow \underline{s} \\
\text{Sort} & & \\
\quad s & ::= & \underline{s} \mid \{a : \underline{s} \mid \theta\} \\
\text{Index} & & \\
\quad i & ::= & a \mid n \mid r \mid () \mid \text{true} \mid \text{false} \mid o_{ui}\ i \mid i\ o_{bi}\ i \mid i\ ?\ i : i \mid \lambda a : \underline{s}.\ i \mid i\ i \\
\text{Proposition} & & \\
\quad \theta & ::= & \top \mid \bot \mid \neg\theta \mid \theta \wedge \theta \mid \theta \vee \theta \mid \theta \longrightarrow \theta \mid i\ o_{br}\ i \mid \forall a : \underline{s}.\ \theta \mid \exists a : \underline{s}.\ \theta \\
\text{Kind} & & \\
\quad \kappa & ::= & * \mid \kappa \Rightarrow \kappa \mid \underline{s} \Rightarrow \kappa \\
\text{Type} & & \\
\quad \tau & ::= & \alpha \mid 1 \mid \text{int} \mid \text{nat}\ i \mid \text{arr}\ \tau\ i \mid \tau \xrightarrow{i} \tau \mid \tau \times \tau \mid \tau + \tau \mid \mu\alpha : \kappa.\ \tau \mid \forall \alpha : \kappa.\ \tau \mid \forall a : s.\ \tau \mid \exists \alpha : \kappa.\ \tau \\
 & \mid & \exists a : s.\ \tau \mid \lambda \alpha : \kappa.\ \tau \mid \tau\ \tau \mid \lambda a : \underline{s}.\ \tau \mid \tau\ i \\
\text{Term} & & \\
\quad e & ::= & x \mid () \mid d \mid \overline{n} \mid \lambda x : \tau.\ e \mid e\ e \mid e\ o_{bt}\ e \mid e \oplus e \mid (e, e) \mid e.1 \mid e.2 \mid \text{l}_\tau.e \mid \text{r}_\tau.e \mid \text{case}\ e\ \text{of}\ x.e\ \text{or}\ x.e \\
 & \mid & \text{fold}_\tau\ e \mid \text{unfold}\ e \mid \Lambda\alpha : \kappa.\ e \mid e\ \tau \mid \Lambda a : s.\ e \mid e\ i \mid \text{pack}_\tau\ \langle \tau, e \rangle \mid \text{unpack}\ e\ \text{as}\ \langle \alpha, x \rangle\ \text{in}\ e \\
 & \mid & \text{pack}_\tau\ \langle i, e \rangle \mid \text{unpack}\ e\ \text{as}\ \langle a, x \rangle\ \text{in}\ e \mid \text{rec}_\tau\ x.e \mid \text{new}\ e\ e \mid e[e] \mid e[e] := e \mid \ell
\end{array}
$$

Fig. 4. TiML syntax

The syntax of TiML is given in Figure 4. We split the core language into four syntactic classes: terms, types, indices, and sorts. Terms are classified by types; types can be indexed by indices, which are classified by sorts. Alternatively one can treat both types and indices as type-level constructors and sorts as kinds. We keep types and indices separate because they are treated differently

Unary Index Operator

o_{ui} ::= ceil | floor | neg | div_n | $\log_n$ | $\exp_n$ | nat2time | bool2nat

Binary Index Operator

o_{bi} ::= $+ \mid - \mid \times \mid$ max | min | and | or | =? | ≤? | <? | ≥? | >?

Binary Index Relation

o_{br} ::= $= \mid \leq \mid < \mid \geq \mid > \mid \leqq$

Binary Term Operator

o_{bt} ::= $+ \mid - \mid \times \mid / \mid \cdots$

Fig. 5. Operators

in our metatheoretical development (e.g. indices are given a denotational semantics while types are not).

TiML's base sorts include natural numbers, time (nonnegative real numbers), unit, Booleans, and functions from base sorts to base sorts. A sort is either a base sort or a base sort refined by a proposition. In refinement sort $\{a : \underline{s} \mid \theta\}$, the variable a stands for the index being refined and can be mentioned by the proposition θ.[1] An index, ranged over by i or j, can be an index variable a, a constant natural number n, a constant nonnegative real number r, the unit value $()$, or a Boolean constant. It can also be formed by unary index operation $o_{ui}\ i$, binary index operation $i\ o_{bi}\ i$, if-then-else, lambda abstraction, or application. All minus operations are bounded below by 0. We use different letter sets to denote different kinds of variables: a and b for index variables, α and β for type variables, and x and y for term variables. All operators are summarized in Figure 5. Propositions include usual logical constructs and binary relations $i\ o_{br}\ i$ between two indices. There is a special binary relation "$f \leqq g$" between two time functions meaning $f \in O(g)$.

TiML has a higher-order kind system with the $\kappa \Rightarrow \kappa$ kind former, besides which there is also the kind former $\underline{s} \Rightarrow \kappa$, allowing type-level functions from base sorts to types, which is needed for kind-indexed datatypes like `list` in Section 1. TiML has four base types: unit, integers, indexed natural numbers, and length-indexed arrays. Integers are just an example of ordinary primitive types. Length-indexed arrays are a generalization of mutable references. The length index is useful in complexity analysis of array-based algorithms, and it also brings the benefit of static array-bound checking.

Indexed natural numbers serve as a bridge between runtime values and static indices. For example, the only value of type $\text{nat}\ 3$ is the term $\overline{3}$. If we did not have indexed natural numbers, then all integers would just have type int, and at compile time we could not tell their runtime values from their types, despite the fact that e.g. array-bound checking is done entirely based on types (TiML is not dependently typed). Indexed natural numbers give us the opportunity to tell a variable's runtime value from its type.

The function type (arrow type) is indexed by an upper bound on the function's running time, which is the most crucial new feature in TiML. Product/sum/recursive types are supported to enable user-defined datatypes. For polymorphic types, existential types, type-level abstractions, and type-level applications, we have two versions of each, one for type arguments and one for index arguments. We use the same notation in this paper for the two versions (and the corresponding introduction and elimination term forms), relying on context to tell them apart.

Some terms are annotated with types (shown as subscripts) to facilitate syntax-directed type-checking, and many of these annotations can be inferred or disguised as datatypes. We use d and $\overline{n}$ as distinct syntax classes to represent integer constants and natural number constants, of types int and $\text{nat}\ i$ respectively. $e\ o_{\text{bt}}\ e$ stands for all primitive binary operations working on primitive types such as int. $e \oplus e$ is the plus operation on natural numbers. It is different from $e\ o_{\text{bt}}\ e$

[1]For readers familiar with liquid types: it corresponds to the dedicated variable γ in liquid types.

Value

$$\begin{array}{lll} v & ::= & () \mid d \mid \overline{n} \mid \lambda x : \tau.\ e \mid (v, v) \mid \mathrm{l}_\tau.v \mid \mathrm{r}_\tau.v \mid \mathrm{fold}_\tau\ v \mid \Lambda\alpha : \kappa.\ e \mid \Lambda a : s.\ e \mid \mathrm{pack}_\tau\ \langle \tau, v\rangle \\ & \mid & \mathrm{pack}_\tau\ \langle i, v\rangle \mid \ell \end{array}$$

Evaluation Context

$$\begin{array}{lll} E & ::= & \square \mid E\ e \mid v\ E \mid E\ o_{bt}\ e \mid v\ o_{bt}\ E \mid E \oplus e \mid v \oplus E \mid (E, e) \mid (v, E) \mid E.1 \mid E.2 \mid \mathrm{l}_\tau.E \mid \mathrm{r}_\tau.E \\ & \mid & \mathrm{case}\ E\ \mathrm{of}\ x.e\ \mathrm{or}\ x.e \mid \mathrm{fold}_\tau\ E \mid \mathrm{unfold}\ E \mid E\ \tau \mid E\ i \mid \mathrm{pack}_\tau\ \langle \tau, E\rangle \mid \mathrm{unpack}\ E\ \mathrm{as}\ \langle \alpha, x\rangle\ \mathrm{in}\ e \\ & \mid & \mathrm{pack}_\tau\ \langle i, E\rangle \mid \mathrm{unpack}\ E\ \mathrm{as}\ \langle a, x\rangle\ \mathrm{in}\ e \mid \mathrm{new}\ E\ e \mid \mathrm{new}\ v\ E \mid E[e] \mid v[E] \mid E[e] := e \\ & \mid & v[E] := e \mid v[v] := E \end{array}$$

Heap

$$\begin{array}{lll} h & \in & \mathrm{loc} \rightharpoonup \overrightarrow{v} \end{array}$$

Configuration

$$\begin{array}{lll} \sigma & = & (h, e, r) \end{array}$$

Fig. 6. Definitions in operational semantics

because the result type has to be properly indexed to reflect the computation. For example, supposing $\vdash e_1 : \mathrm{nat}\ 3$ and $\vdash e_2 : \mathrm{nat}\ 4$, then $\vdash e_1 \oplus e_2 : \mathrm{nat}\ 7$. Notice that $(e_1 \oplus e_2)$'s type is indexed, and the index is computed from the indices of the operands' types. On the contrary, if $\vdash e_1 : \mathrm{int}$ and $\vdash e_2 : \mathrm{int}$, then $\vdash e_1\ o_{\mathrm{bt}}\ e_2 : \mathrm{int}$ (where o_{bt} is e.g. plus), an unindexed type. We use addition as an example for other possible natural-number operations. The next few term formers are the introduction and elimination forms of the corresponding types. We use the notation $x.e$ to mean a binding where x is locally bound in e. $\mathrm{rec}_\tau\ x.e$ is a general fixpoint. The next three terms are array allocation, read, and write. Locations ℓ only arise during reduction, as in standard operational semantics for mutable references.

3.2 Operational Semantics

TiML's operational semantics (Figures 6 and 7) is a standard small-step operational semantics instrumented with a "fuel" parameter. Fuel (a nonnegative real number) is consumed by certain reductions, and a reduction without enough fuel to proceed is stuck. A starting fuel amount in an execution that does not get stuck will thus be an upper bound on the execution's total time cost (assuming fuel consumption coincides with time consumption). Formally, reductions are defined between configurations, each a triple of a heap, a program, and a fuel amount. A heap is a finite partial map (denoted as $\rightharpoonup$) from locations to lists of values (denoted as $\overrightarrow{v}$). Note that each location points to a list of values instead of a single value because each location corresponds to an array. The reduction relation $\sigma \rightsquigarrow \sigma'$ is defined via evaluation contexts and the atomic reduction relation $\sigma \rightarrowtriangle \sigma'$. In our current setting, only the beta reduction (function application) consumes fuel, by one unit. Fixpoint unrolling does not consume fuel, which appears to make our counting scheme too lax, as one may suspect that we are counting too few steps, where fixpoint unrollings can cause unbounded reduction sequences on their own. However, our typing rules induce a syntactic restriction ruling out two consecutive fixpoint unrollings without a beta reduction (explained more later). Array reads and writes check that the offset is within bounds before performing the operation. We use notation $m[k \mapsto v]$ to mean updating a map or a list at k, and we use v^n to mean a list of v repeated n times. $[\![o_{\mathrm{bt}}]\!]$ is an interpretation of the primitive binary operation, which is a partial function that can result in an error when given illegal arguments. For symmetric pairs like $e.1$ and $e.2$, we only show one.

$$\boxed{\sigma \rightharpoonup \sigma'}$$

$$\frac{r \geq 1}{(h,\, (\lambda x : \tau.\ e)\ v,\, r) \rightharpoonup (h,\, e[v/x],\, r-1)} \qquad \frac{}{(h,\, \mathrm{rec}_\tau\ x.e,\, r) \rightharpoonup (h,\, e[\mathrm{rec}_\tau\ x.e/x],\, r)}$$

$$\frac{}{(h,\, \text{unpack}\ (\text{pack}_{\tau'}\ \langle \tau, v\rangle)\ \text{as}\ \langle \alpha, x\rangle\ \text{in}\ e,\, r) \rightharpoonup (h,\, e[\tau/\alpha][v/x],\, r)}$$

$$\frac{}{(h,\, \text{unpack}\ (\text{pack}_{\tau}\ \langle i, v\rangle)\ \text{as}\ \langle a, x\rangle\ \text{in}\ e,\, r) \rightharpoonup (h,\, e[i/a][v/x],\, r)}$$

$$\frac{}{(h,\, (\Lambda\alpha : \kappa.\ e)\ \tau,\, r) \rightharpoonup (h,\, e[\tau/\alpha],\, r)} \qquad \frac{}{(h,\, (\Lambda a : s.\ e)\ i,\, r) \rightharpoonup (h,\, e[i/a],\, r)} \qquad \frac{}{(h,\, (v_1, v_2).1,\, r) \rightharpoonup (h,\, v_1,\, r)}$$

$$\frac{}{(h,\, \text{case}\ \mathrm{l}_\tau.v\ \text{of}\ x.e_1\ \text{or}\ x.e_2,\, r) \rightharpoonup (h,\, e_1[v/x],\, r)} \qquad \frac{h(\ell) = \overrightarrow{v} \qquad n < |\overrightarrow{v}|}{(h,\, \ell[\overline{n}],\, r) \rightharpoonup (h,\, v_n,\, r)}$$

$$\frac{h(\ell) = \overrightarrow{v} \qquad n < |\overrightarrow{v}|}{(h,\, \ell[\overline{n}] := v',\, r) \rightharpoonup (h[\ell \mapsto \overrightarrow{v}[n \mapsto v']],\, (),\, r)} \qquad \frac{\ell \notin h}{(h,\, \text{new}\ \overline{n}\ v,\, r) \rightharpoonup (h[\ell \mapsto v^n],\, \ell,\, r)}$$

$$\frac{[\![o_{\mathrm{bt}}]\!](v_1, v_2) = v}{(h,\, v_1\ o_{\mathrm{bt}}\ v_2,\, r) \rightharpoonup (h,\, v,\, r)} \qquad \frac{}{(h,\, \overline{n_1} \oplus \overline{n_2},\, r) \rightharpoonup (h,\, \overline{n_1 + n_2},\, r)} \qquad \frac{}{(h,\, \text{unfold}\ (\text{fold}_\tau\ v),\, r) \rightharpoonup (h,\, v,\, r)}$$

$$\boxed{\sigma \rightsquigarrow \sigma'}$$

$$\frac{(h,\, e,\, r) \rightharpoonup (h',\, e',\, r')}{(h,\, E[e],\, r) \rightsquigarrow (h',\, E[e'],\, r')}$$

Fig. 7. Operational semantics

$$\begin{array}{lll}
\text{Sorting Context} & \text{Kinding Context} & \text{Typing Context} \\
\quad \Omega \ ::= \ \cdot \mid \Omega,\ a :: s & \quad \Lambda \ ::= \ \cdot \mid \Lambda,\ \alpha :: \kappa & \quad \Gamma \ ::= \ \cdot \mid \Gamma,\ x : \tau \\
\text{Heap Type} & \text{Full Context} & \\
\quad \Sigma \ ::= \ \cdot \mid \Sigma,\ l : (\tau, i) & \quad \Delta \ = \ (\Omega, \Lambda, \Gamma, \Sigma) &
\end{array}$$

Fig. 8. Typing contexts

3.3 Type System

TiML's type system consists of various forms of judgments including sorting, kinding, typing, type equivalence, wellformedness of various entities, etc. Here we only describe sorting and typing rules, since they are most relevant to complexity analysis. Readers are referred to the supplemental material (Coq formalization) for all other judgments. The rules use several kinds of contexts, which are summarized in Figure 8. A sorting/kinding/typing context maps index/type/term variables to sorts/kinds/types respectively. A heap type maps locations to type-index pairs specifying the types and lengths of arrays. A full context (used in typing rules) consists of all the above contexts. Some judgments (e.g. sorting) do not need the full context. To reduce notational noise, we still pass a full

$$\boxed{\Omega \vdash i :: s}$$

$$\frac{\Omega(a) = s}{\Omega \vdash a :: s}\ \textsc{Var} \qquad \frac{}{\Omega \vdash n :: \mathrm{Nat}}\ \textsc{Nat} \qquad \frac{}{\Omega \vdash r :: \mathrm{Time}}\ \textsc{Time} \qquad \frac{}{\Omega \vdash () :: 1}\ () \qquad \frac{}{\Omega \vdash \mathrm{true} :: 2}\ \textsc{True}$$

$$\frac{}{\Omega \vdash \mathrm{false} :: 2}\ \textsc{False} \qquad \frac{\Omega \vdash i : o_{\mathrm{ui}}.\underline{s}_1}{\Omega \vdash o_{\mathrm{ui}}\ i : o_{\mathrm{ui}}.\underline{s}_{\mathrm{res}}}\ \mathrm{UO} \qquad \frac{\Omega \vdash i_m : o_{\mathrm{bi}}.\underline{s}_m\ (m = 1, 2)}{\Omega \vdash i_1\ o_{\mathrm{bi}}\ i_2 : o_{\mathrm{bi}}.\underline{s}_{\mathrm{res}}}\ \mathrm{BO}$$

$$\frac{\Omega, a :: \underline{s}_1 \vdash i :: \underline{s}_2}{\Omega \vdash \lambda a : \underline{s}_1 .\ i :: \underline{s}_1 \Rightarrow \underline{s}_2}\ \Rightarrow\mathrm{I} \qquad \frac{\Omega \vdash i_1 :: \underline{s}_1 \Rightarrow \underline{s}_2 \qquad \Omega \vdash i_2 :: \underline{s}_1}{\Omega \vdash i_1\ i_2 :: \underline{s}_2}\ \Rightarrow\mathrm{E} \qquad \frac{\Omega \vdash i :: \underline{s} \qquad \Omega \vdash \theta[i/a]}{\Omega \vdash i :: \{a : \underline{s} \mid \theta\}}\ \{\}\mathrm{I}$$

$$\frac{\Omega \vdash i :: \{a : \underline{s} \mid \theta\} \qquad \Omega, a :: \underline{s} \vdash \mathrm{wf}\ \theta}{\Omega \vdash i :: \underline{s}}\ \{\}\mathrm{E} \qquad \frac{\Omega \vdash i : 2 \qquad \Omega \vdash i_m : s\ (m = 1, 2)}{\Omega \vdash i\ ?\ i_1 : i_2 : s}\ \textsc{Ite}$$

Fig. 9. Sorting rules

context to those judgments and elide the selection of needed parts. Similarly we will write $\Delta, x : \tau$ and $\Delta(x)$ when it is clear which component of Δ is operated on.

Operators can have associated information like the type of the first argument or the result, written as $o.\tau_1$ and $o.\tau_{\mathrm{res}}$. In Figure 9, the most important rules are the introduction and elimination rules for refinement sorts: rules {}I and {}E. An inhabitant of a base sort can be admitted into a refinement sort if it satisfies the refinement predicate. We use the validity judgment $\Omega \vdash \theta$ to mean that proposition θ is true under the context Ω, which may contain refinements to be used as premises. Its definition is sketched in Section 5. A member of a refinement sort can automatically be used as a member of the base sort. A subsorting rule can be derived where subsorting is defined by implication between refinement predicates.

Typing judgments have the form $\Delta \vdash e : \tau \triangleright i$ where i represents an upper bound on time needed to reduce e to a value. i is an open index that may refer to index variables in Δ. Typing rules are defined preserving an invariant that if $\Delta \vdash e : \tau \triangleright i$ then τ is of kind $*$ and i is of sort Time. The introduction and elimination rules →I and →E for function types reflect the intuition of how to count beta reductions. Namely, the running time of a function is the running time of the function body, and the time to evaluate a function application $e_1\ e_2$ is the sum of that needed for e_1, e_2, the function body, plus 1 (the beta reduction). All other rules do not incur extra time cost. The rule Rec for fixpoints requires that the body be a function abstraction, possibly wrapped by some index polymorphism, which ensures that any two consecutive fixpoint unrollings will trigger at least one beta reduction. Note that we support index-polymorphic recursion where the index argument can change in a recursive call. Being able to make a recursive call with a different index is necessary to reflect the change of argument size. In rules μI and μE, c denotes either a type or an index, so $\overrightarrow{c}$ denotes a list of mixed types and indices. $\mathrm{FTV}(\cdot)$ and $\mathrm{FIV}(\cdot)$ stand for free type and index variables respectively. In rules Rd and Wr for array read and write, we perform static bound checking by requiring $\Delta \vdash i_2 < i_1$. Structural (non-syntax-directed) rules Relax and TyEq are for relaxing the time bound and using an equivalent type.

Note that the form restriction in rule Rec means that fixpoints always define functions, and terms such as recursive lists are ruled out. The constraint is also present in SML, since in SML one can only define a fixpoint by the "fun" keyword, which defines a function that must take at least one argument.

$\boxed{\Delta \vdash e : \tau \triangleright i}$

$$\frac{\Delta(x) = \tau}{\Delta \vdash x : \tau \triangleright 0}\ \textsc{Var} \qquad \frac{\Delta \vdash \tau_1 :: * \qquad \Delta, x : \tau_1 \vdash e : \tau_2 \triangleright i}{\Delta \vdash \lambda x : \tau_1.\ e : \tau_1 \xrightarrow{i} \tau_2 \triangleright 0}\ {\rightarrow}\mathrm{I} \qquad \frac{\Delta \vdash e_1 : \tau_1 \xrightarrow{i} \tau_2 \triangleright i_1 \qquad \Delta \vdash e_2 : \tau_1 \triangleright i_2}{\Delta \vdash e_1\ e_2 : \tau_2 \triangleright i_1 + i_2 + 1 + i}\ {\rightarrow}\mathrm{E}$$

$$\frac{e = \Lambda \overrightarrow{a : s}.\ \lambda y : \tau_1.\ e_1 \qquad \Delta \vdash \tau :: * \qquad \Delta, x : \tau \vdash e : \tau \triangleright 0}{\Delta \vdash \mathrm{rec}_\tau\ x.e : \tau \triangleright 0}\ \textsc{Rec} \qquad \frac{}{\Delta \vdash () : 1 \triangleright 0}\ 1 \qquad \frac{}{\Delta \vdash d : \mathrm{int} \triangleright 0}\ \textsc{Int}$$

$$\frac{}{\Delta \vdash \overline{n} : \mathrm{nat}\ n \triangleright 0}\ \textsc{Nat} \qquad \frac{\Delta \vdash e_m : \tau_m \triangleright i_m\ (m = 1, 2)}{\Delta \vdash (e_1, e_2) : \tau_1 \times \tau_2 \triangleright i_1 + i_2}\ {\times}\mathrm{I} \qquad \frac{\Delta \vdash e : \tau_1 \times \tau_2 \triangleright i}{\Delta \vdash e.1 : \tau_1 \triangleright i}\ {\times}\mathrm{E}_1$$

$$\frac{\Delta \vdash e : \tau_1 \triangleright i \qquad \Delta \vdash \tau_2 :: *}{\Delta \vdash \mathrm{l}_{\tau_2}.e : \tau_1 + \tau_2 \triangleright i}\ {+}\mathrm{I}_1 \qquad \frac{\Delta \vdash e : \tau_1 + \tau_2 \triangleright i \qquad \Delta, x : \tau_m \vdash e_m : \tau \triangleright i_m\ (m = 1, 2)}{\Delta \vdash \mathrm{case}\ e\ \mathrm{of}\ x.e_1\ \mathrm{or}\ x.e_2 : \tau \triangleright i + \max\{i_1, i_2\}}\ {+}\mathrm{E}$$

$$\frac{\Delta \vdash \tau\ \overrightarrow{c} :: * \qquad \tau = \mu\alpha :: \kappa.\ \tau_1 \qquad \Delta \vdash e : \tau_1[\tau/\alpha]\ \overrightarrow{c} \triangleright i}{\Delta \vdash \mathrm{fold}_{\tau\ \overrightarrow{c}}\ e : \tau\ \overrightarrow{c} \triangleright i}\ \mu\mathrm{I} \qquad \frac{\tau = \mu\alpha :: \kappa.\ \tau_1 \qquad \Delta \vdash e : \tau\ \overrightarrow{c} \triangleright i}{\Delta \vdash \mathrm{unfold}\ e : \tau_1[\tau/\alpha]\ \overrightarrow{c} \triangleright i}\ \mu\mathrm{E}$$

$$\frac{e\ \text{is value} \qquad \Delta, \alpha :: \kappa \vdash e : \tau \triangleright 0}{\Delta \vdash \Lambda\alpha : \kappa.\ e : \forall\alpha : \kappa.\ \tau \triangleright 0}\ \forall\mathrm{I} \qquad \frac{\Delta \vdash e : \forall\alpha : \kappa.\ \tau \triangleright i \qquad \Delta \vdash \tau_1 :: \kappa}{\Delta \vdash e\ \tau_1 : \tau[\tau_1/\alpha] \triangleright i}\ \forall\mathrm{E}$$

$$\frac{\Delta \vdash \mathrm{wf}\ s \qquad e\ \text{is value} \qquad \Delta, a :: s \vdash e : \tau \triangleright 0}{\Delta \vdash \Lambda a : s.\ e : \forall a : s.\ \tau \triangleright 0}\ \forall_i\mathrm{I} \qquad \frac{\Delta \vdash e : \forall a : s.\ \tau \triangleright j \qquad \Delta \vdash i :: s}{\Delta \vdash e\ i : \tau[i/a] \triangleright j}\ \forall_i\mathrm{E}$$

$$\frac{\Delta \vdash (\exists\alpha :: \kappa.\ \tau) :: * \qquad \Delta \vdash \tau_1 :: \kappa \qquad \Delta \vdash e : \tau[\tau_1/\alpha] \triangleright i}{\Delta \vdash \mathrm{pack}_{\exists\alpha::\kappa.\ \tau}\ \langle \tau_1, e \rangle : \exists\alpha :: \kappa.\ \tau \triangleright i}\ \exists\mathrm{I}$$

$$\frac{\Delta \vdash e_1 : \exists\alpha :: \kappa.\ \tau \triangleright i_1 \qquad \Delta, \alpha :: \kappa, x : \tau \vdash e_2 : \tau_2 \triangleright i_2 \qquad \alpha \notin \mathrm{FTV}(\tau_2)}{\Delta \vdash \mathrm{unpack}\ e_1\ \mathrm{as}\ \langle \alpha, x \rangle\ \mathrm{in}\ e_2 : \tau_2 \triangleright i_1 + i_2}\ \exists\mathrm{E}$$

$$\frac{\Delta \vdash (\exists a :: s.\ \tau) :: * \qquad \Delta \vdash i :: s \qquad \Delta \vdash e : \tau[i/a] \triangleright j}{\Delta \vdash \mathrm{pack}_{\exists a::s.\ \tau}\ \langle i, e \rangle : \exists a :: s.\ \tau \triangleright j}\ \exists_i\mathrm{I}$$

$$\frac{\Delta \vdash e_1 : \exists a :: s.\ \tau \triangleright i_1 \qquad \Delta, a :: s, x : \tau \vdash e_2 : \tau_2 \triangleright i_2 \qquad a \notin \mathrm{FIV}(\tau_2, i_2)}{\Delta \vdash \mathrm{unpack}\ e_1\ \mathrm{as}\ \langle a, x \rangle\ \mathrm{in}\ e_2 : \tau_2 \triangleright i_1 + i_2}\ \exists_i\mathrm{E}$$

$$\frac{\Delta \vdash e_m : o_{\mathrm{bt}}.\tau_m \triangleright i_m\ (m = 1, 2)}{\Delta \vdash e_1\ o_{\mathrm{bt}}\ e_2 : o_{\mathrm{bt}}.\tau_{\mathrm{res}} \triangleright i_1 + i_2}\ \textsc{Prim} \qquad \frac{\Delta \vdash e_m : \mathrm{nat}\ i_m \triangleright j_m\ (m = 1, 2)}{\Delta \vdash e_1 \oplus e_2 : \mathrm{nat}\ (i_1 + i_2) \triangleright j_1 + j_2}\ \textsc{Nat}_+$$

$$\frac{\Delta \vdash e_1 : \mathrm{nat}\ i \triangleright j_1 \qquad \Delta \vdash e_2 : \tau \triangleright j_2}{\Delta \vdash \mathrm{new}\ e_1\ e_2 : \mathrm{arr}\ \tau\ i \triangleright j_1 + j_2}\ \textsc{New} \qquad \frac{\Delta \vdash e_1 : \mathrm{arr}\ \tau\ i_1 \triangleright j_1 \qquad \Delta \vdash e_2 : \mathrm{nat}\ i_2 \triangleright j_2 \qquad \Delta \vdash i_2 < i_1}{\Delta \vdash e_1[e_2] : \tau \triangleright j_1 + j_2}\ \textsc{Rd}$$

$$\frac{\Delta \vdash e_1 : \mathrm{arr}\ \tau\ i_1 \triangleright j_1 \qquad \Delta \vdash e_2 : \mathrm{nat}\ i_2 \triangleright j_2 \qquad \Delta \vdash i_2 < i_1 \qquad \Delta \vdash e_3 : \tau \triangleright j_3}{\Delta \vdash e_1[e_2] := e_3 : 1 \triangleright j_1 + j_2 + j_3}\ \textsc{Wr} \qquad \frac{\Delta(\ell) = (\tau, i)}{\Delta \vdash \ell : \mathrm{arr}\ \tau\ i \triangleright 0}\ \textsc{Loc}$$

$$\frac{\Delta \vdash e : \tau \triangleright i_1 \qquad \Delta \vdash i_2 :: \mathrm{Time} \qquad \Delta \vdash i_1 \leq i_2}{\Delta \vdash e : \tau \triangleright i_2}\ \textsc{Relax} \qquad \frac{\Delta \vdash e : \tau_1 \triangleright i \qquad \Delta \vdash \tau_2 :: * \qquad \Delta \vdash \tau_1 \equiv \tau_2 :: *}{\Delta \vdash e : \tau_2 \triangleright i}\ \textsc{TyEq}$$

Fig. 10. Typing rules

3.4 Typing Examples

To help the reader build the right intuition about the syntax and typing rules, we write the fold-left example in the formal syntax:

$$\begin{array}{rcl}
\text{list} & \overset{\text{def}}{=} & \mu\gamma : * \Rightarrow \text{Nat} \Rightarrow *.\ \lambda\alpha : *.\ \lambda a : \text{Nat}.(\exists\_ : \{\_ | a = 0\}.\ 1) + \\
& & \qquad\qquad\qquad\qquad (\exists b : \text{Nat},\ \_ : \{\_ | a = b + 1\}.\ \alpha \times \gamma\ \alpha\ b) \\
\text{foldl} & \overset{\text{def}}{=} & \Lambda\alpha\ \beta : *.\ \text{rec}\ g.e \\
e & \overset{\text{def}}{=} & \Lambda m\ n : \text{Nat}.\ \lambda f : \alpha \times \beta \xrightarrow{m} \beta.\ \lambda y : \beta.\ \lambda l : \text{list}\ \alpha\ n. \\
& & \text{case unfold}\ l\ \text{of} \\
& & \quad z.\text{unpack}\ z\ \text{as}\ \langle\_,\_\rangle\ \text{in}\ y \\
& & \text{or}\ z.\text{unpack}\ z\ \text{as}\ \langle n', w\rangle\ \text{in unpack}\ w\ \text{as}\ \langle\_, u\rangle\ \text{in}\ g\ m\ n'\ f\ (f(u.1, y))\ u.2 \\
e & : & \forall m\ n : \text{Nat}.\ (\alpha \times \beta \xrightarrow{m} \beta) \xrightarrow{0} \beta \xrightarrow{0} \text{list}\ \alpha\ n \xrightarrow{T\ m\ n} \beta \\
T & \overset{\text{def}}{=} & \lambda m\ n.\ (m + 4) \times n.
\end{array}$$

Comparing with the source code in Listing 1, we can see that datatypes are translated into recursive types where the restriction on the index argument is translated into a refinement in each constructor. The unnamed index of the refinement sort as well as any extra index arguments in each constructor are existentially quantified. Pattern matching is translated into unfolding, case-analysis, and series of unpackings, the last of which makes the refinements in each constructor available in that branch. The running time of each branch should be bounded by the overall bound of the function. The first branch requires us to prove the trivial VC: $0 \leq T\ m\ n$; the second branch requires us to prove: $n = n' + 1 \rightarrow m + 4 + T\ m\ n' \leq T\ m\ n$.

Another illustrative example is a diverging recursive function, which is not typable in TiML:

$$\text{rec}\ f.\Lambda a : \text{Nat}.\ \lambda x : 1.\ f\ (a - 1)\ x$$

Its untypability is implied by the soundness theorem, which guarantees that every well-typed TiML program terminates. A more intuitive explanation is that one of the VCs it generates is

$$\forall a : \text{Nat}.\ a - 1 + 1 \leq a$$

which is not true without the premise $a \geq 1$ (our subtraction for Nat and Time is bounded below by zero). A proper structural recursion like foldl typechecks because the Cons branch gives us the premise $n = n' + 1$ (hence $n \geq 1$), which we do not have here (refining a to $\{a \mid a \geq 1\}$ will not work because the call $f\ (a - 1)$ will be rejected).

3.5 Soundness Theorem

The soundness theorem states the usual "nonstuckness" property that "well-typed terms cannot get stuck." It uses configuration typing defined by rule Config in Figure 11, which declares that the term and heap are well-typed, and the available fuel is no lower than what is statically estimated by the type system.

Definition 1 (Unstuck). *A configuration σ is unstuck iff $\sigma.2$ is a value or there exists σ' such that $\sigma \rightsquigarrow \sigma'$.*

Theorem 1 (Soundness). *For all Σ, τ, i, σ, and σ', if $\Sigma \vdash \sigma : \tau \triangleright i$ and $\sigma \rightsquigarrow^* \sigma'$, then σ' is unstuck.*

Section 5 sketches the proof (mechanized in Coq).

Note that in rule Config, we only require that if a location is well-typed then it contains a value of the expected type. We do not need to require the other way around (i.e. if a location contains a value then it should be well-typed). The intuition is that a heap type Σ is just an underspecification

$\boxed{\Sigma \vdash h}$

$$\frac{\forall \ell\ \tau\ i.\ \Sigma(\ell) = (\tau, i) \rightarrow \exists \vec{v}.\ h(l) = \vec{v} \land |\vec{v}| = [\![i]\!] \land \forall v \in \vec{v}.\ (\cdot, \cdot, \cdot, \Sigma) \vdash v : \tau \triangleright 0}{\Sigma \vdash h}\ \textsc{Heap}$$

$\boxed{\Sigma \vdash \sigma : \tau \triangleright i}$

$$\frac{(\cdot, \cdot, \cdot, \Sigma) \vdash e : \tau \triangleright i \qquad \vdash \mathrm{wf}\ \Sigma \qquad \Sigma \vdash h \qquad [\![i]\!] \leq r}{\Sigma \vdash (h, e, r) : \tau \triangleright i}\ \textsc{Config}$$

Fig. 11. Configuration typing

of the actual heap h. Well-typed programs will only access locations in h that are specified by Σ. h can have ill-typed junk values outside Σ's domain, and they will not affect programs' behavior.

3.6 Decidability

We aim for relative decidability, meaning that TiML typechecking *always* produces VCs that are true iff the original program should typecheck, even though the VCs do not obviously fall in a decidable theory. The relative decidability can be witnessed by a syntax-directed algorithmic version of the typing rules in Figure 10 by inlining rules Relax and TyEq. In practice, we use SMT solvers to decide the VCs, which are in theory not SMT-decidable because of nonlinear formulas like $m \times n$ from e.g. time bounds or array-bound checking, though Z3 seems to be pretty good at handling them on all our benchmarks.

4 BIG-O INFERENCE

TiML supports Hindley-Milner type inference and some index inference, particularly inferring big-O classes from recurrences. Types and indices can be omitted with underscores, and the typechecker generates unification variables (we call them *uvars* from here on) in place of these underscores. Type uvars are unified or generalized during typechecking per the Hindley-Milner algorithm; after typechecking there should not be any type uvars left. Index uvars are unified as much as possible during typechecking, though after typechecking there could remain some index uvars in the program and in VCs. Index uvars in VCs are converted into existentially quantified variables, and these VCs with existential quantifiers are sent to our heuristic pattern-matching-based recurrence solver.

The task of the recurrence solver is to massage the VCs in order to find patterns such as (1) of Section 1. A pattern-matching-based solver has the strength of being flexible and versatile, despite the weakness of being fragile in the face of superficial syntactical irregularities. The solver first lifts all irrelevant conjuncts out of existential quantifiers, so that under each existential quantifier are only conjuncts that are relevant in finding a value for the existential variable. Then it looks for VCs of the forms

$$\exists T.\ T \leqq g \land \forall m, n.\ A \leq T(m, n) \quad \text{or} \quad \exists g, T.\ T \leqq g \land \forall m, n.\ A \leq T(m, n).$$

The first form corresponds to the case where the programmer has provided a big-O specification (e.g. `msort`), while the second case corresponds to where the programmer has omitted the big-O class (e.g. `split`). The programmer-provided g will be ignored first in the first form, and the solver will come up with its own inference of g, which will be compared to the programmer-supplied specification.

The bulk of the solver's work is analyzing the A part. It treats A as a sum of terms and tries to find among these terms those of the form $T(m, \lceil q_i/b \rceil)$ or $T(m, \lfloor q_i/b \rfloor)$ with a common divisor b but possibly different q_i's where $q_i \leq n$. It uses an SMT solver to do equality and inequality tests to be more robust. After collecting these "subproblem" terms, it tries to find the big-O classes of the remaining terms. A big-O class has the form $mn^c \log^d n$. Some terms have their big-O classes in the premise context, such as T_{split} and T_{merge} in (1). Big-O classes are easy to combine for addition, multiplication, logarithm, and max. Finally, these collected terms are used to match the cases of the Master Theorem, which is shown below as a reminder.

Theorem 2 (Master Theorem). *For recurrence $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$:*
(1) if $f(n) \in O(n^c)$ where $c < \log_b a$, then $T(n) \in \Theta(n^{\log_b a})$;
(2) if $f(n) \in \Theta(n^c \log^k n)$ where $c = \log_b a$ and $k \geq 0$, then $T(n) \in \Theta(n^c \log^{k+1} n)$;
(3) if $f(n) \in \Omega(n^c)$ where $c > \log_b a$, and $af(n/b) \leq kf(n)$ for some $k < 1$ and sufficiently large n, then $T(n) \in \Theta(f)$.

To apply it to inequality recurrences, we only use the first two cases with Θ changed to O in the conclusion to get a sound (but sometimes not tight) bound. The Master Theorem gives a solution for g (an asymptotic bound), not a solution for T (a concrete bound), so subsequent use of T cannot rely on its properties other than being of $O(g)$. Aside from using the Master Theorem for divide-and-conquer-like VCs, the solver also looks for VCs of the form $T(n-1) + O(f(n)) \leq T(n)$ and infers big-O class $T(n) \in O(nf(n))$. This heuristic is useful for simple "remove one" recursion schemes common in list processing. We can see from the above procedure that we only support a limited form of multivariate complexities, where n is the main variable and m is just a passive factor. Such passive factors are mainly used in cases where an algorithm takes in a primitive operation as parameter (like the `le` comparator taken in by function `msort`). Another limitation is that bounds such as $n \log n$ are only supported through this big-O mechanism. The programmer cannot specify a precise $n \log n$-like bound (not a big-O class) and tell the SMT solver to discharge the VCs, since we have not been able to teach the SMT solver to discharge VCs involving $n \log n$.

5 PROOF

We formalized TiML and its soundness proof in Coq, which we believe is the first mechanized soundness proof for a complexity type system. The Coq proof can be found in file `proof/Soundness.v` in the source-code tarball submitted as supplemental material. The Soundness Theorem (Theorem 1 in Section 3, Theorem `soundness` in the Coq proof) is proved by the usual "preservation + progress" approach, with the preservation and progress lemmas shown below. $[\![i]\!]$ stands for the denotational semantics (i.e. interpretation) of index i, explained later in this section.

Lemma 1 (Progress). *For all Σ, τ, i, and σ, if $\Sigma \vdash \sigma : \tau \triangleright i$, then* $\text{unstuck}(\sigma)$.

Proof. Induction on the typing derivation. See Coq proof of lemma `progress` for details. □

Lemma 2 (Preservation). *For all Σ, τ, i, σ, and σ', if $\Sigma \vdash \sigma : \tau \triangleright i$ and $\sigma \rightsquigarrow \sigma'$, then there exist Σ' and i' such that $\Sigma' \vdash \sigma' : \tau \triangleright i'$.*

Proof. Appeal to Lemmas 3 and 4. □

Lemma 3 (Atomic Preservation). *For all $\Sigma, \tau, i, \sigma,$ and σ', letting Δr be $\sigma.3 - \sigma'.3$, if $\Sigma \vdash \sigma : \tau \triangleright i$ and $\sigma \twoheadrightarrow \sigma'$, then $[\![i]\!] \geq \Delta r$ and there exists Σ' such that $\Sigma' \vdash \sigma' : \tau \triangleright (i - \Delta r)$ and $\Sigma \subseteq \Sigma'$.*

Proof. Induction on the atomic stepping derivation (the second premise). See Coq proof of lemma `preservation_atomic` for details. □

Lemma 4 (Ectx Typing). *Let e be $E[e_1]$ (the plugging of term e_1 into evaluation context E). For all Σ, τ, and i, letting Δ be $(\cdot, \cdot, \cdot, \Sigma)$, if $\Delta \vdash e : \tau \triangleright i$ and $\vdash \mathrm{wf}\ \Delta$, then there exist τ_1 and i_1 such that $\Delta \vdash e_1 : \tau_1 \triangleright i_1$, $[\![i_1]\!] \leq [\![i]\!]$, and for all e'_1, Σ', and i'_1, letting Δ' be $(\cdot, \cdot, \cdot, \Sigma')$ and e' be $E[e'_1]$, if $\Delta' \vdash e'_1 : \tau'_1 \triangleright i'_1$, $\vdash \mathrm{wf}\ \Delta'$, $[\![i'_1]\!] \leq [\![i_1]\!]$, and $\Sigma \subseteq \Sigma'$, then $\Delta' \vdash e' : \tau \triangleright i - i_1 + i'_1$.*

Proof. Induction on the definition of the context-plugging operation. See Coq proof of lemma `ectx_typing` for details. □

Lemma 3 is the version of the preservation lemma for atomic steps, also strengthened with the time bound on the post-configuration explicitly specified as $i - \Delta r$. Lemma 4 is a characterization of the typing property of evaluation contexts. It says that if a compound term is well-typed, then the inner term is well-typed, and if one replaces the inner term with another term of the same type, the type of the compound term will not change. It also reflects the intuition that the running time of a compound term is the running time of the inner term plus that of the evaluation context. The proofs of the above lemmas make use of various versions (for sorting/kinding/typing) of substitution lemmas, canonical-value-form lemmas, and weakening lemmas, for each of which we show one example below. Another important lemma is the invariant that the typing judgments guarantee the result type and time are of proper kind/sort.

Lemma 5 (Substitution). *For all Δ, e_1, τ_1, i_1, e_2, x, and τ, if $\Delta, x : \tau \vdash e_1 : \tau_1 \triangleright i_1$, $\Delta \vdash e_2 : \tau \triangleright 0$ and $\vdash \mathrm{wf}\ \Delta$, then $\Delta \vdash e_1[e_2/x] : \tau_1 \triangleright i_1$.*

Proof. Induction on the typing derivation for e_1. See Coq proof of lemma `typing_subst_e_e` for details. □

Note that in the above lemma we require time 0 in e_2. The reason is that variable x may have multiple appearances in e_1. If e_2 has nonzero running time, then after the substitution there may be multiple copies of e_2, and the resulting time will not be simply $i_1 + i_2$. We got away with fixing i_2 to 0 because in all the proofs, term substitution only happens when the substitute is a value.

Lemma 6 (Canonical Value Form). *For all Σ, v, τ_1, τ_2, i, and i', letting Δ be $(\cdot, \cdot, \cdot, \Sigma)$, if $\Delta \vdash v : \tau_1 \xrightarrow{i} \tau_2 \triangleright i'$ and $\vdash \mathrm{wf}\ \Delta$, then there exists e such that $v = \lambda x.\ e$.*

Proof. Induction on the typing derivation. See Coq proof of lemma `canon_TArrow` for details. □

Lemma 7 (Weakening). *For all Δ, e, τ, i, x, and τ', if $\Delta \vdash e : \tau \triangleright i$, then $\Delta, x : \tau' \vdash e : \tau \triangleright i$.*

Proof. Induction on the typing derivation. See Coq proof of lemma `typing_shift_e_e` for details. □

Lemma 8 (Typing-Kinding). *For all Δ, e, τ, and i, if $\Delta \vdash e : \tau \triangleright i$ and $\vdash \mathrm{wf}\ \Delta$, then $\Delta \vdash \tau :: *$ and $\Delta \vdash i :: \mathrm{Time}$.*

Proof. Induction on the typing derivation. See Coq proof of lemma `typing_kinding` for details. □

One complication during the proof is the treatment of type equivalence. The definition of type equivalence should admit both equivalence rules (particularly transitivity) and good inversion lemmas such as Lemma 9. We follow the method in Chapter 30 of Pierce [2002], defining type equivalence with congruence, reduction, and equivalence rules (see the Coq definition of `tyeq`), and then we prove inversion lemmas via a "parallel reduction" version of type equivalence that enjoys a "diamond" property. Our solution is more involved because even for comparing normalized types we cannot use a syntactic equality test, for we allow two semantically equivalent indices to be treated as equal. In the Coq code we use the relation `cong` to compare normalized types, which uses a semantic-equivalence test to compares indices. Because all properties about the denotational semantics (see below) of indices require well-sortedness, we need to put kinding constraints in type equivalence rules. Particularly, the transitivity rule needs a kinding constraint on the intermediate type. All judgments involving types should be morphisms on `tyeq` equivalence, expressed as lemmas such as Lemma 10.

Lemma 9 (Invert TyEq TArrow). *For all Δ, τ_1, i, τ_2, τ_1', i', τ_2, and κ, if $\Delta \vdash \tau_1 \xrightarrow{i} \tau_2 \equiv \tau_1' \xrightarrow{i'} \tau_2' :: \kappa$, $\Delta \vdash \tau_1 \xrightarrow{i} \tau_2 :: \kappa$, $\Delta \vdash \tau_1' \xrightarrow{i'} \tau_2' :: \kappa$, and $\vdash \text{wf } \Delta$, then $\Delta \vdash \tau_1 \equiv \tau_1' :: \kappa$, $\Delta \vdash i = i'$, and $\Delta \vdash \tau_2 \equiv \tau_2' :: \kappa$.*

Proof. See Coq proof of lemma `invert_tyeq_TArrow` for details. □

The judgment $\Delta \vdash i = i'$ used above is just an instance of the validity judgment $\Omega \vdash \theta$. Notice the paper's convention that we may write Δ when only its Ω part is needed.

Lemma 10 (TCtx TyEq). *For all $\Omega, \Lambda, \Sigma, \Gamma, \Gamma', e, \tau$, and i, if $(\Omega, \Lambda, \Sigma, \Gamma) \vdash e : \tau \triangleright i$, $(\Omega, \Lambda) \vdash \text{wf } \Gamma'$, Γ and Γ' have the same domain and have equivalent types at each variable, then $(\Omega, \Lambda, \Sigma, \Gamma') \vdash e : \tau \triangleright i$.*

Proof. See Coq proof of lemma `tctx_tyeq` for details. □

We had two unsuccessful attempts to formalize properties of type equivalence before we embarked on the current approach. The intuition of the two failed approaches was the same as the current one: bake reduction and equivalence rules into the definition and prove that the definition coincides with another relation that admits good inversion lemmas. We first tried to prove that type equivalence coincides with a set of logical relations. Logical relations are a technique for proving contextual equivalence of two open terms, by first defining logical equivalence on closed terms and then extending it to open terms via equivalent substitutions. We ran into trouble with this approach because our types are indexed with open indices, and the index/sort system is a dependent type system. In our second attempt, we tried a fully denotational approach by interpreting open types as functions that return closed type normal forms, proving that type equivalence coincides with equivalence of those functions (assuming functional extensionality). This approach disallows impredicative polymorphic types, whose presence makes type normal forms undefinable, because the cardinality of one of the normal form's constructors (the polymorphic type case) is larger than that of the normal form itself.

Our final type-equivalence definition includes three categories of rules: (1) congruence rules, for example, $\tau_1 \times \tau_2 \equiv \tau_1' \times \tau_2'$ if $\tau_1 \equiv \tau_1'$ and $\tau_2 \equiv \tau_2'$; (2) reduction rules, for example, $(\lambda\alpha.\ \tau_1)\ \tau_2 \equiv \tau_1[\tau_2/\alpha]$; (3) equivalence rules, i.e., reflexivity, transitivity, and symmetry. For a compound example, $(\lambda\alpha.\ \tau_1 \times \alpha)\ \tau_2 \times \tau_3 \equiv \tau_1 \times \tau_2 \times \tau_3$.

In the Coq formalization, we use de Bruijn indices throughout, and since we have the three-tier index, type, and term sublanguages, we have to define multiple forms of substitution and shifting corresponding to each pair of sublanguages. A major part of the proof is establishing harmonious interactions between various forms of judgments and various forms of substitution and shifting.

Another technical hurdle is formalizing the denotational semantics of indices. The denotational semantics (i.e. the interpretation function) should have good reduction behavior to be used as an evaluator, and it needs to admit substitution lemmas such as Lemmas 11 and 12. In our Coq formalization, the interpretation function $[\![\cdot]\!]$ takes in a base-sorting context, a result base sort, and an index, and interprets the index according to the context and the result sort. For example: $[\![a+b+1]\!]^{\mathrm{Nat}}_{[a:\mathrm{Nat}, b:\mathrm{Nat}]}$ = `(fun a b : nat => a+b+1)`, where `(fun a b : nat => a+b+1)` is a Coq term. The intuition is straightforward: closed indices, when interpreted in an empty context, denote just numbers (e.g. $[\![3]\!]^{\mathrm{Nat}}_{[]}$=3); open indices with variables, when interpreted in a proper context, denote functions (e.g. the $a+b+1$ example) that can also be seen as numbers that are parameterized on the values of the free variables. The result base sort is needed to make the Coq type of $[\![\cdot]\!]$ simpler; otherwise $[\![\cdot]\!]$'s type is complexly dependent on the index. We omit the subscript when the context is empty and the superscript when it can be inferred. The denotational semantics is fixed once and for all. Please see the Coq definition of function `interp_idx` for details.

Lemma 11 (Interp Subst Index). *For all Δ, a, i_1, s_1, i_2, and s_2, if $\Delta, a :: s_2 \vdash i_1 :: s_1$ and $\Delta \vdash i_2 :: s_2$, then $[\![i_1[i_2/a]]\!] = [\![i_1]\!]([\![i_2]\!])$.*

Proof. See Coq proof of lemma `interp_subst_i_i` for details. □

Lemma 12 (Interp Subst Prop). *For all Δ, a, θ, i, and s, if $\Delta, a :: s \vdash \theta$, $\Delta \vdash i :: s$, $\Delta \vdash \mathrm{wf}\ \theta$ and $\vdash \mathrm{wf}\ \Delta$, then $\Delta \vdash \theta[i/a]$.*

Proof. See Coq proof of lemma `interp_subst_i_p` for details. □

The validity judgment $\Omega \vdash \theta$ is defined as first collecting refinements in Ω as θ' and then interpreting the syntactic proposition $\theta' \rightarrow \theta$ in the meta-logic. The latter is done in a similar way as the interpretation of indices. For example, $a : \mathrm{Nat}, b : \{\mathrm{Nat} | a = b\} \vdash a + 1 = b + 1$ is defined as $\forall a\ b : \mathrm{nat},\ a = b \rightarrow a + 1 = b + 1$.

6 EVALUATION

We implemented the TiML typechecker in SML from scratch, not using existing parser or typechecker implementations for similar languages, for maximal flexibility. We have tested the TiML typechecker on 17 benchmarks, incorporating classic data structures and algorithms including trees, doubly linked lists, insertion sort, array-based merge sort (copying and in-place), quicksort, binary search, array-based binary heap, k-median search, red-black trees, Braun trees, Dijkstra's algorithm for graph shortest paths, functional queues (amortized analysis), and dynamic tables (amortized analysis). The test is run on a 2.5-GHz quad-core Intel Core i7 CPU with 16GB RAM (actual memory usage is within 256MB). The SMT solver we use is Z3 [De Moura and Bjørner 2008] 4.4.1. Table 1 lists each benchmark's filename, description, total time of typechecking (including time for parsing, typechecking, inference, and VC solving), lines of code, lines of code that contain time annotations, and asymptotic complexities of the most representative top-level functions. Every benchmark finishes within 1 second, most of them within 0.3 seconds. The code of all benchmarks is available in the `examples` directory in the source-code tarball submitted as supplemental material.

As an empirical study of the usability of TiML, we explain each benchmark and analyze the annotations in it. Most of the annotations are at two places: the types of recursive functions and pattern matches. The former is akin to pre/post-conditions and loop invariants in program logics. The latter sometimes require annotations because of the "forgetting problem" of existential-type eliminations: the running time and result type of unpack should not contain the locally introduced type/index variables, but it can be hard for the typechecker to figure out how to forget them. We allow `using` and `return` clauses on `case` to specify the common running time and result type of all branches, which cannot reference branch-local variables. It is similar to the `return` clauses of dependent pattern-matching in Coq. We use some syntactical tricks to guess these clauses. For

Table 1. Benchmarks. Columns show total time of typechecking (parse+typecheck+inference+VC solve), lines of code, lines of code containing time annotations, and asymptotic complexities of the most representative top-level functions.

Name	Description	Time (s)	LoC	LoC.A.	A. Comp.
list	List operations	0.155	48	9	n, mn
ragged-matrix	Ragged matrices	0.113	16	1	$m^2 n$
tree	Trees	0.18	86	10	mn
msort	Merge sort	0.221	49	8	$mn \log n$
insertion-sort	Insertion sort	0.142	25	4	mn^2
braun-tree	Braun trees	0.199	98	11	$\log n$, $\log^2 n$
rbt	Red-black trees	0.422	316	19	$\log n$
dynamic-table	Dynamic tables	0.153	126	10	(amortized) 1
functional-queue	Functional queues	0.137	95	6	(amortized) 1
array-bsearch	Binary search	0.149	44	2	$m \log n$
array-heap	Binary heap	0.221	139	6	$m \log n$
array-msort	Merge sort on arrays	0.228	112	7	$mn \log n$
array-msort-inplace	In-place merge sort on arrays	0.255	133	9	mn^2
array-kmed	k-median search	0.16	70	8	mn^2
dlist	Doubly linked lists	0.26	112	10	mn
qsort	Quicksort	0.128	43	7	mn^2
dijkstra	Dijkstra's alg. (shortest paths)	0.12	75	0	$(m_+ + m_\leq)n^2$

example, if a case analysis is directly under a recursive function, we copy the `using` and `return` annotations in the recursive-function signature.

All annotations in benchmark `list` (list operations) are types of recursive functions similar to `foldl`. Benchmark `ragged-matrix` contains lists of lists with one index being the length of the outer list and another index being the maximal length of the inner lists. Benchmark `tree` contains binary trees and operations on them such as map, fold, and flatten. The annotations in these two files are similar to those in `list`. Benchmark `msort` has been discussed in Section 1. Benchmark `insertion-sort` is specified using big-O similarly to `msort`.

Benchmark `braun-tree` contains Braun trees [Okasaki 1997], a kind of balanced binary trees for functional implementation of priority queues. In a Braun tree, each node stores a value that is smaller than all values in the children, and the size of the left child is equal to or larger by one than that of the right child. It supports enqueue and dequeue in $O(\log n)$ and $O(\log^2 n)$ time respectively. We define Braun trees as binary trees indexed by size (i.e. number of nodes), and for a Braun tree of size $n + 1$, we require the sizes of its left and right child to be $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ respectively. All functions in `braun-tree` are specified with big-O complexities, so the time-annotation burden is on par with `msort`'s. In this benchmark, some implicit-index-argument inference failed, so we have to supply index arguments explicitly using the `@fun_name` syntax (similar to Coq). In benchmark `rbt` for red-black trees, we have to put two extra invariants in the definition of `rbt` other than those shown in Section 1. These two invariants can be derived from the other invariants, but because in TiML lemmas, lemma invocations, and inductions must be written as ordinary functions (like in Dafny) which increase a program's running time, deriving these invariants on the fly will increase asymptotic complexity. Annotation burden for time is again on par with `msort`'s since big-O complexities are used. In the jump from black-height to logarithm of tree size, an assumed lemma is used relating logarithms and exponentials.

Benchmarks `functional-queue` and `dynamic-table` are two examples showing how to use TiML to conduct amortized analysis. In a standard cost analysis, a function's time is specified in terms of only the input size. Let us write such a type as $\forall n.\ \tau_n \xrightarrow{g(n)} \tau'$. In the "potential method" for doing amortized complexity analysis [Cormen et al. 2009], the running time c is specified by an inequality $c + \Phi_1 \leq c_a + \Phi_0$. Here Φ is a (nonnegative) *potential function* defined on configurations (i.e. states), and Φ_0 and Φ_1 are the potentials before and after the function. c_a is called "amortized cost." We can write the type of such a function as $\forall n.\ \exists c\ n'.\ \tau_n \xrightarrow{c} \tau_{n'} \wedge P(n, c, n')$, where $P(n, c, n') \overset{\text{def}}{=} c + \Phi(n') \leq c_a + \Phi(n)$ (c_a is a parameter). Because sometimes c and n' need to depend on the input value (not just its type), the existential quantifiers need to be pushed later: $\forall n.\ \tau_n \xrightarrow{k} \exists c.\ 1 \xrightarrow{c} \exists n'.\ \tau_{n'} \wedge P(n, c, n')$, where k is a constant. We call type $\exists n'.\ \tau_{n'} \wedge P(n, c, n')$ `some_output_and_cost_constraint` and type $\exists c.\ 1 \xrightarrow{c} \exists n'.\ \tau_{n'} \wedge P(n, c, n')$ `amortized_comp` ("amortized computation") in the TiML code.

A functional queue [Okasaki 1999] is a queue implemented by two stacks, one for receiving input, the other for supplying output. When the output stack is empty, the content in the input stack is dumped to the output stack, in reverse. A dynamic table [Cormen et al. 2009] (like the "vector" container in C++'s STL) is a dynamically allocated buffer that enlarges itself when the load factor becomes too high after an insertion, and shrinks itself when the load factor becomes too low after a deletion. Both of these two data structures enjoy amortized constant-time insertion and deletion. Note that TiML does not have any built-in support for amortized analysis, so being able to do it somewhat surprised us. It is in line with many language designers' experience that when one starts with primitives to encode the most basic concepts, many computational phenomena will arise naturally.

Benchmarks `array-bsearch`, `array-heap`, `array-msort`, and `array-msort-inplace` are array-based implementations for binary search, binary heaps, and merge sort (copying and in-place). Their time-annotation burdens are on par with `foldl` and `msort`'s. Benchmark `array-kmed` does k-median search on an array. Big-O inference fails in this benchmark, so precise bounds are given. The VC that fails the Big-O inference is $15 + m + \max(T(m, n-1), T(m, n-1) + 4) \leq T(m, n) - 8$. The culprit is the "$-8$" on the right-hand side. The Big-O inferrer only recognizes a recurrence whose right-hand side is $T(\dots)$. We can add annotations to massage the VC by moving "-8" to the left-hand side, which may or may not be better than just spelling out $T()$ in this case. Function `array_kth_median_on_range` uses local time annotation to forget a local index variable. Benchmark `dlist` implements doubly linked lists using references (i.e. arrays). Each function just needs one big-O annotation. Benchmark `qsort` for quicksort requires a rather detailed time annotation for function `list_qsort` to forget the two local index variables that are the lengths of the two partitions. The running time of `list_qsort` is first calculated in terms of these two lengths, and it is very hard for the typechecker to figure out how to replace these two lengths with the total length of the input list, hence the annotation. Benchmark `dijkstra` implements Dijkstra's algorithm for calculating shortest paths, which surprisingly does not require any time annotation. The reason is that the algorithm is implemented by mainly using standard iterators such as `app`, `appi`, and `foldli` provided by the `Array` module in the standard library, demonstrating the power of modularity preserved by TiML from SML.

Among the benchmarks, `braun-tree`, `rbt`, `dynamic-table`, `functional-queue`, and all the array-based algorithms crucially rely on the refinement mechanism to encode data-structure invariants and algorithm preconditions.

As an empirical data point, an undergraduate student with background in SML took just one day to become fluent in writing and annotating TiML programs.

7 RELATED WORK

The design of TiML is highly influenced by the Dependent ML (DML) language of Xi and Pfenning [1999]. The ideas of indexed types and refinement kinds are from their DML work. They did not explore the possibility of using their techniques to tackle the problem of static guarantees of program execution time. A follow-on project [Grobauer 2001] also missed TiML's central idea that arrow types can be indexed by the function's time bound. The work of Xi and Pfenning [1998, 1999] has shown that their approach is powerful enough to accomplish tasks like static array-bound checking and dead-code elimination. We take these abilities for granted and only focus on time-complexity-related capabilities in this paper.

The Automatic Amortized Resource Analysis (AARA) line of work was initiated by Hofmann and Jost [2003] and successfully pursued by Hoffmann et al. The original idea was to associate a uniform potential with each list node in an affine type system. Aliasing is allowed by partitioning the potential among the aliases. Typechecking generates linear inequalities involving these (yet-unknown) potential coefficients, and solving this linear constraint system by a linear-programming solver can give a consistent assignment to these potential coefficients. The power of this idea is that it bypasses the difficult problem of recurrence solving altogether, yielding a push-button approach, at the cost of only supporting linear bounds on monomorphic first-order programs. Later work extended it to univariate [Hoffmann and Hofmann 2010] and multivariate [Hoffmann et al. 2011] polynomial bounds by associating a uniform potential to not only one node but every tuple of nodes (of certain types); other extensions support higher-order programs [Jost et al. 2010], parallel programs [Hoffmann and Shao 2015], and a large portion of OCaml [Hoffmann et al. 2017]. The bounds must be polynomial, and invariants are not supported. The latest treatment of higher-order functions [Hoffmann et al. 2017] is somewhat unsatisfactory in that a higher-order function does not have a type that fully abstracts its behavior, meaning that at call sites the callee's code (not only its type) must be available to do typechecking and resource analysis. As a consequence, it is not possible to do separate typechecking/compilation where library functions' source code is unavailable. In contrast, TiML's typechecking is fully modular in that at a function's call site only its type is needed. The AARA amortization scheme is akin to "the banker's method" [Tarjan 1985] and ours to "the physicist's method."

Aspinall et al. [2007] introduced a program logic on JVM bytecode for resource verification formalized in Isabelle/HOL. It is a partial-correctness program logic enjoying soundness and relative completeness, complemented by a termination logic. Parameterized on a resource algebra, the logic is very general, but verification (i.e. proving) is all manual, and resource recurrences (arising from the Consequence rule) must be solved by hand. It is intended as a target language compiled from the type system of Hofmann and Jost [2003] in a proof-carrying-code [Necula 1997] setting. We agree that program logics should be treated as one level lower on a stack of formal systems than type systems. Atkey [2010] proposed another program logic based on separation logic with resource potentials, together with a VC generator (requiring loop-invariant annotations) and a proof-search system that can automatically discharge VCs and use a linear-programming solver to infer resource annotations (influenced by Hofmann and Jost [2003]). This system, formalized in Coq, is the closest to TiML's design goal of supporting both highly automatic analysis and expressive invariants, though it is only for a first-order language. Designing program logics for higher-order languages is generally a hard problem. Charguéraud and Pottier [2015] proved the inverse-Ackermann bound of union-find in Coq using their characteristic formulae (CFML) framework augmented with potentials. The proof is manual, but the CFML framework (essentially an axiomatic semantics for OCaml) is shown to be very expressive. McCarthy et al. [2016] introduced another Coq framework for time verification, which is based on a Coq monad that is indexed not by time but by a predicate mentioning time (similar to the Dijkstra monad of Swamy et al. [2013]) which can serve as both time and correctness invariant. Proofs are written manually. Both systems require "pay" or "tick" primitives to be inserted at time-consumption sites, either manually or by a program transformation. Danielsson [2008] described an Agda library whose core constituent is a graded monad called "Thunk" whose index stands for the running time of this thunk. The novelty is the treatment of lazy evaluation with memoization, manifested by the "pay" primitive and the operational semantics for the thunked language. On the practical side, it suffers from the usual nuisance of working in an intensional dependent type system: indices must be definitionally equal for two types to be unifiable.

Sized types [Hughes et al. 1996; Reistad and Gifford 1994; Vasconcelos and Hammond 2004] are similar to indexed types in our setting, though the latter do not have any built-in size-related meaning while Hughes et al. [1996] gave a denotational semantics to the former relating types to their sizes. Vasconcelos and Hammond [2004] described an algorithm for automatically generating cost recurrences from sized-type programs (but did not deal with recurrence solving). Sized types do not support refinements. Çiçek et al. conducted a line of work [Çiçek et al. 2017, 2016; Çiçek et al. 2015] on analyzing incremental and relational time complexity. They use the term "refinement types" in the broad sense of "types enriched with other information," in contrast with our use in the narrow sense of "types of the form $\{x : t \mid P(x)\}$". They do not support refinements in the latter sense. Because the asynchronous rules in their type systems arbitrarily make the choice of relating which subpart of the first program to which subpart of the second program, the authors were unable to devise a typechecking algorithm. Crary and Weirich [2000] presented a type system for resource-bound certification with indexed types. Indices and types are unified in their language, and inductive kinds and primitive recursion on the type level are supported. The program is intended to be machine-generated, so annotation burden is heavy (e.g. one cannot relax a time bound without annotating the "padding" amount). Refinements are not supported.

Madhavan and Kuncak [2014] studied complexity analysis of a first-order language where they focused on inferring constants in postcondition templates that can mention time, which is represented by an instrumented counter variable. Other than postconditions, it does not allow refinements in other places, so we do not see how it can, for example, relate a red-black tree's black-height to its size, which requires invariants of the data structure. Madhavan et al. [2017] verified resource usage for higher-order functions with memoization by transforming the source program into a first-order program instrumented by resources and then generating VCs in Hoare-logic style. They also support index inference by counterexample-guided search. The idea of defunctionalization is also exploited by Avanzini et al. [2015]. We do not want to use defunctionalization because we want to do fully modular complexity analysis, which requires analyzing higher-order functions without knowing the set of all possible argument functions. We can learn from Madhavan and Kuncak [2014] and Madhavan et al. [2017] when it comes to index-inference techniques.

Gimenez and Moser [2016] introduced a resource-annotated operational semantics and type system for interaction nets, with the novel notions of "space-time complexity" and "scheduled types" for guaranteeing the availability pace of data. Since the interaction-net language lacks recursion, the programmer (not the language designer) has to define a new node for each operation like map/fold and its potential function. The paper gives potential functions for the nodes it uses but does not show how to choose such potential functions for new operations. Annotation inference is not addressed, and invariants are not supported. Dal Lago and Petit [2013] did complexity analysis with linear dependent types, which are indexed linear types with a special index i meaning "the ith copy of this term." Linear dependent types enjoy relative completeness. Ghica and Smith [2011] analyzed the complexity of a concurrent Algol-like language that is compiled to hardware circuits directly, by using indices in types to control contraction in parallel compositions. Danner et al.

[2015] proposed a procedure to automatically transform a program into a certain form and read off the complexity-recurrence equations from there, but they did not address recurrence solving. Benzinger [2001, 2004] used the Mathematica computer-algebra system to solve some forms of recurrences.

Liquid types [Rondon et al. 2008; Vazou et al. 2015, 2013] popularized refinement types as a practical middle ground between traditional ML types and full dependent types. The appeal of liquid types lies in their support of automatic refinement inference, made possible by fixing a set of qualifiers and iteratively weakening unsatisfied VCs by removing offending qualifiers. We would like to apply liquid-type techniques to enable automatic refinement inference in TiML, though these techniques are not very good at inferring constants in numerical formulas compared to e.g. counterexample-guided approaches.

Aside from the community working on type systems and software verification, complexity analysis has been studied for many years by the program-analysis community. The COSTA line of work [Albert et al. 2008, 2007, 2008, 2009] aimed at cost analysis of Java bytecode. Albert et al. [2008] made inroads in recurrence solving by converting cost relations to direct recursions and analyzing their evaluation trees. The latter is done by bounding the branching factors, the tree depths, and the sizes of the nodes. The SPEED line of work [Gulavani and Gulwani 2008; Gulwani et al. 2009; Gulwani and Zuleger 2010] did automatic complexity analysis of first-order imperative programs by instrumenting the program with multiple counters and using off-the-shelf abstract-interpretation-based linear invariant-generation tools to infer invariants on these counters automatically. Brockschmidt et al. [2014] did complexity analysis of integer programs by alternating time and size analysis on small parts of the program. Srikanth et al. [2017] handled nonlinear theories by lazily instantiating theorems that are sufficient to approximate a nonlinear theory.

8 LIMITATIONS AND FUTURE WORK

Because of its middle-way approach, TiML cannot match push-button methods' convenience or the full proof ability of systems embedded in a proof assistant. Other than these two fundamental limitations, the current state of TiML has several shortcomings. (1) The pattern-matching-based recurrence solver is versatile albeit fragile. If the syntactic form of the VC is not something we anticipate, the solver tends to fail to recognize the recurrence pattern. (2) The invariants are fixed in a datatype's definition, so if another algorithm on the datatype requires another set of invariants, a new datatype has to be defined. (3) Existential types bring with them the "forgetting problem," which incurs extra annotation burden. (4) The index language is (so far) restricted to numbers. It cannot specify, for example, `map` with an argument function whose running time depends on the sizes of individual list elements. Doing that requires indexing lists with indices of sort "List". (5) TiML does not have built-in support for amortized analysis or memoization. The former can be simulated to some degree as shown in Section 6, while we have not investigated the latter.

We also did not treat subtyping. A natural source of subtyping is allowing $\tau_1 \xrightarrow{i} \tau_2$ to be a subtype of $\tau_1 \xrightarrow{j} \tau_2$ when $i \leq j$ and then propagating it via co- and contra-variance. This relaxation can be useful, for example, when a higher-order function expects a function argument of type $\mathrm{int} \xrightarrow{100} \mathrm{int}$ while an actual argument of type $\mathrm{int} \xrightarrow{90} \mathrm{int}$ is supplied. We did not add subtyping simply because we did not come across this scenario in our benchmarks. The reason is that our benchmarks are always polymorphic in the running time of function arguments (e.g. `fun {m : Time} (f : int -`$\boxed{\mathtt{m}}$`→ int) ...`). So polymorphism saved us and allowed us to get away with not having subtyping. We agree subtyping is still useful and hard to mimic by polymorphism in situations like a function in a record, and we would like to add it in the future. We suspect it will not disrupt the soundness proof too much (we should just need to change type equivalence to subtyping and tweak the type-equivalence-inversion and canonical-value-form lemmas). It does make index inference harder because many VCs of the form $i = j$ will become $i \leq j$.

We are working to address some of these limitations and exploring future extensions. Some easy extensions are a richer index language (possibly with inductive indices), a more detailed cost model with parameterized constants representing the costs of operations on an architecture, and deterministic parallelism measured by "work" and "depth." To count the "depth" measurement of parallel programs, we just need to change $i_1 + i_2$ to $\max\{i_1, i_2\}$ in rules like →E. On the solver back-end, we want to make recurrence solving more robust by using a more sophisticated recurrence solver, and we want to conduct more aggressive index and refinement inference using software-synthesis techniques (e.g. liquid types and counterexample-guided approaches). Zooming out to the bigger picture, we are working on a complexity-preserving compiler that translates TiML's typing derivations (i.e. complexity proofs) down to the typing derivations of a time-annotated assembly language, and on an extended version of TiML that takes into account the time cost of garbage collection.

9 CONCLUSION

We described TiML, a higher-order polymorphic functional language with refinements and static complexity guarantees. It allows verifying complexities of data structures with intricate invariants without resorting to manual proof. Annotation burden is lowered by type and index inference and furthermore by a pattern-matching-based recurrence solver. We studied the metatheory of TiML by a mechanically checked soundness proof, and we argued for the practicality of TiML by implementing its typechecker and empirically evaluating it on a suite of benchmarks. We want to promote this middle-way approach between push-button analysis systems and manual proof systems to have useful and affordable static guarantees with the programmer's help.

ACKNOWLEDGMENTS

We thank Benjamin Barenblat, Jan Hoffmann, Clément Pit–Claudel, Paul Steckler, Murali Vijayaraghavan, and the anonymous reviewers for their careful review of this paper and helpful feedback. This work has been supported in part by National Science Foundation grant CCF-1512611 and by DARPA under agreement number FA8750-16-C-0007. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2008. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Proceedings of the 15th International Symposium on Static Analysis (SAS 2008)*. Springer-Verlag, 221–237.

E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. 2007. Cost Analysis of Java Bytecode. In *Proceedings of the 16th European Symposium on Programming (ESOP 2007)*. Springer-Verlag, 157–172.

Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. 2008. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Formal Methods for Components and Objects: 6th International Symposium (FMCO 2007)*. Springer Berlin Heidelberg, 113–132.

Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa Gil. 2009. Live Heap Space Analysis for Languages with Garbage Collection. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM 2009)*. ACM, 129–138.

David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. 2007. A Program Logic for Resources. *Theoretical Computer Science* 389, 3 (Dec. 2007), 411–445.

Robert Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP 2010)*. Springer-Verlag, 85–103.

Martin Avanzini, Ugo Dal Lago, and Georg Moser. 2015. Analysing the Complexity of Functional Programs: Higher-order Meets First-order. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, 152–164.

Ralph Benzinger. 2001. Automated Complexity Analysis of Nuprl Extracted Programs. *Journal of Functional Programming* 11, 1 (Jan. 2001), 3–31.

Ralph Benzinger. 2004. Automated Higher-order Complexity Analysis. *Theoretical Computer Science* 318, 1-2 (June 2004), 79–103.

Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2014. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference (TACAS 2014)*. Springer Berlin Heidelberg, 140–155.

Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 316–329.

Ezgi Çiçek, Zoe Paraskevopoulou, and Deepak Garg. 2016. A Type Theory for Incremental Computational Complexity with Control Flow Changes. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, 132–145.

Arthur Charguéraud and François Pottier. 2015. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving: 6th International Conference (ITP 2015)*. Springer International Publishing, 137–153.

Ezgi Çiçek, Deepak Garg, and Umut Acar. 2015. Refinement Types for Incremental Computational Complexity. In *Programming Languages and Systems: 24th European Symposium on Programming (ESOP 2015)*. Springer Berlin Heidelberg, 406–431.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

Karl Crary and Stephnie Weirich. 2000. Resource Bound Certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000)*. ACM, 184–198.

Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM 2003)*. USENIX Association, 3–3.

Ugo Dal Lago and Barbara Petit. 2013. The Geometry of Types. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013)*. ACM, 167–178.

Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*. ACM, 133–144.

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2015. Denotational Cost Semantics for Functional Languages with Inductive Types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, 140–151.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008/ETAPS 2008)*. Springer-Verlag, 337–340.

Dan R. Ghica and Alex Smith. 2011. Geometry of Synthesis III: Resource Management Through Type Inference. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*. ACM, 345–356.

Stéphane Gimenez and Georg Moser. 2016. The Complexity of Interaction. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM, 243–255.

Bernd Grobauer. 2001. Cost Recurrences for DML Programs. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP 2001)*. ACM, 253–264.

Bhargav S. Gulavani and Sumit Gulwani. 2008. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*. Springer-Verlag, 370–384.

Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*. ACM, 127–139.

Sumit Gulwani and Florian Zuleger. 2010. The Reachability-bound Problem. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010)*. ACM, 292–304.

Robert Harper. 2014. Structure and Efficiency of Computer Programs. (2014). http://www.cs.cmu.edu/~rwh/papers/secp/secp.pdf

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate Amortized Resource Analysis. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*. ACM, 357–370.

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 359–373.

Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential: A Static Inference of Polynomial Bounds for Functional Programs. In *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP 2010)*. Springer-Verlag, 287–306.

Jan Hoffmann and Zhong Shao. 2015. Automatic Static Cost Analysis for Parallel Programs. In *Proceedings of the 24th European Symposium on Programming Languages and Systems (ESOP 2015)*. Springer-Verlag New York, Inc., 132–157.

Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-order Functional Programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003)*. ACM, 185–197.

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1996)*. ACM, 410–423.

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-order Programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*. ACM, 223–236.

Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO 1996)*. Springer-Verlag, 104–113.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2010)*. Springer-Verlag, 348–370.

Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. 2017. Contract-based Resource Verification for Higher-order Functions with Memoization. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 330–343.

Ravichandhran Madhavan and Viktor Kuncak. 2014. Symbolic Resource Bound Inference for Functional Programs. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559 (CAV 2014)*. Springer-Verlag New York, Inc., 762–778.

Jay McCarthy, Burke Fetscher, Max New, Daniel Feltey, and Robert Bruce Findler. 2016. A Coq Library for Internal Verification of Running-Times. In *Functional and Logic Programming: 13th International Symposium (FLOPS 2016)*. Springer International Publishing, 144–162.

George C. Necula. 1997. Proof-carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997)*. ACM, 106–119.

Chris Okasaki. 1997. Three Algorithms on Braun Trees. *Journal of Functional Programming* 7, 6 (Nov. 1997), 661–666.

Chris Okasaki. 1999. *Purely Functional Data Structures*. Cambridge University Press.

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

Brian Reistad and David K. Gifford. 1994. Static Dependent Costs for Estimating Execution Time. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming (LFP 1994)*. ACM, 65–78.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*. ACM, 159–169.

Akhilesh Srikanth, Burak Sahin, and William R. Harris. 2017. Complexity Verification Using Guided Theorem Enumeration. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 639–652.

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-order Programs with the Dijkstra Monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. ACM, 387–398.

Robert Endre Tarjan. 1985. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods* 6, 2 (1985), 306–318.

Pedro B. Vasconcelos and Kevin Hammond. 2004. Inferring Cost Equations for Recursive, Polymorphic and Higher-order Functional Programs. In *Proceedings of the 15th International Conference on Implementation of Functional Languages (IFL 2003)*. Springer-Verlag, 86–101.

Niki Vazou, Alexander Bakst, and Ranjit Jhala. 2015. Bounded Refinement Types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, 48–61.

Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP 2013)*. Springer-Verlag, 209–228.

Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI 1998).* ACM, 249–257.

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1999).* ACM, 214–227.