Relational Cost Analysis for Functional-Imperative Programs

WEIHAO QU, University at Buffalo, SUNY, USA
MARCO GABOARDI, University at Buffalo, SUNY, USA
DEEPAK GARG, MPI-SWS, Germany

Relational cost analysis aims at formally establishing bounds on the difference in the evaluation costs of two programs. As a particular case, one can also use relational cost analysis to establish bounds on the difference in the evaluation cost of the same program on two different inputs. One way to perform relational cost analysis is to use a relational type-and-effect system that supports reasoning about relations between two executions of two programs.

Building on this basic idea, we present a type-and-effect system, called ARel, for reasoning about the relative cost of array-manipulating, higher-order functional-imperative programs. The key ingredient of our approach is a new lightweight type refinement discipline that we use to track relations (differences) between two mutable arrays. This discipline combined with Hoare-style triples built into the types allows us to express and establish precise relative costs of several interesting programs which imperatively update their data. We have implemented ARel using ideas from bidirectional type checking.

CCS Concepts: • **Theory of computation → Type structures**; **Program verification**.

Additional Key Words and Phrases: relational type systems, refinement types, type-and-effect systems

ACM Reference Format:
Weihao Qu, Marco Gaboardi, and Deepak Garg. 2019. Relational Cost Analysis for Functional-Imperative Programs. *Proc. ACM Program. Lang.* 3, ICFP, Article 92 (August 2019), 29 pages. https://doi.org/10.1145/3341696

1 INTRODUCTION

Standard cost analysis aims at statically establishing an upper or a lower bound on the evaluation cost of a program. The evaluation cost is usually measured in abstract units, e.g., the number of reduction steps in an operational semantics, the number of recursive calls made by the program, the maximum number of abstract units of memory used during the program's evaluation, etc. Cost analysis has been developed using a variety of techniques such as type systems [Avanzini and Dal Lago 2017; Dal Lago and Gaboardi 2011; Danielsson 2008; Grobauer 2001; Hoffmann et al. 2012b], term rewriting and abstract interpretation [Brockschmidt et al. 2014; Hermenegildo et al. 2005; Sinn et al. 2014], and Hoare logics [Atkey 2010; Carbonneaux et al. 2015; Charguéraud and Pottier 2015].

Relational cost analysis, the focus of this paper, is a more recently developed problem that aims at statically establishing an upper bound on the *difference* in the evaluation costs of two related programs or two runs of the same program with different inputs [Çiçek et al. 2017; Ngo et al. 2017; Radicek et al. 2018]. This difference is called the *relative cost* of the two programs or runs. Relational

Authors' addresses: Weihao Qu, University at Buffalo, SUNY, USA, weihaoqu@buffalo.edu; Marco Gaboardi, University at Buffalo, SUNY, USA, gaboardi@buffalo.edu; Deepak Garg, MPI-SWS, Germany, dg@mpi-sws.org.

2475-1421/2019/8-ART92
https://doi.org/10.1145/3341696

cost analysis has many applications: It can show that an optimized program is not slower than the original program on stipulated inputs; in cryptography, it can show that an algorithm's run time is independent of secret inputs and, hence, that there are no leaks on the timing side-channel; in algorithmic analysis, it can help understand the sensitivity of an algorithm's cost to input changes, which can be useful for resource allocation.

There are two reasons for examining relational cost analysis as a separate problem, as opposed to performing standard unary cost analysis separately on the two programs and taking a difference of the established costs. First, in many cases, relational cost analysis is easier than unary cost analysis, since it can focus only on the differences between the two programs. As a trivial example, if t is a complex closed program, it may be very difficult to perform unary cost analysis on it, but it is obvious that the cost of t relative to itself is 0. Second, in many cases, a direct relational cost analysis may be more precise than the difference of two unary analyses, since the relational analysis can exploit relations between intermediate values in the programs that the unary analyses cannot. As an example, the relative cost of two runs of merge sort on lists of length n that differ in at most k positions is in $O(n \cdot (1 + \log(k)))$. This relative cost can be established by a direct relational analysis [Çiçek et al. 2017], but two separate unary analyses can only establish the coarser relative cost $O(n \cdot \log(n))$.

Hitherto, literature on relational cost analysis has been limited to functional languages. However, many practical programs are stateful and use destructive updates, which are more difficult to reason about. Consequently, our goal in this work is to develop relational cost analysis for functional languages with mutable state (i.e., for functional-imperative programs).

To this end, we design a *refinement type-and-effect* system, ARel, for relational cost analysis in a functional, higher-order language with mutable state. The first question we must decide on is *what* kind of state to consider. One option could be to work with standard references as found in many functional languages like ML. However, from the perspective of cost analysis it is often more interesting to consider programs that operate on entire *data structures* (e.g., a sorting algorithm), not just on individual references. Consequently, we consider *mutable arrays*, the standard data structure available in almost all imperative languages. This makes our type system more complicated than it would be with standard references but allows us to verify more interesting examples.

Second, we must decide *how* to treat state in our functional language. Broadly, we have two choices: State could be a pervasive effect as in ML, or it could be confined to a monad as in Haskell, which limits the side-effect to only those sub-computations that actually access the heap. In ARel, we choose the latter option since this separates the pure and impure (state-affecting) parts of the language at the level of types and reduces the complexity of our typing rules.

The primary typing judgment of ARel, $\vdash t_1 \ominus t_2 \lesssim r : \tau$, states that the programs t_1 and t_2 are related at type τ, which can specify relational properties of their results and, importantly, that their relative cost (cost of t_1 minus the cost of t_2) is upper-bounded by r.[1] To reason about array-manipulating programs, we also need to express relations between corresponding arrays across the two runs. For this, our monadic type (the type of impure computations that can access state) has a *refinement* that specifies how arrays are related across the two runs *before* and *after* a heap-accessing computation. Specifically, our monadic type has the form $\{P\}\ \exists\vec{\gamma}.\tau\ \{Q\}^{\mathrm{diff}(r)}$. This type represents a pair of computations which, when starting from arrays related by the relational pre-condition P, end with arrays related by the relational post-condition Q, return values related at τ, newly generated arrays referred by static names $\vec{\gamma}$, and have relative cost at most r. This design is inspired

[1] This judgment is inspired by Çiçek et al. [2017] proposing a type-and-effect system for relational cost analysis of functional programs *without state*. Notice that one can use this typing judgment also to reason about *lower bounds* on the relative cost, by exchanging t_2 and t_1 and considering a negative cost $-r$.

by relational Hoare logics [Benton 2004; Nanevski et al. 2013], but there are two key differences: 1) Our pre- and post-conditions are *minimal*—they only specify the indices at which a pair of arrays differ across the two runs, not full functional properties. This suffices for relational cost analysis of many programs and simplifies our metatheory and, importantly, the implementation. 2) Our monadic types carry a relative cost, and the monad's constructs combine and propagate the different costs.

Additionally, ARel supports establishing lower and upper bounds on the cost of a *single* expression, and falling back to such unary analysis in the middle of a proof of relative cost. Improving over previous type-and-effect systems for relational cost analysis, ARel permits combinations of these two kinds of reasoning in the definition of recursive functions. ARel provides typing rules for the fix-point operator that allow one to *simultaneously* reason about unary and relational cost. This is useful for the analysis of several programs.

To prove that our type system is sound, we develop a logical relations model of our types. This model combines unary and binary logical relations and it supports two different effects, cost and state, that are structurally dissimilar. For the state aspect, we build on step-indexed Kripke logical relations [Ahmed et al. 2009; Ahmed 2004]. Specifically, our logical relations are indexed by a "step"—a standard device for inductive proofs that counts how many steps of computation the logical relation is good for [Ahmed 2006; Appel and McAllester 2001]. Owing to the simplicity of our pre- and post-conditions, we do not need state-dependent worlds as in some other work [Neis et al. 2011; Turon et al. 2013].

To show the effectiveness of our approach, we implemented a bidirectional type-checker for ARel. Thanks to the simplified form of our pre- and post-conditions, we can solve the constraints generated by the type-checker using SMT solvers. The type-checker also uses a restricted number of heuristics in order to address some of the non-determinism coming from the relational reasoning. We used our implementation to type-check a broad set of examples showing some of the challenges of relational cost analysis in programs manipulating arrays.

Our overarching contribution lies in extending relational cost analysis to higher-order functional-imperative programs. Our specific contributions are:

- ARel, a type system for relational cost analysis of functional-imperative programs with mutable arrays.
- A design for lightweight (relational) refinements of array-based computations.
- A soundness proof for our type system via a new step-indexed logical relation.
- An implementation of ARel, based on bidirectional type checking, which we use to type check several functional-imperative examples.

2 AREL THROUGH EXAMPLES

In this section, we illustrate the key ideas behind ARel through two simple examples.

Inplace Map. Consider the following imperative map function taking as input a pure function f, a mutable array a, an index k and the array's length n. For all $i \in [k, n]$, the function replaces the current value in the ith cell of a with $f(a[i])$, thus performing a destructive update.

$$\begin{aligned}\mathsf{fix}\ \mathsf{map}\,(f).\lambda a.\lambda k.\lambda n.\ \mathsf{if}\ k \le n\ \mathsf{then}\ \big(&\mathsf{let}\ \{x\} = \mathsf{read}\ a\ k\ \ \mathsf{in}\\ &\mathsf{let}\{\_\} = \mathsf{updt}\ a\ k\ (f\ x)\ \ \mathsf{in}\ \mathsf{map}\ f\ a\,(k+1)\,n\big)\\ \mathsf{else}\ \mathsf{return}()\end{aligned}$$

The expression (read $a\ k$) returns the element at index k in the array a, and (updt $a\ k\ v$) updates the index k in a to v. Our language uses a state monad to isolate all side-effects like array reads

and updates, so (read $a\ k$) and (updt $a\ k\ v$) are actually expressions of monadic types, also called *computations*. The construct (let $\{x\} = t_1$ in t_2) is monadic sequencing, often called "bind".

Consider the problem of establishing an upper-bound on the *relative* cost of two runs of map that use the *same* function f but two *different* arrays a. Intuitively, the relative cost should be upper-bounded by the product of the maximum variation in the cost of the function f (across inputs) and the number of indices in the range $[k, n]$ at which the two arrays differ.

To support reasoning about two runs as in this example, ARel supports *relational types* that ascribe a pair of related values or related expressions in the two runs. Relational types are written τ. In general, when we say $x : \tau$, we mean that the variable x may be bound to two different values in the two runs, but these two values will be related by the type τ. Specifically, $x : \tau_1 \to \tau_2$ means that x can be bound to two different functions f_1, f_2 in the two runs, satisfying the property that for any two v_1, v_2 of relational type τ_1, the two expressions $f_1\ v_1, f_2\ v_2$ have relational type τ_2. ARel also supports *unary types*, denoted A, that ascribe a value or expression in a single run, but we will have no occasion to use unary types in this example, so we postpone their discussion.

To establish the relative cost of map, we first need a way to represent that the *same* function f will be given to map in both runs. For this, ARel offers the type annotation $\Box$. The type $\Box\tau$ relates expressions in two runs that are (syntactically) equal and are additionally related at the relational type τ. Note that $\Box$ is a *relational refinement*: It refines the relation defined by the underlying type τ. Specifically, the relational typing assumption $f : \Box(\tau_1 \to \tau_2)$ means that, in the two runs, f will be bound to two copies of the *same* function, say f, that given arguments v_1, v_2 related at type τ_1, give expressions $f\ v_1$ and $f\ v_2$ related at type τ_2. In our example, if the array's elements have type τ, the type of f would be $\Box(\tau \to \tau)$.

Next, we need to represent the maximum possible variation in the cost of applying f. The possible variation in the cost can be seen as an *effect*, and the cost of applying a function can be seen as the effect associated with the body of the function, in particular. Hence as is common in effect systems [Nielson and Nielson 1999], we can record the possible variation in cost by means of a refinement of the function type. ARel offers a refinement of this kind. We write $\tau_1 \overset{\mathrm{diff}(r)}{\longrightarrow} \tau_2$ to represent two functions of relational type $\tau_1 \to \tau_2$, the relative cost of whose bodies is upper-bounded by r. Accordingly, if f's cost can vary by r, its type can be further refined to $\Box(\tau \overset{\mathrm{diff}(r)}{\longrightarrow} \tau)$.

Next, we need a way to specify *where* the arrays given as inputs to map in the two runs differ. There are various design choices for supporting this. One obvious but problematic option would be to refine the type of an array itself, to specify where the two ascribed arrays differ across two runs. However, this design quickly runs into an issue: An update on the arrays might be different in the two runs, so it might change the arrays' *type*. This would be highly unsatisfactory since we don't expect the type of an array to change due to an update; in particular, this design would not satisfy (semantic or syntactic) type preservation.

Consequently, we use a different approach inspired by relational Hoare logics: We provide a relational refinement type $\{P\}\ \exists\vec{\gamma}.\tau\ \{Q\}^{\mathrm{diff}(r)}$ for monadic expressions that manipulate arrays. The number r is an upper-bound on the relative cost of the computation, similar to the one we have in function types, and τ is the relational type of the pure values the computation returns. The *pre-condition* P specifies for each pair of related arrays in scope where (at which indices) the arrays are allowed to differ *before* the computation runs, while the *post-condition* Q specifies where the arrays may differ *after* the computation completes. More specifically, P and Q are lists of annotations of the form $\gamma \to \beta$, where γ is a *static name* for an array and β is a set of indices where the array identified by γ *may* differ in the two runs. At any index not in β, the array must be the same in the two runs. Note that even at indices in β, the corresponding values must be related at τ, but our type

system includes types that do not force equality of the related values. One such type is $U(A, B)$ that only insists that the left and right values have (unary) types A and B, without requiring any other relation between them. (The existentially quantified $\vec{\gamma}$ in $\{P\}\ \exists\vec{\gamma}.\tau\ \{Q\}^{\mathrm{diff}(r)}$ is the list of static names of arrays that are allocated during the computation.)

For example, if $x : \Box\tau$, i.e., x is the same in two runs, and b is an array of static name γ, then (updt b 5 x) can be given the type $\{\gamma \to \beta\}\ \exists\_.\mathrm{unit}\ \{\gamma \to (\beta \setminus \{5\})\}^{\mathrm{diff}(0)}$ relative to itself for any β.[2] This type means that if the array b differs at the set of indices β before (updt b 5 x) executes in two runs, then afterwards it can still differ in the indices β *except* at the index 5, which has been overwritten by the same value x. If we replace the assumption $x : \Box\tau$ with $x : \tau$, so that x may differ in the two runs, then the type of (updt b 5 x) relative to itself would be $\{\gamma \to \beta\}\ \exists\_.\mathrm{unit}\ \{\gamma \to (\beta \cup \{5\})\}^{\mathrm{diff}(0)}$, indicating that the arrays may differ at index 5 after the update (even if they did not differ at that index before the update).

We also need a way to tie static names γ appearing in computation types to specific arrays. For this, we refine the type of arrays to include γ. In fact, we also refine the type of arrays to track the length of the array. This doubly refined type is written $\mathrm{Array}_\gamma[l]\ \tau$—a pair of arrays of length l each, identified statically by the name γ, and carrying elements related pointwise at type τ. Finally, we refine integers very precisely: The type int$[n]$ is the *singleton type* containing only the integer n in both runs. The n in the type is a static representation of the runtime values the type ascribes.

With all these components we can now represent the relative cost of map that we are interested in by the judgment:

$$\vdash \mathsf{map} \ominus \mathsf{map} \lesssim 0 : \begin{array}{l} \forall r : .\Box(\tau \overset{\mathrm{diff}(r)}{\longrightarrow} \tau) \to \forall k, n, \gamma, \beta.(k \le n) \supset \\ \mathrm{Array}_\gamma[n]\ \tau \to \mathrm{int}[k] \to \mathrm{int}[n] \to \{\gamma \to \beta\}\ \exists\_.\mathrm{unit}\ \{\gamma \to \beta\}^{\mathrm{diff}(|\beta \cap [k,n]| * r)} \end{array}$$

This typing means that map relates to itself in the following way. Consider two runs of map with the same function f of relative cost r (type $\Box(\tau \overset{\mathrm{diff}(r)}{\longrightarrow} \tau)$), two arrays of static length n, statically named γ (type $\mathrm{Array}_\gamma[n]\ \tau$), two indices, both k (type int$[k]$), and two lengths, both n (type int$[n]$). Then, the two runs return computations with the following relational property: If the two arrays differ at most at indices β before they are passed to map, then they differ at most at the same positions after the computations and the relative cost of the two computations is upper-bounded by $|\beta \cap [k, n]| * r$, i.e., the number of positions in the range $[k, n]$ at which the arrays may differ times r. This is exactly the expected relative cost because at positions where the arrays are equal, f will have the same cost in the two runs (we are assuming language-level determinism here). Note that the variables r, k, n, γ and β are universally quantified in the type above. Also note how γ links the input array to the β in the pre- and post-condition of the computation type.

Consider now a slightly different situation where *different* functions f may be passed to map in the two runs. Suppose that the relative cost of the bodies of the two fs passed is upper-bounded by r, i.e., f has the type $\tau \overset{\mathrm{diff}(r)}{\longrightarrow} \tau$ (without the prefix $\Box$). In this case, the relative cost of the two runs of map can only be upper-bounded by $|[k, n]| * r$, since even at indices where the arrays agree, the cost of applying the two different fs may differ by as much as r. Moreover, the final arrays may differ in all positions in the range $[k, n]$. This is formalized in the following, second relational type

[2] As usual, _ represents a variable whose name is unimportant.

for map.

$$\vdash \mathsf{map} \ominus \mathsf{map} \lesssim 0 : \begin{array}{l} \forall r.(\tau \xrightarrow{\mathrm{diff}\,(r)} \tau) \to \forall k, n, \gamma, \beta.(k \le n) \supset \\ \mathsf{Array}_\gamma[n]\ \tau \to \mathsf{int}[k] \to \mathsf{int}[n] \to \{\gamma \to \beta\}\ \exists\_.\mathsf{unit}\ ^{\mathrm{diff}((n-k)*r)}\{\gamma \to \beta \cup [k, n]\} \end{array}$$

Boolean Or. Next, we describe how high-level reasoning about relative cost is internalized in the typing. ARel supports two kinds of typing modes: *relational typing* as shown in the map example above, and *unary typing* which supports traditional (unary) min- and max-cost analysis for a single run of a program. We will introduce these modes formally in the next section but here we want to show with the following example how they can be meaningfully combined.

$$\begin{array}{l} \mathsf{fix}\,\mathsf{BoolOr}\,(a).\,\lambda k.\lambda n.\mathsf{if}\ k < n\ \mathsf{then}\ \big(\mathsf{let}\{x\} = \mathsf{read}\ a\ k\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ \mathsf{return}\ \mathit{true}\ \mathsf{else}\ \mathsf{BoolOr}\ a\ (k+1) n\big) \\ \qquad\qquad \mathsf{else}\ \mathsf{return}\ \mathit{false} \end{array}$$

This function, given as input an array of booleans a, an index k and the array's length n tells whether there exists an element in a with index $\ge k$ and value *true*.

Given two arbitrary arrays a in two runs, a simple upper-bound on the relative cost of BoolOr is $(n - k) * c$ where c is the cost of one iteration. This is because in one run we can find an element with value *true* in position k, and so the computation can return immediately, while in the other run we may not find any such element, and would need to visit every element of the array with its index greater than k. This kind of high-level reasoning corresponds to a worst-case, best-case analysis of the two individual runs. ARel supports this kind of reasoning by supporting worst-case, best-case (unary) cost analysis in unary mode, and by means of a rule R-S, presented formally in Section 3, allows us to derive a relational typing from two unary typings, with relative cost equal to the difference between the max and the min costs of the unary typings.

However, this kind of reasoning does not account for the case where the two input arrays have a meaningful relation, e.g., they may be equal in some positions. In such cases, a better upper bound on the relative cost would be expressed in term of the first index i (if any) where the two arrays differ. That is, we could have the upper bound $(n - i) * c$. Showing this upper bound in a formal way is more involved. We first need to proceed by case analysis on whether the element x we are reading at each step is the same in the two runs or not. Case analysis in ARel is provided by the rule R-P, presented in Section 3. Using this rule we can consider the two cases separately in typing the subexpression $\mathsf{if}\ x\ \mathsf{then}\ (\mathsf{return}\ \mathit{true})\ \mathsf{else}\ \mathsf{BoolOr}\ a\ (k+1)\ n$.

If x is the same in the two runs, there is no difference in cost because we either return *true* in both runs or we perform the recursive call in both runs. In case the two x's differ, we must switch to unary analysis of the two individual runs, since in one run we will return immediately while in the other we will make a recursive call, so there is no way to continue reasoning relationally. Hence, in order to derive the required upper bound on the overall relative cost we need to have information about the *unary* type of BoolOr. However, since we started by trying to type the body of BoolOr relationally, the standard fixpoint rule only allows us to assume its *relational* type.

One solution to this impasse is to automatically transform relational types of variables in context to unary types when switching from relational to unary reasoning. This approach was adopted by Çiçek et al. [2017] for analyzing pure functional programs but it provides only trivial lower and upper bounds (0 and ∞) on the costs of function variables in the context during the unary analysis. In our example here, this approach yields the trivial upper bound ∞, which is not what we want.

To allow for a more precise analysis, ARel includes a new rule R-FIX-EXT which we introduce formally in Section 3. This rule allows us to assume the result of a *unary* typing of two recursive functions, when typing their bodies *relationally*. With this rule, we can use the (assumed) relational

Index terms $I, L, U, D, \alpha, \beta ::= i \mid b \mid n \mid r \mid I_1 + I_2 \mid I_1 * I_2 \mid I_1 - I_2 \mid max(I_1, I_2) \mid min(I_1, I_2)$
$\mid log_2(I) \mid \lfloor I \rfloor \mid \lceil I \rceil \mid \{I_i\}_{i \in K} \mid \beta \cup \beta \mid \beta \setminus \beta \mid \beta \cap \beta$

Terms $t ::= x \mid n \mid r \mid () \mid \lambda x.t \mid \mathsf{fix}\, f(x).t \mid t_1\ t_2 \mid \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \mid \mathsf{inl}\ t \mid \mathsf{inr}\ t$
$\mid \mathsf{case}\ (t, x.t_1, y.t_2) \mid \Lambda.t \mid t\,[] \mid \mathsf{pack}\ t \mid \mathsf{unpack}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 \mid \mathsf{celim}\ t$
$\mid \mathsf{return}\ t \mid \mathsf{let}\{x\} = t_1\ \mathsf{in}\ t_2 \mid \mathsf{alloc}\ t_1\ t_2 \mid \mathsf{read}\ t_1\ t_2 \mid \mathsf{updt}\ t_1\ t_2\ t_3$

Values $v ::= n \mid l \mid r \mid () \mid \lambda x.t \mid \mathsf{fix}\, f(x).t \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \Lambda.t \mid \mathsf{pack}\ v$
$\mid \mathsf{return}\ t \mid \mathsf{alloc}\ t_1\ t_2 \mid \mathsf{updt}\ t_1\ t_2\ t_3 \mid \mathsf{read}\ t_1\ t_2 \mid \mathsf{let}\{x\} = t_1\ \mathsf{in}\ t_2$

Unary types $A ::= c \mid \mathsf{int}[I] \mid \{P\}\ \overset{\mathsf{exec}(L,U)}{\exists \vec{\gamma}.A}\ \{Q\} \mid \overset{\mathsf{exec}(L,U)}{\forall i{::}S.\ A} \mid \exists i :: S.\ A \mid \overset{\mathsf{exec}(L,U)}{A \longrightarrow A} \mid \mathsf{Array}_{\gamma}[I]\ A$
$\mid \mathsf{list}[I]\ A \mid A_1 + A_2 \mid C \& A \mid C \supset A$

Relat. types $\tau ::= c \mid \mathsf{int}[I] \mid \{P\}\ \overset{\mathsf{diff}(D)}{\exists \vec{\gamma}.\tau}\ \{Q\} \mid \overset{\mathsf{diff}(D)}{\forall i{::}S.\ \tau} \mid \exists i{::}S.\ \tau \mid \overset{\mathsf{diff}(D)}{\tau \longrightarrow \tau} \mid \mathsf{Array}_{\gamma}[I]\ \tau$
$\mid \mathsf{list}^{\alpha}[I]\ \tau \mid \tau_1 + \tau_2 \mid C \& \tau \mid C \supset \tau \mid U(A_1, A_2) \mid \Box \tau$

Unary Type Env.	$\Omega ::= \emptyset \mid \Omega, x : A$
Relational Type Env.	$\Gamma ::= \emptyset \mid \Gamma, x : \tau$
Sort Env.	$\Delta ::= \emptyset \mid \Delta, i :: S$
Loc Env.	$\Sigma ::= \emptyset \mid \Sigma, \gamma :: \mathbb{L}$
Sorts	$S ::= \mathbb{R} \mid \mathbb{N} \mid \mathbb{B} \mid \mathbb{P} \mid \mathbb{L}$
Constraints	$C ::= I_1 = I_2 \mid I_1 < I_2 \mid \neg C \mid I_1 \in I_2$
Constraint Env.	$\Phi ::= \top \mid C \wedge \Phi$
Assertions	$P, Q ::= \mathsf{empty} \mid \gamma \rightarrow \beta \mid P \star Q$
Heaps	$H ::= [] \mid [l \rightarrow z] \mid H_1 \uplus H_2$
Arrays	$z ::= [v_1, \ldots, v_m]$

Judgments

$\Sigma; \Delta; \Phi; \Omega \vdash_L^U t : A$
$\Sigma; \Delta; \Phi; \Gamma \vdash t_1 \ominus t_2 \lesssim D : \tau$
$\Sigma; \Delta \vdash I :: S$
$\Sigma; \Delta; \Phi \vdash A\ \mathsf{wf}$
$\Sigma; \Delta; \Phi \vdash \tau\ \mathsf{wf}$
$\Delta \vdash C\ \mathsf{wf}$
$\Omega \vdash H\ \mathsf{wf}$
$\Sigma; \Delta \vdash P\ \mathsf{wf}$
$\Sigma; \Delta; \Phi \models A_1 \sqsubseteq A_2$
$\Sigma; \Delta; \Phi \models \tau_1 \sqsubseteq \tau_2$

Fig. 1. Syntax of ARel where $n \in \mathbb{N}$, $r \in \mathbb{R}$, $x \in Var$, $i \in iVar$, $\gamma \in iLoc$, $l \in Loc$.

type of BoolOr and its unary type in typing the subexpression BoolOr $a\,(k+1)\,n$. Hence, we can conclude the inductive step and assign the precise relative cost $(n - i) * c$ to BoolOr.

3 AREL FORMALLY

3.1 Syntax

We summarize ARel's syntax in Figure 1. The term language underlying ARel is a simply typed λ-calculus with recursion and constructs for mutable arrays. Most of these constructs are inherited from RelCost [Çiçek et al. 2017], a type system for relative cost analysis in *pure* functional programs. Following that work, ARel also has type refinements in the style of DML [Xi and Pfenning 1999]. The term constructs $\Lambda.t$ and $t\,[]$, pack t and unpack t_1 as x in t_2 correspond to the introduction and elimination of universal and existential types. The construct celim t eliminates the constraint implication $C \supset \tau$.

New here are the constructs to deal with arrays: for *allocating* arrays (alloc t_1 t_2, where t_1 specifies the number of array cells to be allocated, and t_2 the initial value to be stored in each array cell), for *reading* from arrays (read t_1 t_2, where t_1 specifies the array to read from, and t_2 the position in the array to read from), and for *updating* arrays (updt t_1 t_2 t_3, where t_1 specifies the array to be updated, t_2 the position in the array to be updated, and t_3 the value to be used for the update). All

$$\frac{}{v \Downarrow^{0,0} v}\ \text{E-V} \qquad \frac{t_1 \Downarrow^{c_1,k_1} \lambda x.t' \quad t_2 \Downarrow^{c_2,k_2} v \quad t'[v/x] \Downarrow^{c_3,k_3} v_1}{t_1\ t_2 \Downarrow^{c_1+c_2+c_3+c_{\text{app}},\, k_1+k_2+k_3+1} v_1}\ \text{E-A}$$

$$\frac{t_1 \Downarrow^{c_1,k_1} \text{fix}\, f\, x.t' \quad t_2 \Downarrow^{c_2,k_2} v \quad t'[\text{fix}\, f\, x.t'/f][v/x] \Downarrow^{c_3,k_3} v_1}{t_1\ t_2 \Downarrow^{c_1+c_2+c_3+c_{\text{fapp}},\, k_1+k_2+k_3+1} v_1}\ \text{E-F}$$

$$\frac{t_1 \Downarrow^{c_1,k_1} v \quad v; H \Downarrow_f^{c_2,k_2} v_1; H_1 \quad t_2[v_1/x] \Downarrow^{c_3,k_3} v_2 \quad v_2; H_1 \Downarrow_f^{c_4,k_4} v_3; H_2}{\text{let}\{x\} = t_1 \text{ in } t_2; H \Downarrow_f^{c_1+c_2+c_3+c_4+c_{\text{let}},\, k_1+k_2+k_3+k_4+1} v_3; H_2}\ \text{F-E}$$

$$\frac{t_1 \Downarrow^{c_1,k_1} l \quad t_2 \Downarrow^{c_2,k_2} n \quad t_3 \Downarrow^{c_3,k_3} v}{\text{updt}\ t_1\ t_2\ t_3; H \Downarrow_f^{c_1+c_2+c_3+c_{\text{update}},\, k_1+k_2+k_3+1} (); H(l)[n] \leftarrow v}\ \text{F-U}$$

$$\frac{t \Downarrow^{c,k} v}{\text{return}\, t; H \Downarrow_f^{c+c_{\text{ret}},\, k+1} v; H}\ \text{F-T} \qquad \frac{t_1 \Downarrow^{c_1,k_1} l \quad t_2 \Downarrow^{c_2,k_2} n_2 \quad H(l)[n] = v}{\text{read}\ t_1\ t_2; H \Downarrow_f^{c_1+c_2+c_{\text{read}},\, k_1+k_2+1} v; H}\ \text{F-R}$$

$$\frac{t_1 \Downarrow^{c_1,k_1} n \quad t_2 \Downarrow^{c_2,k_2} v \quad z = [\overbrace{v, \ldots, v}^{n}] \quad l \text{ fresh}}{\text{alloc}\ t_1\ t_2; H \Downarrow_f^{c_1+c_2+c_{\text{alloc}},\, k_1+k_2+1} l; H \uplus [l \to z]}\ \text{F-L}$$

Fig. 2. Selection of rules for pure evaluation $t \Downarrow^{c,k} v$, and forcing evaluation $t; H \Downarrow_f^{c,k} v; H'$.

imperative (array-manipulating) constructs are confined to a monad. The constructs return t and $\text{let}\{x\} = t_1$ in t_2 are the usual return and bind of the monad. Impure expressions are distinguished by monadic types, but not syntactically distinguished in the syntax of expressions. Impure expressions (expressions of monadic types) are values, but can be *forced* using a special forcing semantics that we describe below. Finally, arrays are referenced through locations, $l \in Loc$. Although locations do not appear in programs, they do show up during evaluation, so they are included in the syntax.

3.2 Operational Semantics

We define a cost-annotated, big-step operational semantics for our language. Part of this semantics is based on heap manipulation. We represent heaps as mappings $H = [l_1 \to z_1, \ldots, l_n \to z_n]$ from memory locations to concrete arrays $z = [v_1, \ldots, v_n]$. The notation $H(l)[n] = v$ expresses that the value v is stored in the heap H in the array pointed by the location pointer l at the index n, the notation $H(l)[n] \leftarrow v$ represents the heap H where the array pointed by l is updated with the value v at index n, and the notation $H_1 \uplus H_2$, in the spirit of separation logic, denotes a disjoint union of the heaps H_1 and H_2. We give a selection of the evaluation rules in Figure 2. We have two kinds of evaluation judgments: *pure evaluation* $t \Downarrow^{c,k} v$ states that the (pure) expression t evaluates to the value v with cost c, using k steps, while *forcing evaluation* $t; H \Downarrow_f^{c,k} v; H'$ states that the impure expression t can be forced in the heap H to the value v and to the updated heap H' with cost c, consuming k steps.

Steps k are a proof artifact, needed only in our soundness proof that relies on a step-indexed logical relation (Section 4). We count a unit step for every elimination and monadic construct. Readers may ignore steps for now. The costs c are what we seek to upper bound (relatively) using our type system and are, therefore, important. At every elimination form or monadic construct, the semantics add a construct-dependent cost. For example, the cost c_{app} appearing in the rules is the cost of an application. By changing these costs and setting some of them to 0, we can get different cost models. Our type system is parametric in the costs of individual constructs.

Most of the pure evaluation rules are standard. The forcing evaluation rules are used to evaluate impure (monadic) expressions manipulating heaps (arrays). The rule F-T forces the evaluation of an expression return t by evaluating the underlying pure expression t using the pure evaluation semantics. The cost consists of the cost of the pure evaluation of t and the constant cost c_{ret} for the monadic return. The rule F-E combines pure and forcing evaluations in order to evaluate a bind fully. An additional cost c_{let} is added. The rule F-R forces the evaluation of a read expression in the heap H by first extracting the heap location l from which to read, the index of the element n to read, and then returning the value stored in l at index n. The rule F-U forces the evaluation of an update expression in a similar way; it returns a unit value. Finally, the rule F-L forces the evaluation of an alloc expression by creating a new array with the length specified by the first argument and initial values specified by the second argument, and by allocating it in the heap at a new location l, which is returned.

3.3 Index Terms and Constraints

In the spirit of DML [Xi and Pfenning 1999], types are indexed using *static* index terms that are defined in Figure 1. Index terms include booleans, natural numbers and real numbers. A subclass of index terms specific to ARel and that allows us to reason about arrays is one representing (potentially infinite) sets of natural numbers. We denote this class β. These sets can be used to represent at the type level different information on arrays. In relational types, they represent where two related arrays may differ (as explained earlier), while in unary types, they represent the write permissions for the array. We will return to this point later, after we explain the types. We can explicitly form a set through an indexed set comprehension of the form $\{I_i\}_{i \in K}$, where $K \subseteq \mathbb{N}$, and we can take the union $\beta_1 \cup \beta_2$ or the difference $\beta_1 \setminus \beta_2$ of two sets β_1, β_2. We consider only *well-sorted* index terms. To this end, we have a sorting judgment of the form $\Delta \vdash I :: S$ where Δ is a *sort environment*, assigning sorts to index variables, and S is a sort. Our language has five sorts: $\mathbb{N}$ of natural numbers, used for sizes of arrays; $\mathbb{R}$ of real numbers, used to express costs; $\mathbb{B}$ of booleans; $\mathbb{P}$ of sets β just described; and, $\mathbb{L}$ of static names γ of arrays. We omit the sorting judgment because it is straightforward. As a convention, we use L, U to represent unary minimum and maximum costs, and D to denote a maximum relational cost (L, U and D are always of sort $\mathbb{R}$). Index terms can also appear in constraints C. Figure 1 shows some constraints built out of equalities and inequalities over index terms.

3.4 Unary and Relational Types

In ARel we have two typing modes: *unary* and *relational*. This separation is also reflected at the type level where we have two different type languages: *unary types* A and *relational types* τ.

Unary types ascribe expressions in a single run. They use index terms to represent size information, as in the case of the type $\text{list}[I]\ A$ where I represents the size of the list, and costs, as in the case of the type $A \xrightarrow{\text{exec}(L,U)} A'$ where L and U represent lower- and upper-bounds on the cost of the body of the function being typed. Cost can also be seen as a type system effect. Index terms are also used for size in basic types like integers, booleans, etc. and for cost in universal quantification. Additionally, unary types can contain constraints in types $C \& A$ and $C \supset A$, that can be used to implement conditional typing.

We also have a type for arrays and a type for impure computations. The type $\text{Array}_\gamma[I]\ A$ is the type of arrays of length I containing objects of type A. The annotation γ associates a *static name* to the array that is typed. This static name can be used to refer to the array in other types. Impure expressions are typed with monadic types. In our case, a monadic unary type is a cost-annotated

Hoare triple type of the shape $\{P\}\ \exists\vec{\gamma}.A\ \{Q\}^{\mathrm{exec}(L,U)}$, which is inspired by Hoare Type Theory [Nanevski et al. 2008]. Assertions P, Q are sets $\{\gamma_1 \to \beta_1, \ldots, \gamma_n \to \beta_n\}$ assigning to each static location γ_i a set of natural numbers β_i called the (write) permissions. The idea is that the array named γ_i can be written only at indices in β_i (although it may read anywhere). The index terms L and U are lower- and upper-bounds on the execution cost of the (forcing) evaluation of the typed expression.

Relational types ascribe pairs of expressions, one from each of the two runs and, as we will see in Section 4, they are actually interpreted as sets of pairs of expressions in our model. In relational types, index terms carry not just size information but also information about the *relation* between the two values from the two runs. The type $\mathsf{list}^{\alpha}[I]\ \tau$ ascribes a pair of lists, each of length I, whose elements are pointwise related at type τ. Importantly, the relational refinement α specifies an upper bound on the number of positions at which the corresponding elements may differ. In other words, at at least $I - \alpha$ positions, the two lists must have equal elements, even if τ allows them to be completely unrelated. The type $\mathsf{int}[I]$ represents pairs of integers both of which are equal to I. In arrow types $\tau \xrightarrow{\mathrm{diff}(D)} \tau'$, the index term D represents an upper bound on the relative cost of the underlying pair of functions.

Given a pair of unary types A_1, A_2, the relational type $U(A_1, A_2)$ represents arbitrary pairs of expressions of types A_1, A_2, respectively. This offers a principle of relationally typing two "unrelated" values. As explained in Section 2, we also have a comonadic relational type $\Box\tau$ which represents pairs of expressions of type τ which are syntactically equal. In particular, $\Box U(A_1, A_2)$ is the diagonal relation on $A_1 \cap A_2$.

The relational type $\mathsf{Array}_{\gamma}[I]\ \tau$ is similar to the unary array type but it represents two arrays, each of length I, containing values related at τ pointwise. γ is the static name for both arrays. As we will see in Section 4, our logical relation relates γ to two arrays in two different heaps. Relational impure computations, illustrated in the map example of Section 2, are typed using a relational cost-annotated monadic type of the form $\{P\}\ \exists\vec{\gamma}.\tau\ \{Q\}^{\mathrm{diff}(D)}$. This looks similar to the unary type $\{P\}\ \exists\vec{\gamma}.A\ \{Q\}^{\mathrm{exec}(L,U)}$. However, the type means something very different. In the relational type, the pre- and post-conditions P, Q of form $\{\gamma_1 \to \beta_1, \ldots, \gamma_n \to \beta_n\}$ have a relational interpretation, namely, that (for all i) the two arrays named γ_i must carry equal values at all positions not in β (and the values must be related at τ). At positions in β, the values must still be related at τ, but they need not be equal (unless τ forces this). D is an upper bound on the relative cost of forcing the two impure expressions.

As usual, we consider only types that are well-formed. We have well-formedness judgments $\Sigma; \Delta; \Phi \vdash A\ \mathsf{wf}$ for unary types, and $\Sigma; \Delta; \Phi \vdash \tau\ \mathsf{wf}$ for relational types. Here, Σ is a *location environment* listing the locations that can appear in the rest of the judgment, Δ is a sort environment listing all free index variables, and Φ is a *constraint environment* to support conditional typing.

3.5 Unary and Relational Typing

Unary Typing Judgments. ARel's unary typing uses the judgment form

$$\Sigma; \Delta; \Phi; \Omega \vdash_L^U t : A$$

where t is an expression, Σ is a location environment, Δ is a sort environment, Φ is a constraint environment, Ω is a *unary type environment* assigning unary types to variables, A is a unary type, and L and U are index terms representing a lower bound and an upper bound on the cost of evaluating t, respectively. We give a selection of the typing rules for deriving unary typing judgments in Figure 3. Rules U-I and U-V are similar to the ones available in indexed type systems,

$$\frac{}{\Sigma;\Delta;\Phi;\Omega \vdash^{0}_{0} n : \mathrm{int}[n]}\ \text{U-I} \qquad \frac{\Sigma;\Delta;\Phi;x:A, f :^{\mathrm{exec}(L,U)} A \longrightarrow B, \Omega \vdash^{U}_{L} e : B}{\Sigma;\Delta;\Phi;\Omega \vdash^{0}_{0} \mathsf{Fix}\, f(x).e :^{\mathrm{exec}(L,U)} A \longrightarrow B}\ \text{U-F}$$

$$\frac{\Omega(x) = A}{\Sigma;\Delta;\Phi;\Omega \vdash^{0}_{0} x : A}\ \text{U-V} \qquad \frac{\Sigma;\Delta;\Phi;\Omega \vdash^{U}_{L} t : A \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}}{\Sigma;\Delta;\Phi;\Omega \vdash^{0}_{0} \mathsf{return}\, t :^{\mathrm{exec}(L,U)} \{P\}\ \exists \gamma.A\ \{P\}}\ \text{U-T}$$

$$P = P_1 \star P_2$$

$$\frac{\Sigma;\Delta;\Phi;\Omega \vdash^{U_1}_{L_1} t_1 :^{\mathrm{exec}(L,U)} \{P_1\}\ \exists \vec{\gamma_1}.A\ \{Q_1 \star Q_2\} \qquad \Sigma;\Delta,\vec{\gamma_1};\Phi;\Omega, x:A \vdash^{U_2}_{L_2} t_2 :^{\mathrm{exec}(L',U')} \{Q_1 \star P_2\}\ \exists \vec{\gamma_2}.B\ \{Q\}}{\Sigma;\Delta;\Phi;\Omega \vdash^{U_1+U_2+U_l}_{L_1+L_2+L_l} \mathsf{let}\{x\} = t_1\ \mathsf{in}\ t_2 :^{\mathrm{exec}(L+L',U+U')} \{P\}\ \exists \vec{\gamma_1},\vec{\gamma_2}.B\ \{Q \star Q_2\}}\ \text{U-E}$$

$$\frac{\Sigma;\Delta;\Phi;\Omega \vdash^{U_1}_{L_1} t_1 :^{\mathrm{exec}(L,U)} A \longrightarrow B \qquad \Sigma;\Delta;\Phi;\Omega \vdash^{U_2}_{L_2} t_2 : A}{\Sigma;\Delta;\Phi;\Omega \vdash^{U_1+U_2+U+U_{app}}_{L_1+L_2+L+L_{app}} t_1\ t_2 : B}\ \text{U-A}$$

$$\frac{\Sigma;\Delta;\Phi;\Omega \vdash^{U_1}_{L_1} t_1 : \mathrm{int}[I] \qquad \Sigma;\Delta;\Phi;\Omega \vdash^{U_2}_{L_2} t_2 : A \qquad \gamma\ \text{fresh} \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}}{\Sigma;\Delta;\Phi;\Omega \vdash^{0}_{0} \mathsf{alloc}\, t_1\ t_2 :^{\mathrm{exec}(L_1+L_2+L_a,U_1+U_2+U_a)} \{P\}\ \exists \gamma.\ \mathsf{Array}_{\gamma}[I]\ A\ \ \{P \star \gamma \rightarrow \mathbb{N}\}}\ \text{U-L}$$

$$\frac{\Sigma;\Delta;\Phi;\Omega \vdash^{U_1}_{L_1} t_1 :\mathsf{Array}_{\gamma}[I]\ A \qquad \Sigma;\Delta;\Phi;\Omega \vdash^{U_2}_{L_2} t_2 : \mathrm{int}[I'] \qquad \Delta;\Phi \vDash I' \leq I \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}}{\Sigma;\Delta;\Phi;\Omega \vdash^{0}_{0} \mathsf{read}\, t_1\ t_2 :^{\mathrm{exec}(L_1+L_2+L_r,U_1+U_2+U_r)} \{P\}\ \exists \_.A\ \{P\}}\ \text{U-R}$$

$$\frac{\begin{array}{c}\Sigma;\Delta;\Phi;\Omega \vdash^{U_1}_{L_1} t_1 :\mathsf{Array}_{\gamma}[I]\ A \qquad \Sigma;\Delta;\Phi;\Omega \vdash^{U_2}_{L_2} t_2 : \mathrm{int}[I'] \\ \Sigma;\Delta;\Phi;\Omega \vdash^{U_3}_{L_3} t_3 : A \qquad \Delta;\Phi \vDash I' \leq I \qquad \Sigma;\Delta \vdash P\ \mathsf{wf} \qquad \Delta;\Phi \vDash I' \in \beta\end{array}}{\Sigma;\Delta;\Phi;\Omega \vdash^{0}_{0} \mathsf{updt}\, t_1\ t_2\ t_3 :^{\mathrm{exec}(L_1+L_2+L_3+L_u,U_1+U_2+U_3+U_u)} \{P \star \gamma \rightarrow \beta\}\ \exists \_.\mathrm{unit}\ \{P \star \gamma \rightarrow \beta\}}\ \text{U-U}$$

$$\frac{\Sigma;\Delta;\Phi;\Omega \vdash^{U}_{L} t : A \qquad \Sigma;\Delta;\Phi \models A \sqsubseteq A' \qquad \Delta;\Phi \vDash U \leq U' \qquad \Delta;\Phi \vDash L' \leq L}{\Sigma;\Delta;\Phi;\Omega \vdash^{U'}_{L'} t : A'}\ \text{U-X}$$

Fig. 3. Selection of unary typing rules.

with explicit cost 0. Rules U-F and U-A are similar to the ones available in classical effect systems. We present them here to show how the costs change in the typing.

The remaining rules concern impure expressions of monadic types. Rules U-T and U-E type the unit and the bind of the monad, respectively. They combine the different costs and assertions in the monadic type, using a style similar to separation logic. The rule for allocations, U-L, introduces a new static location γ and creates a new monadic type whose postcondition assigns to γ all the natural numbers ($\mathbb{N}$), indicating that all the continuation has the permission to write all positions of the array. Additionally, like all other rules, this rule also adds a cost accounting for the forcing of the

$$\frac{\Sigma;\Delta;\Phi;\Gamma \vdash t \ominus t \lesssim D : \tau \qquad \forall x \in dom(\Gamma).\Sigma;\Delta;\Phi \models \Gamma(x) \sqsubseteq \Box\Gamma(x)}{\Sigma;\Delta;\Phi;\Gamma \vdash t \ominus t \lesssim 0 : \Box\tau}\ \text{R-NC}$$

$$\frac{\Sigma;\Delta;\Phi,C;\Gamma \vdash t_1 \ominus t_2 \lesssim D : \tau \qquad \Sigma;\Delta;\Phi,\neg C;\Gamma \vdash t_1 \ominus t_2 \lesssim D : \tau}{\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_2 \lesssim D : \tau}\ \text{R-P}$$

Fig. 4. Selection of pure relational synchronous typing rules.

$$\frac{\Sigma;\Delta;\Phi;|\Gamma|_1 \vdash^{U_1}_{L_1} t_1 : A_1 \qquad \Sigma;\Delta;\Phi;|\Gamma|_2 \vdash^{U_2}_{L_2} t_2 : A_2}{\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_2 \lesssim U_1 - L_2 : U(A_1,A_2)}\ \text{R-S}$$

$$\frac{\Sigma;\Delta;\Phi;|\Gamma|_1 \vdash^{U_1}_{L_1} t_1 : A_1 \qquad \Sigma;\Delta;\Phi;\Gamma,x : U(A_1,A_1) \vdash t_2 \ominus t_2' \lesssim D_2 : \tau}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \ominus t_2' \lesssim U_1 + D_2 + c_{lt} : \tau}\ \text{R-LT-E}$$

$$\frac{\Sigma;\Delta;\Phi;|\Gamma|_2 \vdash^{U_1}_{L_1} t_1' : A_1' \qquad \Sigma;\Delta;\Phi;\Gamma,x : U(A_1',A_1') \vdash t_2 \ominus t_2' \lesssim D_2 : \tau'}{\Sigma;\Delta;\Phi;\Gamma \vdash t_2 \ominus \mathsf{let}\ x = t_1'\ \mathsf{in}\ t_2' \lesssim D_2 - L_1 - c_{lt} : \tau'}\ \text{R-E-LT}$$

$$\frac{\Sigma;\Delta;\Phi;|\Gamma|_1 \vdash^{U_1}_{L_1} t_1 : A_1 \overset{\mathrm{exec}(L,U)}{\longrightarrow} A_2 \qquad \Sigma;\Delta;\Phi;\Gamma \vdash t_2 \ominus t_2' \lesssim D_2 : U(A_1,A_2')}{\Sigma;\Delta;\Phi;\Gamma \vdash t_1\ t_2 \ominus t_2' \lesssim U_1 + U + D_2 + c_{app} : U(A_2,A_2')}\ \text{R-APP-E}$$

$$\frac{\begin{array}{c}\Sigma;\Delta;\Phi;|\Gamma|_1 \vdash^{U_1}_{L_1} t : A_1 + A_2 \\ \Sigma;\Delta;\Phi;\Gamma,x : U(A_1,A_1) \vdash t_1 \ominus t' \lesssim D_2 : \tau \qquad \Sigma;\Delta;\Phi;\Gamma,y : U(A_2,A_2) \vdash t_2 \ominus t' \lesssim D_2 : \tau\end{array}}{\Sigma;\Delta;\Phi;\Gamma \vdash\ \mathsf{case}\ (t, x.t_1, y.t_2) \ominus t' \lesssim U_1 + D_2 + c_{case} : \tau}\ \text{R-CASE-E}$$

Fig. 5. Selection of pure relational asynchronous typing rules.

allocation. Finally, note that the upper- and lower-bounds on the judgment are 0. This is because alloc $t_1\ t_2$ is a value. In the pure evaluation, it returns without cost. Cost arises only when the term is forced; this is accounted in the cost annotations in the monadic type. The rule for reading, U-R, merely checks that the index being read is within the array bounds. The rule for updating, U-U, also performs a similar check but, in addition, it also requires that the updated index is contained in the permissions available for the array in the precondition. Finally, the rule U-X allows weakening the upper and lower bounds on the cost and applying subtyping.

Relational Typing Judgments. ARel's relational typing uses the judgment form

$$\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_2 \lesssim D : \tau$$

Here, t_1 and t_2 are two expressions, Σ, Δ, and Φ are environments similar to the ones used by unary typing judgments, Γ is a *relational type environment* assigning relational types to variables, τ is a relational type for t_1, t_2, and D is an index term representing an *upper bound* on the relative

cost of evaluating t_1 and t_2, i.e., $\text{cost}(t_1) - \text{cost}(t_2)$. We have two kinds of relational typing rules: *synchronous rules* which relate two structually similar programs, and *asynchronous rules* which relate programs that are not necessarily structurally similar. We first present a selection of the *pure* typing rules which include both synchronous rules (Figure 4) and asynchronous rules (Figure 5), inspired by the work of Çiçek et al. [2017]. Then, we present a selection of the *monadic* typing rules which support relational cost analysis for arrays. We present the synchronous rules (Figure 6) and the asynchronous rules (Figure 7). The rest of the typing rules can be found in the Appendix.

Pure Synchronous Rules. We present two synchronous rules R-P and R-NC in Figure 4. The other rules are similar to the one from Çiçek et al. [2017] and they can be found in the Appendix. Rule R-P allows reasoning by cases on any constraint in the constraint environment. Rule R-NC is the introduction rule for $\Box$-ed types. Briefly, t can be related to itself at type $\Box\tau$ when t relates to itself at type τ and, additionally, all variables in the context morally have $\Box$-ed types. The latter ensures that variables can only be substituted by equal terms. In this case, the relative cost is trivially 0.

Pure Asynchronous Rules. We present a selection of the pure asynchronous rules in Figure 5 including the generic rule R-S, and rules for the pure let binding, function application and case elimination. Rule R-S allows switching from relational reasoning about t_1 and t_2 to unary reasoning about the two terms, independently. Notice that the relational type in the conclusion is the embedding of the two unary types without any meaningful relation ($U(A_1, A_2)$). The rule uses a straightforward map $|\Gamma|_i$ from relational environments to unary environments, whose definition can be found in the appendix. Importantly, the relative cost in the conclusion is the difference of the unary costs in the premises. Rule R-LT-E relates a pure let binding expression to an arbitrary expression. In this rule, we use the metavariable c_{lt} to denote the cost of a let elimination. Notice that one of the assumptions in this rule, the one for the expression t_1, is a unary typing judgment. This is needed to provide guarantees on typability of t_1 and to provide the cost of evaluating it, which is used in the bound on the relative cost in the conclusion of the rule. The rule R-E-LT is dual to R-LT-E – it relates an arbitrary expression with a let. Notice that while the rule R-LT-E uses the upper bound on the unary cost of t_1, the rule R-E-LT uses the lower bound. Rule R-APP-E relates a function application with an arbitrary expression, while rule R-CASE-E relates a case expression with an arbitrary expression. Also in these rules we use some unary typing assumptions to guarantee typability and to provide unary costs which are used in giving upper bounds on the relative costs. We also have dual rules which we present in the Appendix.

Synchronous Rules. Figure 6 shows a selection of relational synchronous typing rules pertaining to monadic constructs and arrays. Rules R-T and R-LET relationally type the return and bind of our monad. The rules introduce the trivial relational Hoare-triple and combine two relational Hoare triples by sequencing, respectively. In particular, the rule R-LET uses the style of separation logic. Rule R-FIX-EXT types fixpoint expressions relationally. During the relational reasoning, it also allows assuming the *unary* types of the two functions, which are established in separate premises. This rule, introduces a weak form of *intersection types* in the environment which can be used in combination with the rule R-S (Figure 5) to give precise bounds on relative cost.

For each operation on arrays we have two rules, one that is general and the other that works under some assumption about equality of arguments in the two runs. Consider, for example, the rules R-L and R-LB for relationally typing the alloc construct. The rules are similar, e.g., both create a new static name γ for the two allocated arrays and both account for relative costs very similarly. However, R-LB applies only when the expressions initializing the two arrays are related at a $\Box$-ed type (second premise). As a result, it is guaranteed that the arrays allocated in the two runs will

$$\frac{P = P_1 \star P_2 \qquad \Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_1' \lesssim D_1 : \{P_1\}\ \exists \vec{\gamma_1}.\tau\ \{Q_1 \star Q_2\}^{\mathrm{diff}(D)} \qquad \Sigma;\Delta,\vec{\gamma_1}:\vec{\mathbb{L}};\Phi;\Gamma,x:\tau \vdash t_2 \ominus t_2' \lesssim D_2 : \{Q_1 \star P_2\}\ \exists \vec{\gamma_2}.\sigma\ \{Q\}^{\mathrm{diff}(D')}}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathsf{let}\{x\} = t_1\ \mathsf{in}\ t_2 \ominus \mathsf{let}\{x\} = t_1'\ \mathsf{in}\ t_2' \lesssim D_1 + D_2 : \{P\}\ \exists \vec{\gamma_1}\vec{\gamma_2}.\sigma\ \{Q \star Q_2\}^{\mathrm{diff}(D+D')}}\ \textsc{r-let}$$

$$\frac{\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_2 \lesssim D : \tau \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathsf{return}\ t_1 \ominus \mathsf{return}\ t_2 \lesssim 0 : \{P\}\ \exists \_.\tau\ \{P\}^{\mathrm{diff}(D)}}\ \textsc{r-t}$$

$$\frac{\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_1' \lesssim D_1 : \mathsf{int}[I] \quad \Sigma;\Delta;\Phi;\Gamma \vdash t_2 \ominus t_2' \lesssim D_2 : \tau \quad \gamma\ \mathsf{fresh} \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathsf{alloc}\, t_1\, t_2 \ominus \mathsf{alloc}\, t_1'\, t_2' \lesssim 0 : \{P\}\ \exists \gamma.\ \mathsf{Array}_\gamma[I]\ \tau\ \{P \star \gamma \to \mathbb{N}\}^{\mathrm{diff}(D_1+D_2)}}\ \textsc{r-l}$$

$$\frac{\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_1' \lesssim D_1 : \mathsf{int}[I] \qquad \Sigma;\Delta;\Phi;\Gamma \vdash t_2 \ominus t_2' \lesssim D_2 : \square\tau \qquad \gamma\ \mathsf{fresh} \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathsf{alloc}\, t_1\, t_2 \ominus \mathsf{alloc}\, t_1'\, t_2' \lesssim 0 : \{P\}\ \exists \gamma.\ \mathsf{Array}_\gamma[I]\ \tau\ \{P \star \gamma \to \emptyset\}^{\mathrm{diff}(D_1+D_2)}}\ \textsc{r-lb}$$

$$\frac{\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_1' \lesssim D_1 : \mathsf{Array}_\gamma[I]\ \tau \qquad \Sigma;\Delta;\Phi;\Gamma \vdash t_2 \ominus t_2' \lesssim D_2 : \mathsf{int}[I'] \qquad \Delta;\Phi \models I' \le I \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathsf{read}\ t_1\ t_2 \ominus \mathsf{read}\ t_1'\ t_2' \lesssim 0 : \{P\}\ \exists \_.\tau\ \{P\}^{\mathrm{diff}(D_1+D_2)}}\ \textsc{r-r}$$

$$\frac{\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_1' \lesssim D_1 : \mathsf{Array}_\gamma[I]\ \tau \qquad \Sigma;\Delta;\Phi;\Gamma \vdash t_2 \ominus t_2' \lesssim D_2 : \mathsf{int}[I'] \qquad \Delta;\Phi \models I' \le I \qquad I' \notin \beta \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathsf{read}\ t_1\ t_2 \ominus \mathsf{read}\ t_1'\ t_2' \lesssim 0 : \{P \star \gamma \mapsto \beta\}\ \exists \_.\square\tau\ \{P \star \gamma \mapsto \beta\}^{\mathrm{diff}(D_1+D_2)}}\ \textsc{r-rb}$$

$$\frac{\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_1' \lesssim D_1 : \mathsf{Array}_\gamma[I]\ \tau \qquad \Sigma;\Delta;\Phi;\Gamma \vdash t_2 \ominus t_2' \lesssim D_2 : \mathsf{int}[I'] \qquad \Sigma;\Delta;\Phi;\Gamma \vdash t_3 \ominus t_3' \lesssim D_3 : \tau \qquad \Delta;\Phi \models I' \le I \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathsf{updt}\ t_1\ t_2\ t_3 \ominus \mathsf{updt}\ t_1'\ t_2'\ t_3' \lesssim 0 : \{P \star \gamma \mapsto \beta\}\ \exists \_.\mathsf{unit}\ \{P \star \gamma \mapsto \beta \cup \{I'\}\}^{\mathrm{diff}(D_1+D_2+D_3)}}\ \textsc{r-u}$$

$$\frac{\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_1' \lesssim D_1 : \mathsf{Array}_\gamma[I]\ \tau \qquad \Sigma;\Delta;\Phi;\Gamma \vdash t_2 \ominus t_2' \lesssim D_2 : \mathsf{int}[I'] \qquad \Sigma;\Delta;\Phi;\Gamma \vdash t_3 \ominus t_3' \lesssim D_3 : \square\tau \qquad \Delta;\Phi \models I' \le I \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathsf{updt}\ t_1\ t_2\ t_3 \ominus \mathsf{updt}\ t_1'\ t_2'\ t_3' \lesssim 0 : \{P \star \gamma \mapsto \beta\}\ \exists \_.\mathsf{unit}\ \{P \star \gamma \mapsto \beta \setminus \{I'\}\}^{\mathrm{diff}(D_1+D_2+D_3)}}\ \textsc{r-ub}$$

$$\frac{\Sigma;\Delta;\Phi;x:\tau_1, f\ :\overset{\mathrm{diff}\,(D)}{\longrightarrow}\tau_2,\Gamma, f\ :U(A_1,A_2) \vdash t_1 \ominus t_2 \lesssim D : \tau_2 \qquad \Sigma;\Delta;\Phi;|\Gamma|_1 \vdash_0^0 \mathsf{Fix}\, f(x).t_1 : A_1 \qquad \Sigma;\Delta;\Phi;|\Gamma|_2 \vdash_0^0 \mathsf{Fix}\, f(x).t_2 : A_2}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathsf{Fix}\, f(x).t_1 \ominus \mathsf{Fix}\, f(x).t_2 \lesssim D : \tau_1 \overset{\mathrm{diff}\,(D)}{\longrightarrow} \tau_2}\ \textsc{r-fix-ext}$$

Fig. 6. Selection of monadic synchronous relational typing rules.

$$\frac{\Sigma;\Delta;\Phi;|\Gamma|_1 \vdash_{L_1}^{U_1} t_1 :\{P_1\}\ \overset{\text{exec}(L,U)}{\exists \vec{\gamma}_1 : A_1}\ \{Q_1\} \qquad \Sigma;\Delta;\Phi;|\Gamma|_2 \vdash_{L_2}^{U_2} t_2' :\{P_2\}\ \overset{\text{exec}(L',U')}{\exists \vec{\gamma}_1 : A_1'}\ \{Q_2\} \qquad dom(P) = dom(P_1) \qquad \Sigma;\Delta;\Phi;\Gamma, x : U(A_1,A_1) \vdash t_2 \ominus t_2' \lesssim D_2 : \{P \sqcup P_1\}\ \overset{\text{diff}(D')}{\exists \vec{\gamma}_1.\tau}\ \{Q\}}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathsf{let}\{x\} = t_1\ \mathsf{in}\ t_2 \ominus t_2' \lesssim -L_2 : \quad \{P\}\ \overset{\text{diff}(U_1+U+(D_2+U_2)+D'+c_{let})}{\exists \vec{\gamma}_1.\tau}\ \{Q\}}\ \text{R-LET-E}$$

$$\frac{\Sigma;\Delta;\Phi;|\Gamma|_2 \vdash_{L_1}^{U_1} t_1' :\{P_1\}\ \overset{\text{exec}(L,U)}{\exists \vec{\gamma}_1 : A_1'}\ \{Q_1\} \qquad \Sigma;\Delta;\Phi;|\Gamma|_1 \vdash_{L_2}^{U_2} t_2 :\{P_2\}\ \overset{\text{exec}(L',U')}{\exists \vec{\gamma}_1 : A_1}\ \{Q_2\} \qquad dom(P) = dom(P_1) \qquad \Sigma;\Delta;\Phi;\Gamma, x : U(A_1',A_1') \vdash t_2 \ominus t_2' \lesssim D_2 : \{P \sqcup P_1\}\ \overset{\text{diff}(D')}{\exists \vec{\gamma}_1.\tau'}\ \{Q\}}{\Sigma;\Delta;\Phi;\Gamma \vdash t_2 \ominus \mathsf{let}\{x\} = t_1'\ \mathsf{in}\ t_2' \lesssim U_2 : \quad \{P\}\ \overset{\text{diff}(D'+(D_2-L_2)-L_1-L-c_{\text{let}})}{\exists \vec{\gamma}_1.\tau'}\ \{Q\}}\ \text{R-E-LET}$$

Fig. 7. Selection of monadic asynchronous relational typing rules.

have equal values in all positions. This is reflected in the assertion $\gamma \to \emptyset$ in the postcondition of the monadic type in the rule, which says that there are no locations where the newly allocated arrays (named γ) can differ. In contrast, the rule R-L does not require the initializing expressions to be related at a $\Box$-ed type, but it has $\gamma \to \mathbb{N}$ in the postcondition, meaning that the two arrays may differ anywhere. A similar difference arises in the rules R-R and R-RB for relationally typing the construct read. In R-RB, the read index I' must not be in the β of the array being read in the precondition; as a result, the values read must be equal in the two runs. Hence, the resulting type has a $\Box$ on it. R-R is similar, but, here, there is no requirement that I' is not in the β, so two different values may be read, and there is no $\Box$ on the result type. The rules R-U and R-UB for updt follow the principle of alloc: In R-UB, the values being written in the two runs are known to be equal (via a $\Box$-ed type), so the index I' that is updated is removed from β in the postcondition. This is not the case in R-R, where it must be added to β, since the two values at index I' might differ after the update.[3] In all these rules, the premise $\Delta;\Phi \vDash I' \leq I$ denotes a constraint entailment which reads as follows: under the substitution of all the variables in the index environment Δ, under the assumption of the constraint Φ, the constraint $I' \leq I$ holds. This premise gurantees that the array bound is not exceeded. We omit here the rules for deriving this judgment since they are standard.

Finally, note that all monadic rules "propagate" relative costs from the premises to the monadic types. This is similar to the unary rules; the difference is that the costs propagated here are relative, whereas the unary type system propagates unary lower- and upper-bounds.

Asynchronous Rules. Figure 7 shows the two asynchronous rules R-LET-E and R-E-LET, relating a monadic binding construct and an arbitrary expression. We explain only the rule R-LET-E which relates the monadic binding construct $\mathsf{let}\{x\} = t_1\ \mathsf{in}\ t_2$ to an arbitrary expression t_2' (the rule R-E-LET is its dual and it can be understood similarly). The first premise of the rule R-LET-E requires a unary typing for the monadic expression t_1. This typing has two kinds of costs: the lower bound L_1 and upper bound U_1 for the unary execution cost of t_1, and the lower bound L and upper bound U for the execution cost of the resulting computation evaluated from t_1, this is embedded in the monadic type of t_1. The second premise requires a unary typing for the monadic expression t_2'. This gives

[3]The astute reader will note that the set of γs in any pre- or postcondition must be written down explicitly, i.e., we have not introduced sophisticated constructors (like set comprehension) for pre- and postconditions. This means that we cannot meaningfully specify monadic computations that allocate a data-dependent number of arrays. This hasn't been a problem for our examples, and we believe an extension to lift this restriction will be straightforward.

us an upper bound U_2 on the cost of evaluating this expression. The premise $dom(P) = dom(P_1)$ requires that the execution of the computation resulting from the expression t_1 can only affect arrays that appear in both P_1 and P. Finally, the last premise requires relating the subexpression t_2 to t_2' with the relative cost upper-bounded by D_2 under the assumption that the values substituted for the variable x are related at the type $U(A_1, A_1)$. Notice that this is the weakest requirement in terms of types that we can have. Additionally, this typing judgment also gives us the upper bound D' on the relative cost for executing the two computations resulting from evaluating the two expressions. To put the information of the unary and relational typing together we use the precondition $P \sqcup P_1$ in this premise, where the operation $\sqcup$ gives a precondition where a name γ which is used in P, e.g. $\gamma \mapsto \beta \in P$, and in P_1, e.g. $\gamma \mapsto \beta_1 \in P_1$, now points to the union of the two corresponding sets, i.e. e.g. $\gamma \mapsto \beta \cup \beta_1 \in P \sqcup P_1$. The conclusion of the rule uses all the cost information we discussed to compute an upper bound on the relative cost of the two expressions, where, as usual, we use the metavariable c_{let} to denote the cost of evaluating the monadic binding construct.

One can also design similar asynchronous rules for the other monadic constructs. However, the syntactic forms of the other constructs considerably constrain their asynchronous typing rules, making the scope of application of such rules rather narrow. For this reason we do not commit to the design of such rules here.

Subtyping. Subtyping is important in ARel. It serves several purposes. First, as in all refinement type systems, subtyping equates terms up to refinement, e.g., it allows replacing int$[2 + i]$ with int$[5]$ under the constraint $i = 3$. Second, specific to cost analysis, subtyping allows weakening costs, e.g., the relational type $\tau_1 \overset{\text{diff}(D)}{\longrightarrow} \tau_2$ can be subtyped to $\tau_1 \overset{\text{diff}(D')}{\longrightarrow} \tau_2$ when $D \leq D'$ since the D on the arrow is an upper bound on relative cost. Third, subtyping allows "massaging" of modalities $\Box$ and U, e.g., $\Box\tau$ can be subtyped to τ. Finally, specific to ARel, subtyping allows weakening of pre- and postconditions in monadic types. The first three uses are standard (e.g., see Çiçek et al. [2017]), so we only describe the last use here. The unary and relational subtyping judgments have the forms $\Sigma; \Delta; \Phi \models A_1 \sqsubseteq A_2$ and $\Sigma; \Delta; \Phi \models \tau_1 \sqsubseteq \tau_2$, respectively. Figure 8 shows selected subtyping rules. The notation $P \subseteq P'$ means that $P = \{\gamma_1 \to \beta_1, \gamma_2 \to \beta_2, \ldots, \gamma_n \to \beta_n\}$, $P' = \{\gamma_1 \to \beta_1', \gamma_2 \to \beta_2', \ldots, \gamma_n \to \beta_n'\}$, and $\forall i \in \{1, \ldots, n\}.\beta_i \subseteq \beta_i'$.

Rule S-UM allows subtyping on the unary monadic type. It says that we can subtype by weakening the costs, adding more (write) permissions to the pre-condition and removing permissions from the postcondition, as manifest in the premises $P \subseteq P'$ and $Q' \subseteq Q$. Rule S-RM similarly allows subtyping on the relational monadic type. This rule says that we can subtype by weakening the relative cost, making the precondition more precise and the postcondition less precise, where P' is more precise than P when P' tells us more about which values are equal. In particular, $\gamma \to \beta$ is more precise than $\gamma \to \beta'$ when $\beta' \subseteq \beta$. This is why the premises of S-RM check $P' \subseteq P$ and $Q \subseteq Q'$. Note that the checks on P, P' and Q, Q' are dual in the two rules. This is pure coincidence; the meanings of the pre-(post-)condition in the unary and relational monadic types are completely different. Finally, rule S-RUM allows subtyping from U applied to two unary monadic types to a single relational monadic type. This rule is best read as follows: If we have two computations that modify an array (γ_i) at positions in T_i and T_i', respectively (left side of $\sqsubseteq$), then running them on two arrays that agree at all positions outside the set β will result in two arrays that agree at all positions outside the set $\beta \cup T_i \cup T_i$' (right side of $\sqsubseteq$).

4 LOGICAL RELATIONS

To prove the soundness of ARel we build a step-indexed logical relation for its types. We give two interpretations—one unary and one monadic, that interact at the type $U(A_1, A_2)$.

$$\frac{\Sigma;\Delta;\Phi \models A \sqsubseteq A' \quad \vec{\gamma}_1 \subseteq \vec{\gamma}_2 \qquad \Sigma,\Delta;\Phi \models P \subseteq P' \qquad \Delta;\Phi \models L' \leq L \quad \Delta;\Phi \models U \leq U' \qquad \Sigma,\vec{\gamma}_1;\Delta;\Phi \models Q' \subseteq Q}{\Sigma;\Delta;\Phi \models \{P\}\ \overset{\text{exec}(L,U)}{\exists \vec{\gamma}_1.A}\ \{Q\} \sqsubseteq \{P'\}\ \overset{\text{exec}(L',U')}{\exists \vec{\gamma}_2.A'}\ \{Q'\}}\ \text{S-UM}$$

$$\frac{\Sigma;\Delta;\Phi \models \tau \sqsubseteq \tau' \quad \Delta;\Phi \models D \leq D' \qquad \Sigma,\vec{\gamma}_1;\Delta;\Phi \models Q \subseteq Q' \qquad \Sigma;\Delta;\Phi \models P' \subseteq P \qquad \vec{\gamma}_1 \subseteq \vec{\gamma}_2}{\Sigma;\Delta;\Phi \models \{P\}\ \overset{\text{diff}(D)}{\exists \vec{\gamma}_1.\tau}\ \{Q\} \sqsubseteq \{P'\}\ \overset{\text{diff}(D')}{\exists \vec{\gamma}_2.\tau'}\ \{Q'\}}\ \text{S-RM}$$

S-RUM

$$\frac{\beta_i' = \beta_i \cup T_i \cup T_i'}{\Sigma;\Delta;\Phi \vdash U(\{\gamma_i \to T_i\}\ \overset{\text{exec}(L,U)}{\exists \vec{\gamma}_1'.A_1}\ \{Q_1\}, \{\gamma_i \to T_i'\}\ \overset{\text{exec}(L',U')}{\exists \vec{\gamma}_2'.A_2}\ \{Q_2\}) \sqsubseteq \{\gamma_i \to \beta_i\}\ \overset{\text{diff}(U-L')}{\exists \vec{\gamma}_1, \vec{\gamma}_2.U(A_1,A_2)}\ \{\gamma_i \to \beta_i'\}}$$

Fig. 8. Selection of subtyping rules.

Unary Interpretation. The *value interpretation* $[\![A]\!]_{g,k}$ of a unary type A is, as usual, a set of values. Also, as usual, this interpretation is indexed by a "step-index" $k \in \mathbb{N}$, which is merely a proof device for induction [Ahmed 2006; Appel and McAllester 2001]. The step index counts the "steps" in our operational semantics. Importantly, the interpretation is also refined by a mapping g from static names γ to triples (l, n, A) expressing the location, the length of the array, and the *syntactic* type of the elements of the array named γ. g is our version of a Kripke world from the literature on logical relations [Neis et al. 2011; Turon et al. 2013]. We give a selection of the clauses defining the value interpretation of unary types in Figure 9.

For example, the value interpretation $[\![\text{Array}_\gamma[n]\ A]\!]_{g,k}$ of an array type is a set of locations l which are assigned to γ in g. The value interpretation $[\![\overset{\text{exec}(L,U)}{\{P\}\ \exists \vec{\gamma}.A}\ \{Q\}]\!]_{g,k}$ of a monadic type uses a heap relation $H \models_{g,k} P$ which says that the assertion P holds for the heap H at world g. All the static names γ in P refer through g to concrete arrays in H that have the right type and length. The value interpretation for monadic types is a set of monadic values v that when forced using a heap H validating the precondition P, yield a heap H_1 validating the postcondition Q. Additionally, the interpretation only allows those computations v that update arrays at locations for which the precondition P asserts permissions. We can extend the value interpretation to expressions:

$$[\![A]\!]_{g,k}^{E,(L,U)} = \{t \mid \forall v, k' \leq k.\ t \Downarrow^{c,k'} v \Rightarrow v \in [\![A]\!]_{g,k-k'} \wedge L \leq c \leq U\}$$

This definition accounts for costs. The interpretation is also extended to environments in a standard way. In the following theorem we use $\vdash \delta : \Delta$ and $\models \delta\Phi$ to denote that δ is a substitution for the index variables in Δ satisfying the constraint environment Φ.

Theorem 4.1 (Fundamental Theorem for Unary Typing). *If* $\Sigma;\Delta;\Phi;\Omega \vdash_L^U t : A$, $\vdash \delta : \Delta$ *and* $\models \delta\Phi$, *and* $\sigma \in [\![\delta\Omega]\!]_{g,k}$, *then* $(\delta\sigma t) \in [\![\delta A]\!]_{g,k}^{E,(\delta L,\delta U)}$.

Relational Interpretation. We give a selection of clauses for the definition of the *value interpretation* $(\!|\tau|\!)_{G,k}$ of relational types in Figure 9. The interpretation of a relational type is a set of pairs of related values. G, the Kripke world of the relational interpretation, is a mapping from static array names γ to 4-tuples (l_1, l_2, n, τ). If $G(\gamma) = (l_1, l_2, n, \tau)$, then l_1, l_2 are the locations where the arrays statically named γ are stored in the two runs, n is the length of these two arrays, and τ is the type at whose relational interpretation the two arrays' corresponding elements should be related. This is used for instance in the interpretation of the type $\text{Array}_\gamma[n]\ \tau$. To define the interpretation of

$$[\![\mathsf{int}[n]]\!]_{g,k} = \{\, n \,\} \quad (\!|\mathsf{int}[n]|\!)_{G,k} = \{\, (n,n) \} \quad [\![\mathsf{Array}_\gamma[n]\ A]\!]_{g,k} = \{\, l \mid g(\gamma) = (l,n,A) \}$$

$$(\!|\mathsf{Array}_\gamma[n]\ \tau|\!)_{G,k} = \{\, (l_1,l_2) \mid G(\gamma) = (l_1,l_2,\tau,n) \} \quad (\!|\Box\tau|\!)_{G,k} = \{\, (v,v) \mid (v,v) \in (\!|\tau|\!)_{G,k} \}$$

$$(\!|U(A_1,A_2)|\!)_{G,k} = \{\, (v_1,v_2) \mid \forall k', v_1 \in [\![A_1]\!]_{G|_1,k'} \wedge v_2 \in [\![A_2]\!]_{G|_2,k'} \}$$

$$[\![\{P\}\ \overset{\mathrm{exec}(L,U)}{\exists\vec{\gamma}.A}\ \{Q\}]\!]_{g,k} = \left\{ \begin{array}{l} v \mid \forall g_1 \supseteq g, \forall k_1 \le k, \forall k_2 < k_1, c, \forall H. \big(H \vDash_{g_1,k_1} P \wedge v; H \Downarrow_f^{c,k_2}\ \big) \\ \Rightarrow \exists g_2 \supseteq g_1, H_1, v_1, \vec{\gamma}. (v; H \Downarrow_f^{c,k_2}\ v_1; H_1) \wedge L \le c \le U \wedge H_1 \vDash_{g_2,k_1-k_2} Q \\ \wedge v_1 \in [\![A]\!]_{g_2,k_1-k_2} \wedge \big((\exists n.P = \{\gamma_1 \to T_1, \ldots, \gamma_n \to T_n\} \\ \wedge \forall i \in [1,n].g(\gamma_i) = (l_i,A,m)) \Rightarrow \forall j.(H[l_i][j]) \neq H_1[l_i][j] \Rightarrow j \in T_i\big) \end{array} \right\}$$

$$(\!|\{P\}\ \overset{\mathrm{diff}(D)}{\exists\vec{\gamma}.\tau}\ \{Q\}|\!)_{G,k} = \left\{ \begin{array}{l} (v_1,v_2) \mid \forall G_1 \supseteq G, k_1 \le k, k_2 < k_1, k_3, c_1, c_2, H_1, H_2. \\ \Big((H_1,H_2) \vDash_{G_1,k_1} P \wedge v_1; H_1 \Downarrow_f^{c_1,k_2} \wedge v_2; H_2 \Downarrow_f^{c_2,k_3}\ \Big) \Rightarrow \\ \exists G_2 \supseteq G_1, H_1', H_2', v_1', v_2', \vec{\gamma}. \Big(\ v_1; H_1 \Downarrow_f^{c_1,k_2}\ v_1'; H_1' \wedge\ v_2; H_2 \Downarrow_f^{c_2,k_3}\ v_2'; H_2' \\ \wedge (H_1',H_2') \vDash_{G_2,k_1-k_2} Q \wedge\ (v_1',v_2') \in (\!|\tau|\!)_{G_2,k_1-k_2} \wedge c_1 - c_2 \le D\Big) \end{array} \right\}$$

Fig. 9. Selection of the clauses defining the unary interpretation and the relational interpretation of types.

monadic types we need a relation $(H_1,H_2) \vDash_{G,k} P$ which says that the relational assertion P holds of the heaps H_1, H_2 at world G. We show the definition for the case $P = \{\gamma \to \beta\}$ here:

$$\begin{array}{rl} (H_1,H_2) \vDash_{G,k} (\gamma \to \beta)\ \ \mathsf{iff}\ \ & \exists l_1,l_2,\tau,n : G(\gamma) = (l_1,l_2,\tau,n) \\ & \wedge \big(\forall i \le n.\ \big(H_1(l_1)[i], H_2(l_2)[i]\big) \in (\!|\tau|\!)_{G,k-1}\big) \\ & \wedge \big(\forall i \le n.\ \big(H_1(l_1)[i] \neq H_2(l_2)[i]\ \Rightarrow\ i \in \beta\big)\big) \end{array}$$

Two heaps H_1, H_2 satisfy the assertion $\gamma \to \beta$ when all the related elements in the arrays $H_1(l_1)$ and $H_2(l_2)$ are in the value interpretation $(\!|\tau|\!)_{G,k-1}$ and, additionally, for indices $i \notin \beta$, the corresponding array elements are *equal*. Thus, β tracks positions where the two arrays *may* differ, consistent with our earlier description. Whether elements at indices in β can actually differ or not depends on τ. For example, when τ is $\mathsf{int}[n]$ (for some n) or even $\exists i.\mathsf{int}[i]$, this forces corresponding elements to be equal at all indices (not just at those outside β) since the relational interpretation of $\mathsf{int}[n]$ is the singleton $\{(n,n)\}$. However, when $\tau = U(A_1,A_2)$, elements at indices not in β can be arbitrary values of types A_1, A_2 since the relational interpretation of $U(A_1,A_2)$ is morally $A_1 \times A_2$.

The definition of the relational interpretation for a monadic type $\{P\}\ \overset{\mathrm{diff}(D)}{\exists\vec{\gamma}.\tau}\ \{Q\}$ (Figure 9) is the set of pairs of values (v_1,v_2) that when forced starting from heaps H_1, H_2 satisfying the precondition P, result in heaps H_1', H_2' satisfying the postcondition Q. The relative cost of forcing must be upper-bounded by D. We extend the relational interpretation to pairs of expressions as follows:

$$(\!|\tau|\!)^{E,D}_{G,k} = \{(t_1,t_2) \mid \forall k_1 \le k, v_1\ v_2. (t_1 \Downarrow^{c_1,k_1} v_1 \wedge t_2 \Downarrow^{c_2,k_2} v_2) \Rightarrow (v_1,v_2) \in (\!|\tau|\!)_{G,k-k_1} \wedge c_1 - c_2 \le D\}$$

We also extend the interpretation to environments in the obvious way and prove a fundamental theorem for the relational typing.

Theorem 4.2 (Fundamental Theorem for Relational Typing). *If* $\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_2 \lesssim D : \tau$ *and* $\vdash \delta : \Delta$ *and* $\vDash \delta\Phi$ *and* $(\sigma_1,\sigma_2) \in (\!|\delta\Gamma|\!)_{G,k}$, *then* $(\delta\sigma_1 t_1, \delta\sigma_2 t_1) \in (\!|\delta\tau|\!)^{E,(\delta D)}_{G,k}$.

For readers familiar with Kripke logical relations, we note that our worlds g and G are *not* step-indexed (only our logical relations are step-indexed). This is unlike some prior work [Neis et al.

2011; Turon et al. 2013]. We do not need step-indexed worlds since we include syntactic types, A or τ, for mutable locations (arrays) in the worlds. This suffices for our purposes. Our logical relations are well-founded. The relational interpretation for heaps, $(H_1, H_2) \vDash_{G,k} P$, refers back to the value interpretation only at a step index strictly smaller than k, while the value interpretation refers back to the heap relation (via the clause for the monadic type, $\{P\}\ \exists\vec{\gamma}.A\ \{Q\}^{\mathrm{exec}(L,U)}$) at a smaller or equal step index. Consequently, the relation is well-founded by the lexicographic order ⟨step-index, size of type⟩. Our unary interpretation is well-founded for a similar reason.

5 MORE EXAMPLES

We discuss here three more examples demonstrating how we perform relational cost analysis on programs with arrays. To improve readability, we omit some annotations and use syntactic sugar.

Cooley Tukey FFT Algorithm. This example shows how to use relational cost analysis to reason about *constant-time* programs with imperative updates. We consider the following implementation of the Cooley Tukey algorithm for fast Fourier transforms [Cooley and Tukey 1965].

$$\begin{array}{l}\mathsf{fix}\ \mathsf{FFT}\ (\_).\lambda x.\lambda y.\lambda n.\lambda p.\\ \quad \mathsf{if}\ 2 \le n\ \mathsf{then}\ \mathsf{let}\ \{\_\} = \mathsf{separate}\ ()\ x\ n\ y\ p\ \mathsf{in}\ \mathsf{let}\ \{\_\} = \mathsf{FFT}\ ()\ x\ y\ (n/2)\ p\ \mathsf{in}\\ \qquad \mathsf{let}\ \{\_\} = \mathsf{FFT}\ ()\ x\ y\ (n/2)\ (p + n/2)\ \mathsf{in}\ \mathsf{loop}()\ 0\ n\ x\ p\ \mathsf{else}\ \mathsf{return}\ ()\end{array}$$

FFT implements a divide-and-conquer algorithm. x is the input array, y is another array used for temporary storage, n is the length of the two arrays, and p is an index pointing to the index where the array should be split in the recursive call. This function uses a helper function separate to relocate elements in even positions to the lower half of the array x and elements in odd positions to the upper half of the input array x respectively, using y as a scratchpad. We omit the code of separate here; it can be found in the Appendix. Another helper function loop simulates a for loop in which the input array x is updated using the mathematical manipulations needed for the Fourier transform.

$$\begin{array}{l}\mathsf{fix}\ \mathsf{loop}\ (\_).\lambda k.\lambda n.\lambda x.\lambda p.\ \mathsf{if}\ k < (n/2)\ \mathsf{then}\\ \quad \mathsf{let}\ \{e\} = \mathsf{read}\ x\ (k + p)\ \ \mathsf{in}\ \mathsf{let}\ \{o\} = \mathsf{read}\ x\ (k + p + n/2)\ \ \mathsf{in}\\ \quad \mathsf{let}\ w = \exp(-2\pi k/n)\ \mathsf{in}\ \mathsf{let}\ \{\_\} = \mathsf{updt}\ x\ (k + p)\ (e + w * o)\ \ \mathsf{in}\\ \quad \mathsf{let}\ \{\_\} = \mathsf{updt}\ x\ (k + p + n/2)\ (e - w * o)\ \ \mathsf{in}\ \mathsf{loop}\ ()\ (k + 1)\ n\ x\ p\ \mathsf{else}\ \mathsf{return}\ ()\end{array}$$

Intuitively, this example is constant time (for arrays with fixed length) because array manipulations depend only on the positions of the elements, and not on their values (assuming constant time addition and multiplication). One way to internalize this observation in the typing process is using only relational rules and relational types, always with relative cost 0. We do this for the auxiliary functions separate and loop first, and with these we can easily give the following relational type to FFT, witnessing the constant-time nature of this function.

$$\begin{array}{ll}\vdash \mathsf{FFT} \ominus \mathsf{FFT} \lesssim 0 : & \mathrm{unit} \to \forall\gamma_1,\gamma_2,\beta_1,M,N,P.(P + M < N) \supset\\ & \mathrm{Array}_{\gamma_1}[N]\ U(\mathrm{int},\mathrm{int}) \to \mathrm{Array}_{\gamma_2}[N]\ U(\mathrm{int},\mathrm{int}) \to \mathrm{int}[M] \to \mathrm{int}[P] \to\\ & \{\gamma_1 \to \beta_1, \gamma_2 \to \mathbb{N}\}\ \exists\_.\mathrm{unit}\ \{\gamma_1 \to \mathbb{N}, \gamma_2 \to \mathbb{N}\}^{\mathrm{diff}(0)}\end{array}$$

Another way to achieve the same result is to first compute the precise lower and upper bounds on the unary cost of FFT and then show that they are, in fact, equal. However, computing the precise unary cost of FFT is more difficult. We first establish the precise cost of the helper functions. For

example, we need to give the following unary type to loop.

$$\vdash \text{loop} : \begin{array}{l} \text{unit} \to \forall \gamma_1, K, M, N, P.(P + M < N) \supset \text{int}[K] \to \text{int}[M] \to \text{Array}_{\gamma_1}[N]\ \text{int} \to \\ \text{int}[P] \to \{\gamma_1 \xrightarrow{\text{exec}(4*(M-K),4*(M-K))} \mathbb{N}\}\ \exists\_.\text{unit}\ \{\gamma_1 \to \mathbb{N}\} \end{array}$$

Similarly, for the function separate we would need to establish the precise cost, $4 * M$. Once these unary costs are available, we can conclude that the function FFT has the same min and max costs: $8 * M * \log(M)$ and is, thus, constant time (using the rule R-S).[4]

While both the unary and relational reasoning can show that this example is constant time, the relational reasoning is *much* easier in this case since the relative cost is 0 everywhere.

Naive String Search. We show how a combination of unary and relational reasoning can give a precise relative cost to a class array algorithm: substring search.

$$\begin{array}{l} \text{NSS} = \text{fix}\ F\ (s).\ \lambda w.\lambda m.\lambda l_s.\lambda l_w.\lambda p. \\ \quad \text{if}\ (m + l_w) \le l_s\ \text{then let}\ \_ = \text{search}\ s\ w\ m\ 0\ l_s\ l_w\ p \\ \quad\quad \text{in}\ F(s)\ w\ (m+1)\ l_s\ l_w\ p\ \text{else return}\ () \end{array}$$

Here, strings are represented as arrays of integers (storing the ASCII code of each character). The function NSS takes as input, a "long" string s and a "short" string w in arrays, the lengths l_s and l_w of these arrays, and an array p of length l_s (we call this the result array). This function iteratively searches the substring w at each position in s and records whether the substring is found at that position (1) or not (0). To do this, NSS uses the following helper function search.

$$\begin{array}{l} \text{fix search}\ (\_).\ \lambda s.\ \lambda w.\ \lambda m.\ \lambda i.\ \lambda l_s.\ \lambda l_w.\ \lambda p. \\ \quad \text{let}\ \{x\} = \text{read}\ s\ (m+i)\ \text{in let}\ \{y\} = \text{read}\ w\ i\ \text{in} \\ \quad\quad \text{if}\ (i + 1 == l_w)\ \text{then}\ \big(\text{if}\ (x == y)\ \text{then}\ \text{updt}\ p\ m\ 1\ \text{else}\ \text{updt}\ p\ m\ 0\ \big) \\ \quad\quad \text{else}\big(\text{if}\ (x == y)\ \text{then search}\ ()\ s\ w\ m\ (i+1)\ l_s\ l_w\ p\ \text{else}\ \text{updt}\ p\ m\ 0\ \big) \end{array}$$

The function search has the same inputs as NSS except for the additional index i, that iterates over the positions of l_w. The two conditionals check whether search is in its final step ($i + 1 == l_w$), and whether the two corresponding characters in s and w coincide. When the two characters differ, p is updated with 0. When the two conditionals are satisfied at the same time, p is updated with 1.

Intuitively, search runs fastest when the first character of w does not appear in s. It runs slowest when the suffix of w starting at index i occurs in s at offset $m + i$. The difference of these two costs is a bound on the relative cost of search. However, in the specific case where we consider two runs of search on the same string s, the same index i and where the two ws agree on some prefix, we can see that the two runs behave identically until we reach an index where the ws start to differ. In this case, we can give a better bound. To write this bound, we need to express the first index in the range $[i, l_w]$ where the two ws differ. In ARel, the index term $MIN(\beta_2 \cap [I, \infty))$ represents this index (assuming β_2 is the relational pre-condition of w and I is the static index refinement for i's size). Then, search incurs a nontrivial relative cost only *after* this index is reached. Using this idea, we can show:

$$\vdash \text{search} \ominus \text{search} \lesssim 0 : \begin{array}{l} \text{unit} \to \forall \gamma_1, \gamma_2, \gamma_3, I, M, R, N, \beta_2, \beta_3.(I < R < N \wedge M + I < N) \supset \\ \text{Array}_{\gamma_1}[N]\ U(\text{int}) \to \text{Array}_{\gamma_2}[R]\ U(\text{int}) \to \text{int}[M] \to \text{int}[I] \to \text{int}[N] \to \\ \text{int}[R] \xrightarrow{\text{diff}((R-1-min(MIN(\beta_2 \cap [I,\infty)),R-1))*r)} \text{Array}_{\gamma_3}[N]\ U(\text{bool}) \to \{P, \gamma_3 \to \beta_3\}\ \exists\_.\text{unit}\ \{P, \gamma_3 \to \beta_3 \cup \{M\}\} \end{array}$$

[4]It is not hard to see that FFT has unary cost in $O(M * \log(M))$: The unary costs of separate and the call to loop() are both linear in M, so the cost $f(M)$ of FFT satisfies the recurrence $f(M) = 2 * f(M/2) + O(M)$, which has the standard solution $O(M * \log(M))$. However, proving this in the type system is much harder than the direct relational proof of 0 relative cost.

where $P = \gamma_1 \to \emptyset, \gamma_2 \to \beta_2$, R is the static size of l_w, and r is the (constant) cost of two read operations. To account for the case where w is the same in the two executions we also add a lower bound $R - 1$ in the cost. The relative cost we establish here is more precise than the one we would achieve with a non-relational analysis ($(R - 1 - I) * r$).

We stress here that to obtain this relative cost, the rule R-FIX-EXT is essential. At a high level, typing proceeds by case analysis on $I \in \beta_2$. When $I \notin \beta_2$ we can proceed relationally with relative cost 0 in the recursive call. When $I \in \beta_2$ the control flows may differ in the two runs and we need to switch to unary reasoning via the rule R-S. To obtain our bound using unary worst- best-case analysis we need the precise unary type of search, which is available in the context thanks to the rule R-FIX-EXT. The details of this proof are in our appendix. By using the typing above for search, we can also obtain an improved relative cost for NSS relative to itself: $(R - 1 - min(MIN(\beta_2 \cap [I, \infty)), R - 1)) * r * (N - M - R)$. This is simply the number of times search is called multiplied by the relative cost of search.

Inplace Insertion Sort. Our next example, *inplace insertion sort*, implements the insertion sort algorithm without any temporary arrays. The relative cost is complex but we can show that under reasonable assumptions, ARel provides a more precise relative cost than a unary analysis. The algorithm is written in our language as follows.

$$\begin{aligned}&\mathsf{fix}\ \mathsf{ISort}\ (\_).\ \lambda s.\ \lambda i.\ \lambda l_s.\ \mathsf{if}\ (i < l_s)\ \mathsf{then}\ \mathsf{let}\ \{a\} = \mathsf{read}\ s\ i\ \ \mathsf{in}\\ &\quad \mathsf{let}\ \{b\} = \mathsf{insert}\ ()\ s\ a\ 0\ i\ \mathsf{in}\ \mathsf{ISort}\ ()\ s\ (i+1)\ l_s\ \mathsf{else}\ \mathsf{return}\ ()\end{aligned}$$

Intuitively, we observe that the cost of ISort relative to itself should be the sum of the possible cost variation of every recursive call, which is mainly decided by the auxiliary function insert below.

$$\begin{aligned}&\mathsf{fix}\ \mathsf{insert}\ (\_).\ \lambda s.\ \lambda a.\ \lambda x.\ \lambda i.\ \mathsf{let}\ \{b\} = \mathsf{read}\ s\ x\ \ \mathsf{in}\\ &\quad \mathsf{if}\ (a \geq b)\ \mathsf{then}\ \mathsf{insert}\ ()\ s\ a\ (x+1)\ i\\ &\quad \mathsf{else}\ \mathsf{let}\ \{\_\} = \mathsf{shift}\ ()\ s\ x\ (i-1)\ \mathsf{in}\ \ \mathsf{updt}\ s\ x\ a\end{aligned}$$

The recursive function insert implements the standard operation of inserting an element into an array by finding the right position x to insert the element at and shifting elements behind x in the array backward before updating the value at index x to a. This function uses a helper function shift, which performs the shift operation. We omit the code here but note that shift uses one read operation and one update operation at every index, and finding the right position only needs one read operation. The unary cost of ISort is maximum when the input array is initially sorted in descending order. In contrast, the unary cost is minimum when the input array is initially sorted ascending. Assuming that read and update operations incur unit cost, the unary type of insert is as follows.

$$\begin{aligned}\vdash \mathsf{insert}: \ &\mathsf{unit} \to \forall\gamma_1.\forall N, X, I.(X \leq N \wedge I \leq N) \supset \mathsf{Array}_{\gamma_1}[N]\ \mathsf{int}\\ &\xrightarrow{\mathsf{exec}(I-X+1,\,2*(I-X)+2)} \mathsf{int} \to \mathsf{int}[X] \to \mathsf{int}[I] \to \{\gamma_1 \to \mathbb{N}\}\ \exists\_.\mathsf{unit}\ \{\gamma_1 \to \mathbb{N}\}\end{aligned}$$

With this unary type of insert in hand, we can obtain the relative cost of insert by switching to unary reasoning and then taking the difference. An interesting observation is that if the input arrays of the two runs coincide in the insertion range $[0, I]$ and the elements 'a' being inserted also agree, then insert's cost relative to itself is 0. The corresponding relational type is shown below.

$$\begin{aligned}\vdash \mathsf{insert} \ominus \mathsf{insert} \lesssim 0: \ &\mathsf{unit} \to \forall\gamma_1, \beta_1.\forall N, A, X, I.(X \leq N \wedge I \leq N \wedge \beta_1 \cap [X, I] = \emptyset) \supset\\ &\xrightarrow{\mathsf{diff}(0)} \mathsf{Array}_{\gamma_1}[N]\ U(\mathsf{int}) \to \mathsf{int} \to \mathsf{int}[X] \to \mathsf{int}[I] \to \{\gamma_1 \to \beta_1\}\ \exists\_.\mathsf{unit}\ \{\gamma_1 \to \beta_1\}\end{aligned}$$

This observation can be used in typing ISort: For every I, we split cases on whether $\beta_1 \cap [0, I] = \emptyset$ or not (using rule R-P). While $\beta_1 \cap [0, I] = \emptyset$, we proceed relationally (with 0 relative cost). Once $\beta_1 \cap [0, I] \neq \emptyset$, we switch to unary reasoning using rule R-S since control flow may differ in the two runs. As in the previous example, we need the rule R-FIX-EXT for this. Using this idea, we obtain a very precise relational cost for ISort.

$$\vdash \mathsf{ISort} \ominus \mathsf{ISort} \lesssim 0 : \begin{array}{l} \mathsf{unit} \to \forall \gamma_1, \beta_1, N, I.(I \leq N) \supset \\ \mathsf{Array}_{\gamma_1}[N]\ U(\mathsf{int}) \to \mathsf{int}[I] \to \mathsf{int}[N] \to \{\gamma_1 \to \beta_1\}\ \exists \_.\mathsf{unit}\ \{\gamma_1 \to \mathbb{N}\}^{\mathrm{diff}(\frac{N*(N+1)-k*(k+1)}{2})} \end{array}$$

where the index term $k = max(I, min(MIN(\beta_1), N))$ represents the first index where the two arrays differ. The relative cost $\frac{N*(N+1)-k*(k+1)}{2}$ is the sum of all the relative costs generated in the recursive calls corresponding to indices in the range $[k, N]$. Recursive calls up to index k incur 0 relative cost, as noted above. More details are provided in the appendix. We note that the cost obtained here is more precise than the relative cost that can be obtained using unary reasoning alone.

6 BIDIRECTIONAL TYPE CHECKING

Our next goal is to implement our type system ARel to try out our examples. However, implementing ARel naively results in an immediate challenge: Some of the rules are not syntax-directed and lead to nondeterminism in an implementation. For example, in the split rule R-P, we can choose any constraint to split on (and this is an infinite space); we can apply the switch rule R-S anywhere; in the rule R-FIX-EXT we have to guess the unary types of the functions (again an infinite space); and, there are two rules for every array operator. To resolve this nondeterminism, we introduce an extended expression language with annotations to guide type-checking. For example, the term $(\mathsf{split}\ \ t\ \text{with}\ C)$ marks uses of rule R-P that split on constraint C in type-checking t. The use of rule R-FIX-EXT is indicated by the construct $(\mathsf{FIXEXT}\ f(x).t\ \text{with}\ \ A)$ that provides the unary type A of $(\mathsf{fix} f(x).t)$. Similarly, we introduce two variants of array operations, e.g., alloc and $\mathsf{alloc}_\mathsf{b}$ corresponding to the two rules R-L and R-LB, respectively.

Further, there is nondeterminism in subtyping due to the modal types $\Box$ and U. We resolve this by adding explicit type coercions where subtyping would be needed. Prior work shows that this approach is complete for these two modal types [Çiçek et al. 2019].

Beyond this, we face the usual challenge of algorithmizing any type system: The need to either annotate or infer the types of bound variables. Here, the problem is more nuanced than would be in a simply typed or even a refinement type calculus, since we must also deal with cost bounds in function and monadic types. To address this challenge, we rely on the well-known middle ground of bidirectional type-checking or local type inference [Pierce and Turner 2000], where type annotations must be provided only at explicit β-redexes and for top-level functions, but everything else can be inferred. We design a bidirectional type system for ARel which is similar in spirit to the one for RelCost [Çiçek et al. 2019], but extended in nontrivial ways to support bidirectional type-checking for array operations and for our fixpoint extension. This system can derive in an algorithmic way four typing judgments, two relational and two unary. The relational typing judgment of ARel splits into two relational judgments in the bidirectional version, one for the "checking mode" and one for "inference mode". The relational checking judgment has the form:

$$\Sigma; \Delta; \psi_a; \Phi_a; \Gamma \vdash t_1 \ominus t_2 \downarrow \tau, D \Rightarrow \Phi.$$

Given the location environment Σ, the index variable environment Δ, the existential variable context Ψ_a, the current constraint environment Φ_a, the relational typing context Γ, and terms t_1 and t_2, we *check* against the relational type τ and the relative cost D, and we generate the constraint Φ, which

$$\frac{\Delta;\psi_a;\Phi_a;|\Gamma| \vdash t_1 \uparrow A_1 \Rightarrow [\psi_1], \_, U_1, \Phi_1 \qquad \Delta;\psi_a;\Phi_a;|\Gamma| \vdash t_2 \uparrow A_2 \Rightarrow [\psi_2], L_2, \_, \Phi_2}{\Delta;\psi_a;\Phi_a;\Gamma \vdash \text{switch } t_1 \ominus \text{switch } t_2 \uparrow U\,(A_1, A_2) \Rightarrow [\psi_1, \psi_2], U_1 - L_2, \Phi_1 \wedge \Phi_2}\ \textbf{alg-r-switch}\uparrow$$

$$\frac{\begin{array}{c} L_1, U_1, L_2, U_2 \in \text{fresh}(\mathbb{R}) \qquad \Delta;U_1, L_1, \psi_a;\Phi_a;|\Gamma| \vdash t_1 \downarrow A_1, L_1, U_1 \Rightarrow \Phi_1 \\ \Delta;U_2, L_2, \psi_a;\Phi_a;|\Gamma| \vdash t_2 \downarrow A_2, L_2, U_2 \Rightarrow \Phi_2 \quad \Phi = \Phi_1 \wedge \exists L_2, U_2 :: \mathbb{R}.\Phi_2 \wedge U_1 - L_2 \doteq D \end{array}}{\Sigma;\Delta;\psi_a;\Phi_a;\Gamma \vdash \text{switch } t_1 \ominus \text{switch } t_2 \downarrow U\,(A_1, A_2), D \Rightarrow \exists L_1, U_1 :: \mathbb{R}.(\Phi)}\ \textbf{alg-r-switch}\downarrow$$

$$\frac{\begin{array}{c} \Sigma;\Delta;\psi_a;C \wedge \Phi_a;\Gamma \vdash t_1 \ominus t_1' \downarrow \tau, D \Rightarrow \Phi_1 \\ \Sigma;\Delta;\psi_a;\neg C \wedge \Phi_a;\Gamma \vdash t_1 \ominus t_1' \downarrow \tau, D \Rightarrow \Phi_2 \qquad \Delta \vdash C\ \text{wf} \end{array}}{\Sigma;\Delta;\psi_a;\Phi_a;\Gamma \vdash \text{split } (t_1) \text{ with } C \ominus \text{split } (t_1') \text{ with } C \downarrow \tau, D \Rightarrow C \rightarrow \Phi_1 \wedge \neg C \rightarrow \Phi_2}\ \textbf{alg-r-split}\downarrow$$

alg-fixext$\downarrow$

$$\frac{\begin{array}{c} \Delta;\psi_a;\Phi_a;|\Gamma| \vdash \text{fix} f(x).t \downarrow A_1, 0, 0 \Rightarrow \Phi_1 \qquad \Delta;\psi_a;\Phi_a;|\Gamma| \vdash \text{fix} f(x).t' \downarrow A_2, 0, 0 \Rightarrow \Phi_2 \\ \Sigma;\Delta;\psi_a;\Phi_a;f : \tau_1 \xrightarrow{\text{diff}(D')} \tau_2, f : U(A_1, A_2), x : \tau_1, \Gamma \vdash t \ominus t' \downarrow \tau_2, D' \Rightarrow \Phi \qquad \Phi_r = \Phi \wedge \Phi_1 \wedge \Phi_2 \end{array}}{\Sigma;\Delta;\psi_a;\Phi_a;\Gamma \vdash \text{FIXEXT} f(x).t \text{ with } A_1 \ominus \text{FIXEXT} f(x).t' \text{ with } A_2 \downarrow \tau_1 \xrightarrow{\text{diff}(D')} \tau_2, D \Rightarrow \Phi_r \wedge 0 \doteq D}$$

Fig. 10. Selection of algorithmic typing rules

must be discharged separately. In contrast, the relational inference judgment has the form:

$$\Sigma;\Delta;\psi_a;\Phi_a;\Gamma \vdash t_1 \ominus t_2 \uparrow \tau \Rightarrow [\psi], D, \Phi.$$

Here, we *synthesize* the relational type τ and the relative cost D, and we generate the constraint Φ with all the newly generated (existential) variables in ψ. We have similar judgments for the unary case. The unary checking judgment has the form $\Sigma;\Delta;\psi_a;\Phi_a;\Omega \vdash t \downarrow A, L, U \Rightarrow \Phi$, while the unary inference judgment has the form $\Sigma;\Delta;\psi_a;\Phi_a;\Omega \vdash t \uparrow A \Rightarrow [\psi], L, U, \Phi$. Both these judgments can be understood in a way similar to their relational counterparts. In all the judgments, we write all the outputs (inferred components) in red and inputs in black. Notice that in comparison with the typing judgments in ARel, the algorithmic typing judgments have one more input context Ψ_a which records previously eliminated existential variables.

We show selected algorithmic typing judgments in Figure 10 to explain how we handle ARel's non-determinism. The switch rule (R-S) exists in both checking and inference modes because we find it convenient to use the rule in both modes in our examples. Both algorithmic rules relate the annotated terms (switch t_1) and (switch t_2) at the type $U\,(A_1, A_2)$ and generate the final constraint based on the constraints from subterms t_1 and t_2 obtained in unary mode. The relative cost D must be the difference of the maximal unary cost of t_1 (U_1) and the minimal unary cost of t_2 (L_2). In the checking rule, **alg-r-switch**$\downarrow$, this is forced in the output constraint. The split rule (R-P) exists only in checking mode (**alg-r-split**$\downarrow$). The terms (split t_1 with C) and (split t_2 with C) determine that this rule must be applied, splitting on constraint C. The final output constraint $C \rightarrow \Phi_1 \wedge \neg C \rightarrow \Phi_2$ also analyzes C. The algorithmic counterpart of the rule R-FIX-EXT in checking mode, **alg-fixext**$\downarrow$, relates the annotated terms (FIXEXT $f(x).t$ with A_1) and (FIXEXT $f(x).t$ with A_1) and checks the subterms $\text{fix} f(x).t$ and $\text{fix} f(x).t'$ at the unary types A_1 and A_2, respectively. The final constraint is the combination of the constraints generated from the unary checking of the two subterms and the relational checking of the two function bodies.

Next, we discuss selected rules for array operations. These operations constitute the main challenge in our bidirectional type system relative to prior work. We show a selection of bidirectional

alg-r-alc-↓

$$\frac{\begin{array}{c}D_1, D_2 \in \mathsf{fresh}(\mathbb{R}) \\ \Sigma; \Delta; D_1, \psi_a; \Phi_a; \Gamma \vdash t_1 \ominus t_1' \downarrow \mathsf{int}[I], D_1 \Rightarrow \Phi_1 \qquad \Sigma; \Delta; D_2, \psi_a; \Phi_a; \Gamma \vdash t_2 \ominus t_2' \downarrow \tau, D_2 \Rightarrow \Phi_2 \\ \Phi = \Phi_2 \wedge D_1 + D_2 \doteq D \qquad \Sigma \vdash \gamma \text{ fresh} \qquad \Sigma; \Delta \vdash P \text{ wf} \qquad \Phi_r = \exists D_1 :: \mathbb{R}.\Phi_1 \wedge (\exists D_2 :: \mathbb{R}.\Phi)\end{array}}{\Sigma; \Delta; \psi_a; \Phi_a; \Gamma \vdash \mathsf{alloc}\ t_1\ t_2 \ominus \mathsf{alloc}\ t_1'\ t_2' \downarrow \ \{P\}\ \exists \gamma.\ \mathsf{Array}_\gamma[I]\ \tau\ \{P \star \gamma \rightarrow \mathbb{N}\}, 0 \overset{\mathsf{diff}(D)}{\Rightarrow} \Phi_r}$$

alg-r-alcB-↓

$$\frac{\begin{array}{c}D_1, D_2 \in \mathsf{fresh}(\mathbb{R}) \\ \Sigma; \Delta; D_1, \psi_a; \Phi_a; \Gamma \vdash t_1 \ominus t_1' \downarrow \mathsf{int}[I], D_1 \Rightarrow \Phi_1 \qquad \Sigma; \Delta; D_2, \psi_a; \Phi_a; \Gamma \vdash t_2 \ominus t_2' \downarrow \square\tau, D_2 \Rightarrow \Phi_2 \\ \Phi = \Phi_2 \wedge D_1 + D_2 \doteq D \qquad \Sigma \vdash \gamma \text{ fresh} \qquad \Sigma; \Delta \vdash P \text{ wf} \qquad \Phi_r = \exists D_1 :: \mathbb{R}.\Phi_1 \wedge (\exists D_2 :: \mathbb{R}.\Phi)\end{array}}{\Sigma; \Delta; \psi_a; \Phi_a; \Gamma \vdash \mathsf{alloc}_\mathsf{b}\ t_1\ t_2 \ominus \mathsf{alloc}_\mathsf{b}\ t_1'\ t_2' \downarrow \ \{P\}\ \exists \gamma.\ \mathsf{Array}_\gamma[I]\ \tau\ \{P \star \gamma \rightarrow \emptyset\}, 0 \overset{\mathsf{diff}(D)}{\Rightarrow} \Phi_r}$$

alg-r-read-↓

$$\frac{\begin{array}{c}\Delta; \psi_a; \Phi_a; \Gamma \vdash t_1 \ominus t_1' \uparrow \mathsf{Array}_\gamma[I]\ \tau \Rightarrow [\psi_1], D_1, \Phi_1 \\ \Delta; \psi_1, \psi_a; \Phi_a; \Gamma \vdash t_2 \ominus t_2' \uparrow \mathsf{int}[I'] \Rightarrow [\psi_2], D_2, \Phi_2 \qquad P = P' \star \gamma \rightarrow \_ \\ \Delta; \psi_a; \Phi_a \vDash I' \leq I \qquad \Phi = \Phi_2 \wedge D_1 + D_2 \doteq D \qquad \Sigma; \Delta \vdash P \text{ wf} \qquad \Phi_r = \exists(\psi_1).\Phi_1 \wedge (\exists(\psi_2).\Phi)\end{array}}{\Delta; \psi_a; \Phi_a; \Gamma \vdash \mathsf{read}\ t_1\ t_2 \ominus \mathsf{read}\ t_1'\ t_2' \downarrow \ \{P\}\ \exists \_.\tau\ \{P\}, 0 \overset{\mathsf{diff}(D)}{\Rightarrow} \Phi_r}$$

alg-r-readB-↓

$$\frac{\begin{array}{c}\Delta; \psi_a; \Phi_a; \Gamma \vdash t_1 \ominus t_1' \uparrow \mathsf{Array}_\gamma[I]\ \tau \Rightarrow [\psi_1], D_1, \Phi_1 \qquad P = P' \star \gamma \rightarrow \beta \\ \Delta; \psi_1, \psi_a; \Phi_a; \Gamma \vdash t_2 \ominus t_2' \uparrow \mathsf{int}[I'] \Rightarrow [\psi_2], D_2, \Phi_2 \qquad \Delta; \psi_a; \Phi_a \vDash I' \leq I \wedge \neg(I' \in \beta) \\ \Phi = \Phi_2 \wedge D_1 + D_2 \doteq D \qquad \Sigma; \Delta \vdash P \text{ wf} \qquad \Phi_r = \exists(\psi_1).\Phi_1 \wedge (\exists(\psi_2).\Phi)\end{array}}{\Delta; \psi_a; \Phi_a; \Gamma \vdash \mathsf{read}_\mathsf{b}\ t_1\ t_2 \ominus \mathsf{read}_\mathsf{b}\ t_1'\ t_2' \downarrow \ \{P\}\ \exists \_.\square\tau\ \{P\}, 0 \overset{\mathsf{diff}(D)}{\Rightarrow} \Phi_r}$$

alg-r-updt-↓

$$\frac{\begin{array}{c}\Delta; \psi_a; \Phi_a; \Gamma \vdash t_1 \ominus t_1' \uparrow \mathsf{Array}_\gamma[I]\ \tau \Rightarrow [\psi_1], D_1, \Phi_1 \qquad D_3 \in \mathsf{fresh}(\mathbb{R}) \\ \Delta; \psi_1, \psi_a; \Phi_a; \Gamma \vdash t_2 \ominus t_2' \uparrow \mathsf{int}[I'] \Rightarrow [\psi_2], D_2, \Phi_2 \qquad \Delta; \psi_a; \Phi_a \vDash I' \leq I \wedge \beta' = \beta \cup \{I'\} \\ \Delta; D_3, \psi_2, \psi_1, \psi_a; \Phi_a; \Gamma \vdash t_3 \ominus t_3' \downarrow \tau, D_3 \Rightarrow \Phi_3 \qquad P = P' \star \gamma \rightarrow \beta \qquad Q = P' \star \gamma \rightarrow \beta' \\ \Phi = \Phi_2 \wedge D_1 + D_2 + D_3 \doteq D \qquad \Sigma; \Delta \vdash P' \text{ wf} \qquad \Phi_r = \exists(\psi_1).(\Phi_1 \wedge (\exists(\psi_2).(\Phi_2 \wedge \exists D_3 :: \mathbb{R}.\Phi))\end{array}}{\Delta; \psi_a; \Phi_a; \Gamma \vdash \mathsf{updt}\ t_1\ t_2\ t_3 \ominus \mathsf{updt}\ t_1'\ t_2'\ t_3' \downarrow \ \{P\}\ \exists \_.\mathsf{unit}\ \{Q\}, 0 \overset{\mathsf{diff}(D)}{\Rightarrow} \Phi_r}$$

alg-r-updtB-↓

$$\frac{\begin{array}{c}\Delta; \psi_a; \Phi_a; \Gamma \vdash t_1 \ominus t_1' \uparrow \mathsf{Array}_\gamma[I]\ \tau \Rightarrow [\psi_1], D_1, \Phi_1 \qquad D_3 \in \mathsf{fresh}(\mathbb{R}) \\ \Delta; \psi_1, \psi_a; \Phi_a; \Gamma \vdash t_2 \ominus t_2' \uparrow \mathsf{int}[I'] \Rightarrow [\psi_2], D_2, \Phi_2 \qquad \Delta; \psi_a; \Phi_a \vDash I' \leq I \wedge \beta' = \beta \setminus \{I'\} \\ \Delta; D_3, \psi_2, \psi_1, \psi_a; \Phi_a; \Gamma \vdash t_3 \ominus t_3' \downarrow \square\tau, D_3 \Rightarrow \Phi_3 \qquad P = P' \star \gamma \rightarrow \beta \qquad Q = P' \star \gamma \rightarrow \beta' \\ \Phi = \Phi_2 \wedge D_1 + D_2 + D_3 \doteq D \qquad \Sigma; \Delta \vdash P' \text{ wf} \qquad \Phi_r = \exists(\psi_1).(\Phi_1 \wedge (\exists(\psi_2).(\Phi_2 \wedge \exists D_3 :: \mathbb{R}.\Phi))\end{array}}{\Delta; \psi_a; \Phi_a; \Gamma \vdash \mathsf{updt}_\mathsf{b}\ t_1\ t_2\ t_3 \ominus \mathsf{updt}_\mathsf{b}\ t_1'\ t_2'\ t_3' \downarrow \ \{P\}\ \exists \_.\mathsf{unit}\ \{Q\}, 0 \overset{\mathsf{diff}(D)}{\Rightarrow} \Phi_r}$$

Fig. 11. Selection of algorithmic typing rules for array operations

rules for array operations in Figure 11. As mentioned, to resolve the non-determinism between the $\square$-ed and non-$\square$-ed rules for each array operation, we use distinct expressions, e.g., $\mathsf{alloc}_\mathsf{b}\ t_1\ t_2$ vs $\mathsf{alloc}\ t_1\ t_2$. To start understanding the rules, note that the conclusion of every array operation is typed in checking mode. The two allocation rules **alg-r-alc-↓** and **alg-r-alcB-↓** check the first

arguments t_1 and t_1' against the relational type int$[I]$ and relative cost D_1, then check the second arguments t_2 and t_2' against the relational type τ (or $\Box\tau$) and relative cost D_2. The final constraint $\Phi_r = \exists D_1 :: \mathbb{R}.\Phi_1 \wedge (\exists D_2 :: \mathbb{R}.\Phi)$ requires that there exist D_1 and D_2 such that Φ_1 and Φ_2 hold and that $D_1 + D_2$ equals the given cost D. The algorithmic typing rules for read and updt have other interesting aspects. These rules are in checking mode but the types of the first two arguments are inferred, not checked. This is because, although we know that the first argument of read $t_1\ t_2$ or $\text{updt}_\text{b}\ t_1\ t_2\ t_3$ must be an array and the second argument must be a number, we do not know the size of the array or the size (refinement index) of the number. Hence, we must infer this information. Additionally, these rules make checks on the pre- and post-conditions. As an example, the condition $\neg(I' \in \beta)$ is checked on the rule **alg-r-readB-↓** to guarantee that we are indeed reading the same element on the two sides. Similarly, in the rules **alg-r-updt-↓** and **alg-r-updtB-↓**, the β' in the post-condition, representing the differences between the two arrays, must be the same as the β in the pre-condition except for the index I' which has been updated. For this, in the rule **alg-r-updt-↓** we check that $\beta' = \beta \cup \{I'\}$, while in the rule **alg-r-updtB-↓** we check that $\beta' = \beta \setminus \{I'\}$, consistent with the (non-algorithmic) typing rules.

7 IMPLEMENTATION AND EXPERIMENTS

We implemented the bidirectional type checking system for ARel described in Section 6. Using this implementation, we checked all the examples described in this paper as well as some others that are described in the appendix. We explain the results of our experiments in this section. One small difference between the system described in Section 6 and our implementation is that rather than support two syntactic variants of every array operation, we use heuristics to infer whether to apply the $\Box$-ed rule or the non-$\Box$-ed rule. For example, to decide to apply the rule **alg-r-readB-↓** as opposed to **alg-r-read-↓**, we check that $I \notin \beta$. We always try the $\Box$-ed rules first. These heuristics suffice for our examples and reduce our annotation burden at the cost of some extra constraint solving time. Our typechecker is implemented in OCaml and we plan to open source it.

Constraint Solving. The primary difficulty in our implementation and the most time-consuming step in type checking is solving the constraints that the bidirectional type system generates. For this we rely on an SMT solver. Specifically, we use Alt-Ergo [Bobot et al. 2013] through the Why3 frontend [Filliâtre and Paskevich 2013]. A fundamental difficulty here is that the SMT solver struggles with constraints that have too many existential quantifiers. To alleviate this concern, we rely on a solution proposed in the implementation of RelCost [Çiçek et al. 2019]: We implement a simple algorithm that generates candidate substitutions for existentially quantified variables by examining equality and inequality constraints that mention the variables. This works remarkably well (we refer to [Çiçek et al. 2019] for details). A new challenge for ARel is how to represent and solve constraints involving the sets β. For this, we rely on the library for set theory from Why3.

Experiments. Table 1 summarizes some statistics about the performance of our type checker on different examples. For each example, we show the number of lines of code (LOC), the number of type annotations that are needed (#TYP), the number of annotations needed to disambiguate rules (#ESF), the time needed for type checking (TC), the time needed for solving the constraints that arise as premises during type checking (TC-SMT), and the time needed for solving the final constraint which is the output of the type checking (TF-SMT). Our experiments were performed on a 3.1 GHz Intel Core i5 processor with 8GB of RAM.

The programs map(1), map(2), boolOr, FFT, NSS and ISort are implementations of the corresponding examples discussed in Section 2 and Section 5. For FFT, which uses the auxiliary functions separate and loop, we report statistics for the whole program and individually for each auxiliary function. The program ISort uses helper functions insert and shift. These are also shown separately.

The programs merge(1) and merge(2) consider an imperative versions of merge sort, typed with two different relational types. The function SAM (square-and-multiply) computes a positive power of a number represented as an array of bits, while comp checks the equality of two passwords represented as arrays of bits. These last two examples are array-based implementations of similar list-based implementations presented in Çiçek et al. [2017]. More details are in the appendix.

Table 1. Summary of experimental results

Benchmark	LOC	#TYP	#ESF	TC	TC-SMT	TF-SMT
map(1)	19	3	0	0.802s	1.051s	0.01s
map(2)	12	2	1	1.247s	0.994s	0.02s
boolOr	48	8	3	1.574s	1.131	2.38s
separate	36	8	0	1.351s	2.148s	0.01s
loop	23	5	0	1.167s	2.114s	0.01s
FFT	66	17	0	2.591s	4.268s	0.01s
Search	62	10	3	3.753s	4.43s	6.56s
NSS	94	12	3	4.158s	4.413s	10.03s
shift	14	3	0	0.660s	1.394s	0.01s
insert	22	6	0	1.001s	3.019s	0.01s
iSort	134	12	3	2.897s	6.181s	10.70s
merge(1)	29	8	0	2.203s	2.232s	0.01s
merge(2)	64	11	2	3.231s	0.349s	0.02s
sam	19	4	1	0.946s	0.083s	0.02s
comp	20	3	0	1.138s	0.112s	0.01s

The results in Table 1 show that ARel can be used effectively to reason about the relative cost of functional-imperative programs. Unsurprisingly, examples combining relational and unary reasoning (using rules R-FIX-EXT and R-S) such as boolOr, NSS and ISort need more annotations and need more time for both type checking and SMT solving. In some examples like ISort, TC-SMT, the time taken for solving constraints in the premises of the rules is very high. This is because of the heuristic we described at the beginning of this section where we try □-ed rules before non-□-ed rules. The SMT solver first tries to prove that the □-ed rule can be applied, but in some cases it *times out*. This timeout period is counted in TC-SMT. It is set to 1s in all examples, except ISort and Insert, where we try for 2s. TF-SMT, the time taken to check the final output constraint, is also high for some examples like ISort, but this is due to the complexity of the constraint. Further improving our heuristics and the constraint solving process remains a direction for future work.

8 RELATED WORK

A lot of prior work has studied static cost analysis. We discuss some of this work here. Reistad and Gifford [1994] present a type and effect system for cost analysis where, like ARel, the cost can depend on the size of the input. Danielsson [2008] uses a cost-annotated monad similar in spirit to the one we use here. Dal Lago and Gaboardi [2011] present a linear dependent type system using index terms to analyze time complexity. Hoffmann et al. [2012a] present an automated amortized cost analysis for programs with complex data structures such as matrices. Wang et al. [2017] develop a type system for cost analysis with time complexity annotations in types. However, none of these systems consider relational analysis of costs.

Charguéraud and Pottier [2015] present an amortized resource analysis based on an extension of separation logic with time credits. Our use of triples and separation-based management of arrays

references is similar to theirs. However, their technique is based on separation logic, while ours is based on a type-and-effect system. Moreover, they consider only unary reasoning while we are interested primarily in relational reasoning. Lichtman and Hoffmann [2017] present an amortized resource analysis for arrays and reference based on arrays with potentials. Their technique represents the available "potential" before and after a computation, similar to our triples. Again, they focus only on unary cost analysis and, consider mostly first-order programs and linear potentials.

Outside of cost analysis, a lot of work has considered relational verification techniques for other applications. Lahiri et al. [2010] present a differential static analysis to find code defects looking at two pieces of code relationally. Probabilistic relational verification has seen many applications in cryptography [Barthe et al. 2014] and differential privacy [Barthe et al. 2015; Gaboardi et al. 2013]. The indexed types used by Gaboardi et al. [2013] are similar in spirit to ours. Zhang et al. [2015] introduce dependent labels into the type of SecVerilog, an extension of Verilog with information flow control. The use of a lightweight invariant on variables and security levels in SecVerilog is similar to our use of β, which is also an invariant on static location variables. Unno et al. [2017] present an automated approach to verification based on induction for Horn clauses, which can also be used for relational verification. Benton et al. [Benton et al. 2014, 2016] introduce abstract effects to reason about abstract locations. This is conceptually similar to the way our preconditions and postconditions allow us to reason about different independent locations.

Our work is inspired by RelCost [Çiçek et al. 2017] and DuCostIt [Çiçek et al. 2016]. These are refinement type and effect systems for pure functional languages *without mutable state*. RelCost supports relational cost analysis of pure programs. In contrast, ARel supports imperative arrays. The difference is substantial: Besides significant changes to the model, the type system has to be enriched with Hoare-like triples, whose design is a key contribution of our work. RelCost has an implementation via an SMT back-end [Çiçek et al. 2019]; we extend this approach with imperative features and support for sets of indices (our βs). Ngo et al. [2017] combine information flow and amortized resource analysis to guarantee constant-resource implementations. Their type system allows relational reasoning about resources through precise unary analysis. Their focus is on first-order functional programs and on the constant time guarantee, while we want to support functional-imperative programs and more general relative costs. Radicek et al. [2018] add a cost monad to a relational refinement type system, where refinements reason about relational cost, for programs without state. This system is expressive: it supports a combination of cost analysis with value-sensitivity and full functional specifications (RelCost can also be embedded in it). However, it requires a framework for full functional verification. Our approach is complementary in that we use lighter refinements that are easier to implement, but do not support full functional verification.

9 CONCLUSION

We presented ARel, a relational type-and-effect system that can be used to reason about the relative cost of functional-imperative programs with mutable arrays. Our key contribution is a set of lightweight relational refinements allowing one to establish different relations between pairs of state-affecting computations, including upper-bounds on cost difference. We have discussed how ARel is implemented and used ARel to reason about the relational cost of several nontrivial examples.

ACKNOWLEDGMENTS

This work is in part supported by the National Science Foundation under Grant No. 1718220.

REFERENCES

Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*.

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

Amal G Ahmed. 2004. Semantics of types for mutable state.

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (2001), 657–683.

Robert Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*.

Martin Avanzini and Ugo Dal Lago. 2017. Automating sized type inference for complexity analysis. In *Proceedings of DICE-FOPARA*.

Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella Béguelin. 2014. Probabilistic relational verification for cryptographic implementations. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations.. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

Nick Benton, Martin Hofmann, and Vivek Nigam. 2014. Abstract effects and proof-relevant logical relations. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

Nick Benton, Martin Hofmann, and Vivek Nigam. 2016. Effect-dependent transformations for concurrent programs. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*.

François Bobot, Sylvain Conchon, E Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. 2013. The Alt-Ergo automated theorem prover, 2008.

Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2014. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Alg. for the Constr. and Anal. of Systems - 20th Int. Conf. (TACAS)*.

Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional Certified Resource Bounds. In *Proceedings of the 36th Conference on Programming Language Design and Implementation (PLDI)*.

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

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*.

Ezgi Çiçek, Zoe Paraskevopoulou, and Deepak Garg. 2016. A Type Theory for Incremental Computational Complexity With Control Flow Changes. In *Proceedings of the International Conference on Functional Programming(ICFP)*.

Ezgi Çiçek, Weihao Qu, Gilles Barthe, Marco Gaboardi, and Deepak Garg. 2019. Bidirectional type checking for relational properties. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. 533–547.

James W. Cooley and John W. Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.* (1965).

Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *Proceedings of the IEEE 26th Annual Symposium on Logic in Computer Science (LICS)*.

Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3: Where Programs Meet Provers. In *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*.

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear Dependent Types for Differential Privacy. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

Bernd Grobauer. 2001. Cost recurrences for DML programs. In *Proceedings of the 6th International Conference on Functional Programming (ICFP)*.

M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming* 58, 1–2 (October 2005), 115–140.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012a. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.* (2012).

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012b. Resource Aware ML. In *Computer Aided Verification - 24th International Conference, CAV*.

Shuvendu K. Lahiri, Kapil Vaswani, and C. A. R. Hoare. 2010. Differential static analysis: opportunities, applications, and challenges. In *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Gruia-Catalin Roman and Kevin J. Sullivan (Eds.).

Benjamin Lichtman and Jan Hoffmann. 2017. Arrays and References in Resource Aware ML. In *The 2nd International Conference on Formal Structures for Computation and Deduction, FSCD*.

Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2013. Dependent Type Theory for Verification of Information Flow and Access Control Policies. *ACM Trans. Program. Lang. Syst.* 35, 2 (2013).

Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. 2008. Hoare type theory, polymorphism and separation. *J. Funct. Program.* 18, 5-6 (2008).

Georg Neis, Derek Dreyer, and Andreas Rossberg. 2011. Non-parametric parametricity. *J. Funct. Program.* 21, 4-5 (2011), 497–562.

Van Chan Ngo, Mario Dehesa-Azuara, Matt Fredrikson, and Jan Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *2017 IEEE Symposium on Security & Privacy*.

Flemming Nielson and HanneRiis Nielson. 1999. Type and Effect Systems. In *Correct System Design*. Lecture Notes in Computer Science, Vol. 1710. 114–136.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44.

Ivan Radicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2018. Monadic refinements for relational cost analysis. *PACMPL* 2, POPL (2018).

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 '94)*. 65–78.

Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th International Conference, CAV*.

Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. 343–356.

Hiroshi Unno, Sho Torii, and Hiroki Sakamoto. 2017. Automating Induction for Solving Horn Clauses. In *Computer Aided Verification - 29th International Conference, CAV*.

Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*.