JFP **31**, e27, 62 pages, 2021.

doi:10.1017/S0956796821000071

Relational cost analysis in a functional-imperative setting

WEIHAO QU
Boston University, Computer Science Department, Boston, MA 02215, USA
(*e-mail:* weihaoqu@bu.edu)

MARCO GABOARDI
Boston University, Computer Science Department, Boston, MA 02215, USA
(*e-mail:* gaboardi@bu.edu)

DEEPAK GARG
Max Planck Institute for Software Systems, Saarbrücken, Germany
(*e-mail:* dg@mpi-sws.org)

Abstract

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 (the difference in the evaluation 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 that imperatively update their data. We have implemented ARel using ideas from bidirectional type checking.

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, for example, 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 (Grobauer, 2001; Danielsson, 2008; Dal Lago & Gaboardi, 2011; Hoffmann *et al.*, 2012b; Avanzini & Dal Lago, 2017), term rewriting and abstract interpretation (Hermenegildo *et al.*, 2005; Sinn *et al.*, 2014; Brockschmidt *et al.*, 2014), and Hoare logics (Atkey, 2010; Carbonneaux *et al.*, 2015; Charguéraud & 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 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. For example, consider a programmer who would like to update a piece of code of a distributed system, where the cost is local memory and this resource is limited. A unary cost analysis of the overall system may be impractical, while it may be easy to perform an analysis of the local difference memory consumption between the original piece of code and the updated one. 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 relational analysis as shown by Çiçek *et al.* (2017), but two separate unary analyses can only establish the coarser relative cost $O(n \cdot \log(n))$.

Hitherto, the 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 propose 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 side effects 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 in the relational setting, we also need to express relations between corresponding arrays across the two runs of the two programs. For this reason, 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 precondition P, end with arrays related by the relational postcondition 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 by relational Hoare logics (Benton, 2004; Nanevski *et al.*, 2013), but there are two key differences: (1) Our pre- and postconditions are *minimal*—they only specify the indices at which a pair of corresponding 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 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. Specifically, ARel provides typing rules for the fix-point operator that allow one to *simultaneously* reason about the cost in the unary and relational setting. This is useful for the analysis of several programs that we show later.

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, 2004; Ahmed *et al.*, 2009). 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 & McAllester (2001). Owing to the simplicity of our pre- and postconditions, 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 implement a bidirectional type checker for ARel. Thanks to the simplified form of our pre- and postconditions, we can solve the constraints generated by the type checker using SMT solvers. The type checker also uses a restricted number of heuristics to address some of the non-determinism coming from the relational reasoning, and the array operations. In order to evaluate the performance of our implementation, we consider a broad set of examples showcasing different challenges for 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 as follows:

- 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.

[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 $-r$.

- 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.

Improvement with respect to the conference version. This paper is an extended version of a paper published at the ICFP 2019 conference. The main additions to the conference version are as follows:

- A comprehensive presentation of the logical relations (Section 4) used to prove soundness, along with a full definition of the type interpretation. Additionally, representative cases of the proof of the fundamental theorem (Section 4.3) are included.
- Two new examples (Section 5). The first one is Mergesort. This example further illustrates the expressiveness of ARel. The second example is *Loop unswitching*, a common technique used in compiler optimization. This example aims at giving our readers insights about how ARel handles two programs that are not structurally similar.
- A comprehensive presentation of the algorithmic version of ARel (Section 6). This section briefly introduces bidirectional type checking along with the difficulties of directly applying this technique to ARel. This motivates the introduction of a core language, ARelCore, as a theoretical midway point, for the algorithmization of ARel (Section 6.1). The concrete algorithmic type system BiARel works on the core language (Section 6.3). This section also shows the soundness and completeness of ARelCore with respect to BiARel and the soundness and completeness of the elaboration from ARel to ARelCore (Section 6.2).
- An annotated example mapi used to show how the type checker works and the extent of the required annotations (Section 7.3). This section also include an in-depth discussion of the limitations of our implementation and directions for improvement.
- The code of the type checker as well as the examples used in Section 7 have been released publicly at `https://github.com/haddyclipk/ICFP2019_BiArel`. An appendix with full proofs is also available in the repository.

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, named mapi, 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.

$$
\begin{array}{l}
\mathsf{fix}\ \mathsf{mapi}\ (f).\lambda a.\lambda k.\lambda n. \\
\quad \mathsf{if}\ k \leq n\ \mathsf{then} \\
\qquad \mathsf{let}\ \{x\} = \mathsf{read}\ a\ k\ \ \mathsf{in} \\
\qquad \mathsf{let}\ \{\_\} = \mathsf{updt}\ a\ k\ (f\ x)\ \ \mathsf{in} \\
\qquad \mathsf{mapi}\ f\ a\ (k+1)\ n \\
\quad \mathsf{else} \\
\qquad \mathsf{return}\ ()
\end{array}
$$

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 mapi that use the *same* function f but two *different* arrays a. Intuitively, the relative cost should be upper bounded by the product of the maximal 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 arguments 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. Naturally, ARel also supports *unary types*, denoted A, that ascribe only one 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 mapi, we first need a way to represent that the *same* function f will be given to mapi in both runs. To this end, 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 & 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 $\tau_1 \xrightarrow{\mathrm{diff}(r)} \tau_2$ 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 $\Box(\tau \xrightarrow{\mathrm{diff}(r)} \tau)$.

Next, we need a way to specify *where* the arrays given as inputs to mapi 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 computations in two runs, similar to the one we have in function types, and τ is the relational type relating the pair of *pure* values returned by the computations. The *precondition* P specifies for each pair of related arrays in scope where (at which indices) these corresponding arrays are allowed to differ *before* the execution of the computations, while the *postcondition* Q specifies where these arrays may differ *after* the completion of the computations. More specifically, P and Q are lists of annotations of the form $\gamma \to \beta$, where γ is a *static name* identifying a pair of related arrays and β is a set of indices where this pair of arrays *may* differ in the two runs. In other words, at any index not in β, the corresponding arrays 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 $\{P\}\ \exists\vec{\gamma}.\tau\ \{Q\}$ is the list of static names of arrays that are allocated during the computation.)

To be more concrete, let us consider an example: If $x : \Box\tau$ (i.e., x is the same in two runs) and b represents a pair of arrays associated with the static name γ, then two equal update operations $(\mathsf{updt}\ b\ 5\ x)$ can be given the relational monadic type $\{\gamma \to \beta\}\ \exists\_.\mathrm{unit}\ \{\gamma \to (\beta \setminus \{5\})\}$, for any β.[2] This type means that if the two corresponding arrays b differ at most in the set of indices β before $\mathsf{updt}\ b\ 5\ x$ executes, then afterwards the arrays can still differ in the indices β *except* at the index 5, which has been overwritten by the same value x (indicated by the box type $\Box\tau$). If we replace the assumption $x : \Box\tau$ with $x : \tau$, so that x may differ in the two runs, then the type of $\mathsf{updt}\ b\ 5\ x$ relative to itself would be $\{\gamma \to \beta\}\ \exists\_.\mathrm{unit}\ \{\gamma \to (\beta \cup \{5\})\}$, indicating that the arrays may differ at index 5 after the update (even if they did not differ at that index before the update).

In order to make this reasoning formal, we need a way to tie the static names γ appearing in computation types to specific arrays. To this end, 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 $\mathsf{Array}_\gamma[l]\ \tau$—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 $\mathrm{int}[n]$ is the *singleton type* containing only the integer n in both runs. The n in the type is a static representation of the runtime value the type ascribes.

With all these components we can now represent the relative cost of mapi by the following judgment:

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

This typing means that mapi relates to itself in the following way. Consider two runs of mapi with the same function f of relative cost r (type $\Box(\tau \xrightarrow{\mathrm{diff}(r)} \tau)$), two arrays of static length n, statically named γ (type $\mathsf{Array}_\gamma[n]\ \tau$), two indices, both k (type $\mathrm{int}[k]$), and two lengths, both n (type $\mathrm{int}[n]$). 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

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

mapi, then they differ at most at the same positions after the computations, and the relative cost of the two computations is upper bounded by $|\beta \cap [k,n]| * r$, that is, 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 postcondition of the computation type.

As is usual in effect systems, when we apply mapi, we have two kinds of costs. For example, suppose that we provide arguments f, a, k, and n. Then, we have some cost D such that:

$$\vdash \mathsf{mapi}\ f\ a\ k\ n \ominus \mathsf{mapi}\ f\ a\ k\ n \lesssim D : \{\gamma \to \beta\}\ \overset{\mathrm{diff}(|\beta\cap[k,n]|*r)}{\exists}\_.\mathsf{unit}\ \{\gamma \to \beta\}$$

Here, we can think of D as a bound on the relative cost of the computation before we get to the array evaluation, while $|\beta \cap [k,n]| * r$ is a bound on the relative cost bounds for the part of the computation involving arrays.

Consider now a slightly different situation where *different* functions f may be passed to mapi in the two runs. Suppose that the relative cost of the bodies of the two fs passed is upper bounded by r, that is, 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 mapi 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 for mapi.

$$\begin{aligned}\vdash \mathsf{mapi} \ominus \mathsf{mapi} \lesssim 0 : {} & \forall r :: \mathbb{R}.(\tau \overset{\mathrm{diff}(r)}{\longrightarrow} \tau) \to \forall k, n, \gamma, \beta.(k \le n) \supset \\ & \mathsf{Array}_\gamma[n]\ \tau \to \mathsf{int}[k] \to \mathsf{int}[n] \to \{\gamma \to \beta\}\ \overset{\mathrm{diff}((n-k)*r)}{\exists}\_.\mathsf{unit}\ \{\gamma \to \beta \cup [k,n]\}\end{aligned}$$

BooleanOr. 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 imperative 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 via an extended fix rule **r-fix-ext**.

```
fix BoolOr (a). λk.λn.
  if k < n then
    let {x} = read a k in
    if x then
      return true
    else
      BoolOr a (k + 1) n
  else
    return false
```

This function, given as input an array of booleans a, an index k and the array's length n says 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-switch, 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, for example, they may be equal in some positions. In such cases, a better upper bound on the relative cost would be expressed in terms 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-split, presented in Section 3. Using this rule we can consider the two cases separately in typing the subexpression: if x then return *true* else 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 the 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 more precise analysis, ARel includes a new rule r-fix-ext which we introduce formally in Section 3. This rule allows us to assume *unary* typing of two recursive functions, when typing their bodies *relationally*. With this rule, we can use the (assumed) relational type of BoolOr and its unary type in typing the subexpression BoolOr $a\ (k+1)\ n$. With this, we can conclude the inductive step and assign the precise relative cost $(n-i)*c$ to BoolOr.

3 ARel formally

In this section, we present the syntax, semantics, and the type system of ARel. We can think about ARel as composed of two parts, a *pure* part, inspired by Çiçek *et al.* (2017), which allows one to reason about the difference in the execution costs of two pure programs and

an *impure* part, which allows one to reason about the difference in the execution costs of two programs involving array operations. In our paper, we will mostly focus on the impure part, giving details of the pure part as needed.

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 the pure constructs (the ones in black in Figure 1) are similar to the ones one can find in a pure standard functional language. We have variables x, natural numbers n, and real numbers r, unit (), lambda abstraction $\lambda x.t$, and recursion $\mathsf{fix}\, f(x).t$. and application $t_1\ t_2$. We have the introduction and elimination constructs for product, sum, and inductive list. Additionally, we have some constructs to deal with type level information. We have term constructs $\Lambda.t$ and $t\ []$, $\mathsf{pack}\ t$ and $\mathsf{unpack}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2$ corresponding to the introduction and elimination of universal and existential types over index terms, respectively, and a term construct $\mathsf{celim}\ t$, which is used to eliminate type-level constraints. We discuss these constructs further when we introduce types.

The impure part at the term level consists of constructs to deal with arrays, which we highlight with blue underlines in Figure 1. We have constructs for *allocating* arrays ($\mathsf{alloc}\ t_1\ t_2$, 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 ($\mathsf{read}\ t_1\ t_2$, where t_1 specifies the array to read from, and t_2 the position in the array to read from), and for *updating* arrays ($\mathsf{updt}\ t_1\ t_2\ t_3$, 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 imperative (array-manipulating) constructs are confined to a monad. The constructs $\mathsf{return}\ t$ and $\mathsf{let}\ \{x\} = t_1\ \mathsf{in}\ t_2$ are the usual return and bind of the monad. We do not distinguish between impure expressions and pure expressions at the syntactic level; this distinction is enforced by types. 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, $l \in Loc$, where Loc is a fixed set of heap locations. Although locations do not appear in programs, they are needed for the evaluation, so they are included in the syntax.

Types can contain index terms. We use $iVar$ to denote the set of index term variables, and $iLoc$ to denote the set of index term variables that refer to locations statically. These static identifiers for locations are denoted γ and belong to a specific sort written $\mathbb{L}$. We discuss index terms in detail in Section 3.3.

3.2 Operational semantics

We define a cost-annotated, big-step operational semantics for our language. Part of this semantics is based on manipulation of heaps, also described in Figure 1. 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 an update to the heap H: The array pointed by l in H is updated with the value v

Index terms	$I, L,$ $U, D,$ $\underline{\alpha}, \underline{\beta}$	::=	$i \mid b \mid n \mid r \mid I_1 + I_2 \mid I_1 - I_2 \mid I_1 \cdot I_2 \mid \lceil I \rceil \mid \lfloor I \rfloor \mid log_2 I \mid I_1^{I_2} \mid \mathsf{min}(I_1, I_2) \mid \mathsf{max}(I_1, I_2) \mid \frac{I_1}{I_2} \mid \sum_{i=I_1}^{I_n} I \mid \underline{\{I_i\}_{i \in K} \mid \beta \cup \beta \mid \beta \setminus \beta \mid \beta \cap \beta}$
Terms	t	::=	$x \mid n \mid r \mid () \mid \lambda x.t \mid \mathsf{fix}\ f(x).t \mid t_1\, t_2 \mid \langle t_1, t_2 \rangle \mid \pi_1(t) \mid \pi_2(t) \mid \mathsf{case}\ (t, x.t_1, y.t_2) \mid \mathsf{inl}\ t \mid \mathsf{inr}\ t \mid \mathsf{nil} \mid \mathsf{cons}(t_1, t_2) \mid \Lambda.t \mid t\ [] \mid (\ \mathsf{case}\ t\ \mathsf{of\ nil} \to t_1 \mid h :: tl \to t_2) \mid \mathsf{pack}\ t \mid \mathsf{celim}\ t \mid \mathsf{unpack}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 \mid \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \mid \underline{\mathsf{return}\ t \mid \mathsf{alloc}\ t_1\ t_2 \mid \mathsf{updt}\ t_1\ t_2\ t_3 \mid \mathsf{read}\ t_1\ t_2 \mid \mathsf{let}\ \{x\} = t_1\ \mathsf{in}\ t_2}$
Values	v	::=	$n \mid l \mid r \mid () \mid \lambda x.t \mid \mathsf{fix}\ f(x).t \mid \langle v_1, v_2 \rangle \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \mathsf{nil} \mid \Lambda.t \mid \mathsf{pack}\ v \mid \underline{\mathsf{return}\ t \mid \mathsf{alloc}\ t_1\ t_2 \mid \mathsf{updt}\ t_1\ t_2\ t_3 \mid \mathsf{read}\ t_1\ t_2 \mid \mathsf{let}\ \{x\} = t_1\ \mathsf{in}\ t_2}$
Unary types	A	::=	$\mathsf{unit} \mid \mathsf{int} \mid \overset{\mathsf{exec}(L,U)}{\forall i{::}S.\ A} \mid \exists i :: S.\ A \mid \overset{\mathsf{exec}(L,U)}{A \longrightarrow A} \mid \mathsf{list}[I]\ A \mid A_1 \times A_2 \mid A_1 + A_2 \mid C \& A \mid C \supset A \mid \mathsf{int}[I] \mid \mid \underline{\mathsf{Array}_\gamma[I]\ A} \mid \underline{\overset{\mathsf{exec}(L,U)}{\{P\}\ \exists \vec{\gamma}.A\ \{Q\}}}$
Relat. types	τ	::=	$\mathsf{unit} \mid \mathsf{int} \mid \overset{\mathsf{diff}(D)}{\forall i{::}S.\ \tau} \mid \exists i{::}S.\ \tau \mid \overset{\mathsf{diff}(D)}{\tau \longrightarrow \tau} \mid \mathsf{list}^\alpha[I]\ \tau \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid C \& \tau \mid C \supset \tau \mid U(A_1, A_2) \mid \Box\tau \mid \mathsf{int}[I] \mid \underline{\overset{\mathsf{diff}(D)}{\{P\}\ \exists \vec{\gamma}.\tau\ \{Q\}} \mid \mathsf{Array}_\gamma[I]\ \tau}$
Constraints	C	::=	$I_1 = I_2 \mid I_1 < I_2 \mid \neg C \mid \underline{I_1 \in I_2}$
Heaps	$\underline{H}$	::=	$\underline{[] \mid [l \to z] \mid H_1 \uplus H_2}$
Arrays	$\underline{z}$	::=	$\underline{[v_1, \ldots, v_m]}$
Assertions	$\underline{P, Q}$	::=	$\underline{\mathsf{empty} \mid \gamma \to \beta \mid P \star Q}$
Sorts	S	::=	$\mathbb{R} \mid \mathbb{N} \mid \underline{\mathbb{B} \mid \mathbb{P} \mid \mathbb{L}}$

Loc Env	$\underline{\Sigma ::= \emptyset \mid \Sigma, \gamma :: \mathbb{L}}$	**Sort Env.**	$\Delta ::= \emptyset \mid \Delta, i :: S$
Unary Type Env.	$\Omega ::= \emptyset \mid \Omega, x : A$	**Relational Type Env.**	$\Gamma ::= \emptyset \mid \Gamma, x : \tau$
Constraint Env.	$\Phi ::= \top \mid C \wedge \Phi$		

Unary Typing Judgment	$\Sigma; \Delta; \Phi; \Omega \vdash^U_L t : A$
Relational Typing Judgment	$\Sigma; \Delta; \Phi; \Gamma \vdash t_1 \ominus t_2 \lesssim D : \tau$
Sorting Judgment	$\Sigma; \Delta \vdash I :: S$
Unary Subtyping	$\Sigma; \Delta; \Phi \models A_1 \sqsubseteq A_2$
Relational Subtyping	$\Sigma; \Delta; \Phi \models \tau_1 \sqsubseteq \tau_2$
Unary Type Well-formedness	$\Sigma; \Delta; \Phi \vdash A\ \mathsf{wf}$
Relational Type Well-formedness	$\Sigma; \Delta; \Phi \vdash \tau\ \mathsf{wf}$
Constraint Well-formedness	$\Delta \vdash C\ \mathsf{wf}$
Heap Well-formedness	$\underline{\Omega \vdash H\ \mathsf{wf}}$
Assertion Well-formedness	$\underline{\Sigma; \Delta \vdash P\ \mathsf{wf}}$

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

$$\frac{}{v \Downarrow^{0,0} v}\ \textbf{e-val} \qquad \frac{t \Downarrow^{c,k} v}{\mathsf{inl}\ t \Downarrow^{c,k} \mathsf{inl}\ v}\ \textbf{e-inl}$$

$$\frac{t \Downarrow^{c_1,k_1} \mathsf{inl}\ v \qquad t_1[v/x] \Downarrow^{c_2,k_2} v_r}{\mathsf{case}\ (t, x.t_1, y.t_2) \Downarrow^{c_1+c_2+c_{\mathsf{case}},k_1+k_2+1} v_r}\ \textbf{e-casel}$$

$$\frac{t_1 \Downarrow^{c_1,k_1} \lambda x.t' \quad t_2 \Downarrow^{c_2,k_2} v \quad t'[v/x] \Downarrow^{c_3,k_3} v_1}{t_1\, t_2 \Downarrow^{c_1+c_2+c_3+c_{\mathsf{app}},k_1+k_2+k_3+1} v_1}\ \textbf{e-app}$$

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

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

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

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

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

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

at index n. 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 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. We give a selection of the evaluation rules in Figure 2. We include all the rules for the impure part and a selection for the pure part, all the other rules can be found in the Appendix.

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. At every elimination form or monadic construct, the semantics adds a construct-dependent cost. For example, the cost c_{app} appearing in the rules stands for the cost of the function application operation. By changing these costs and setting some of them to 0, we can get different cost models. In other words, our type system is parametric in the costs of individual constructs.

The pure evaluation rules are mostly standard. They track how the cost and the steps change when a pure expression evaluates. The rule e-val says that a value v evaluates to itself with no cost and in zero steps. The rules e-inl and e-casel describe the evaluation of

the introduction and elimination terms for the sum type (we only show the rule for inl as the rule for inr is similar), where in addition we record the cost c_{case} for the case elimination, and we increment the steps. Rules a-app and e-fix are similar but for the two application cases, the latter one includes recursion.

The forcing evaluation rules are used to evaluate impure (monadic) expressions manipulating heaps (arrays). The rule f-ret forces the evaluation of an expression $\mathsf{return}\, t$, representing the unit of the monad, 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. As one would expect, the unit of the monad wraps a pure expression into a monadic computation, and it accounts for the cost of this operation by means of the cost c_{ret}. The rule f-bind combines pure and forcing evaluations in order to fully evaluate a monadic let. This rule shows how an impure computation (involving arrays) is evaluated in our semantics. We first evaluate an expression t_1 to a value v using the pure evaluation semantics. Then, we force evaluate this value v to a value v_1 of the underlying type. We can then perform the substitution and evaluate the resulting expression $t_2[v_1/x]$ to a value v_2 using the pure evaluation semantics. The resulting value v_2 is then force evaluated to another value v_3 which is also the result of the overall let expression. The heap also changes accordingly. c_{let} accounts for additional costs associated with the bind operation itself.

The rule f-read forces the evaluation of a read expression in the heap H by first evaluating the heap location l from which to read, then the index of the element n to read, and then returning the value stored in l at index n. The rule f-updt forces the evaluation of an update expression in a similar way; it returns a unit value. Finally, the rule f-alloc 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.

It is worth noticing that, in the forcing evaluation rules, all the subexpressions evaluate using the pure semantics to values of base types, since in our language we only allow arrays of base types.

3.3 Index terms and constraints

In the spirit of DML (Xi & Pfenning, 1999), types are indexed using *static* index terms that are defined in Figure 1. Index terms include natural numbers and real numbers, which we use to express size and cost information, respectively. We equip index terms with several operations including ceiling, floor, log, min, and max. Moreover, we have special index terms denoting (potentially infinite) sets of natural numbers, representing sets of array indexes, and operations over them (we identify these with blue underlines in Figure 1). We denote elements of this class with the letter 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 use index terms also in the relational type for lists to specify the number of

values that differ pointwise in the lists across two runs. We denote index terms with this specific meaning with the metavariable α.

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: $\mathbb{N}$ for natural numbers, used for sizes of arrays, lists, and other data structures; $\mathbb{R}$ for real numbers, used to express costs; $\mathbb{B}$ for booleans; $\mathbb{P}$ for sets of natural numbers as just described; and $\mathbb{L}$ for static names of arrays (sorting rules can be found in the Appendix). As a convention, we use L, U to represent the 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, which express equalities and inequalities over index terms and can are used to represent conditional typing.

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 grammars: *unary types* A and *relational types* τ.

Unary types describe values (expressions) in a single run. They use index terms to represent size information, as in the case of the type $\mathsf{list}[I]\,A$ where I represents the size of the list, and costs, as in the case of the type $A \overset{\mathrm{exec}(L,U)}{\longrightarrow} A'$ where L and U represent lower and upper bounds on the cost of the body of the function being typed. The cost can also be seen as an effect. We also have other basic types, as well as types for products and sums. Index terms are also used for size in basic types like integers, booleans, etc., for costs in universal quantifications, and in constraints.

Besides the pure types we just discussed, we have a type for arrays and a type for impure computations (with blue underlines in Figure 1). The type $\mathsf{Array}_\gamma[I]\,A$ is the type of arrays of length I containing objects of type A. We limit A to base types like int, bool, etc., to simplify our technical development. In particular, we do not support arrays of arrays here. 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 $\{P\}\ \exists\vec{\gamma}.A\ \{Q\}$, 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 describing (writing) 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 domain of Q may be larger than the domain of P, to account for newly allocated arrays. The index terms L and U are lower and upper bounds on the execution cost of the (forcing) evaluation of the typed expression.

Additionally, index terms are used in constraints which can appear in types of the shape $C \supset A$, which reads as "the constraint C implies A", and of the shape $C \& A$, which reads as "A and the constraint C is true". These constraints support conditional typing and they are quite useful to restrict the properties of arrays. For example, the type $I > 0\ \&\ \mathsf{Array}_\gamma[I]\,A$

ascribes non-empty arrays, and the constraint C in the type $C \supset A$ can be used to restrict array index bounds, as we will see in the examples in Section 5.

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

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 way to trivially relate 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, and we have corresponding subtyping rule s-r-T and s-r-D in Figure 9. In particular, $\Box U(A_1, A_2)$ is the diagonal sub-relation of $U(A_1, A_2)$, that is, $\Box U(A_1, A_2)$ is the subset of $U(A_1, A_2)$ where the left and right components are equal.

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. The static name γ is the name for both arrays. As we will see in Section 4, our logical relation relates γ to two arrays in two different heaps. Related impure computations, illustrated in the imperative 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 $\{P\}\ \exists\vec{\gamma}.A\ \{Q\}^{\mathrm{exec}(L,U)}$ but it means something different. In the relational type, the pre- and postconditions 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, e.g., with a prefix $\Box$). D is an upper bound on the relative cost of forcing the evaluation of 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. Well-formedness rules are provided in the Appendix.

3.5 Unary and relational typing

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

$$\Sigma; \Delta; \Phi; \Omega \vdash^{U}_{L} t : A$$

$$\frac{}{\Sigma;\Delta;\Phi;\Omega \vdash^{0}_{0} n : \mathrm{int}[n]}\ \textbf{u-int} \qquad \frac{\Sigma;\Delta;\Phi;x:A, f : A \overset{\mathrm{exec}(L,U)}{\longrightarrow} B,\Omega \vdash^{U}_{L} t : B}{\Sigma;\Delta;\Phi;\Omega \vdash^{0}_{0} \mathsf{fix}\ f(x).t : A \overset{\mathrm{exec}(L,U)}{\longrightarrow} B}\ \textbf{u-fix}$$

$$\frac{\Omega(x) = A}{\Sigma;\Delta;\Phi;\Omega \vdash^{0}_{0} x : A}\ \textbf{u-var} \qquad \frac{\Sigma;\Delta;\Phi;\Omega \vdash^{U_1}_{L_1} t_1 : A \overset{\mathrm{exec}(L,U)}{\longrightarrow} B \qquad \Sigma;\Delta;\Phi;\Omega \vdash^{U_2}_{L_2} t_2 : A}{\Sigma;\Delta;\Phi;\Omega \vdash^{U_1+U_2+U+U_{\mathsf{app}}}_{L_1+L_2+L+L_{\mathsf{app}}} t_1\, t_2 : B}\ \textbf{u-app}$$

$$\frac{\Sigma;\Delta;\Phi;\Omega \vdash^{U}_{L} t : A_1 \qquad \Sigma;\Delta;\Phi \vdash A_2\ \mathsf{wf}}{\Sigma;\Delta;\Phi;\Omega \vdash^{U}_{L} \mathrm{inl}\ t : A_1 + A_2}\ \textbf{u-inl} \qquad \frac{\Sigma;\Delta;\Phi;\Omega \vdash^{U}_{L} t : A_2 \qquad \Sigma;\Delta;\Phi \vdash A_1\ \mathsf{wf}}{\Sigma;\Delta;\Phi;\Omega \vdash^{U}_{L} \mathrm{inr}\ t : A_1 + A_2}\ \textbf{u-inr}$$

$$\frac{\begin{array}{c}\Sigma;\Delta;\Phi;\Omega \vdash^{U_1}_{L_1} t : A_1 + A_2 \\ \Sigma;\Delta;\Phi;x:\ A_1,\Omega \vdash^{U_2}_{L_2} t_1 : A \qquad \Sigma;\Delta;\Phi;y:\ A_2,\Omega \vdash^{U_2}_{L_2} t_2 : A\end{array}}{\Sigma;\Delta;\Phi;\Omega \vdash^{U_1+U_2+U_c}_{L_1+L_2+L_c} \mathrm{case}\ (t, x.t_1, y.t_2) : A}\ \textbf{u-case}$$

$$\frac{}{\Sigma;\Delta;\Phi;\Omega \vdash^{0}_{0} \mathsf{nil} : \mathsf{list}[0]\ A}\ \textbf{u-nil}$$

$$\frac{\Sigma;\Delta;\Phi;\Omega \vdash^{U_1}_{L_1} t_1 : A \qquad \Sigma;\Delta;\Phi;\Omega \vdash^{U_2}_{L_2} t_2 : \mathsf{list}[I]\ A}{\Sigma;\Delta;\Phi;\Omega \vdash^{U_1+U_2}_{L_1+L_2} \mathsf{cons}(t_1,t_2) : \mathsf{list}[I+1]\ A}\ \textbf{u-cons}$$

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

Fig. 3. Selection of pure unary typing rules.

where t is an expression, Σ is a location environment, Δ is a sort environment listing all the free index variables as mentioned before, 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 Figures 3 (for the pure constructs) and 4 (for the impure part).

We first show the rules for pure expressions in Figure 3. These rules are similar to the ones proposed by Çiçek *et al.* (2017). The main difference is that our rules have one more environment Σ, used to store the locations in the heap. Rules u-int and u-var are relatively self-explanatory. Rules u-fix and u-app are similar to the ones available in classical effect systems, where the lower bound and upper bound of the cost of the (recursive) function body is recorded in the function type. The cost of application considers also the cost of executing the function body. Rules u-inl and u-inr are dual. Notice that the introduction of the sum type $A_1 + A_2$ requires well-formedness for the type that is introduced in the sum. Rule u-case for eliminating the sum type requires the same upper and lower bound in both branches. Rules u-nil and u-cons are similar and specify sizes of lists. Finally, rule u-sub, internalizes weakening for the upper and lower bounds and subtyping.

The rules for the unary typing of impure expressions are presented in Figure 4. Since costs of operations like reading and writing memory are *variable* on most architectures, these rules rely on given upper- and lower bounds on the cost of each operation. For example, L_{read} and U_{read} denote the minimum and maximum cost of reading a heap location, respectively. The costs $U_{\mathsf{let}}, U_{\mathsf{alloc}}, L_{\mathsf{read}}, L_{\mathsf{updt}}$ are similar.

$$\frac{\Sigma;\,\Delta;\,\Phi;\,\Omega \vdash_{L}^{U} t : A \qquad \Sigma;\,\Delta \vdash P\ \mathsf{wf}}{\Sigma;\,\Delta;\,\Phi;\,\Omega \vdash_{0}^{0} \mathsf{return}\ t : \{P\}\ \exists \gamma.A\ \{P\}}\ \textbf{u-ret}$$

$$\frac{\Sigma;\,\Delta;\,\Phi;\,\Omega \vdash_{L_1}^{U_1} t_1 : \{P\}\ \exists \vec{\gamma}_1.A\ \{P'\} \qquad \Sigma;\,\Delta, \vec{\gamma}_1;\,\Phi;\,\Omega, x : A \vdash_{L_2}^{U_2} t_2 : \{P'\}\ \exists \vec{\gamma}_2.B\ \{Q\}}{\Sigma;\,\Delta;\,\Phi;\,\Omega \vdash_{0}^{0} \mathsf{let}\ \{x\} = t_1\ \mathsf{in}\ t_2 : \{P\}\ \exists \vec{\gamma}_1, \vec{\gamma}_2.B\ \{Q\}}\ \textbf{u-bind}$$

$$\frac{\Sigma;\,\Delta;\,\Phi;\,\Omega \vdash_{L_1}^{U_1} t_1 : \mathrm{int}[I] \qquad \Sigma;\,\Delta;\,\Phi;\,\Omega \vdash_{L_2}^{U_2} t_2 : A \qquad \gamma\ \text{fresh} \qquad \Sigma;\,\Delta \vdash P\ \mathsf{wf}}{\Sigma;\,\Delta;\,\Phi;\,\Omega \vdash_{0}^{0} \mathsf{alloc}\ t_1\ t_2 : \{P\}\ \exists \gamma.\ \mathsf{Array}_{\gamma}[I]\ A\ \ \{P \star \gamma \to \mathbb{N}\}}\ \textbf{u-alloc}$$

$$\frac{\begin{array}{c}\Sigma;\,\Delta;\,\Phi;\,\Omega \vdash_{L_1}^{U_1} t_1 : \mathsf{Array}_{\gamma}[I]\ A \\ \gamma \in dom(P) \qquad \Sigma;\,\Delta;\,\Phi;\,\Omega \vdash_{L_2}^{U_2} t_2 : \mathrm{int}[I'] \qquad \Delta;\,\Phi \vDash I' \le I \qquad \Sigma;\,\Delta \vdash P\ \mathsf{wf}\end{array}}{\Sigma;\,\Delta;\,\Phi;\,\Omega \vdash_{0}^{0} \mathsf{read}\ t_1\ t_2 : \{P\}\ \exists \_.A\ \{P\}}\ \textbf{u-read}$$

$$\frac{\begin{array}{c}\Sigma;\,\Delta;\,\Phi;\,\Omega \vdash_{L_1}^{U_1} t_1 : \mathsf{Array}_{\gamma}[I]\ A \qquad \Sigma;\,\Delta;\,\Phi;\,\Omega \vdash_{L_2}^{U_2} t_2 : \mathrm{int}[I'] \\ \Sigma;\,\Delta;\,\Phi;\,\Omega \vdash_{L_3}^{U_3} t_3 : A \qquad \Delta;\,\Phi \vDash I' \le I \qquad \Sigma;\,\Delta \vdash P\ \mathsf{wf} \qquad \Delta;\,\Phi \vDash I' \in \beta\end{array}}{\Sigma;\,\Delta;\,\Phi;\,\Omega \vdash_{0}^{0} \mathsf{updt}\ t_1\ t_2\ t_3 : \{P \star \gamma \to \beta\}\ \exists \_.\mathrm{unit}\ \{P \star \gamma \to \beta\}}\ \textbf{u-updt}$$

Fig. 4. Selection of impure unary typing rules.

Rules u-ret and u-bind 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. For example, the assertion $P_1 \star P_2$ corresponds to disjoint parts of the heap. The rule for allocations, u-alloc, 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 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 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. Cost arises only when the impure expression alloc t_1 t_2 is forced; this is accounted for in the cost annotations of the monadic type. The rule for reading, u-read, merely checks that the index being read is within the array bounds. The rule for updating, u-updt, 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.

Relational Typing Judgment. 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, that is, $\mathrm{cost}(t_1) - \mathrm{cost}(t_2)$.

$$\frac{}{\Sigma;\Delta;\Phi;\Gamma \vdash n \ominus n \lesssim 0 : \mathrm{int}[n]}\ \textbf{r-int} \qquad \frac{\Gamma(x) = \tau}{\Sigma;\Delta;\Phi;\Gamma \vdash x \ominus x \lesssim 0 : \tau}\ \textbf{r-var}$$

$$\frac{\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_1' \lesssim D_1 : \tau_1 \xrightarrow{\mathrm{diff}(D)} \tau_2 \qquad \Sigma;\Delta;\Phi;\Gamma \vdash t_2 \ominus t_2' \lesssim D_2 : \tau_1}{\Sigma;\Delta;\Phi;\Gamma \vdash t_1\ t_2 \ominus t_1'\ t_2' \lesssim D + D_1 + D_2 : \tau_2}\ \textbf{r-app}$$

$$\frac{\Sigma;\Delta;\Phi;x:\tau, f\ :\tau \xrightarrow{\mathrm{diff}(D)} \sigma, \Gamma \vdash t_1 \ominus t_2 \lesssim D : \sigma}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathrm{fix}\, f(x).t_1 \ominus \mathrm{fix}\, f(x).t_2 \lesssim 0 : \tau \xrightarrow{\mathrm{diff}(D)} \sigma}\ \textbf{r-fix}$$

$$\frac{\Sigma;\Delta;\Phi;\Gamma \vdash t \ominus t' \lesssim D : \tau_1 \qquad \Sigma;\Delta;\Phi \vdash \tau_2\ \mathsf{wf}}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathrm{inl}\ t \ominus \mathrm{inl}\ t' \lesssim D : \tau_1 + \tau_2}\ \textbf{r-inl}$$

$$\frac{\Sigma;\Delta;\Phi;\Gamma \vdash t \ominus t' \lesssim D : \tau_2 \qquad \Sigma;\Delta;\Phi \vdash \tau_1\ \mathsf{wf}}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathrm{inr}\ t \ominus \mathrm{inr}\ t' \lesssim D : \tau_1 + \tau_2}\ \textbf{r-inr}$$

$$\frac{\begin{array}{c}\Sigma;\Delta;\Phi;\Gamma \vdash t \ominus t' \lesssim D_1 : \tau_1 + \tau_2 \\ \Sigma;\Delta;\Phi;\Gamma, x:\tau_1 \vdash t_1 \ominus t_1' \lesssim D_2 : \tau \qquad \Sigma;\Delta;\Phi;\Gamma, y:\tau_2 \vdash t_2 \ominus t_2' \lesssim D_2 : \tau\end{array}}{\Sigma;\Delta;\Phi;\Gamma \vdash\ \mathrm{case}\ (t, x.t_1, y.t_2) \ominus\ \mathrm{case}\ (t', x.t_1', y.t_2') \lesssim D_1 + D_2 : \tau}\ \textbf{r-case}$$

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

$$\frac{\Sigma;\Delta;\Phi, C;\Gamma \vdash t_1 \ominus t_2 \lesssim D : \tau \qquad \Sigma;\Delta;\Phi, \neg C;\Gamma \vdash t_1 \ominus t_2 \lesssim D : \tau}{\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_2 \lesssim D : \tau}\ \textbf{r-split}$$

$$\frac{\Sigma;\Delta;\Phi;\Omega \vdash t \ominus t \lesssim D : \tau \qquad \Sigma;\Delta;\Phi \models \tau \sqsubseteq \tau' \qquad \Sigma;\Delta;\Phi \vDash D \leq D'}{\Sigma;\Delta;\Phi;\Gamma \vdash t \ominus t \lesssim D' : \tau'}\ \textbf{r-sub}$$

$$\frac{\begin{array}{c}\Sigma;\Delta;\Phi;x:\tau_1, f\ :\tau_1 \xrightarrow{\mathrm{diff}(D)} \tau_2, \Gamma, f\ : U(A_1, A_2) \vdash t_1 \ominus t_2 \lesssim D : \tau_2 \\ \Sigma;\Delta;\Phi;|\Gamma|_1 \vdash_0^0 \mathrm{fix}\, f(x).t_1 : A_1 \qquad \Sigma;\Delta;\Phi;|\Gamma|_2 \vdash_0^0 \mathrm{fix}\, f(x).t_2 : A_2\end{array}}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathrm{fix}\, f(x).t_1 \ominus \mathrm{fix}\, f(x).t_2 \lesssim 0 : \tau_1 \xrightarrow{\mathrm{diff}(D)} \tau_2}\ \textbf{r-fix-ext}$$

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

That the relational judgements relates two programs naturally leads to two kinds of relational typing rules: *synchronous rules* that relate two structurally similar programs, and *asynchronous rules* for arbitrary programs. We first present a selection of the *pure* typing rules, both synchronous (Figure 5) and asynchronous (Figure 6). These rules are again inspired by the work of Çiçek *et al.* (2017). Then, we present a selection of *impure* typing rules, which support relational cost analysis for arrays. Again, we distinguish synchronous rules (Figure 7) from asynchronous rules (Figure 8). A complete set of typing rules can be found in the Appendix.

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

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

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

$$\frac{\Sigma;\Delta;\Phi;|\Gamma|_1 \vdash^{U_1}_{L_1} t_1 : A_1 \xrightarrow{\mathrm{exec}(L,U)} A_2 \qquad \Sigma;\Delta;\Phi;\Gamma \vdash t_2 \ominus t_2' \lesssim D_2 : U(A_1, A_2')}{\Sigma;\Delta;\Phi;\Gamma \vdash t_1\ t_2 \ominus t_2' \lesssim U_1 + U + D_2 + c_{app} : U(A_2, A_2')}\ \textbf{r-app-e}$$

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

$$\frac{\Sigma;\Delta;\Phi;|\Gamma|_2 \vdash^{U_1}_{L_1} t' : A_1 + A_2 \qquad \begin{array}{c}\Sigma;\Delta;\Phi;\Gamma, x : U(A_1, A_1) \vdash t \ominus t_1' \lesssim D_2 : \tau \\ \Sigma;\Delta;\Phi;\Gamma, y : U(A_2, A_2) \vdash t \ominus t_2' \lesssim D_2 : \tau\end{array}}{\Sigma;\Delta;\Phi;\Gamma \vdash t \ominus\ \mathsf{case}\ (t', x.t_1', y.t_2') \lesssim D_2 - L_1 - c_{case} : \tau}\ \textbf{r-e-case}$$

Fig. 6. Selection of pure relational asynchronous typing rules.

Pure Synchronous Rules. We present selected synchronous rules for pure expressions in Figure 5. The rule r-int relates two copies of the same integer n the singleton type int[n]. The rules r-var, r-fix, and r-app are the relational counterpart of the rules for variables, function abstraction, and application we saw in the unary part. The main difference is that we give an upper bound to the relative cost, rather than lower and upper bounds on the execution cost of a single expression. The rules r-inl, r-inr, and r-case type the introduction and elimination of the relational sum type. In the elimination rule r-case, notice that we require the relative cost D_2 of the two branches to be the same. Rule r-split allows reasoning by cases on any constraint in the constraint environment. Rule r-sub is the relational version of the rule u-sub. It allows weakening the upper bound on the relative cost D as well as subtyping.

Rule r-nc is the introduction rule for $\Box$-ed types. Briefly, t can be related to itself at the 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 0. This rule allows us to assume that given a pair of functions whose type is refined with $\Box$, if we apply them to the same argument, we have two executions following the same path, and at the same cost. We discussed this intuition behind the rule r-nc in Section 2 when we looked at the first relational type of mapi.

Rule r-fix-ext types fixpoint expressions relationally. This rule also requires *unary* typing for the two functions, which are established in separate premises. We require these

additional premises so that we can use the information provided by the unary typing to establish the relative cost. In other words, this rule introduces a weak form of *intersection types* in the environment which can be used in combination with the rule r-switch (Figure 6) to give precise bounds on relative cost.

Pure Asynchronous Rules. We present a selection of the pure asynchronous rules in Figure 6. Rule r-switch allows switching from relational reasoning about t_1 and t_2 in the conclusion, to unary reasoning about the two terms independently in the premises. 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 an erasure map $|\Gamma|_i$ from relational environments to unary environments ($i = 1$ for left and $i = 2$ for the right), 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 pure let construct. (This is different from the cost c_{let} of the monadic bind, which we discussed earlier.) Notice that one of the assumptions in this rule, the one for the expression t_1, is a unary typing judgment. This is needed in order to provide guarantees on the 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 standard 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 to an arbitrary expression, while rule r-case-e relates a case expression with an arbitrary expression and r-e-case does the opposite. Also in these rules, we use some unary typing assumptions to guarantee typability and to provide unary costs that are used in giving upper bounds on the relative costs.

Impure Synchronous Rules. Figure 7 shows a selection of relational synchronous typing rules pertaining to monadic constructs and arrays. Rules r-ret and r-bind 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-bind uses the style of separation logic.

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-alloc and r-allocb for relationally typing the alloc construct. The rules are similar, for example, both create a new static name γ for the two allocated arrays, and both account for relative costs very similarly. However, r-allocb applies only when the expressions initializing the two arrays are related at a $\Box$-ed type (the second premise). As a result, it is guaranteed that the arrays allocated in the two runs will 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-alloc 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 after the allocation.

A similar difference between the rules r-read and r-readb for relationally typing the construct read. In r-readb, the read index I' must not be in the β of the array being read

$$\frac{\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_1' \lesssim D_1 : \{P\}\ \overset{\mathrm{diff}(D)}{\exists \vec{\gamma_1}.\tau}\ \{P'\} \qquad \Sigma;\Delta,\vec{\gamma_1} : \vec{\mathbb{L}};\Phi;\Gamma,x:\tau \vdash t_2 \ominus t_2' \lesssim D_2 : \{P'\}\ \overset{\mathrm{diff}(D')}{\exists \vec{\gamma_2}.\sigma}\ \{Q\}}{\Sigma;\Delta;\Phi;\Gamma \vdash \mathsf{let}\ \{x\} = t_1\ \mathsf{in}\ t_2 \ominus \mathsf{let}\ \{x\} = t_1'\ \mathsf{in}\ t_2' \lesssim 0 : \{P\}\ \overset{\mathrm{diff}(D+D'+D_1+D_2)}{\exists \vec{\gamma_1}\vec{\gamma_2} : \sigma}\ \{Q\}}\ \textbf{r-bind}$$

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

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

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

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

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

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

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

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

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-read 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-updt and r-updtb for updt follow the principle of alloc: In r-updtb, 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-updt, where it must be added to β, since the two values at index I' might differ after the update. In all these rules, the premise $\Delta;\Phi \models I' \le I$ denotes constraint entailment, which means that for any substitution of all index variables in the index environment Δ, if all constraints in Φ hold, then the constraint $I' \le I$ holds. We omit the rules for deriving this judgment since they are standard.

$$\frac{\Sigma;\Delta;\Phi;|\Gamma|_1 \vdash^{U_1}_{L_1} t_1 :\{P_1\}\ \exists\vec{\gamma_1}:A_1\ \{Q_1\}^{\,\mathrm{exec}(L,U)} \qquad \Sigma;\Delta;\Phi;|\Gamma|_2 \vdash^{U_2}_{L_2} t_2' :\{P_2\}\ \exists\vec{\gamma_1}:A_1'\ \{Q_2\}^{\,\mathrm{exec}(L',U')} \qquad dom(P)=dom(P_1) \qquad \Sigma;\Delta;\Phi;\Gamma,x:U(A_1,A_1)\vdash t_2\ominus t_2' \lesssim D_2 : \{P\sqcup P_1\}\ \exists\vec{\gamma_1}.\tau\ \{Q\}^{\,\mathrm{diff}(D')}}{\Sigma;\Delta;\Phi;\Gamma\vdash \mathsf{let}\ \{x\}=t_1\ \mathsf{in}\ t_2\ominus t_2' \lesssim -L_2 : \{P\}\ \exists\vec{\gamma_1}.\tau\ \{Q\}^{\,\mathrm{diff}(U_1+U+(D_2+U_2)+D'+c_{let})}}\ \textbf{r-bind-e}$$

$$\frac{\Sigma;\Delta;\Phi;|\Gamma|_2 \vdash^{U_1}_{L_1} t_1' :\{P_1\}\ \exists\vec{\gamma_1}:A_1'\ \{Q_1\}^{\,\mathrm{exec}(L,U)} \qquad \Sigma;\Delta;\Phi;|\Gamma|_1 \vdash^{U_2}_{L_2} t_2 :\{P_2\}\ \exists\vec{\gamma_1}:A_1\ \{Q_2\}^{\,\mathrm{exec}(L',U')} \qquad dom(P)=dom(P_1) \qquad \Sigma;\Delta;\Phi;\Gamma,x:U(A_1',A_1')\vdash t_2\ominus t_2' \lesssim D_2 : \{P\sqcup P_1\}\ \exists\vec{\gamma_1}.\tau'\ \{Q\}^{\,\mathrm{diff}(D')}}{\Sigma;\Delta;\Phi;\Gamma\vdash t_2\ominus \mathsf{let}\ \{x\}=t_1'\ \mathsf{in}\ t_2' \lesssim U_2 : \{P\}\ \exists\vec{\gamma_1}.\tau'\ \{Q\}^{\,\mathrm{diff}(D'+(D_2-L_2)-L_1-L-c_{\mathrm{let}})}}\ \textbf{r-e-bind}$$

Fig. 8. Selection of monadic asynchronous relational typing rules.

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.

It is worth emphasizing that the set of γs in any pre- or postcondition must be written down explicitly, that is, 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 has not been a problem for our examples, and we believe an extension to lift this restriction is possible by adding language-level constructors and elimination rules for assigning γ. While this change would make our approach more flexible and more expressive, it would also put an additional burden on the programmer.

Impure Asynchronous Rules. Figure 8 shows the two asynchronous rules r-bind-e and r-e-bind, relating a monadic binding construct and an arbitrary expression. We explain only the rule r-bind-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-bind is its dual and it can be understood similarly). The first premise of the rule r-bind-e requires 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 monadic computation. The second premise requires a unary typing for the monadic expression t_2'. This gives 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 bound 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 t_2 and t_2'. 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 $\sqcup$ lifts set union pointwise to partial maps (preconditions P are partial maps from locations γ to sets of indices). The precondition in

the unary monadic type of t_1 in the first premise provides the indices where the computation associated with t_1 has write permission. So, intuitively, P_1 provides the indices that may be overwritten when executing t_1. Hence, we want to use this information when relating expressions t_2 and t_2' because t_1 is supposed to be executed by then. 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 the monadic binding construct. The bounds on the relative cost here deserve some discussion. Following the definition, and observing that monadic let is a value, we have that the relative cost of the two expressions is bound by $-L_2$. However, we want also to have a bound on the relative cost of forcing the evaluation of the two expressions, since this must be recorded in the monadic type. The upper bound is $U_1 + U + (D_2 + U_2) + D' + c_{let}$, where $U_1 + U$ upper bounds the cost of forcing the evaluation of t_1, D_2 upper bounds the difference in cost of evaluating t_2 and t_2' to values, U_2 upper bounds the cost of evaluating t_2 to its value, and D' upper bounds the difference in the costs of forcing the evaluation of the values obtained by evaluating t_2 and t_2', respectively.

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 types up to constraints, for example, it allows replacing int$[2 + i]$ with int$[5]$ under the constraint $i = 3$. Second, specific to cost analysis, subtyping allows weakening costs, for example, the relational type $\tau_1 \xrightarrow{\mathrm{diff}(D)} \tau_2$ can be subtyped to $\tau_1 \xrightarrow{\mathrm{diff}(D')} \tau_2$ 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, for example, $\Box\tau$ can be subtyped to τ. Finally, specific to the monadic types, subtyping allows weakening of pre and postconditions in monadic types. The first three purposes of subtyping in ARel are relatively standard (e.g., see Çiçek *et al.*, 2017) and we will only introduce them briefly. We describe the last use here at length. 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 9 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 $\forall i \in \{1, \ldots, n\}.\beta_i \subseteq \beta_i'$.

The rules s-u-arrow and s-r-arrow subtype unary and relational function types, respectively. The rule s-r-w allows weakening from the relational type τ to its weak version $U(|\tau|_1, |\tau|_2)$, where $|\cdot|_i$ is used to convert from a relational type to a unary type. When $i = 1$, this construct projects the left side of the relational type; when $i = 2$ it projects its right side (see the Appendix for the definitions of these projections). The rules s-r-list and s-r-array subtype relational list and array types, respectively. They impose requirements on lengths. Rule s-r-ua allows weakening the relational type $U(A_1, A_2)$ to $U(A_1', A_2')$ if we know that A_1 and A_2 are subtypes of A_1' and A_2', respectively. As we have seen before, the modality $\Box$ applied to a relational type τ requires the two components related by the type τ to be identical. In fact, $\Box$ has a comonadic flavor and it follows the standard comonadic rules s-r-T and s-r-D. Rule s-r-bd can be read as follows: When two equal functions (of

$$\frac{\Sigma;\Delta;\Phi \models A_1' \sqsubseteq A_1 \qquad \Sigma;\Delta;\Phi \models A_2 \sqsubseteq A_2' \qquad \Delta;\Phi \models L' \leq L \qquad \Delta;\Phi \models U \leq U'}{\Sigma;\Delta;\Phi \models A_1 \overset{\text{exec}(L,U)}{\longrightarrow} A_2 \sqsubseteq A_1' \overset{\text{exec}(L',U')}{\longrightarrow} A_2'}\ \textbf{s-u-arrow}$$

$$\frac{\Sigma;\Delta;\Phi \models A_1 \sqsubseteq A_1' \quad \Sigma;\Delta;\Phi \models A_2 \sqsubseteq A_2'}{\Sigma;\Delta;\Phi \models U(A_1,A_2) \sqsubseteq U(A_1',A_2')}\ \textbf{s-r-ua} \qquad \frac{}{\Sigma;\Delta;\Phi \models \tau \sqsubseteq U(|\tau|_1,|\tau|_2)}\ \textbf{s-r-w}$$

$$\frac{}{\Sigma;\Delta;\Phi \models \Box\tau \sqsubseteq \tau}\ \textbf{s-r-T} \qquad \frac{}{\Sigma;\Delta;\Phi \models \Box\tau \sqsubseteq \Box\Box\tau}\ \textbf{s-r-D}$$

$$\frac{\Sigma;\Delta;\Phi \models \tau_1' \sqsubseteq \tau_1 \qquad \Sigma;\Delta;\Phi \models \tau_2 \sqsubseteq \tau_2' \qquad \Delta;\Phi \models D \leq D'}{\Sigma;\Delta;\Phi \models \tau_1 \overset{\text{diff}(D)}{\longrightarrow} \tau_2 \sqsubseteq \tau_1' \overset{\text{diff}(D')}{\longrightarrow} \tau_2'}\ \textbf{s-r-arrow}$$

$$\frac{}{\Sigma;\Delta;\Phi \models \Box(\tau_1 \overset{\text{diff}(D)}{\longrightarrow} \tau_2) \sqsubseteq \Box\tau_1 \overset{\text{diff}(0)}{\longrightarrow} \Box\tau_2}\ \textbf{s-r-bd}$$

$$\frac{\Sigma;\Delta;\Phi \models \tau \sqsubseteq \tau' \qquad \Sigma;\Delta;\Phi \models I = I' \quad \Sigma;\Delta;\Phi \models \alpha \leq \alpha'}{\Sigma;\Delta;\Phi \models \mathsf{list}^{\alpha}[I]\ A \sqsubseteq \mathsf{list}^{\alpha'}[I']\ A'}\ \textbf{s-r-list}$$

$$\frac{}{\Sigma;\Delta;\Phi \models U(A_1 \overset{\text{exec}(L,U)}{\longrightarrow} A_2, A_1' \overset{\text{exec}(L',U')}{\longrightarrow} A_2') \sqsubseteq U(A_1,A_1') \overset{\text{diff}(U-L')}{\longrightarrow} U(A_2,A_2')}\ \textbf{s-r-execdiff}$$

$$\frac{\Sigma;\Delta;\Phi \models \tau \sqsubseteq \tau' \qquad \Sigma;\Delta;\Phi \models I = I' \qquad \Sigma;\Delta;\Phi \models \gamma = \gamma'}{\Sigma;\Delta;\Phi \models \mathsf{Array}_{\gamma}[I]\ \tau \sqsubseteq \mathsf{Array}_{\gamma'}[I']\ \tau'}\ \textbf{s-r-array}$$

$$\frac{\Sigma;\Delta;\Phi \models A \sqsubseteq A' \qquad \vec{\gamma_1} \subseteq \vec{\gamma_2} \qquad \Delta;\Phi \models L' \leq L \qquad \Delta;\Phi \models U \leq U' \qquad \Sigma,\Delta;\Phi \models P \sqsubseteq P' \qquad \Sigma,\vec{\gamma_1};\Delta;\Phi \models Q' \sqsubseteq Q}{\Sigma;\Delta;\Phi \models \{P\}\ \exists\vec{\gamma_1}.A\ \{Q\} \sqsubseteq \{P'\}\ \exists\vec{\gamma_2}.A'\ \{Q'\}}\ \textbf{s-um}$$

$$\frac{\Sigma;\Delta;\Phi \models \tau \sqsubseteq \tau' \quad \Delta;\Phi \models D \leq D' \qquad \Sigma,\vec{\gamma_1};\Delta;\Phi \models Q \sqsubseteq Q' \qquad \Sigma;\Delta;\Phi \models P' \sqsubseteq P \qquad \vec{\gamma_1} \subseteq \vec{\gamma_2}}{\Sigma;\Delta;\Phi \models \{P\}\ \exists\vec{\gamma_1}.\tau\ \{Q\} \sqsubseteq \{P'\}\ \exists\vec{\gamma_2}.\tau'\ \{Q'\}}\ \textbf{s-rm}$$

$$\frac{\beta_i' = \beta_i \cup T_i \cup T_i'}{\begin{array}{c}\Sigma;\Delta;\Phi \vDash U(\{\gamma_i \to T_i\}\ \exists\vec{\gamma_1}.A_1\ \{Q_1\},\{\gamma_i \to T_i'\}\ \exists\vec{\gamma_2}.A_2\ \{Q_2\}) \\ \sqsubseteq \{\gamma_i \to \beta_i\}\ \exists\vec{\gamma_1},\vec{\gamma_2}.U(A_1,A_2)\ \{\gamma_i \to \beta_i'\}\end{array}}\ \textbf{s-rum}$$

Fig. 9. Selection of subtyping rules.

the $\Box$-ed relational function type $\Box(\tau_1 \xrightarrow{\mathrm{diff}(D)} \tau_2)$) are given equal arguments (type $\Box\tau_1$), the results are equal and the relative cost is 0. We used this rule implicitly in the mapi example in Section 2.

Next, we present the subtyping rules for monadic types. The first 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 precondition 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 because the meanings of the pre(post)condition in the unary and relational monadic types are completely different. Finally, rule s-rum allows subtyping from the modality 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 (associated with the static name 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 $\sqsubseteq$). This is because the indices in the set $T_i \cup T_i$' may be overwritten during at least one of the two executions.

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 relational, that interact at the type $U(A_1, A_2)$.

4.1 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 (Appel & McAllester, 2001; Ahmed, 2006). The step-index counts the "steps" in our operational semantics. Importantly, the interpretation is also indexed by a *world* g mapping from static names γ to triples (l, n, A) specifying the location, the length of the array, and the *syntactic* type of the elements of the array named γ. Technically, we are defining a *Kripke logical relation*, and the world g is a so-called Kripke world (Neis *et al.*, 2011; Turon *et al.*, 2013). We give the clauses defining the value interpretation of unary types in Figure 10.

The value interpretations of standard type constructors like pairs and functions are also standard. The value interpretation of an array type, $[\![\mathsf{Array}_\gamma[I]\ A]\!]_{g,k}$, is a set of locations. A location l is in this set if $g(\gamma)$ is (l, A, I), that is, the element type and length for γ in the world g match those in the array type and the location l corresponds to γ.

The value interpretation of monadic types relies on a heap relation $H \models_{g,k} P$, which is defined in Figure 12. This heap relation means that the assertion P—which could be a pre- or postcondition from a unary monadic type—holds for the heap H at world g at step k. The

$$
\begin{array}{ll}
[\![\mathrm{int}]\!]_{g,k} & = \{\ n \mid n \in \mathbb{N}\ \} \\
[\![\mathrm{unit}]\!]_{g,k} & = \{\ ()\ \} \\
[\![\mathrm{int}[I]]\!]_{g,k} & = \{\ n \mid I = n\ \} \\
[\![\mathsf{Array}_{\gamma}[I]\ A]\!]_{g,k} & = \{\ l \mid I = n\ \wedge\ g(\gamma) = (l, A, n)\ \} \\
[\![A \xrightarrow{\mathrm{exec}(L,U)} A']\!]_{g,k} & = \left\{ \mathsf{fix}\ f(x).t \;\middle|\; \begin{array}{l} \forall k' < k,\ g' \sqsupseteq g.\ \forall v.\ v \in [\![A]\!]_{g',k'} \implies \\ \quad t[v/x][\mathsf{fix}\ f(x).t/f] \in [\![A']\!]^{e}_{g',k',(L,U)} \end{array} \right\} \\
[\![\{P\}\ \exists\vec{\gamma}.A\ \{Q\}]\!]^{\mathrm{exec}(L,U)}_{g,k} & = \left\{ v \;\middle|\; \begin{array}{l} \forall g_1 \sqsupseteq g.\ \forall k_1 \le k,\ k_2 < k_1.\ \forall c.\ \forall H. \\ H \vDash_{g_1,k_1} P \wedge\ v; H \Downarrow_f^{c,k_2} \implies \\ \quad \exists g_2 \sqsupseteq g_1.\ \exists H_1,\ v_1,\ \vec{\gamma}.\ \ v; H \Downarrow_f^{c,k_2} v_1; H_1\ \wedge \\ L \le c \le U\ \wedge\ H_1 \vDash_{g_2,k_1-k_2} Q\ \wedge\ v_1 \in [\![A]\!]_{g_2,k_1-k_2}\ \wedge \\ \Big(\exists n.\ P = \{\gamma_1 \to T_1, \ldots, \gamma_n \to T_n\}\ \wedge \\ \quad \forall i \in [1,n].\ g(\gamma_i) = (l_i, A, m)\ \implies \\ \quad \forall j. H[l_i][j] \neq H_1[l_i][j]\ \implies\ j \in T_i\Big) \end{array} \right\} \\
[\![\forall i{::}S.\ A]\!]^{\mathrm{exec}(L,U)}_{g,k} & = \{\ \Lambda.t \mid \forall I.\ \vdash I :: S\ .\ \wedge\ t \in [\![A[I/i]]\!]^{E,(L[I/i],U[I/i])}_{g,k-1}\ \} \\
[\![\exists i :: S.\ A\ ]\!]_{g,k} & = \{\ \mathsf{pack}\ v\ \mid \exists I.\ \vdash I :: S\ \wedge v \in [\![A[I/i]]\!]_{g,k-1}\ \} \\
[\![A_1 + A_2]\!]_{g,k} & = \{\ \mathrm{inl}\ v \mid v \in [\![A_1]\!]_{g,k}\ \} \cup \{\ \mathrm{inr}\ v \mid v \in [\![A_2]\!]_{g,k}\ \} \\
[\![C \supset A]\!]_{g,k} & = \{\ v \mid \nvDash C\ \vee\ v \in [\![A]\!]_{g,k-1}\ \} \\
[\![C \& A]\!]_{g,k} & = \{\ v \mid \vDash C\ \wedge\ v \in [\![A]\!]_{g,k-1}\ \}
\end{array}
$$

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

$$
\begin{array}{ll}
[\![\cdot]\!]_{g,k} & = \{\ \emptyset\ \} \\
[\![\Omega, x : A]\!]_{g,k} & = \{\ (\sigma[v/x]) \mid \sigma \in [\![\Omega]\!]_{g,k}\ \wedge\ v \in [\![A]\!]_{g,k}\ \}
\end{array}
$$

Fig. 10. The interpretation of the types and contexts.

relation checks that for every (l, A, n) in the range of g, every element of the array named l in H is in the interpretation of type A. To break the cyclicity between the definition of the heap relation and value interpretation, A is interpreted at the smaller step index $k-1$ in the heap relation.

Back to the unary monadic type, the value interpretation $[\![\{P\}\ \exists\vec{\gamma}.A\ \{Q\}]\!]^{\mathrm{exec}(L,U)}_{g,k}$ is a set of monadic values v that when forced in 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 note that the interpretation quantifies universally over worlds g_1 that extend g ($g_1 \sqsupseteq g$) and step-indices k_1 less than or equal to k. This is standard in step-indexed Kripke logical relations and makes the interpretation "monotonic", that is, closed under larger worlds and smaller step indices (Lemma 1).

Next, we extend the value interpretation to an expression interpretation:

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

Compared to the value interpretation, the expression interpretation, which we identify by the superscript e, accounts for lower and upper bound costs L and U. The expression interpretation requires that if the expression t evaluates to value v with the step index k' and cost c, then the cost satisfies the constraint $L \leq c \leq U$ and the resulting value v is in the value interpretation of the type A with the remaining step index $k - k'$.

Next, we extend our interpretation to *open* terms. For this, we first extend our value interpretation to unary contexts: Given a substitution σ, we say that $\sigma \in [\![\Omega]\!]_{g,k}$ if σ maps every variable in Ω to a value in the interpretation of its unary type. We write σt to denote the application of the substitution σ to the term t. With this, we can extend our interpretation to typed open terms, that is, typing judgments, as in the statement of the fundamental theorem (Theorem 2).

4.2 Relational interpretation

Next, we interpret relational types. Figure 11 shows the value and expression interpretations of relational types, and the interpretation of relational contexts. As in the case of the unary interpretation, we use Kripke worlds. Relational Kripke worlds are denoted G and they map 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 each of these two arrays, and τ is the type at whose relational interpretation the two arrays' corresponding elements should be related.

The value interpretation of a relational type τ is written $(\!|\tau|\!)_{G,k}$. It is a set of pairs of related values. Most of the clauses of this definition are straightforward. Somewhat unusually, the interpretation of a function type contains a pair of (recursive) functions that satisfies not one but two conditions: (i) Given related values as arguments, the functions return related results and (ii) Each of the two functions, when given an argument in the (unary) interpretation of the argument type's unary projection, returns a result in the (unary) interpretation of the result type's unary projection. The first condition is standard. The second condition is needed to make our relational-to-unary projections of types sound.

To define the interpretation of the relational monadic types we need a relational heap relation $(H_1, H_2) \vDash_{G,k} P$, defined in Figure 12. The relation says when the heaps H_1, H_2 from two runs satisfy a relational assertion P, which could be a pre- or postcondition from a relational monadic type. Intuitively, this relation holds when for every $\gamma \to \beta$ in P, $G(\gamma)$ is also defined, and if $G(\gamma) = (l_1, l_2, \tau, n)$, then for every index up to n, the two arrays l_1, l_2 in heaps H_1, H_2, respectively, have elements within the relational interpretation of τ, and every index where the two elements differ is in β. This formalizes the intuitive meaning of β from earlier sections.

Note that the condition on elements differing is a one-way implication: We do not insist that at every index in β, the two elements necessarily differ. In fact, depending on τ, in some cases, even elements at indices in β might be forced to be equal. For example, when τ is int[m] (for some m) or even $\exists i.\text{int}[i]$, this forces corresponding elements to be equal at all indices since the relational interpretation of int[m] is the singleton $\{(m, m)\}$. However, when $\tau = U(A_1, A_2)$, elements at indices in β can be arbitrary values of types A_1, A_2 since the relational interpretation of $U(A_1, A_2)$ is morally $A_1 \times A_2$.

Like the unary heap relation, the relational heap relation is well-founded by induction on the step index.

$$
\begin{aligned}
(\!|\mathsf{int}|\!)_{G,k} &= \{\ (n_1,n_2) \mid n_1 = n_2\ \}\\
(\!|\mathsf{unit}|\!)_{G,k} &= \{\ ((),())\ \}\\
(\!|\mathsf{int}[I]|\!)_{G,k} &= \{\ (n,n) \mid I = n\ \}\\
(\!|\mathsf{Array}_\gamma[I]\ \tau|\!)_{G,k} &= \{\ (l_1,l_2) \mid I = n\ \wedge\ G(\gamma) = (l_1,l_2,\tau,n)\ \}\\
(\!|\tau_1 \xrightarrow{\mathrm{diff}(D)} \tau_2|\!)_{G,k} &= \left\{ (\ \mathsf{fix}\ f(x).t_1,\ \mathsf{fix}\ f(x).t_2\ ) \;\middle|\; \begin{array}{l} \forall G' \supseteq G.\ \forall k' \le k-1.\ \forall v_1,\ v_2.\\ (v_1,v_2) \in (\!|\tau_1|\!)_{G',k'} \implies\\ \quad (t_1[v_1/x][\mathsf{fix}\ f(x).t_1/f],\\ t_2[v_2/x][\mathsf{fix}\ f(x).t_2/f]) \in (\!|\tau_2|\!)^e_{G',k',D}\ \wedge\\ \forall j.\ \Big(\mathsf{fix}\ f(x).t_1 \in [\![|\tau_1|_1 \xrightarrow{\mathrm{exec}(0,\infty)} |\tau_2|_1]\!]_{G|_1,j}\ \wedge\\ \mathsf{fix}\ f(x).t_2 \in [\![|\tau_1|_2 \xrightarrow{\mathrm{exec}(0,\infty)} |\tau_2|_2]\!]_{G|_2,j}\Big) \end{array} \right\}\\
(\!|\{P\}\ \exists\vec{\gamma}.\tau\ \{Q\}|\!)^{\mathrm{diff}(D)}_{G,k} &= \left\{ (v_1,v_2) \;\middle|\; \begin{array}{l} \forall G_1 \supseteq G.\ \forall k_1 \le k,\ k_2 < k_1,\ k_3,\ c_1,\ c_2,\ H_1,\ H_2.\\ (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}\\ \implies\quad \exists G_2 \supseteq G_1,\ H_1',\ H_2',\ v_1',\ v_2'.\\ \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\\ \quad (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\\ \quad c_1 - c_2 \le D\Big) \end{array} \right\}\\
(\!|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'}\ \}\\
(\!|\forall i{::}S.\ \tau|\!)^{\mathrm{diff}(D)}_{G,k} &= \left\{ (\Lambda.t_1,\Lambda.t_2) \;\middle|\; \begin{array}{l} \forall I.\ \vdash I :: S\ \wedge\ (t_1,t_2) \in (\!|\tau[I/i]|\!)^{E,D[I/i]}_{G,k-1}\ \wedge\\ \forall j.\ t_1 \in [\![|\tau|_1]\!]^{E,(0,\infty)}_{G|_1,j}\ \wedge\ t_2 \in [\![|\tau|_2]\!]^{E,(0,\infty)}_{G|_2,j} \end{array} \right\}\\
(\!|\exists i :: S.\ \tau\ |\!)_{G,k} &= \{\ (\mathsf{pack}\ v_1, \mathsf{pack}\ v_2) \mid \exists I.\ \vdash I :: S\ \wedge\ (v_1,v_2) \in (\!|\tau[I/i]|\!)_{G,k-1}\ \}\\
(\!|\tau_1 + \tau_2|\!)_{G,k} &= \{\ (\mathrm{inl}\ v_1, \mathrm{inl}\ v_2) \mid (v_1,v_2) \in (\!|\tau_1|\!)_{G,k-1}\ \vee\ (v_1,v_2) \in (\!|\tau_2|\!)_{G,k-1}\}\\
(\!|\Box\tau|\!)_{G,k} &= \{\ (v,v) \mid (v,v) \in (\!|\tau|\!)_{G,k}\ \}\\
(\!|C \supset \tau|\!)_{G,k} &= \{\ (v_1,v_2) \mid \not\vDash C\ \vee\ (v_1,v_2) \in (\!|\tau|\!)_{G,k-1}\ \}\\
(\!|C \& \tau|\!)_{G,k} &= \{\ (v_1,v_2) \mid \vDash C\ \wedge\ (v_1,v_2) \in (\!|\tau|\!)_{G,k-1}\ \}\\
(\!|\tau|\!)^e_{G,k,D} &= \left\{ (t_1,t_2) \;\middle|\; \begin{array}{l} \forall k_1 \le k.\ \forall k_2,\ v_1,\ v_2.\ (t_1 \Downarrow^{c_1,k_1} v_1\ \wedge\ t_2 \Downarrow^{c_2,k_2} v_2)\\ \implies\quad (v_1,v_2) \in (\!|\tau|\!)_{G,k-k_1}\ \wedge\ c_1 - c_2 \le D \end{array} \right\}\\
(\!|\cdot|\!)_{G,k} &= \{\ \emptyset\ \}\\
(\!|\Gamma|\!)_{G,k} &= \{\ (\sigma_1,\sigma_2) \mid \forall x \in dom(\Gamma).\ \forall \tau.\ x : \tau \in \Gamma.\ (\sigma_1(x),\sigma_2(x)) \in (\!|\tau|\!)_{G,k}\ \}
\end{aligned}
$$

Fig. 11. The interpretation of relational types and contexts.

The relational interpretation for a monadic type $\{P\}\ \exists\vec{\gamma}.\tau\ \{Q\}^{\mathrm{diff}(D)}$ (Figure 11) 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.

Next, we extend the value interpretation of relational types to pairs of expressions:

$$
(\!|\tau|\!)^e_{G,k,D} = \left\{ (t_1,t_2) \;\middle|\; \begin{array}{l} \forall k_1 \le k,\ \forall k_2,\ v_1,\ v_2.\ (t_1 \Downarrow^{c_1,k_1} v_1 \wedge t_2 \Downarrow^{c_2,k_2} v_2) \Rightarrow\\ (v_1,v_2) \in (\!|\tau|\!)_{G,k-k_1} \wedge c_1 - c_2 \le D \end{array} \right\}
$$

$$
\begin{array}{rll}
H \vDash_{g,k} \mathit{empty} & \text{iff} & \text{true} \\
H \vDash_{g,k} (\gamma \to \beta) & \text{iff} & \exists l, A, n : g(\gamma) = (l, A, n) \ \wedge \ \left(\forall i \leq n.\ \left(H[l][i]\right) \in (\!|A|\!)_{g,k-1}\right) \\
H \vDash_{g,k} (P * Q) & \text{iff} & \exists H', H'', g', g'' : \left(H = H' \uplus H''\right) \ \wedge \ \left(g = g' \uplus g''\right) \\
 & & \wedge \ \left(H' \vDash_{g',k} P\right) \ \wedge \ \left(H'' \vDash_{g'',k} Q\right) \\
(H_1, H_2) \vDash_{G,k} \mathit{empty} & \text{iff} & \text{true} \\
(H_1, H_2) \vDash_{G,k} (\gamma \to \beta) & \text{iff} & \exists l_1, l_2, \tau, n : G(\gamma) = (l_1, l_2, \tau, n) \\
 & & \wedge \ \left(\forall i \leq n.\ \left(H_1[l_1][i], H_2[l_2][i]\right) \in (\!|\tau|\!)_{G,k-1}\right) \\
 & & \wedge \ \left(\forall i \leq n.\ \left(H_1[l_1][i] \neq H_2[l_2][i] \ \Rightarrow \ i \in \beta\right)\right) \\
(H_1, H_2) \vDash_{G,k} (P * Q) & \text{iff} & \exists H_1', H_1'', H_2', H_2'', G', G'' : \\
 & & \left(H_1 = H_1' \uplus H_1''\right) \ \wedge \ \left(H_2 = H_2' \uplus H_2''\right) \ \wedge \ \left(G = G' \uplus G''\right) \\
 & & \wedge \ \left((H_1', H_2') \vDash_{G',k} P\right) \ \wedge \ \left((H_1'', H_2'') \vDash_{G'',k} Q\right)
\end{array}
$$

Fig. 12. Unary and relational heap relation.

The intepretation simply says that two expressions e_1, e_2 are in the interpretation of type τ if they both evaluate to some values v_1, v_2 and those values are in the value interpretation of τ. Additionally, the costs c_1, c_2 of these two evaluations satisfy $c_1 - c_2 \leq D$. Note that the step index counts steps of only the left evaluation, not both. We could, alternatively, have set up the interpretation to count steps of only the right evaluation or both.

Next, we extend our value interpretation to relational contexts (as for the unary case), and the statement of our fundamental theorem (Theorem 4.3) contains the interpretation of typed open terms, that is, the relational typing judgment.

Note. 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 because our language only considers arrays whose elements are of base type like int, bool, etc.

4.3 Fundamental theorem

In this section, we prove a standard theorem, called a fundamental theorem, for each of our two interpretations, unary and relational. We use $\vdash \delta : \Delta$ to mean that δ is a well-sorted substitution for the index variables in the domain of Δ, and $\vDash \delta\Phi$ to denote that δ satisfies the constraint environment Φ. As a preliminary step, we show a monotonicity lemma, which is useful in the proofs of our fundamental theorems.

Lemma 1 (Monotonicity).

1. *If* $k' \leq k$ *and* $G \subseteq G'$, *then* $(\!|\tau|\!)_{G,k} \subseteq (\!|\tau|\!)_{G',k'}$.
2. *If* $k' \leq k$ *and* $g \subseteq g'$, *then* $[\![A]\!]_{g,k} \subseteq [\![A]\!]_{g',k'}$.
3. *If* $k' \leq k$ *and* $G \subseteq G'$, *then* $(\!|\tau|\!)^{E,D}_{G,k} \subseteq (\!|\tau|\!)^{E,D}_{G',k'}$.
4. *If* $k' \leq k$ *and* $g \subseteq g'$, *then* $[\![A]\!]^{E,(L,U)}_{g,k} \subseteq [\![A]\!]^{E,(L,U)}_{g',k'}$.
5. *If* $k' \leq k$ *and* $G \subseteq G'$, *then* $(\!|\Gamma|\!)_{G,k} \subseteq (\!|\Gamma|\!)_{G',k'}$
6. *If* $k' \leq k$ *and* $g \subseteq g'$, *then* $[\![\Omega]\!]_{g,k} \subseteq [\![\Omega]\!]_{g',k'}$

Proof. Points (1) and (3), related to the interpretation of relational types, are proved simultaneously by induction on τ. Similarly, points (2) and (4), about the interpretation of the unary types, are proved simultaneously by induction on the unary type A. Points (5) and (6), about the contexts, follows directly from points (1) and (2). □

The monotonicity lemma says that all our interpretations are monotone with respect to larger worlds and smaller step indexes. Take the value interpretation of a relational type as an instance. If a pair of values (v_1, v_2) is in this interpretation for some step index k in some world G, then (v_1, v_2) is also in the value interpretation of the same type for any smaller step index k' and any bigger world G'.

Next, we present the fundamental theorem for ARel's unary typing. The theorem states the following. Suppose we have an expression t that is well-typed at a unary type A in contexts Σ, Δ, Φ and Ω with cost lower and upper bounds L and U. If we close everything using a substitution δ for the index variables in Δ (satisfying Φ) and a substitution σ for the term variables Ω satisfying the (context) interpretation of $\delta\Omega$ (at world g and step index k), then the closed term σt is in the interpretation of the closed type δA with bounds L and U (at world g and step index k).

Theorem 2 (Fundamental Theorem for Unary Typing). *If* $\Sigma; \Delta; \Phi; \Omega \vdash_L^U t : A$, $\vdash \delta : \Delta$ *and* $\vDash \delta\Phi$, *and* $\sigma \in [\![\delta\Omega]\!]_{g,k}$, *then* $(\sigma\ t) \in [\![\delta A]\!]^e_{g,k,(\delta L,\delta U)}$.

Proof. The proof is by induction on the derivation of the judgment $\Sigma; \Delta; \Phi; \Omega \vdash_L^U t : A$. We present some of the most relevant cases.

$$\textbf{Case}\ \frac{\Sigma; \Delta; \Phi; \Omega \vdash_{L_1}^{U_1} t_1 : \mathsf{Array}_\gamma[I]\ A \qquad \gamma \in dom(P) \qquad \Sigma; \Delta; \Phi; \Omega \vdash_{L_2}^{U_2} t_2 : \mathbf{int}[I'] \qquad \vDash I' \leq I \qquad \Sigma; \Delta \vdash P\ \mathsf{wf}}{\Sigma; \Delta; \Phi; \Omega \vdash_0^{\mathrm{exec}(L_1+L_2+L_{\mathsf{read}}, U_1+U_2+U_{\mathsf{read}})} \mathsf{read}\ t_1\ t_2 : \{P\}\ \exists\_ : A\ \{P\}}\ \textbf{u-read}$$

By assumption we have $\vdash \delta : \Delta$, $\vDash \delta\Phi$ and $\sigma \in [\![\delta\Omega]\!]_{g,k}$. We need to show:

$$\mathsf{read}\ (\sigma\ t_1)\ (\sigma\ t_2) \in [\![\delta\ \overset{\mathrm{exec}(L_1+L_2+L_{\mathsf{read}}, U_1+U_2+U_{\mathsf{read}})}{\{P\}\ \exists\_ : A\ \{P\}}\ ]\!]^e_{g,k,(0,0)}$$

Since $\mathsf{read}\ (\sigma\ t_1)\ (\sigma\ t_2)$ is a value, and its evaluation incurs no cost, it is sufficient to show:

$$\mathsf{read}\ (\sigma\ t_1)\ (\sigma\ t_2) \in [\![\delta\ \overset{\mathrm{exec}(L_1+L_2+L_{\mathsf{read}}, U_1+U_2+U_{\mathsf{read}})}{\{P\}\ \exists\_ : A\ \{P\}}\ ]\!]_{g,k}$$

By the definition of forcing evaluation we have:

$$\frac{\sigma\ t_1 \Downarrow^{c_1,k_1} l \qquad \sigma\ t_2 \Downarrow^{c_2,k_2} n_2 \qquad H(l)[n] = v}{\mathsf{read}\ (\sigma\ t_1)\ (\sigma\ t_2); H \Downarrow_f^{c_1+c_2+c_{\mathsf{read}}, k_1+k_2+1} v; H}\ \textbf{f-read}$$

So, unfolding the definition of interpretation, for $k' \leq k$, $g' \supseteq g$, and an arbitrary heap H such that $H \vDash_{g',k'} P$ and such that $k_1 + k_2 + 1 < k' \leq k$ we must show:

1. $\delta\ L_1 + \delta\ L_2 + \delta\ L_{\mathsf{read}} \leq c_1 + c_2 + c_{\mathsf{read}} \leq \delta\ U_1 + \delta\ U_2 + \delta\ U_{\mathsf{read}}$.
2. $H \vDash_{g',k'-(k_1+k_2+1)} P$.
3. $v \in [\![\delta A]\!]_{g',k'-(k_1+k_2+1)}$.

4. $\exists n.P = \{\gamma_1 \to T_1, \ldots, \gamma_n \to T_n\} \wedge \forall i \in [1,n].g(\gamma_i) = (l_i, A, m) \Rightarrow \forall j.(H[l_i][j]) \neq H[l_i][j] \Rightarrow j \in T_i$.

By induction hypothesis, from the first premise and Lemma 1 we have:

$$\sigma\ t_1 \in [\![\mathsf{Array}_\gamma[\delta\ I]\ (\delta A)]\!]^e_{g',k',(\delta\ L_1, \delta\ U_1)}$$

From this we have:

$$l \in [\![\mathsf{Array}_\gamma[\delta\ I]\ (\delta A)]\!]_{g',k'-k_1} \wedge \delta\ L_1 \leq c_1 \leq \delta\ U_1$$

which in turn tells us: $g'(\gamma) = (l, \delta A, \delta\ I)$.

By induction hypothesis, from the premise $\Sigma; \Delta; \Phi; \Omega \vdash^{U_2}_{L_2} t_2 : \mathrm{int}[I']$ and Lemma 1 we have:

$$\sigma\ t_2 \in [\![\mathrm{int}[\delta\ I']]\!]^e_{g',k',(\delta\ L_2, \delta\ U_2)}$$

From this we have that: $n_2 \in [\![\mathrm{int}[\delta\ I']]\!]_{g',k'-k_2}$, which in turn tells us that

$$n_2 = \delta\ I' \wedge \delta\ L_2 \leq c_2 \leq \delta\ U_2$$

Now to conclude we can proceed as follows:

1. the inequality $\delta\ L_1 + \delta\ L_2 + \delta\ L_{\mathsf{read}} \leq c_1 + c_2 + c_{\mathsf{read}} \leq \delta\ U_1 + \delta\ U_2 + \delta\ U_{\mathsf{read}}$ can be shown using the fact that $L_{\mathsf{read}} \leq c_{\mathsf{read}} < U_{\mathsf{read}}$ and the fact that in our language L_{read} and U_{read} are constants , which means $\delta\ L_{\mathsf{read}} = L_{\mathsf{read}}$ and $\delta\ U_{\mathsf{read}} = U_{\mathsf{read}}$.
2. $H \vDash_{g',k'-(k_1+k_2+1)} P$ is proved by unfolding the definition of our assumption $H \vDash_{g',k'} P$.
3. For $v \in [\![\delta A]\!]_{g',k'-(k_1+k_2+1)}$, we know that from the heap relation $H \vDash_{g',k'} P$ and the premise $\gamma \in dom(P)$, we have $\forall i \leq n, (H_1(l)(i)) \in [\![A]\!]_{g',k'-1}$, which in turn tells us $v \in [\![A]\!]_{g',k'-1}$. We can then show our goal by using Lemma 1.
4. $\exists n.P = \{\gamma_1 \to T_1, \ldots, \gamma_n \to T_n\} \wedge \forall i \in [1,n].g(\gamma_i) = (l_i, A, m) \Rightarrow \forall j.(H[l_i][j]) \neq H[l_i][j] \Rightarrow j \in T_i$ follows because the heap H is not changed, so the claim is trivial.

$$\textbf{Case}\ \frac{\Sigma; \Delta; \Phi; \Omega \vdash^{U_1}_{L_1} t_1 : \mathsf{Array}_\gamma[I]\ A \qquad \Sigma; \Delta; \Phi; \Omega \vdash^{U_2}_{L_2} t_2 : \mathbf{int}[I'] \qquad \Sigma; \Delta; \Phi; \Omega \vdash^{U_3}_{L_3} t_3 : A \qquad \Delta; \Phi \vDash I' \leq I \qquad \Sigma; \Delta; \vdash P\ \mathsf{wf} \qquad \Delta; \Phi \vDash I' \in \beta}{\Sigma; \Delta; \Phi; \Omega \vdash^{0}_{0} \mathsf{updt}\ t_1\ t_2\ t_3 : \{P \star \gamma \to \beta\}\ \exists\_ : \mathbf{unit}\ \{P \star \gamma \to \beta\}}^{\mathrm{exec}(L_1+L_2+L_3+L_{\mathsf{updt}}, U_1+U_2+U_3+U_{\mathsf{updt}})}\ \textbf{u-updt}$$

By assumption we have $\vdash \delta : \Delta, \vDash \delta\Phi$ and $\sigma \in [\![\delta\Omega]\!]_{g,k}$, we need to show:

$$\mathsf{updt}\ (\sigma\ t_1)\ (\sigma\ t_2)\ (\sigma\ t_3) \in [\![\delta\ \{P \star \gamma \to \beta\}\ \exists\_ : \mathrm{unit}\ \{P \star \gamma \to \beta\}]\!]^{e\ \mathrm{exec}(L_1+L_2+L_3+L_{\mathsf{updt}}, U_1+U_2+U_3+U_{\mathsf{updt}})}_{g,k,(0,0)}$$

Since $\mathsf{updt}\ (\sigma\ t_1)\ (\sigma\ t_2)\ (\sigma\ t_3)$ is a value, it is sufficient to show:

$$\mathsf{updt}\ (\sigma\ t_1)\ (\sigma\ t_2)\ (\sigma\ t_3) \in [\![\delta\ \{P \star \gamma \to \beta\}\ \exists\_ : \mathrm{unit}\ \{P \star \gamma \to \beta\}]\!]^{\mathrm{exec}(L_1+L_2+L_3+L_{\mathsf{updt}}, U_1+U_2+U_3+U_{\mathsf{updt}})}_{g,k}$$

By the definition of forcing evaluation we have:

$$\frac{\sigma\ t_1 \Downarrow^{c_1,k_1} l \qquad \sigma\ t_2 \Downarrow^{c_2,k_2} n \qquad \sigma\ t_3 \Downarrow^{c_3,k_3} v}{\mathsf{updt}\ (\sigma\ t_1)\ (\sigma\ t_2)\ (\sigma\ t_3); H \Downarrow_f^{c_1+c_2+c_3+c_{\mathsf{update}}, k_1+k_2+k_3+1} (); H(l)[n] \leftarrow v}\ \textbf{f-updt}$$

So, unfolding the definition of the interpretation, for $k' \leq k$, $g' \supseteq g$, for an arbitrary heap H such that $H \vDash_{g',k'} P \star \gamma \to \beta$, and $k_1 + k_2 + k_3 + 1 < k'$, we must show:

1. $\delta\, L_1 + \delta\, L_2 + \delta\, L_3 + \delta\, L_{\mathsf{updt}} \leq c_1 + c_2 + c_3 + c_{\mathsf{updt}} \leq \delta\, U_1 + \delta\, U_2 + \delta\, U_3 + \delta\, U_{\mathsf{updt}}$.
2. $(H(l)[n] \leftarrow v) \vDash_{g',k'-(k_1+k_2+k_3+1)} P \star \gamma \to \beta$.
3. $v \in [\![\mathrm{unit}]\!]_{g',k'-(k_1+k_2+k_3+1)}$.
4. $\exists n.P \star \gamma \to \beta = \{\gamma_1 \to T_1, \ldots, \gamma_n \to T_n\} \wedge \forall i \in [1,n].g(\gamma_i) = (l_i, A, m) \Rightarrow \forall j.(H[l_i][j]) \neq (H(l)[n] \leftarrow v)[l_i][j] \Rightarrow j \in T_i$.

By induction hypothesis, from the first premise $\Sigma; \Delta; \Phi; \Omega \vdash^{U_1}_{L_1} t_1 : \mathsf{Array}_\gamma[I]\, A$ and Lemma 1, we have:

$$\sigma\ t_1 \in [\![\mathsf{Array}_\gamma[\delta\, I]\ (\delta A)]\!]^e_{g',k',\delta\,(\delta\, L_1, \delta\, U_1)}$$

From this we have:

$$l \in [\![\mathsf{Array}_\gamma[\delta\, I]\ (\delta A)]\!]_{g',k'-k_1} \Rightarrow g'(\gamma) = (l, \delta A, \delta\, I) \wedge \delta\, L_1 \leq c_1 \leq \delta\, U_1$$

By induction hypothesis, from the second premise $\Delta; \Phi; \Omega \vdash^{U_2}_{L_2} t_2 : \mathrm{int}[I']$ and Lemma 1, we conclude:

$$\sigma\ t_2 \in [\![\mathrm{int}[\delta\, I]]\!]^e_{g',k',(\delta\, L_2, \delta\, U_2)}$$

From this we have that $n \in [\![\mathrm{int}[\delta\, I]]\!]_{g',k'-k_2}$, which in turn tells us that

$$n = I \wedge \delta\, L_2 \leq c_2 \leq \delta\, U_2$$

By induction hypothesis, from the third premise $\Sigma; \Delta; \Phi; \Omega \vdash^{U_3}_{L_3} t_3 : A$ and Lemma 1, we have:

$$\sigma\ t_3 \in [\![\delta A]\!]^e_{g',k',(\delta\, L_3, \delta\, U_3)}$$

From this we have:

$$v \in [\![\delta A]\!]_{g',k'-k_3} \wedge \delta\, L_3 \leq c_1'' \leq \delta\, U_3$$

To conclude, we proceed as follows:

1. the inequality $\delta\, L_1 + \delta\, L_2 + \delta\, L_3 + \delta\, L_{\mathsf{updt}} \leq c_1 + c_2 + c_3 + c_{\mathsf{updt}} \leq \delta\, U_1 + \delta\, U_2 + \delta\, U_3 + \delta\, U_{\mathsf{updt}}$ can be shown using the fact that $L_{\mathsf{updt}} \leq c_{\mathsf{updt}} < U_{\mathsf{updt}}$ and L_{updt} and U_{updt} are constants, which means $\delta\, L_{\mathsf{updt}} = L_{\mathsf{updt}}$ and $\delta\, U_{\mathsf{updt}} = U_{\mathsf{updt}}$.
2. $(H(l)[n] \leftarrow v) \vDash_{g',k'-(k_1+k_2+k_3+1)} P \star \gamma \to \beta$ is proved by unfolding the definition of our assumption $H \vDash_{g',k'} P \star \gamma \to \beta$. We can conlude that $\forall i \leq I.H(l)[i] \in (\!|\delta A|\!)_{g',k'-1}$, and then we can show that $\forall i \leq I.(H(l)[n] \leftarrow v)(l)[i] \in (\!|\delta\, A|\!)_{g',k'-(k_3+1)}$ because of the our previous conclusion $v \in [\![\delta A]\!]_{g',k'-k_3}$. Then our goal can be proved by using Lemma 1.
3. $v \in [\![\mathrm{unit}]\!]_{g',k'-(k_1+k_2+k_3+1)}$ is proved by the definition of the interpretation of unit type.
4. $\exists n.P \star \gamma \to \beta = \{\gamma_1 \to T_1, \ldots, \gamma_n \to T_n\} \wedge \forall i \in [1,n].g(\gamma_i) = (l_i, A, m) \Rightarrow \forall j.(H[l_i][j]) \neq (H(l)[n] \leftarrow v)[l_i][j] \Rightarrow j \in T_i$. It is proved depending on γ_i. When $\gamma_i \neq \gamma$, the array is not changed such that $H[l_i] = (H(l)[n] \leftarrow v)[l_i]$. When $\gamma_i = \gamma$, then $T_i = \beta$, we can show that the only updated index n at this array $H[l]$ is in β from our premise $\Delta; \Phi \vDash I' \in \beta$ in the rule, which proves this case. □

Similarly, we have a fundamental theorem for the relational typing. The theorem says the following. Suppose we have a pair of expressions (t_1,t_2) that is well-typed with the relational type τ in the contexts Σ, Δ, Φ, and Γ with upper bound D on the relative cost. Suppose we close Δ with the substitution δ for index variables, which also satisfies the constraint Φ, and close Γ with a pair of closing substitutions (σ_1, σ_2) satisfying the interpretation of the relational context $\delta\Gamma$. Then, the pair of closed terms $(\sigma_1\ t_1, \sigma_2\ t_2)$ is in the expression interpretation of the closed relational type $\delta\tau$ with the cost upper bound D.

Theorem 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* $(\sigma_1\ t_1, \sigma_2\ t_2) \in (\!|\delta\tau|\!)^e_{G,k,(\delta\ D)}$.

Proof. The proof is by induction on $\Sigma; \Delta; \Phi; \Gamma \vdash t_1 \ominus t_2 \lesssim D : \tau$.

$$\textbf{Case}\ \frac{\Sigma; \Delta; \Phi; \Gamma \vdash t_1 \ominus t_1' \lesssim D_1 : \tau_1 \xrightarrow{\mathrm{diff}(D)} \tau_2 \qquad \Sigma; \Delta; \Phi; \Gamma \vdash t_2 \ominus t_2' \lesssim D_2 : \tau_1}{\Sigma; \Delta; \Phi; \Gamma \vdash t_1\ t_2 \ominus t_1'\ t_2' \lesssim D + D_1 + D_2 : \tau_2}\ \textbf{r-app}$$

By assumption we have $\vdash \delta : \Delta$, $\vDash \delta\Phi$ and $(\sigma_1, \sigma_2) \in [\![\delta\Gamma]\!]_{G,k}$, we need to show:

$$\big((\sigma_1\ t_1)\ (\sigma_1\ t_2), (\sigma_2\ t_1')\ (\sigma_2\ t_2')\big) \in (\!|\delta\tau_2|\!)^e_{G,k,(\delta\ D + \delta\ D_1 + \delta\ D_2)}$$

By the definition of the evaluation we have:

$$\frac{\sigma_1\ t_1 \Downarrow^{c_1,k_1} \mathsf{fix} f(x).t \quad \sigma_1\ t_2 \Downarrow^{c_2,k_2} v \quad t[\mathsf{fix} f(x).t/f][v/x] \Downarrow^{c_3,k_3} v_1}{(\sigma_1\ t_1)\ (\sigma_1\ t_2) \Downarrow^{c_1+c_2+c_3+c_{\mathsf{fapp}},k_1+k_2+k_3+1} v_1}\ \textbf{e-fix}$$

$$\frac{\sigma_2\ t_1' \Downarrow^{c_1',k_1'} \mathsf{fix} f(x).t' \quad \sigma_2\ t_2' \Downarrow^{c_2',k_2'} v' \quad t'[\mathsf{fix} f(x).t'/f][v'/x] \Downarrow^{c_3',k_3'} v_1'}{(\sigma_2\ t_1')\ (\sigma_2\ t_2') \Downarrow^{c_1'+c_2'+c_3'+c_{\mathsf{fapp}},k_1'+k_2'+k_3'+1} v_1'}\ \textbf{e-fix}$$

So, unfolding the interpretation, for $k_1 + k_2 + k_3 + 1 \le k$, we must show:

1. $(v_1, v_1') \in (\!|\delta\tau_2|\!)_{G,k-(k_1+k_2+k_3+1)}$.
2. $(c_1 + c_2 + c_3 + c_{\mathsf{fapp}}) - (c_1' + c_2' + c_3' + c_{\mathsf{fapp}}) \le \delta\ D + \delta\ D_1 + \delta\ D_2$

By induction hypothesis, from the first premise $\Sigma; \Delta; \Phi; \Gamma \vdash t_1 \ominus t_2 \lesssim D_1 : \tau_1 \xrightarrow{\mathrm{diff}(D)} \tau_2$, we have:

$$(\sigma_1\ t_1, \sigma_2\ t_1') \in (\!|\delta\tau_1 \xrightarrow{\mathrm{diff}(\delta\ D)} \delta\tau_2|\!)^e_{G,k,\delta\ D_1}$$

From this we know:

$$(\mathsf{fix} f(x).t, \mathsf{fix} f(x).t') \in (\!|\delta\tau_1 \xrightarrow{\mathrm{diff}(\delta\ D)} \delta\tau_2|\!)_{G,k-k_1} \wedge c_1 - c_1' \le \delta\ D_1$$

By induction hypothesis, from the second premise $\Sigma; \Delta; \Phi; \Gamma \vdash t_2 \ominus t_2' \lesssim D_2 : \tau_1$, we conclude:

$$(\sigma_1\ t_2, \sigma_2\ t_2') \in (\!|\delta\tau_1|\!)^e_{G,k,\delta\ D_2}$$

From this we have:

$$(v, v') \in (\!|\delta\tau_1|\!)_{G,k-k_2} \wedge c_2 - c_2' \le \delta\ D_2$$

So, we unfold the definition of $(\mathsf{fix}\, f(x).t, \mathsf{fix}\, f(x).t') \in (\!|\delta\tau_1 \xrightarrow{\mathrm{diff}(\delta D)} \delta\tau_2|\!)_{G,k-k_1}$, using the previous conclusion $(v, v') \in (\!|\delta\tau_1|\!)_{G,k-(k_2+k_1+1)}$, we have:

$$(t[\mathsf{fix}\, f(x).t/f][v/x], t'[\mathsf{fix}\, f(x).t'/f][v'/x]) \in (\!|\delta\tau_2|\!)^e_{G,k-(k_1+k_2+1),\delta D}$$

From this we know:

$$(v_1, v_1') \in (\!|\delta\tau_2|\!)_{G,k-(k_1+k_2+k_3+1),\delta D} \wedge c_3 - c_3' \leq \delta D$$

Now to conclude we can proceed as follows:

1. $(v_1, v_1') \in (\!|\delta\tau_2|\!)_{G,k-(k_1+k_2+k_3+1)}$ is proved by our aforementioned conclusions
2. $(c_1 + c_2 + c_3 + c_{\mathsf{fapp}}) - (c_1' + c_2' + c_3' + c_{\mathsf{fapp}}) \leq \delta D + \delta D_1 + \delta D_2$ is proved by our aforementioned conclusions.

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

By assumption we have $\vdash \delta : \Delta$, $\vDash \delta\Phi$ and $(\sigma_1, \sigma_2) \in [\![\delta\Gamma]\!]_{G,k}$, we need to show:

$$(\mathsf{alloc}\ (\sigma_1\ t_1)\ (\sigma_1\ t_2), \mathsf{alloc}\ (\sigma_2\ t_1')\ (\sigma_2\ t_2')) \in (\!|\{P\}\ \exists\vec{\gamma}.\ \mathsf{Array}_\gamma[I]\ \delta\tau\ \{P \star \gamma \to \mathbb{N}\}|\!)^e_{G,k,0}$$

Since $\mathsf{alloc}\ (\sigma_1\ t_1)\ (\sigma_1\ t_2)$ and $\mathsf{alloc}\ (\sigma_2\ t_1')\ (\sigma_2\ t_2')$ are values, it is sufficient to show:

$$(\mathsf{alloc}\ (\sigma_1\ t_1)\ (\sigma_1\ t_2), \mathsf{alloc}\ (\sigma_2\ t_1')\ (\sigma_2\ t_2')) \in (\!|\{P\}\ \exists\gamma.\ \mathsf{Array}_\gamma[I]\ (\delta\tau)\ \{P \star \gamma \to \mathbb{N}\}|\!)_{G,k}$$

By the definition of the forcing evaluation we have:

$$\frac{\sigma_1\ t_1 \Downarrow^{c_1,k_1} n \quad \sigma_1\ t_2 \Downarrow^{c_2,k_2} v \quad z = [\overbrace{v, \ldots, v}^{n}] \quad l\ \mathsf{fresh}}{\mathsf{alloc}\ (\sigma_1\ t_1)\ (\sigma_1\ t_2); H_1 \Downarrow_f^{c_1+c_2+c_{\mathsf{alloc}},k_1+k_2+1} l; H_1 \uplus [l \to z]}\ \textbf{f-alloc}$$

$$\frac{\sigma_2\ t_1' \Downarrow^{c_1',k_1'} n' \quad \sigma_2\ t_2' \Downarrow^{c_2',k_2'} v' \quad z' = [\overbrace{v', \ldots, v'}^{n'}] \quad l'\ \mathsf{fresh}}{\mathsf{alloc}\ (\sigma_2\ t_1')\ (\sigma_2\ t_2'); H_2 \Downarrow_f^{c_1'+c_2'+c_{\mathsf{alloc}},k_1'+k_2'+1} l'; H_2 \uplus [l \to z']}\ \textbf{f-alloc}$$

So, unfolding the definition of interpretation, for $G' \supseteq G$, $k' \leq k$, and arbitrary heaps H_1, H_2 such that $(H_1, H_2) \vDash_{G',k'} P$ and $k_1 + k_2 + 1 < k' \leq k$. Now we define $G_2 = G'[r \to (l, l', \delta\tau, \delta I)$, $H_1' = H_1 \uplus [l \to z]$ and $H_2' = H_2 \uplus [l' \to z']$, we must show:

1. $(H_1', H_2') \vDash_{G_2,k'-(k_1+k_2+1)} P \star \gamma \to \mathbb{N}$.
2. $(l, l') \in (\!|\mathsf{Array}_\gamma[\delta I]\ (\delta\tau)|\!)_{G_2,k'-(k_1+k_2+1)}$.
3. $(c_1 + c_2 + c_{\mathsf{alloc}}) - (c_1' + c_2' + c_{\mathsf{alloc}}) \leq (\delta D_1 + \delta D_2)$.

By induction hypothesis, from the first premise $\Sigma; \Delta; \Phi; \Gamma \vdash t_1 \ominus t_1' \lesssim D_1 : \mathrm{int}[I]$ and Lemma 1 we conclude:

$$(\sigma_1\ t_1, \sigma_2\ t_1') \in (\!|\mathrm{int}[\delta I]|\!)^e_{G',k',\delta D_1}$$

From this we have: $(n, n') \in (\!|\mathsf{int}[\delta\, I]|\!)_{G', k'-k_1}$, which in turn tells us:

$$n = n' = \delta\, I \wedge c_1 - c_1' \leq \delta\, D_1$$

By induction hypothesis, from the second premise $\Sigma; \Delta; \Phi; \Gamma \vdash t_2 \ominus t_2' \lesssim D_2 : \tau$ and Lemma 1 we have:

$$(\sigma_1\, t_2, \sigma_2\, t_2') \in (\!|\delta\tau|\!)^e_{G', k', \delta\, D_2}$$

From this we have:

$$(v, v') \in (\!|\delta\tau|\!)_{G', k'-k_2} \wedge c_2 - c_2' \leq \delta\, D_2$$

Now to conclude we can proceed as follows:

1. $(H_1', H_2') \vDash_{G_2, k'-(k_1+k_2+1)} P \star \gamma \to \mathbb{N}$. It is proved depending on γ_i. When $\gamma_i \neq \gamma$, the corresponding arrays is not changed, so the claim is trivial. When $\gamma_i = \gamma$, we can show that $\forall i < \delta\, I. H_1'(l)[i] \neq H_2'(l')[i] \Rightarrow i \in \mathbb{N}$.
2. $(l, l') \in (\!|\mathsf{Array}_\gamma[\delta\, I]\ (\delta\tau)|\!)_{G_2, k'-(k_1+k_2+1)}$ is proved by unfolding its definition using $G_2 = G'[r \to (l, l', \delta\tau, \delta\, I)$.
3. $(c_1 + c_2 + c_{\mathsf{alloc}}) - (c_1' + c_2' + c_{\mathsf{alloc}}) \leq (\delta\, D_1 + \delta\, D_2)$ is proved by the previous conclusions.

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

By assumption we have $\vdash \delta : \Delta$, $\vDash \delta\Phi$ and $(\sigma_1, \sigma_2) \in [\![\delta\Gamma]\!]_{G,k}$, we need to show:

$$(\sigma_1\, t_2, \mathsf{let}\ \{x\} = (\sigma_2\, t_1')\ \mathsf{in}\ (\sigma_2\, t_2')) \in (\!|\quad \{P\}\ \exists \vec{\gamma}_1.\delta\tau'\ \{Q\}^{\mathsf{diff}(\delta(D' + (D_2 - L_2) - L_1 - L - c_{\mathsf{let}}))}\quad |\!)^e_{G,k,\delta\, U_2}$$

By the definition of the evaluation of $\sigma_1\, t_2$ of the form $\sigma_1\, t_2 \Downarrow^{c_1, k_1} v_1$, and the evaluation of the monadic bind of the form, we have:

$$(\mathsf{let}\ \{x\} = \sigma_2\, t_1'\ \mathsf{in}\ \sigma_2\, t_2') \Downarrow^{0,0} \mathsf{let}\ \{x\} = \sigma_2\, t_1'\ \mathsf{in}\ \sigma_2\, t_2'$$

So, unfolding the definition of the expression interpretation, we must show:

1. $c_1 - 0 \leq \delta\, U_2$.
2. $(v_1, (\mathsf{let}\ \{x\} = (\sigma_2\, t_1')\ \mathsf{in}\ (\sigma_2\, t_2'))) \in (\!|\delta(\quad \{P\}\ \exists \vec{\gamma}_1.\delta\tau\ \{Q\}^{\mathsf{diff}(\delta(D' + (D_2 - L_2) - L_1 - L - c_{\mathsf{let}}))}\quad)|\!)_{G,k-k_1}$.

By Theorem 2, from the second premise $\Sigma; \Delta; \Phi; |\Gamma|_1 \vdash_{L_2}^{U_2} t_2 : \{P_2\}\ \exists \vec{\gamma}_1 : A_1\ \{Q_2\}^{\mathsf{exec}(L',U')}$, instantiated with $\sigma_1 \in [\![\delta|\Gamma|_1]\!]_{|G|_1, k}$ inferred from $(\sigma_1, \sigma_2) \in (\!|\delta\Gamma|\!)_{G,k}$, we have:

$$\sigma_1\, t_2 \in [\![\{P_2\}\ \exists \vec{\gamma}_1 : \delta A_1\ \{Q_2\}^{\mathsf{exec}(\delta\, L', \delta\, U')}]\!]^e_{|G|_1, k, (\delta\, L_2, \delta\, U_2)}$$

From this we have:

$$v_1 \in [\![\{P_2\}\ \exists \vec{\gamma}_1 : \delta A_1\ \{Q_2\}]\!]^{\mathrm{exec}(\delta\, L', \delta\, U')}_{|G|_1,k} \wedge \delta\, L_2 \leq c_1 \leq \delta\, U_2$$

which proves our first goal: $c_1 - 0 \leq \delta\, U_2$.

We then need to show:

$$(v_1, \mathsf{let}\ \{x\} = \sigma_2\, t'_1\ \mathsf{in}\ \sigma_2\, t'_2) \in (\!|\ \{P\}\ \exists \vec{\gamma}_1.\delta\tau\ \{Q\}\ |\!)^{\mathrm{diff}(\delta(D' + (D_2 - L_2) - L_1 - L - c_{\mathsf{let}}))}_{G,k-k_1}$$

By the definition of the forcing evaluation we have:

$$v_1; H \Downarrow_f^{c_2,k_2} v_2; H_2$$

$$\frac{\sigma_2\, t'_1 \Downarrow^{c'_1,k'_1} v' \quad v'; H' \Downarrow_f^{c'_2,k'_2} v'_1; H'_1 \quad \sigma_2\, t'_2[v'_1/x] \Downarrow^{c'_3,k'_3} v'_2 \quad v'_2; H'_1 \Downarrow_f^{c'_4,k'_4} v'_3; H'_2}{\mathsf{let}\ \{x\} = \sigma_2\, t'_1\ \mathsf{in}\ \sigma_2\, t'_2; H' \Downarrow_f^{c'_1+c'_2+c'_3+c'_4+c_{\mathsf{let}},k'_1+k'_2+k'_3+k'_4+1} v'_3; H'_2}\ \textbf{f-bind}$$

So, unfolding the definition of the interpretation, for $G' \supseteq G$, $k' \leq k - k_1$, arbitrary heaps H, H' such that $H, H' \vDash_{G',k'} P$ and $k_2 < k' \leq k - k_1$, we must show:

1. $(H_2, H'_2) \vDash_{G',k'-k_2} Q$.
2. $(v_2, v'_3) \in (\!|\delta\tau|\!)_{G',k'-k_2}$.
3. $c_2 - (c'_1 + c'_2 + c'_3 + c'_4 + c_{\mathsf{let}}) \leq \delta(D' + (D_2 - L_2) - L_1 - L - c_{\mathsf{let}})$.

By induction hypothesis using Theorem 2, from the first premise, we have:

$$\sigma_2\, t'_1 \in [\![(\{P_1\}\ \exists \vec{\gamma}_1 : \delta A'_1\ \{Q_1\})]\!]^{e\ \mathrm{exec}(\delta\, L, \delta\, U)}_{|G|_2,k',(\delta\, L_1, \delta\, U_1)}$$

From this we have:

$$v' \in [\![\{P_1\}\ \exists \vec{\gamma}_1 : \delta A'_1\ \{Q_1\}]\!]^{\mathrm{exec}(\delta\, L, \delta\, U)}_{|G|_2,k'-k'_1} \wedge \delta\, L_1 \leq c'_1 \leq \delta\, U_1$$

which in turn tells us the following when we define $g_2 = |G|_2$:

$$H'_1 \vDash_{g_2,k'-k'_2} Q_1 \wedge v'_1 \in [\![\delta A'_1]\!]_{g_2,k'-k'_2} \wedge \delta\, L \leq c'_2 \leq \delta\, U$$

From this, we can conclude:

$$(\sigma_1[v'_1/x], \sigma_2[v'_1/x]) \in (\!|\delta(\Gamma, x : U(A'_1, A'_1))|\!)_{G_2,k}$$

By induction hypothesis, from the third premise instantiated with $(\sigma_1[v'_1/x], \sigma_2[v'_1/x]) \in (\!|\delta(\Gamma, x : U(A'_1, A'_1))|\!)_{G_2,k}$, we have:

$$(\sigma_1\, t_2[v'_1/x], \sigma_2\, t'_2[v'_1/x]) \in (\!|\{P \cup P_1\}\ \exists \vec{\gamma}_1.\delta\tau'\ \{Q\}|\!)^{e\ \mathrm{diff}(\delta\, D')}_{G_2,k,\delta\, D_2}$$

From this we have:

$$(v_1, v'_2) \in (\!|\{P \cup P_1\}\ \exists \vec{\gamma}_1.\delta\tau'\ \{Q\})|\!)^{\mathrm{diff}(\delta\, D')}_{G_2,k-k_1} \wedge c_1 - c'_3 \leq \delta\, D_2$$

We unfold $(v_1, v'_2) \in (\!|\{P \cup P_1\}\ \exists \vec{\gamma}_1.\delta\tau'\ \{Q\})|\!)^{\mathrm{diff}(\delta\, D')}_{G_2,k-k_1}$ and have: $(H, H'_1) \vDash_{G',k'} P \sqcup P_1$ by using the definition of the heap relation and the premise $dom(P) = dom(P_1)$. We choose $G_2 = G'$ and then have:

$$(v_2, v'_3) \in (\!|\delta\tau'|\!)_{G',k'-k_2} \wedge c_2 - c'_4 \leq \delta\, D' \wedge (H_2, H'_2) \vDash_{G',k'-k_2}$$

Now to conclude we can proceed as follows:

1. $(H_2, H_2') \vDash_{G',k'-k_2} Q$ is already proved by the previous conclusions.
2. $(v_2, v_3') \in (\!|\delta\tau|\!)_{G',k'-k_2}$ is already proved by the previous conclusions.
3. $c_2 - (c_1' + c_2' + c_3' + c_4' + c_{\mathsf{let}}) \leq \delta(D' + (D_2 - L_2) - L_1 - L - c_{\mathsf{let}})$ is proved by previous conclusions. □

When we have the fundamental theorems for both the unary and relational typing, we can easily have the soundness of our system.

Lemma 3 (Soundness for Costs).

1. If $\vdash_L^U t : A$ *and* $t \Downarrow^c v$, *then* $L \leq c \leq U$.
2. If $\vdash t_1 \ominus t_2 \lesssim D : \tau$ *and* $t_1 \Downarrow^{c_1,-} v_1$ *and* $t_2 \Downarrow^{c_2,-} v_2$, *then* $c_1 - c_2 \leq D$.

Proof. Proof of statement (1) follows directly by the fundamental theorem for uanry typing.
Proof of statement (2) follows directly by the fundamental theorem for relational typing. □

5 More examples

We discuss here five more examples demonstrating how we perform relational cost analysis on programs with arrays. To improve readability, we omit some annotations and use syntactic sugar. For example, we abbreviate let $\{x\} = t_1$ in t_2 to $x \leftarrow t_1$; t_2 and even to $t_1; t_2$ when x does not appear in t_2. We also shorten the type $U(A, A)$ to $U(A)$ and use if t then t_1 else t_2 as syntactic sugar for $\mathsf{case}(t, x.t_1, y.t_2)$ when x and y do not appear in t_1 and t_2, respectively.

In some of the examples, such as the Cooley–Tukey FFT Algorithm below, we add a dummy first argument of type unit to a recursive function. This allows us to make the recursive function polymorphic in index terms.

5.1 Cooley–Tukey FFT algorithm

The Fast Fourier Transform or FFT is a discrete Fourier Transform of a sequence of numbers. The Cooley–Tukey algorithm is a commonly used FFT algorithm that uses divide-and-conquer to split the sequence (Cooley & Tukey, 1965). Here, we perform a relational cost analysis of an imperative implementation of the algorithm that represents the sequence as an array and uses in-place updates. The objective of the analysis is to prove that the implementation is constant time—any two runs with arbitrary inputs take the same amount of time, assuming that array read/write operations and primitive numerical operations like addition and multiplication are constant time. Our implementation is shown in Figure 13.

The recursive function FFT uses divide-and-conquer. The variable x is the input array, y is another array used for temporary storage, m is the length of the range of the array to

```
fix FFT ().λx.λy.λm.λp.
  if 2 ≤ m then
    separate () x m y p ;
    FFT () x y (m/2) p ;
    FFT () x y (m/2) (p + m/2) ;
    loop () 0 m x p
  else
    return ()

fix separate ().λx.λm.λy.λp.
  sp() x m 0 y p ;
  cp() y x p (m + p)

fix loop ().λk.λm.λx.λp.
  if k < (m/2) then
    e ←read x (k + p) ;
    o ←read x (k + p + m/2) ;
    let w = exp(−2πk/m) in
    updt x (k + p) (e + w ∗ o) ;
    updt x (k + p + m/2) (e − w ∗ o) ;
    loop () (k + 1) m x p
  else
    return ()
```

Fig. 13. Code of FFT.

be transformed, and p is an index that specifies the starting index of the range of the array to 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 array respectively, using y as a scratchpad. The function separate also uses two helper functions—the function sp does the separation work using temporary storage and the function cp copies the separated part from the temporary storage back to the original array. We omit the code of sp and cp here; this code can be found in the Appendix.

Another helper function loop simulates a for-loop in which the input array x is updated to actually perform the Fourier transform.

Intuitively, this example is constant time (for arrays with fixed length) because the sequence of array accesses depends only on the array length, not on array contents. Formally, every function in this example relates to itself with relative cost 0. For instance, the relational types of separate and loop are shown below.

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

$$\vdash \mathsf{loop} \ominus \mathsf{loop} \lesssim 0 : \begin{array}{l} \mathsf{unit} \to \forall \gamma_1, \beta_1. \forall K, M, N, P.(P + M < N) \supset \\ \big(\mathsf{int}[K] \to \underset{\mathrm{diff}(0)}{\mathsf{int}[M]} \to \mathsf{Array}_{\gamma_1}[N]\ U(\mathsf{int}) \to \mathsf{int}[P] \to \\ \{\gamma_1 \to \beta_1\}\ \exists \_.\mathsf{unit}\ \{\gamma_1 \to \mathbb{N}\} \big) \end{array}$$

The constraint $P + M < N$ ensures that we never index the array past its end. Next, we show the relational type of FFT, again with relative cost 0.

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

The typing derivations for the above typing judgments are straightforward, so we omit them here.

We note that a different way to achieve the same result is to first compute precise lower and upper bounds on the unary cost of FFT and then show that they are, in fact, equal. This works, but is much more difficult because computing the precise unary cost of FFT is quite involved. We first establish the precise cost of the auxiliary functions. For example, we need to give the following unary types to separate and loop.

$$\vdash \mathsf{separate}: \begin{array}{l} \mathrm{unit} \to \forall \gamma_1, \gamma_2, M, N, P.(M + P < N) \supset \\ \big(\mathsf{Array}_{\gamma_1}[N]\ \mathrm{int} \to \mathrm{int}[M] \to \mathsf{Array}_{\gamma_2}[N]\ \mathrm{int} \to \mathrm{int}[P] \to \\ \{\gamma_1 \to \mathbb{N}, \gamma_2 \to \mathbb{N}\}\ \overset{\mathrm{exec}(4*M,4*M)}{\exists}\_.\mathrm{unit}\ \{\gamma_1 \to \mathbb{N}, \gamma_2 \to \mathbb{N}\} \big) \end{array}$$

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

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-switch).[3]

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 while the unary reasoning requires a precise computation of the actual complexity of all functions.

5.2 Naive string search

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

We represent strings as arrays of integers (storing the ASCII code of each character). In Figure 14, the function NSS takes as input, a "long" string s and a "short" string w in the form of 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 in p whether the substring is found at that position (1) or not (0). To do this, NSS uses helper function search, which is also shown in Figure 14.

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 between these two costs is a bound on the relative cost of search. Consider two runs of search on the same string s, the same index i and where the two ws agree on some prefix. The runs behave identically until we reach an index i where the two ws differ for the first time. We can use this index to give a better bound on the cost of search relative to itself.

[3] 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.

```
fix NSS (s).λw.λm.λl_s.λl_w.λp.
  if (m + l_w) ≤ l_s then
    search s w m 0 l_s l_w p ;
    NSS s w (m + 1) l_s l_w p
  else
    return ()
```

```
fix search (). λs. λw. λm. λi. λl_s. λl_w. λp.
  x ←read s (m + i) ;
  y ←read w i ;
  if i + 1 == l_w then
    if x == y then
      updt p m 1
    else
      updt p m 0
  else
    if x == y then
      search () s w m (i + 1) l_s l_w p
    else
      updt p m 0
```

Fig. 14. Code of NSS.

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 precondition 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:

$$\vdash \mathsf{search} \ominus \mathsf{search} \lesssim 0 : \begin{array}{l} \mathrm{unit} \to \forall \gamma_1, \gamma_2, \gamma_3, I, M, R, N, \beta_2, \beta_3. \\ (I < R < N \wedge M + I < N) \supset \\ \qquad \overset{\mathrm{diff}((R-1-min(MIN(\beta_2 \cap [I,\infty)),R-1))*r)}{\big(} T \to \{P, \gamma_3 \to \beta_3\}\ \exists \_.\mathrm{unit}\ \{P, \gamma_3 \to \beta_3 \cup \{M\}\}\ \big) \end{array}$$

where $T = \mathsf{Array}_{\gamma_1}[N]\ U(\mathrm{int}) \times \mathsf{Array}_{\gamma_2}[R]\ U(\mathrm{int}) \times \mathrm{int}[M] \times \mathrm{int}[I] \times \mathrm{int}[N] \times \mathrm{int}[R] \times \mathsf{Array}_{\gamma_3}[N]\ U(\mathrm{bool})$

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. The constraint $I < R$ guarantees that the search will not exceed the length of the array w and the constraint $R < N$ guarantees that w is shorter than s. The other constraint $M + I < N$ guarantees that the search will not exceed the length of the array s. The postcondition modifies only the set associated with γ_3, from β_3 to $\beta_3 \cup \{M\}$, representing that only the result array p will be overwritten. To account for the case where w is the same in the two executions we also add a lower bound $R - 1$ to the cost. The relative cost we establish here is more precise than the cost $(R - 1 - I) * r$ we would achieve with a non-relational analysis.

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-switch. To obtain our bound using unary worst- and best-case analysis, we need the precise unary type of search, which is available in the context only due to the rule r-fix-ext. The details of this proof are in our appendix.

```
fix msort (). λa.λb.λl.λu.
  if l ≥ u then
    return ()
  else
    msort a b l (l + ⌊(u − l)/2⌋) ;
    msort a b (l + ⌊(u − l)/2⌋ + 1) u ;
    merge a b l (l + ⌊(u − l)/2⌋) u

merge = λa.λb.λl.λm.λu.
  merge_lp l a l m (m + 1) u b ;
  copy l a b u
```

```
fix merge_lp (k) λa.λl_s.λl_e.λr_s.λr_e.λb.
  if l_s < l_e then
    if r_s < r_e then
      x ←read a l_s ;
      y ←read a r_s ;
      if x < y then
        updt b k x ;
        merge_lp (k + 1) a (l_s + 1) l_e r_s r_e b
      else
        updt b k y ;
        merge_lp (k + 1) a l_s l_e (r_s + 1) r_e b
    else
      x ←read a l_s ;
      updt b k x ;
      merge_lp (k + 1) a (l_s + 1) l_e r_s r_e b
  else
    if r_s < r_e then
      x ←read a r_s ;
      updt b k x ;
      merge_lp (k + 1) a l_s l_e (r_s + 1) r_e b
    else
      return ()
```

Fig. 15. Code of msort.

Using the above type of search, we can also obtain a tight bound on the cost of 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 $(N - M - R)$ multiplied by the relative cost of search.

5.3 Mergesort

As our next example, we consider an imperative version of mergesort. Similar to what we did for the mapi example in Section 2, we present two relational types for mergesort, corresponding to two different assumptions on the inputs. Consider the function msort in Figure 15 which sorts the elements of an array a from index l to u, using another array b as buffer. We use an auxiliary function merge that merges the two sorted partitions of the array. This function is defined in Figure 15. The function merge takes in input the array a, the buffer array b, the starting index l and ending index u, and additionally asks for a "midpoint" index m at which the array range is divided. It uses two helper functions: the function merge_{lp} which implements the standard merging process in a recursive way, and the function copy which copies the merged buffer array back to the working array. We omit the code of the function copy here (it can be found in the Appendix). We show just the code of the function $\mathsf{merge}_{\mathsf{lp}}$. The argument k is the position where to store the merged element in the buffer array b, the four arguments l_s, l_e, r_s, and r_e separate the array a into two portions, the left one is specified by the variables l_s and l_e representing its starting and ending indices, whereas the right portion is specified by r_s and r_e. The idea underlying this function is to check if there is an element available in the left portion and right portion using conditionals, then compare two elements one from each side, before storing the smaller one in the buffer array at the position k. Then this pointer is updated, and the process is repeated recursively.

The first relational type we want to show for msort assumes that the two arrays, which are input to the two runs, are equal in the range $[l, u]$. Intuitively, the fact that we have the same array elements in the range $[l, u]$ implies that we have the same execution path, and so we can prove that the relative cost is 0.

$$\begin{aligned}\vdash \mathsf{msort} \ominus \mathsf{msort} \lesssim 0 : \; & \mathrm{unit} \to \forall l, u, \beta, \gamma, \gamma', N. \\ & l \le u \le N \wedge \beta \cap [l, u] = \emptyset \supset \\ & \big(\mathsf{Array}_{\gamma}[N]\ U(\mathrm{int}) \to \mathsf{Array}_{\gamma'}[N]\ U(\mathrm{int}) \to \mathrm{int}[l] \to \\ & \mathrm{int}[u] \to \{\gamma \to \beta, \gamma' \to \mathbb{N}\}\ \underset{\mathrm{diff}(0)}{\exists \_.\mathrm{unit}}\ \{\gamma \to \beta, \gamma' \to \mathbb{N}\} \big)\end{aligned}$$

The assumption that the two arrays from the two runs are equal in the range $[l, u]$ is represented by the constraint $(\beta \cap [l, u] = \emptyset)$, where β is the set containing indices at which the two arrays can differ. Using this assumption, we can derive relational types asserting relative cost 0 for both $\mathsf{merge}_{\mathsf{lp}}$ and copy, and thus for merge. We show the types of $\mathsf{merge}_{\mathsf{lp}}$ and merge below.

$$\begin{aligned}\vdash \mathsf{merge}_{\mathsf{lp}} \ominus \mathsf{merge}_{\mathsf{lp}} \lesssim 0 : \; & \forall N, l_s, l_e, r_s, r_e, K, \gamma, \gamma', \beta. \\ & l_s \le l_e \le r_s \le r_e \le N \wedge \beta \cap [K, r_e] = \emptyset \supset \\ & \big(\mathrm{int}[K] \to \mathsf{Array}_{\gamma}[N]\ U(\mathrm{int}) \to \mathrm{int}[l_s] \to \mathrm{int}[l_e] \to \\ & \mathrm{int}[r_s] \to \mathrm{int}[r_e] \to \mathsf{Array}_{\gamma'}[N]\ U(\mathrm{int}) \to \\ & \{\gamma \to \beta, \gamma' \to \mathbb{N}\}\ \underset{\mathrm{diff}(0)}{\exists \_.\mathrm{unit}}\ \{\gamma \to \beta, \gamma' \to \mathbb{N} \setminus [l, u]\} \big)\end{aligned}$$

$$\begin{aligned}\vdash \mathsf{merge} \ominus \mathsf{merge} \lesssim 0 : \; & \forall l, m, u, \beta, \gamma, \gamma', N. \\ & l \le u \le N \wedge \beta \cap [l, u] = \emptyset \supset \\ & \big(\mathsf{Array}_{\gamma}[N]\ U(\mathrm{int}) \to \mathsf{Array}_{\gamma'}[N]\ U(\mathrm{int}) \to \mathrm{int}[l] \to \\ & \mathrm{int}[m] \to \mathrm{int}[u] \to \{\gamma \to \beta, \gamma' \to \mathbb{N}\}\ \underset{\mathrm{diff}(0)}{\exists \_.\mathrm{unit}}\ \{\gamma \to \beta, \gamma' \to \mathbb{N}\} \big)\end{aligned}$$

Notice that we also restrict the value of u to be between the starting index l and the array length of a. Moreover, notice that the precondition and the postcondition in merge are equal. This follows again from the assumption we have for this typing: since the two arrays coincide in the range $[l, u]$ before the sort, they will also coincide after the sort, so no new element will be added to β.

Next, we show a more general relational type for msort that does not assume that the two input arrays are equal in the range $[l, u]$. We start a general type for the function $\mathsf{merge}_{\mathsf{lp}}$. Without the assumption, we may have different execution paths in two runs of this function. We use the rule r-switch to switch to the unary analysis to get the relative cost and the rule R-X to use the subtyping rule r-rum for the proper monadic type. Using this approach we get the following relational type for $\mathsf{merge}_{\mathsf{lp}}$:

$$\begin{aligned}\vdash \mathsf{merge}_{\mathsf{lp}} \ominus \mathsf{merge}_{\mathsf{lp}} \lesssim 0 : \; & \forall K, l_s, l_e, r_s, r_e, \beta, \gamma, \gamma', N. \\ & l_s \le l_e \le r_s \le r_e \le N \wedge K \le r_e \supset \\ & \big(T \to \{\gamma \to \beta, \gamma' \to \mathbb{N}\}\ \underset{\mathrm{diff}(\max(l_e - l_s, r_e - r_s))}{\exists \_.\mathrm{unit}}\ \{\gamma \to \beta, \gamma' \to \mathbb{N}\} \big)\end{aligned}$$

where $T = \mathrm{int}[K] \times \mathsf{Array}_{\gamma}[N]\ U(\mathrm{int}) \times \mathrm{int}[l_s] \times \mathrm{int}[l_e] \times \mathrm{int}[r_s] \times \mathrm{int}[r_e] \times \mathsf{Array}_{\gamma'}[N]\ U(\mathrm{int})$

```
fix ISort (). λs.λi.λl_s.
  if i < l_s then
    a ←read s i ;
    b ← insert () s a 0 i ;
    ISort () s (i + 1) l_s
  else
    return ()

fix insert (). λs.λa.λx.λi.
  b ←read s x ;
  if a ≥ b then
    insert () s a (x + 1) i ;
  else
    shift () s x (i − 1) ;
    updt s x a
fix shift (). λs.λidx.λi.
  if idx ≤ i then
    c ←read s i ;
    updt s i + 1 c ;
    shift () s idx (i − 1)
  else
    return ()
```

Fig. 16. Code of ISort.

So, we have that the relative cost of two executions of $\mathsf{merge}_{\mathsf{lp}}$ is $\max(l_e - l_s, r_e - r_s)$. (The function copy does not contain branches and so it does not introduce a nontrivial relative cost.)

Using the relational type described above we can derive the following relational type for merge:

$$\begin{aligned}\vdash \mathsf{merge} \ominus \mathsf{merge} \lesssim 0 : \; & \forall l, m, u. \forall \beta. \forall \gamma, \gamma'. \forall N : \mathbb{N}.\ l \le m \le u \le N \supset \\ & \mathsf{Array}_{\gamma}[N]\ U(\mathrm{int}) \to \mathsf{Array}_{\gamma'}[N]\ U(\mathrm{int}) \to \mathrm{int}[l] \to \mathrm{int}[m] \\ & \xrightarrow{\mathrm{diff}(max(m-l, u-(m+1)))} \\ & \to \mathrm{int}[u] \to \{\gamma \to \beta, \gamma' \to \mathbb{N}\}\ \exists \_.\mathrm{unit}\ \{\gamma \to \beta \cup [l, u], \gamma' \to \mathbb{N}\}\end{aligned}$$

From this, we can give a relational type to msort.

$$\begin{aligned}\vdash \mathsf{msort} \ominus \mathsf{msort} \lesssim 0 : \; & \forall l, u, \beta, \gamma, \gamma', N.\ l \le u \le N \supset \\ & \big(\mathsf{Array}_{\gamma}[N]\ U(\mathrm{int}) \to \mathrm{int}[l] \to \mathrm{int}[u] \to \mathsf{Array}_{\gamma'}[N]\ U(\mathrm{int}) \\ & \xrightarrow{\mathrm{diff}(Q(u-l+1, |\beta \cap [l,u]|))} \\ & \to \{\gamma \to \beta, \gamma' \to \mathbb{N}\}\ \exists \_.\mathrm{unit}\ \{\gamma \to \beta \cup \{l, u\}, \gamma' \to \mathbb{N}\}\big)\end{aligned}$$

where $Q(n, \alpha) = \Sigma_{i=0}^{H} h(\lceil 2^{i-1} \rceil) \cdot \min(\alpha, 2^{H-i})$ and $H = \lceil \log_2(n) \rceil$. To prove that msort has this relative cost, we need some algebraic properties of the recurrence relation Q, which we postpone to the appendix. While the cost looks complicated, we prove that it is in $O(n \cdot (1 + \log_2(|\beta \cap [l, u]|)))$. This is consistent with a previous relational analysis of a *non-imperative* variant of msort (Çiçek *et al.*, 2017).

5.4 Inplace insertion sort

Our next example, *inplace insertion sort*, implements the insertion sort algorithm without any temporary array. This is an involved example which combines most of the ideas we discussed in other examples, in order to derive a precise relative cost. 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 shown in Figure 16. The function

ISort sorts the input array s in the range $[i, l_s]$. 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 determined by the auxiliary function insert shown in Figure 16.

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. The input arguments x and i specify the range in the array to insert into.

The function insert uses a helper function shift, which performs the shift operation. It is not hard to observe that the auxiliary function shift, which shifts the elements in the range $[idx, i]$ backward by one index, uses one read operation and one update operation at every index. 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 a unit cost each, the unary type of insert is as follows:

$$\vdash \mathsf{insert} : \begin{array}{l} \mathrm{unit} \to \forall \gamma_1, N, X, I.\, X \le N \wedge I \le N \supset \\ \big(\mathsf{Array}_{\gamma_1}[N]\ \mathrm{int} \to \mathrm{int} \to \mathrm{int}[X] \to \mathrm{int}[I] \to \{\gamma_1 \to \mathbb{N}\}\ \exists\_.\mathrm{unit}^{\mathrm{exec}(I-X+1,2*(I-X)+2)}\ \{\gamma_1 \to \mathbb{N}\}\big) \end{array}$$

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 $[X, I]$ and the elements "a" being inserted also agree, then insert's cost relative to itself is 0. The corresponding relational type is shown below, where the constraint $\beta_1 \cap [X, I] = \emptyset$ describes our aforementioned assumption.

$$\vdash \mathsf{insert} \ominus \mathsf{insert} \lesssim 0 : \begin{array}{l} \mathrm{unit} \to \forall \gamma_1, \beta_1, N, A, X, I. \\ X \le N \wedge I \le N \wedge \beta_1 \cap [X, I] = \emptyset \supset \\ \big(\mathsf{Array}_{\gamma_1}[N]\ U(\mathrm{int}) \to \mathrm{int} \to \mathrm{int}[X] \to \mathrm{int}[I] \to \\ \{\gamma_1 \to \beta_1\}\ \exists\_.\mathrm{unit}^{\mathrm{diff}(0)}\ \{\gamma_1 \to \beta_1\}\big) \end{array}$$

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-split). 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-switch since control flow may differ in the two runs. We need the rule r-fix-ext to allow us to switch back to unary typing when the control flow actually differs at some point. Using this idea, we obtain a very precise relational cost for ISort.

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

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.

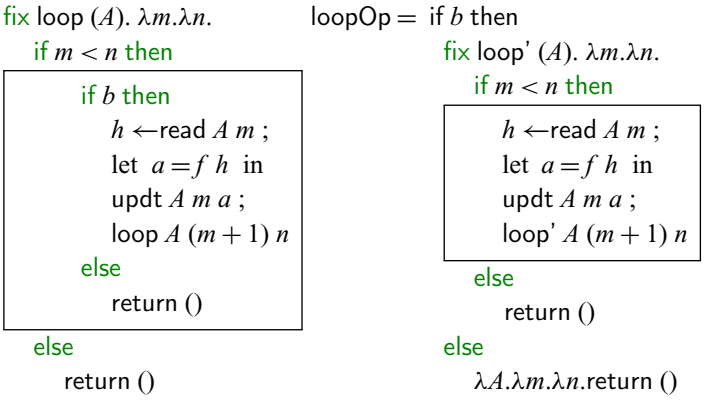

Fig. 17. Code of loop unswitching.

5.5 Loop unswitching

In this example, we examine a classical technique from compiler optimization, loop unswitching. We show how ARel can provide a more precise relative cost than the standard worst-case/best-case analysis when dealing with two programs that are not structurally similar.

In Figure 17, we present a function loop which operates a simple loop over an input array A, from index m, to the end of the array n (of type int[N]). Inside the loop, we have an if statement whose then branch reads the value in the array at index m and does some pure computation $f\ h$ on the value h just read, and then stores the result back to the array, before moving on to the next iteration. For simplicity, the else branch does nothing interesting.

This program can be optimized by pulling out the if conditional from the function body, as we can see in the right part of Figure 17. We call the optimized program loopOp. Suppose that the original program loop and the optimized program loopOp operate on the same array A, run the same pure computation f inside the loop, and share the boolean input b in two runs. Intuitively, the relative cost of loop and loopOp is upper bounded by N (the size of the array) when we count one unit cost for elimination forms because loop checks b at each iteration, while the optimized one checks b only once. Using unary analysis, we can obtain the following unary types.

$$\vdash \lambda f.\lambda b.\mathsf{loop}\ :\ \begin{array}{l}(A \to \mathrm{unit}) \to \mathrm{bool} \to \forall \gamma_1, N, M.\\ \mathsf{Array}_{\gamma_1}[N]\ A \to \mathrm{int}[M] \to \mathrm{int}[N] \to \{\gamma_1 \to \mathbb{N}\}\ \overset{\mathrm{exec}(1,4*(N-M)+1)}{\exists \_.\mathrm{unit}}\ \{\gamma_1 \to \mathbb{N}\}\end{array}$$

$$\vdash \lambda f.\lambda b.\mathsf{loopOp}\ :\ \begin{array}{l}(A \to \mathrm{unit}) \to \mathrm{bool} \to \forall \gamma_1, N, M.\\ \mathsf{Array}_{\gamma_1}[N]\ A \to \mathrm{int}[M] \to \mathrm{int}[N] \to \{\gamma_1 \to \mathbb{N}\}\ \overset{\mathrm{exec}(1,3*(N-M)+1)}{\exists \_.\mathrm{unit}}\ \{\gamma_1 \to \mathbb{N}\}\end{array}$$

In both types above loop, both the precondition and the postcondition map γ_1 to $\mathbb{N}$, which is consistent with our assumption that the array will be updated during the execution of either loop or loopOp. With the help of the subtyping rule s-rum, we obtain the cost of loop relative to loopOp, which is $4 * (N - M)$. When we start the loop from the beginning, which means $M = 0$, then the relative cost is bounded by $4 * N$, which is higher than the N we expected.

We can do better if we use the relational analysis and, in particular, the relational asynchronous rule r-e-case in Figure 6 on p. 20 (recall that if t then t_1 else t_2 is syntactic sugar for the construct case $(t, x.t_1, y.t_2)$). To be precise, we can derive a simplified asynchronous rule r-e-if from r-e-case.

$$\frac{\Sigma;\Delta;\Phi;|\Gamma|_2 \vdash^{U_1}_{L_1} t' : \mathrm{bool} \qquad \Sigma;\Delta;\Phi;\Gamma \vdash t \ominus t'_1 \lesssim D_2 : \tau \qquad \Sigma;\Delta;\Phi;\Gamma \vdash t \ominus t'_2 \lesssim D_2 : \tau}{\Sigma;\Delta;\Phi;\Gamma \vdash t \ominus \mathsf{if}\ t'\ \mathsf{then}\ t'_1\ \mathsf{else}\ t'_2 \lesssim D_2 - L_1 - c_{case} : \tau}\ \textbf{r-e-if}$$

This asynchronous rule allows us to relationally type loop relative to the inner recursive function loop' inside loopOp. When we compare the bodies of loop and loop', and type the "then" branch of the first if conditional (if $m < n$ ...), we come across structurally dissimilar pieces of codes: the piece of code inside the box in loop and the piece of code inside the box in loop'. Here, we use the r-e-if rule above. We want to avoid comparing the "else" branch return () in the loop with the boxed part of loop', when we know that the condition b is the same in the two runs. To this end, we refine our unary boolean type bool to bool$[B]$ and our relational boolean type bool_r to $\mathrm{bool}_r[B]$, where $B \in \{\mathrm{true}, \mathrm{false}\}$. (The erasure operation over the refined relational type is defined as $|\mathrm{bool}_r[B]|_i = \mathrm{bool}[B]$).

With this, we can relationally type the two programs with a precise relative cost as follows:

$$\vdash \lambda f.\lambda b.\mathsf{loop} \ominus \lambda f.\lambda b.\mathsf{loopOp} \lesssim 0 : \begin{array}{l}(U(A) \to \mathrm{unit}) \to \forall B :: \{\mathrm{true}, \mathrm{false}\}.\mathrm{bool}_r[B] \\ \xrightarrow{\mathrm{diff}(-1)} \forall \gamma_1, N, M.\ \mathsf{Array}_{\gamma_1}[N]\ U(A) \xrightarrow{\mathrm{diff}(N-M)} \mathrm{int}[M] \\ \to \mathrm{int}[N] \to \{\gamma_1 \to \mathbb{N}\}\ \exists\_.\mathrm{unit}\ \{\gamma_1 \to \mathbb{N}\}\end{array}$$

The negative cost -1 in the type comes from checking b at the beginning in the optimized version. The relative cost embedded in the monadic type reflects the difference between the boxed code in loop and the boxed code in loop' for every iteration.

6 Bidirectional type checking

In this section we discuss the implementation of ARel. Implementing ARel naively results in an immediate challenge: the relational aspects of ARel create non-determinism in the type system. This non-determinism comes from two aspects. First, some of the rules are not syntax-directed. For example, in the split rule r-split in Figure 5, we can choose any constraint to split on (and this is an infinite choice); we can apply the switch rule r-switch in Figure 6 anywhere; in the rule r-fix-ext shown in Figure 7, we have to guess the unary types of the functions (again an infinite choice); simultaneously, there are two rules for every array operator, which makes a choice between them non-deterministic because $\Box\,\tau$ is a subtype of τ. Second, the existence of the modality $\Box$ and its interaction between other types makes the algorithmization of relational subtyping quite tricky. To be concrete, it is challenging to define relational subtyping algorithmically while preserving transitivity of the subtyping relation.

To address these challenges, inspired by the work of Çiçek *et al.* (2019), we introduce a two-step method to algorithmize ARel. We first introduce an intermediate core language denoted ARelCore corresponding to an annotated, syntax-directed, version of ARel. An elaboration of ARel to ARelCore eliminates the non-determinism of ARel in two steps. First, it adds annotations on term constructors to resolve non-syntax-directed typing rules.

$$
\begin{array}{ll}
\textbf{Terms} & t ::= \cdots \mid \mathsf{return}\ t \mid \mathsf{alloc}\ t\ t \mid \mathsf{alloc}_{\Box}\ t t \mid \mathsf{updt}\ t\ t\ t \mid \mathsf{updt}_{\Box}\ t\ t\ t \\
& \mid \mathsf{read}\ t\ t \mid \mathsf{read}_{\Box}\ t\ t \mid \mathsf{let}\ \{x\} = t\ \mathsf{in}\ t_1 \mid \mathsf{switch}\ t \mid \mathsf{NC}\ t \mid \mathsf{split}\ t\ \mathsf{with}\ C \\
& \mid \mathsf{contra}\ t \mid \mathsf{FIXEXT}\ t\ \mathsf{with}\ A \\
\textbf{Values} & v ::= \cdots \mid \mathsf{return}\ t \mid \mathsf{alloc}\ t\ t \mid \mathsf{alloc}_{\Box}\ t t \mid \mathsf{updt}\ t\ t\ t \mid \mathsf{updt}_{\Box}\ t\ t\ t \\
& \mid \mathsf{read}\ t\ t \mid \mathsf{read}_{\Box}\ t\ t \mid \mathsf{let}\ \{x\} = t\ \mathsf{in}\ t_1
\end{array}
$$

Fig. 18. ARelCore syntax of values and terms.

Second, it eliminates relational subtyping in the core language to avoid its corresponding non-determinism. The elaboration replaces relational subtyping with explicit coercions in the core language. The language ARelCore is sound and complete with respect to ARel, modulo finding the right elaboration. We then develop bidirectional type checking for ARelCore as the main component of our implementation, and heuristics for elaboration.

This section is organized as follows. We first present the ARelCore language through its syntax and typing rules. Then, we demonstrate the elaboration from ARel to ARelCore by showing some of the elaboration rules. We also justify its soundness and completeness. Then, after the introduction of ARelCore and the elaboration, we focus on the algorithmization and discuss how to construct a bidirectional type system with respect to ARelCore.

6.1 ARelCore

The difficulties encountered when trying to algorithmize ARel disappear when we instead algorithmize a core language suitable for bidirectional type checking. This core language can be seen as a theoretical medium for algorithmization, which is sound and complete with respect to our declarative type system ARel.

6.1.1 Syntax

The syntax of ARelCore is an extension of the syntax of ARel with the annotations and corresponding term constructs shown in Figure 18. These annotations and new constructs make ARelCore's type system syntax-directed. For instance, the term construct $\mathsf{split}\ t\ \mathsf{with}\ C$ can be used to mark a use of the rule r-split, which splits on the constraint C in the type checking of t. The use of the rule r-fix-ext of Figure 5 is indicated by the construct $\mathsf{FIXEXT}\ f(x).t\ \mathsf{with}\ A$ that provides the unary type A of $\mathsf{fix}f(x).t$. Similarly, the constructs $\mathsf{NC}\ t$ and $\mathsf{switch}\ t$ specify the use of the rule r-nc which introduces the $\Box$ modality, and the switch rule r-split, respectively. The construct $\mathsf{contra}\ t$ allows us to assign an arbitrary type to the expression t if there is a contradiction in our constraint environment. In addition, we introduce two variants of each array operation, for example, alloc and $\mathsf{alloc}_{\mathsf{b}}$ correspond to the two rules r-alloc and r-allocb, respectively.

6.1.2 Typing rules of ARelCore

The main purpose of ARelCore is to provide a syntax-directed type system that we can use as the target for a translation of ARel programs. Like ARel, ARelCore has two typing judgments. We have the unary judgment $\Sigma; \Delta; \Phi_a; \Omega \vdash_L^U t :^c A$ that assigns to a single term t a type A, and bounds L and U to t's evaluation cost, under the environment $\Sigma; \Delta; \Phi_a; \Omega$. We also have relational typing $\Sigma; \Delta; \Phi_a; \Gamma \vdash t_1 \ominus t_2 \lesssim D :^c \tau$, which says that two terms

$$\frac{\Sigma;\Delta;\Phi_a;\Omega \vdash^{U}_{L} t :^{c} A \qquad \Delta;\Phi_a \models A \sqsubseteq A' \qquad \Delta;\Phi_a \models L' \leq L \qquad \Delta;\Phi_a \models U \leq U'}{\Sigma;\Delta;\Phi_a;\Omega \vdash^{U'}_{L'} t :^{c} A'}\ \textbf{c-}\sqsubseteq$$

$$\frac{\Sigma;\Delta;\Phi_a;\Gamma \vdash t \ominus t' \lesssim D :^{c} \tau \qquad \Delta;\Phi_a \models \tau \equiv \tau' \qquad \Delta;\Phi_a \models D \leq D'}{\Sigma;\Delta;\Phi_a;\Gamma \vdash t \ominus t' \lesssim D' :^{c} \tau'}\ \textbf{c-r-}\equiv$$

$$\frac{\Delta;\Phi_a;|\Gamma| \vdash^{U_1}_{L_1} t_1 :^{c} A \qquad \Delta;\Phi_a;|\Gamma| \vdash^{U_2}_{L_2} t_2 :^{c} A}{\Sigma;\Delta;\Phi_a;\Gamma \vdash \mathsf{switch}\ t_1 \ominus \mathsf{switch}\ t_2 \lesssim U_1 - L_2 :^{c} U\ A}\ \textbf{c-switch}$$

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

$$\frac{\begin{array}{c}\Sigma;\Delta;\Phi_a;\Gamma \vdash t_1 \ominus t_1' \lesssim D_1 :^{c} \mathrm{int}[I] \\ \Sigma;\Delta;\Phi_a;\Gamma \vdash t_2 \ominus t_2' \lesssim D_2 :^{c} \Box\tau \qquad \gamma\ \mathsf{fresh} \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}\end{array}}{\Sigma;\Delta;\Phi_a;\Gamma \vdash \mathsf{alloc}_{\Box}\ t_1\ t_2 \ominus \mathsf{alloc}_{\Box}\ t_1'\ t_2' \lesssim 0 :^{c} \{P\}\ \exists\gamma.\ \mathsf{Array}_{\gamma}[I]\ \tau\ \{P \star \gamma \to \emptyset\}}\ \begin{array}{c}\textbf{c-r-allocB} \\ \mathrm{diff}(D_1+D_2)\end{array}$$

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

$$\frac{\begin{array}{c}\Sigma;\Delta;\Phi_a;\Gamma \vdash t_1 \ominus t_1' \lesssim D_1 :^{c} \mathsf{Array}_{\gamma}[I]\ \tau \\ \Sigma;\Delta;\Phi_a;\Gamma \vdash t_2 \ominus t_2' \lesssim D_2 :^{c} \mathrm{int}[I'] \qquad \Delta;\Phi_a \models I' \leq I \qquad I' \notin \beta \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}\end{array}}{\Sigma;\Delta;\Phi_a;\Gamma \vdash \mathsf{read}_{\Box}\ t_1\ t_2 \ominus \mathsf{read}_{\Box}\ t_1'\ t_2' \lesssim 0 :^{c} \{P \star \gamma \to \beta\}\ \exists\_.\Box\tau\ \{P \star \gamma \to \beta\}}\ \begin{array}{c}\textbf{c-r-readb} \\ \mathrm{diff}(D_1+D_2)\end{array}$$

$$\frac{\begin{array}{c}\Sigma;\Delta;\Phi_a;\Gamma \vdash t_1 \ominus t_1' \lesssim D_1 :^{c} \mathsf{Array}_{\gamma}[I]\ \tau \qquad \Sigma;\Delta;\Phi_a;\Gamma \vdash t_2 \ominus t_2' \lesssim D_2 :^{c} \mathrm{int}[I'] \\ \Sigma;\Delta;\Phi_a;\Gamma \vdash t_3 \ominus t_3' \lesssim D_3 :^{c} \tau \qquad \Delta;\Phi_a \models I' \leq I \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}\end{array}}{\Sigma;\Delta;\Phi_a;\Gamma \vdash \mathsf{updt}\ t_1\ t_2\ t_3 \ominus \mathsf{updt}\ t_1'\ t_2'\ t_3' \lesssim 0 :^{c} \{P \star \gamma \to \beta\}\ \exists\_.\mathrm{unit}\ \{P \star \gamma \to \beta \cup \{I'\}\}}\ \begin{array}{c}\textbf{c-r-update} \\ \mathrm{diff}(D_1+D_2+D_3)\end{array}$$

$$\frac{\begin{array}{c}\Sigma;\Delta;\Phi_a;\Gamma \vdash t_1 \ominus t_1' \lesssim D_1 :^{c} \mathsf{Array}_{\gamma}[I]\ \tau \qquad \Sigma;\Delta;\Phi_a;\Gamma \vdash t_2 \ominus t_2' \lesssim D_2 :^{c} \mathrm{int}[I'] \\ \Sigma;\Delta;\Phi_a;\Gamma \vdash t_3 \ominus t_3' \lesssim D_3 :^{c} \Box\tau \qquad \Delta;\Phi_a \models I' \leq I \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}\end{array}}{\begin{array}{l}\Sigma;\Delta;\Phi_a;\Gamma \vdash \mathsf{updt}_{\Box}\ t_1\ t_2\ t_3 \ominus \mathsf{updt}_{\Box}\ t_1'\ t_2'\ t_3' \lesssim 0 :^{c} \\ \qquad \{P \star \gamma \to \beta\}\ \exists\_.\mathrm{unit}\ \{P \star \gamma \to \beta \setminus \{I'\}\}\end{array}}\ \begin{array}{c}\textbf{c-r-updateb} \\ \mathrm{diff}(D_1+D_2+D_3)\end{array}$$

Fig. 19. Selected typing rules of ARelCore.

t_1 and t_2 are related at type τ, and that their relative cost is bounded by D, under the context Γ.

We present in Figure 19 a selection of the typing rules for ARelCore. We select rules for terms that may incur non-determinism in ARel, and we show how the non-determinism can be resolved at the syntactic level in ARelCore. As an example, to resolve the non-determinism caused by array-based operations we now have two different constructors for each operation and two different rules for them. For instance, for the read operation, we have a pair of terms read t t and read$_{\Box}$ t t which correspond to the two typing rules c-r-read and c-r-readb, respectively. These typing rules of ARelCore are similar to their counterparts in ARel. Consider the rule c-r-read. It is easy to see that it has a structure similar to that of the rule r-read in Figure 7. For example, the same premise $\Delta;\Phi \vDash I' \leq$

I guarantees the array bound limit. Such similarity can also be found between the rule c-r-readb and the rule r-readb, and other pairs of array-related rules.

Subtyping is managed in a different way. For unary subtyping, we have the rule **c-$\sqsubseteq$** which has a form similar to its counterpart in ARel. This is mainly because the unary subtyping relation allows a direct algorithmization. On the other hand, relational subtyping in ARelCore is limited to type *equivalence* as in rule c-r-$\equiv$. ARel's subtyping is then simulated using explicit coercion functions. The main reason for this approach comes from the fact that the modalities $\Box$ and U prevent easy algorithmization. We show that every relational subtyping can be simulated by applying a coercion expression in ARelCore.

Lemma 4 (Simulation of Binary Subtyping in ARelCore). *If* $\Delta; \Phi \models \tau \sqsubseteq \tau'$ *then there exists a term* t *in ARelCore, which we also denote* $coerce_{\tau,\tau'}$*, such that* $\Delta; \Phi; \cdot \vdash t \ominus t \lesssim 0 :^c \tau \xrightarrow{diff(0)} \tau'$.

Proof. By induction on the given subtyping derivation. □

6.2 Elaboration

Now that we have ARelCore, we want to show that we can elaborate any well-typed program from ARel into a well-typed program in ARelCore. The unary elaboration judgment $\Delta; \Phi_a; \Omega \vdash^U_L t \rightsquigarrow t^* : A$ represents the translation of a term t in ARel to its counterpart t^* in ARelCore. Both terms have the unary type A and bounds L, U in the contexts $\Sigma; \Delta; \Phi_a; \Omega$. In the same vein, the relational elaboration judgment has the shape $\Sigma; \Delta; \Phi_a; \Gamma \vdash t_1 \ominus t_2 \rightsquigarrow t_1^* \ominus t_2^* \lesssim D : \tau$. It represents the translation of a pair of terms (t_1, t_2) to (t_1^*, t_2^*). Both pairs are typed at τ in the given contexts, and both pairs have the same relative cost bound D. We show a selection of the unary and relational elaboration rules in Figure 20. In the unary elaboration subsumption rule e-u-sub and the relational rule e-r-sub, we see the two different approaches to subtyping we discussed previously. The rule e-u-sub preserves the unary subtyping relation during the translation. As we mentioned before, ARelCore eliminates relational subtyping using coercions, which are provided by Lemma 4. In the rule e-r-sub, the ARelCore term t' is such a coercion. Elaboration rules for array operations have structures that are quite similar to the structure of the rules in ARel, except that they map terms to their annotated versions.

Elaboration is sound and complete in the sense of the following theorems.

Theorem 5 (Soundness of ARelCore & Type Preservation of Embedding). *The following holds.*

1. *If* $\Sigma; \Delta; \Phi; \Omega \vdash^U_L t \rightsquigarrow t^* : A$*, then* $\Sigma; \Delta; \Phi; \Omega \vdash^U_L t^* :^c A$ *and* $\Sigma; \Delta; \Phi; \Omega \vdash^U_L t : A$.
2. *If* $\Sigma; \Delta; \Phi; \Gamma \vdash t_1 \ominus t_2 \rightsquigarrow t_1^* \ominus t_2^* \lesssim D : \tau$*, then* $\Sigma; \Delta; \Phi; \Gamma \vdash t_1^* \ominus t_2^* \lesssim D :^c \tau$ *and* $\Sigma; \Delta; \Phi; \Gamma \vdash t_1 \ominus t_2 \lesssim D : \tau$.

Proof. By simultaneous induction on the given elaboration derivations. □

$$\frac{\Sigma;\Delta;\Phi_a;\Omega \vdash^{U_1}_{L_1} t_1 \rightsquigarrow t_1^* : \mathsf{Array}_\gamma[I]\ A \qquad \Sigma;\Delta;\Phi_a;\Omega \vdash^{U_2}_{L_2} t_2 \rightsquigarrow t_2^* : \mathrm{int}[I'] \qquad \Delta;\Phi_a \models I' \leq I \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}}{\Sigma;\Delta;\Phi_a;\Omega \vdash^{0}_{0} \mathsf{read}\ t_1\ t_2 \rightsquigarrow \mathsf{read}\ t_1^*\ t_2^* : \overset{\mathrm{exec}(L_1+L_2+L_r,U_1+U_2+U_r)}{\{P\}\ \exists\gamma.A\ \{P\}}}\ \textbf{e-u-read}$$

$$\frac{\begin{array}{c}\Sigma;\Delta;\Phi_a;\Gamma \vdash t_1 \ominus t_1' \rightsquigarrow {t_1}^* \ominus {t_1'}^* \lesssim D_1 : \mathsf{Array}_\gamma[I]\ \tau \\ \Sigma;\Delta;\Phi_a;\Gamma \vdash t_2 \ominus t_2' \rightsquigarrow {t_2}^* \ominus {t_2'}^* \lesssim D_2 : \mathrm{int}[I'] \\ \Delta;\Phi_a \models I' \leq I \qquad \Delta;\Phi_a \models I' \notin \beta \qquad \Sigma;\Delta \vdash P\ \mathsf{wf}\end{array}}{\Sigma;\Delta;\Phi_a;\Gamma \vdash \mathsf{read}\ t_1\ t_2 \ominus \mathsf{read}\ t_1'\ t_2' \rightsquigarrow \mathsf{read}_\Box\ t_1^*\ t_2^* \ominus \mathsf{read}_\Box\ t_1'^*\ t_2'^* \lesssim 0 : \overset{\mathrm{diff}(D_1+D_2)}{\{P \star \gamma \to \beta\}\ \exists\gamma.\Box\tau\ \{P \star \gamma \to \beta\}}}\ \textbf{e-r-readb}$$

$$\frac{\Sigma;\Delta;\Phi_a;\Omega \vdash^{U}_{L} t \rightsquigarrow t^* : A \qquad \Delta;\Phi_a \models A \sqsubseteq A' \qquad \Delta;\Phi_a \models L' \leq L \qquad \Delta;\Phi_a \models U \leq U'}{\Delta;\Phi_a;\Omega \vdash^{U'}_{L'} t \rightsquigarrow t^* : A'}\ \textbf{e-u-sub}$$

$$\frac{\Sigma;\Delta;\Phi_a;\Gamma \vdash t_1 \ominus t_2 \rightsquigarrow {t_1}^* \ominus {t_2}^* \lesssim D : \tau \qquad \Delta;\Phi_a \models \tau \sqsubseteq \tau' \qquad t' = \mathrm{coerce}_{\tau,\tau'} \qquad \Delta;\Phi_a \models D \leq D'}{\Sigma;\Delta;\Phi_a;\Gamma \vdash t_1 \ominus t_2 \rightsquigarrow t'\ t_1^* \ominus t'\ t_2^* \lesssim D' : \tau'}\ \textbf{e-r-sub}$$

Fig. 20. Selected unary and relational elaboration rules.

Theorem 6 (Completeness of ARelCore). *The following holds.*

1. *If* $\Sigma;\Delta;\Phi;\Omega \vdash^{U}_{L} t : A$ *then* $\exists\, t^*$ *such that* $\Sigma;\Delta;\Phi;\Omega \vdash^{U}_{L} t \rightsquigarrow t^* : A$.
2. *If* $\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_2 \lesssim D : \tau$ *then* $\exists\, t_1^*$ *and* $\exists\, t_2^*$ *such that* $\Sigma;\Delta;\Phi;\Gamma \vdash t_1 \ominus t_2 \rightsquigarrow t_1^* \ominus t_2^* \lesssim D : \tau$.

Proof. By simultaneous induction on the given typing derivations. □

6.3 Algorithmization

Next, we algorithmize ARelCore's type system. Here, we face the usual challenge of algorithmizing any type system: The need to either annotate or infer the types of bound variables. 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 bidirectional type checking or local type inference (Pierce & Turner, 2000), where type annotations must be provided only at explicit beta-redexes and at the top level, but everything else can be inferred. Our algorithmic system, which we call BiARel, inherits ARelCore's syntax and adds two annotated constructs $(t : \tau, D)$ and $(t : A, L, U)$, as shown in Figure 21. These are required for bidirectional type checking.

For background, bidirectional type checking is a widely used method to implement type systems. The main idea is to split the typing judgment into two judgments corresponding to two modes: one for checking types and the other for synthesizing types. The combination of these two modes reduces the number of annotations needed to guide a type checker. Rules of a bidirectional type system usually resemble those in standard declarative type systems. This simplifies the proof of soundness and completeness of the bidirectional type

$$\textbf{Terms}\quad t ::= \cdots \mid \mathsf{return}\ t \mid \mathsf{alloc}\ t\ t \mid \mathsf{alloc}_{\square}\ t\ t \mid \mathsf{updt}\ t\ t\ t \mid \mathsf{updt}_{\square}\ t\ t\ t \mid \mathsf{read}\ t\ t \mid \mathsf{read}_{\square}\ t\ t \mid \mathsf{let}\ \{x\} = t\ \mathsf{in}\ t_1 \mid \mathsf{switch}\ t \mid \mathsf{NC}\ t \mid \mathsf{split}\ t\ \mathsf{with}\ C \mid \mathsf{contra}\ t \mid \mathsf{FIXEXT}\ f(x).1\text{`}t\ \mathsf{with}\ A \mid (t : \tau, D) \mid (t : A, L, U)$$

Fig. 21. BiARel syntax of values and terms.

system relative to the declarative type system. Our bidirectional type system for ARelCore is similar to the one for RelCost (Çiçek *et al.*, 2019), but extended in a nontrivial way to support array-related operations and for our extended fixpoint operator.

The core language has two typing judgments, the unary and relational one, while we have four of them in BiARel: two relational and two unary judgments. The relational typing judgment of ARelCore splits into two relational judgments in its bidirectional version, one for the "checking mode" and one for "inference mode". The relational checking judgment has the follwoing form:

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

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

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

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.

Similarly, we have two judgments for the unary case. The unary checking judgment has the form $\Sigma; \Delta; \psi_a; \Phi_a; \Omega \vdash t \downarrow A, L, U \Rightarrow \boxed{\Phi}$, while the unary inference judgment has the form $\Sigma; \Delta; \psi_a; \Phi_a; \Omega \vdash t \uparrow \boxed{A} \Rightarrow [\boxed{\psi}], \boxed{L}, \boxed{U}, \boxed{\Phi}$. 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 $\boxed{red}$ boxes and inputs in black. We can think of our unary checking judgment as a mutually recursive function $\mathsf{check}(t, A, L, U)$, whose inputs include a term t, a type A, the bounds L and U. The generated output is a constraint $\boxed{\Phi}$, which holds exactly when t indeed has type A with execution cost bounded by L and U in our semantics. Likewise, the unary inference judgment performs like a function whose input is a term t. The outputs cover the inferred type $\boxed{A}$, the constraint $\boxed{\Phi}$, the bounds $\boxed{U}$, $\boxed{L}$, and an index variable environment $\boxed{\psi}$ that tracks all the new generated index variables during the procedure of the inference. Here, the type and bounds hold iff $\exists\boxed{\psi}.\boxed{\Phi}$ holds. Notice that algorithmic typing judgments have one more input context ψ_a, which records previously eliminated existential variables.

We show selected algorithmic typing judgments in Figure 22 to explain how we handle ARel's non-determinism. The switch rule (r-switch) exists in both checking and inference modes. Both algorithmic rules relate the annotated terms $\mathsf{switch}\ t_1$ and $\mathsf{switch}\ t_2$ 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 is 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-split) exists only in checking mode

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

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

$$\frac{\Sigma;\Delta;\psi_a;C \wedge \Phi_a;\Gamma \vdash t_1 \ominus t_1' \downarrow \tau,D \Rightarrow \Phi_1 \quad \Sigma;\Delta;\psi_a;\neg C \wedge \Phi_a;\Gamma \vdash t_1 \ominus t_1' \downarrow \tau,D \Rightarrow \Phi_2 \qquad \Delta \vdash C\ \mathsf{wf}}{\Sigma;\Delta;\psi_a;\Phi_a;\Gamma \vdash \mathsf{split}\ (t_1)\ \mathsf{with}\ C \ominus \mathsf{split}\ (t_1')\ \mathsf{with}\ C \downarrow \tau,D \Rightarrow C \to \Phi_1 \wedge \neg C \to \Phi_2}\ \textbf{alg-r-split}\downarrow$$

$$\frac{\Sigma;\Delta;\psi_a;\Phi_a;\Gamma \vdash t \ominus t' \uparrow \tau' \Rightarrow [\psi], D', \Phi_1 \qquad \Sigma;\Delta;\psi,\psi_a;\Phi_a \models \tau' \equiv \tau \Rightarrow \Phi_2}{\Sigma;\Delta;\psi_a;\Phi_a;\Gamma \vdash t \ominus t' \downarrow \tau,D \Rightarrow \exists(\psi).\Phi_1 \wedge \Phi_2 \wedge D' \leq D}\ \textbf{alg-r-}\uparrow\downarrow$$

$$\frac{\Sigma;\Delta;\psi_a;\Phi_a;\Gamma \vdash t \ominus t' \downarrow \tau,D \Rightarrow \Phi \qquad \Delta;\Phi_a \vdash \tau\ \mathsf{wf} \qquad \mathrm{FIV}(\tau,D) \in \Delta}{\Sigma;\Delta;\psi_a;\Phi_a;\Gamma \vdash (t:\tau,D) \ominus (t':\tau,D) \uparrow \tau \Rightarrow [\cdot], D, \Phi}\ \textbf{alg-r-anno-}\uparrow$$

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

Fig. 22. Selection of algorithmic typing rules in BiARel.

(alg-r-split↓). 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 $C \to \Phi_1 \wedge \neg C \to \Phi_2$ also analyzes C.

The algorithmic counterpart of the rule r-fix-ext in checking mode, alg-fixext↓, relates the annotated terms $\mathsf{FIXEXT} f(x).t$ with A_1 and $\mathsf{FIXEXT} f(x).t$ with A_2 and checks the subterms $\mathsf{fix} f(x).t$ and $\mathsf{fix} f(x).t'$ 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.

The rule alg-r-↑↓ provides the possiblity of tranfering from inference mode to checking mode, while the rule alg-r-anno-↑ allows the opposite transfer. Notice that we check the equivalence of two types in the rule alg-r-↑↓. In most bidirectional type systems, one would check subtyping here but, as explained earlier, ARelCore only has type equivalence. We emphasize the presence of the annotated term $(t:\tau,D)$ in the rule alg-r-↑↓. This annotated term allows the user to provide the type τ and the effect (relative cost D) to be checked in checking mode, as shown in the rule alg-r-↑↓. It helps when the bidirectional type checker has difficulty inferring the type of a term t by transferring the inferring challenge to a task that is easier, namely, checking a user-provided type. The unary annotated term (t,A,L,U) is useful in a similar way.

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 rules for array operations in Figure 23. As mentioned, to resolve the nondeterminism between the □-ed and non-□-ed rules for each array operation, we use distinct expressions, for example, $\mathsf{alloc}_{\square}\ t_1\ t_2$ versus $\mathsf{alloc}\ t_1\ t_2$. Notice 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 $\square\tau$) 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 $\mathsf{read}\ t_1\ t_2$ or $\mathsf{updt}_{\square}\ t_1\ t_2\ t_3$ 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 check the pre and postconditions. As an example, the condition $\neg(I' \in \beta)$ is checked in the rule alg-r-readB-↓ to guarantee that we indeed read the same value on the two sides. Similarly, in the rules alg-r-updt-↓ and alg-r-updtB-↓, the β' in the postcondition, representing the differences between the two arrays, must be the same as the β in the precondition 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 corresponding declarative typing rules of ARel.

Finally, we show the soundness and completeness of BiARel with respect to ARelCore. Soundness says if the constraints in the output of a provable BiARel typing judgment for term t are satisfiable, then a corresponding typing judgment for the type-erased term $|t|$ is provable in ARelCore. Completeness is the converse: If a term t has a typing derivation in ARelCore, then an annotation of t has a typing derivation in BiARel and the constraints in the output of this derivation are satisfiable. In the following theorem, the function FIV(r)eturns the free index variables in its argument. $\Delta \rhd \theta_a : \psi_a$ means θ_a is a valid substitution for ψ_a under the index variable environment Δ. This substitution is used in the theorem, for example, free index variables in constraints Φ are substituted using ψ_a, written $\Phi[\theta_a]$. We also define the type erasure operation $|t|$, which erases $(t : A, L, U)$ and $(t : \tau, D)$ to t.

Theorem 7 (Soundness of the Algorithmic Typechecking in ARelCore). *The following hold.*

1. *Assume that* $\Sigma; \Delta; \psi_a; \Phi_a; \Omega \vdash t \downarrow A, L, U \Rightarrow \boxed{\Phi}$ *and*
   a. $FIV(\Phi_a, \Omega, A, L, U) \subseteq dom(\Delta, \psi_a)$
   b. $\Sigma; \Delta; \Phi_a[\theta_a] \models \Phi[\theta_a]$ *is provable for some* θ_a *such that* $\Delta \rhd \theta_a : \psi_a$ *is derivable.*

   Then $\Sigma; \Delta; \Phi_a[\theta_a]; \Omega[\theta_a] \vdash^{U[\theta_a]}_{L[\theta_a]} |t| :^c A[\theta_a]$.
2. *Assume that* $\Sigma; \Delta; \psi_a; \Phi_a; \Omega \vdash t \uparrow \boxed{A} \Rightarrow [\boxed{\psi}], \boxed{L}, \boxed{U}, \boxed{\Phi}$ *and*

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

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

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

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

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

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

$$\frac{\begin{array}{c} \Sigma;\Delta;\psi_a;\Phi_a;\Gamma \vdash t_1 \ominus t_1' \uparrow \boxed{\overset{\mathrm{diff}(D)}{\{P\}\ \exists\vec{\gamma_1} : \tau_1\ \{P'\}}} \Rightarrow [\boxed{\psi_1}], \boxed{D_1}, \boxed{\Phi_1} \\ \overset{\mathrm{diff}(D')}{D_2 \in \mathrm{fresh}(\mathbb{R}) \qquad \Sigma;\Delta;D_2,\psi_1,\psi_a;\Phi_a;\Gamma \vdash t_2 \ominus t_2' \downarrow \{P'\}\ \exists\vec{\gamma_2} : \tau_2\ \{Q\}}, D_2 \Rightarrow \boxed{\Phi_2} \\ \Phi = \Phi_2 \wedge D_1 + D_2 + D + D' \doteq D'' \qquad \Phi' = \exists\psi_1.(\Phi_1 \wedge (\exists D_2 :: \mathbb{R}.\Phi) \end{array}}{\Sigma;\Delta;\psi_a;\Phi_a;\Gamma \vdash \mathsf{let}\ \{x\} = t_1\ \mathsf{in}\ t_2 \ominus \mathsf{let}\ \{x\} = t_1'\ \mathsf{in}\ t_2' \downarrow \overset{\mathrm{diff}(D'')}{\{P\}\ \exists\_.\tau_2\ \{Q\}}, 0 \Rightarrow \boxed{\Phi'}}\ \textbf{alg-r-bind-}\downarrow$$

Fig. 23. Selection of algorithmic typing rules for array operations.

   a. $FIV(\Phi_a, \Omega) \subseteq dom(\Delta, \psi_a)$
   b. $\forall\theta\ \forall\theta_a.\ \Delta; \Phi_a[\theta_a] \models \Phi[\theta\ \theta_a]$ *is provable s.t* $\Delta \rhd \theta : \psi$ *and* $\Delta \rhd \theta_a : \psi_a$ *are derivable*

   Then $\Sigma; \Delta; \Phi_a[\theta_a]; \Omega[\theta_a] \vdash^{U[\theta\ \theta_a]}_{L[\theta\ \theta_a]} |t| :^c A[\theta\ \theta_a]$.
3. *Assume that* $\Delta; \psi_a; \Phi_a; \Gamma \vdash t \ominus t' \downarrow \tau, D \Rightarrow \boxed{\Phi}$ *and*

   a. $FIV(\Phi_a, \Gamma, \tau, D) \subseteq dom(\Delta, \psi_a)$
   b. $\Delta; \Phi_a[\theta_a] \models \Phi[\theta_a]$ *is provable for some* θ_a *such that* $\Delta \rhd \theta_a : \psi_a$ *is derivable*

   Then $\Sigma; \Delta; \Phi_a[\theta_a]; \Gamma[\theta_a] \vdash |t| \ominus |t'| \lesssim D[\theta_a] :^c \tau[\theta_a]$.
4. *Assume that* $\Delta; \psi_a; \Phi_a; \Gamma \vdash t \ominus t' \uparrow \boxed{\tau} \Rightarrow [\boxed{\psi}], \boxed{D}, \boxed{\Phi}$ *and*

   a. $FIV(\Phi_a, \Gamma) \subseteq dom(\Delta, \psi_a)$
   b. $\forall\theta\ \forall\theta_a.\ \Delta; \Phi_a[\theta_a] \models \Phi[\theta\ \theta_a]$ *is provable s.t* $\Delta \rhd \theta : \psi$ *and* $\Delta \rhd \theta_a : \psi_a$ *are derivable*

   Then $\Delta; \Phi_a[\theta_a]; \Gamma[\theta_a] \vdash |t| \ominus |t'| \lesssim D[\theta\ \theta_a] :^c \tau[\theta\ \theta_a]$.

Proof. Statements (1–4) follow from simultaneous structural induction on the given algorithmic typing derivations. □

Theorem 8 (Completeness of the Algorithmic Typechecking in ARelCore). *The following hold.*

1. *Assume that* $\Sigma; \Delta; \Phi_a; \Omega \vdash^U_L t :^c A$. *Then,* $\exists t'$ *such that*

   a. $\Sigma; \Delta; \cdot; \Phi_a; \Omega \vdash t' \downarrow A, L, U \Rightarrow \boxed{\Phi}$
   b. $\Delta; \Phi_a \models \Phi$
   c. $|t'| = t$
2. *Assume that* $\Sigma; \Delta; \Phi_a; \Gamma \vdash t_1 \ominus t_2 \lesssim D :^c \tau$. *Then,* $\exists t'_1, t'_2$ *such that*

   a. $\Sigma; \Delta; \cdot; \Phi_a; \Gamma \vdash t'_1 \ominus t'_2 \downarrow \tau, D \Rightarrow \boxed{\Phi}$
   b. $\Delta; \Phi_a \models \Phi$
   c. $|t'_1| = t_1$ *and* $|t'_2| = t_2$

Proof. By simultaneous induction on the given ARelCore typing derivations. □

7 Implementation and experiments

We have 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. Our type checker is implemented in OCaml and the code as well as the examples are available in the public Git repository at https://github.com/haddyclipk/ICFP2019_BiArel. We summarize salient points of our implementation and the results of our experiments in this section.

7.1 Heuristics

Our implementation uses heuristics to automatically backtrack over some typing rules of ARel. We do this to reduce the annotations programemrs have to write manually.

One heuristic that our type checker implements is to automatically determine whether to apply the □-ed rule or the non-□-ed rule, rather than forcing the programmer to make this choice by providing an annotation on every array operation. Our heuristic applies the □-ed rule first and tries to solve the generated constraints (with an SMT-solver, as explained later). If the constraint cannot be solved, the heuristic tries the non-□-ed rule. For example, when processing a read operation we always try the alg-r-readB-↓ rule first. The generated constraint is $I \notin \beta$. We pass this constraint to the SMT solver and if it says yes (satisfiable), we just continue. If the SMT solver says no, we backtrack and try the alg-r-read-↓ rule.

Another heuristic is that we switch from relational to unary reasoning only when absolutely necessary. There are three cases when this happens: a) the unary type is explicitly mentioned with the construct FIXEXT t with A_1, b) the switch term switch t is used, and c) no other relational rules apply.

These heuristics suffice for our examples and reduce our annotation burden at the cost of some extra type checking time.

7.2 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 & 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. From a simple inspection of the algorithmic rules, we can see that generated constraints contain inequalities on index terms such as $I' \leq I$ to check the array bound limit (for instance, in the rule alg-r-read-↓), and equalities such as $D_1 + D_2 = D$ to show that the inferred relative cost matches the relative cost we want to check. Çiçek *et al.* (2019)'s simple algorithm works remarkably well on these kinds of constraints.

A new challenge for ARel is how to represent and solve constraints involving the sets of integers β. There are three kinds of constraints involving these sets. (1) equalities of two sets $\beta = \beta'$ which are generated when the rule compares whether the precondition and postcondition are the same (e.g., in the rule alg-r-read-↓); (2) containments between sets such as $\beta \subseteq \beta'$ which are generated when the postcondition is updated (e.g., alg-r-updt-↓); (3) index inclusions $I \in \beta$ which are generated when certain indexes appear or disappear after the execution of the computation (e.g., alg-r-updtB-↓). Index inclusions are also used by our heuristic to decide whether to use □-ed rules or not. To express these constraints, we rely on the library for set theory from Why3. Its operations for membership, equality, inclusion, empty set, union, intersection, and difference are enough to solve our constraints, and work well in our experience.

7.3 Type checking example

We illustrate our implementation of type checking by walking the reader through the type checking of the following *annotated* version of the mapi function from Section 1.

```
let ≤ = (contra : int → int → bool, 0) in
let plusOne = (contra : ∀x.int[x] → int[x + 1], 0) in
fix mapi (f).λa.λk.λn.
  if(≤ k n) then
    SPLIT{
    let {x} = (read a k, τ1, 0 ; τ1', 0) in
    let {_} = (updt a k (f x), τ2, 0 ; τ2', 0) in
    mapi f a (plusOne k) n
    } with (i ∈ b)
  else
    return ()
```

$$\leq 0 \ : \ \forall r : .\Box(U(\text{int}, \text{int}) \xrightarrow{\text{diff}(r)} U(\text{int}, \text{int})) \rightarrow \forall i, m, g, b.(i < m) \supset$$
$$\text{Array}_g[m]\ U(\text{int}, \text{int}) \rightarrow \text{int}[i] \rightarrow \text{int}[m] \rightarrow \{g \rightarrow b\}\ \exists g.\text{unit}\ \{g \rightarrow b\}$$

First, we introduce two primitive functions for $\leq$ and plusOne. We use a trivial expression contra to simplify the actual implementations of these primitives. For $\leq$, contra appears in the term $(\text{contra}, \text{int} \rightarrow \text{int} \rightarrow \text{bool}, 0)$, which also includes the relational type and relative cost of $\leq$. We omit the relational costs on the arrows here because they are 0, which is also the default cost in our concrete syntax. (In experimental results below, we do not count these functions as annotations, since they are just primitive functions that actually should be inserted by the compiler automatically.)

Different from the mapi function in Section 1, our map example uses the annotated term SPLIT$\{t\}$ with C to specify the use of the split rule r-split in Figure 5. This eliminates non-determinism and guides the type checking. We also have the annotated terms $(\text{read}\ a\ k, \tau_1, 0\ ;\ \tau_1', 0)$ and $(\text{updt}\ a\ k\ (f\ x), \tau_2, 0\ ;\ \tau_2', 0)$, which vary from our standard terms because they now contain the two relational types and two relative cost upper bounds. As a reminder, the annotated terms aim to provide the necessary type and effect to help the type checker.

When we want to provide the relational type and relative cost for terms related to array operations, we need to provide two types and two relative costs, one for the $\Box$-ed rule and the other for the non-$\Box$-ed rule, as depicted in our heuristics in Section 7.1. As an example, consider the expression $(\text{updt}\ a\ k\ (f\ x), \tau_2, 0\ ;\ \tau_2', 0)$. In this expression, $\tau_2 = \{g \rightarrow b\}\ \exists g.\text{unit}^{\text{diff}(0)}\ \{g \rightarrow b \setminus \{i\}\}$ is the relational type we want our type checker to use for the expression $\text{updt}\ a\ k\ (f\ x)$ when it tries the rule alg-r-updtB-$\downarrow$. The second type, $\tau_2' = \{g \rightarrow b\}\ \exists g.\text{unit}^{\text{diff}(r)}\ \{g \rightarrow b \cup \{i\}\}$ is the type for the non-$\Box$-ed rule alg-r-updt-$\downarrow$, which is tried only if the rule alg-r-updtB-$\downarrow$ generates an unsatisfiable constraint.

Overall, this example uses three annotations, which are shown in boxes in the code above.

7.4 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

Table 1. *Summary of experimental results*

Benchmark	LOC	#TYP	#ESF	TC (s)	TC-SMT (s)	TF-SMT (s)
mapi(1)	12	2	0	0.802	1.051	0.01
mapi(2)	19	3	1	1.247	0.994	0.02
boolOr	48	8	3	1.574	1.131	2.38
separate	36	8	0	1.351	2.148	0.01
loop	23	5	0	1.167	2.114	0.01
FFT	66	17	0	2.591	4.268	0.01
Search	62	10	3	3.753	4.430	6.56
NSS	94	12	3	4.158	4.413	10.03
shift	14	3	0	0.660	1.394	0.01
insert	22	6	0	1.001	3.019	0.01
iSort	134	12	3	2.897	6.181	10.70
merge(1)	29	8	0	2.203	2.232	0.01
merge(2)	64	11	2	3.231	0.349	0.02
sam	19	4	1	0.946	0.083	0.02
comp	20	3	0	1.138	0.112	0.01

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 mapi(1), mapi(2), boolOr, FFT, NSS, and ISort are implementations of the corresponding examples discussed in Sections 2 and 5. The example mapi(1) is mapi where we do not assume that the input functions are equal in the two runs. In mapi(2), we assume that the inputs functions are equal in the two runs (we presented the full annotated code for this example in Section 7.3). 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) are the two typings of imperative merge discussed in Section 5.

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 of these examples are in the Appendix.

The results in Table 1 show that ARel can be used to reason about the relative cost of functional-imperative programs. Unsurprisingly, examples combining relational and unary reasoning (using rules r-fix-ext and r-switch) 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, the time taken for solving constraints in the premises of the rules (TC-SMT),

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 need 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.

7.5 Limitations and future directions

One obvious limitation of our current prototype is efficiency, as we mentioned, for example, NSS and iSort. Type checking slows down for two reasons: (1) The heuristic to determine whether to apply a □-ed rule to array-based operations has to wait for SMT to timeout in some cases. The time for alt-ergo (our SMT solver) to solve constraints varies considerably depending on the examples. When dealing with examples with many array-based operations, the problem is exacerbated. Unfortunately, we use Why3 to connect to alt-ergo and have to set a large timeout to guarantee enough time for alt-ergo to deal with the constraint on all connections, which accounts for the unnecessary time consumption. (2) The complexity of the final constraint grows with the number of array-based operations. This complexity translates to longer SMT-solving times.

Another limitation of our implementation is that some annotations are still needed (despite our heuristics). We saw this in the example of Section 7.3.

We plan to improve our prototype by improving our heuristics and the constraint solving process. We would like to find a way to decrease connection times, the constraint solving time, and to make our backtracking more efficient. We also plan to investigate the use of other SMT solvers in order to improve efficiency further.

8 Related work

A lot of prior work has studied static cost analysis. We discuss some of this work here. Reistad & 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 & Gaboardi (2011) present a linear dependent type system using index terms to analyze time complexity. Hoffmann *et al.* (2012*a*) 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. Recurrence extraction analyzes the cost by extracting recurrences which express the run time cost in terms of sizes of inputs, under either call-by-value, call-by-name, or call-by-push-value evaluation strategies (Danner *et al.*, 2015; Kavvos *et al.*, 2019; Cutler *et al.*, 2020). However, none of these systems consider relational costs.

Charguéraud & 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 & Hoffmann (2017) present an amortized resource analysis for arrays and references using. 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 (Gaboardi *et al.*, 2013; Barthe *et al.*, 2015). Barthe *et al.* (2015) propose HOARe^2, which uses relational refinements to reason about differential privacy and other probabilistic relational properties. ARel also relies on relational refinements to reason about pairs of arrays via assertions P, Q in our monadic types. The difference is that we choose lightweight assertions and use them to reason only about difference of arrays. Conversely, HOARe^2 uses arbitrary relational refinement types, which are more expressive, but which also require many more annotations.

The indexed types used by Gaboardi *et al.* (2013) are similar in spirit to ours. Their indices cover the size of the data types as we do, but they also track the sensitivity, which is useful for differential privacy. Our indices instead focus more on effects and differences of arrays. 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.* (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 directly 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 have presented ARel, a relational type-and-effect system for reasoning about the relative cost of two 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.

ARel currently supports arrays whose elements are of base types, due to our choice of the lightweight monadic types for the array-based operations. Support for more complicated but common data types such as matrices (arrays of arrays) is something we would like to develop in the future. Other limitations come from the current implementation, as discussed in Section 7.5. Another possible direction for future work is to add other imperative data structures besides arrays.

Acknowledgments

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

Conflicts of interest

None.

Supplementary materials

For supplementary material for this article, please visit doi.org/10.1017/S0956796821000071

References

Ahmed, A. (2006) Step-indexed syntactic logical relations for recursive and quantified types. In Proceedings of the European Conference on Programming Languages and Systems (ESOP).

Ahmed, A., Dreyer, D. & Rossberg, A. (2009) State-dependent representation independence. In Proceedings of the Symposium on Principles of Programming Languages (POPL).

Ahmed, A. G. (2004) Semantics of types for mutable state, Princeton University.

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

Atkey, R. (2010) Amortised resource analysis with separation logic. In Proceedings of the European Conference on Programming Languages and Systems (ESOP).

Avanzini, M. & Dal Lago, U. (2017) Automating sized type inference for complexity analysis. In Proceedings of DICE-FOPARA.

Barthe, G., Fournet, C., Grégoire, B., Strub, P.-Y., Swamy, N. & Béguelin, S. Z. (2014) Probabilistic relational verification for cryptographic implementations. In Proceedings of the Symposium on Principles of Programming Languages (POPL).

Barthe, G., Gaboardi, M., Arias, E. J. G., Hsu, J., Roth, A. & Strub, P.-Y. (2015) Higher-order approximate relational refinement types for mechanism design and differential privacy. In Proceedings of the Symposium on Principles of Programming Languages (POPL).

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

Benton, N., Hofmann, M. & Nigam, V. (2014) Abstract effects and proof-relevant logical relations. In Proceedings of the Symposium on Principles of Programming Languages (POPL).

Benton, N., Hofmann, M. & Nigam, V. (2016) Effect-dependent transformations for concurrent programs. In Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming.

Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Lescuyer, S. & Mebsout, A. (2013) *The alt-ergo automated theorem prover, 2008*.

Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C. & Giesl, J. (2014) Alternating runtime and size complexity analysis of integer programs. In Tools and Algorithms for the Construction and Analysis of Systems – 26th International Conference (TACAS).

Carbonneaux, Q., Hoffmann, J. & Shao, Z. (2015) Compositional certified resource bounds. In Proceedings of the 36th Conference on Programming Language Design and Implementation (PLDI).

Çiçek, E., Barthe, G., Gaboardi, M., Garg, D. & Hoffmann, J. (2017) Relational cost analysis. In Proceedings of the Symposium on Principles of Programming Languages (POPL).

Charguéraud, A. & Pottier, F. (2015) Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In Interactive Theorem Proving - 6th International Conference, ITP.

Çiçek, E., Paraskevopoulou, Z. & Garg, D. (2016) A type theory for incremental computational complexity with control flow changes. In Proceedings of the International Conference on Functional Programming(ICFP).

Çiçek, E., Qu, W., Barthe, G., Gaboardi, M. & Garg, D. (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, pp. 533–547

Cooley, J. W. & Tukey, J. W. (1965) An algorithm for the machine calculation of complex fourier series. *Math. Comput.* **19**(90), 297–301.

Cutler, J. W., Licata, D. R. & Danner, N. (2020) Denotational recurrence extraction for amortized analysis. *Proc. ACM Program. Lang.* **4**(ICFP), 1–29.

Dal Lago, U. & Gaboardi, M. (2011) Linear dependent types and relative completeness. In Proceedings of the IEEE 26th Annual Symposium on Logic in Computer Science (LICS).

Danielsson, N. A. (2008) Lightweight semiformal time complexity analysis for purely functional data structures. In Proceedings of the Symposium on Principles of Programming Languages (POPL).

Danner, N., Licata, D. R. & Ramyaa, R. (2015) Denotational cost semantics for functional languages with inductive types. In Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. ICFP 2015.

Filliâtre, J.-C. & Paskevich, A. (2013) Why3: Where programs meet provers. In Proceedings of the European Conference on Programming Languages and Systems (ESOP).

Gaboardi, M., Haeberlen, A., Hsu, J., Narayan, A. & Pierce, B. C. (2013) Linear dependent types for differential privacy. In Proceedings of the Symposium on Principles of Programming Languages (POPL).

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

Hermenegildo, M. V., Puebla, G., Bueno, F. & López-García, P. (2005) Integrated program debugging, verification, and optimization using abstract interpretation (and the ciao system preprocessor). *Sci. Comput. Program.* **58**(1–2), 115–140.

Hoffmann, J., Aehlig, K. & Hofmann, M. (2012a) Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.* **34**(3), 1–62.

Hoffmann, J., Aehlig, K. & Hofmann, M. (2012b) Resource aware ML. In Computer Aided Verification - 24th International Conference, CAV.

Kavvos, G. A., Morehouse, E., Licata, D. R. & Danner, N. (2019) Recurrence extraction for functional programs through call-by-push-value. *Proc. ACM Program. Lang.* **4**(POPL), 1–31.

Lahiri, S. K., Vaswani, K. & Hoare, C. A. R. (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, Roman, G.-C. & Sullivan, K. J. (eds).

Lichtman, B. & Hoffmann, J. (2017) Arrays and references in resource aware ML. In The 2nd International Conference on Formal Structures for Computation and Deduction, FSCD.

Nanevski, A., Banerjee, A. & Garg, D. (2013) Dependent type theory for verification of information flow and access control policies. *ACM Trans. Program. Lang. Syst.* **35**(2), 1–41.

Nanevski, A., Morrisett, J. G. & Birkedal, L. (2008) Hoare type theory, polymorphism and separation. *J. Funct. Program.* **18**(5–6), 865–911.

Neis, G., Dreyer, D. & Rossberg, A. (2011) Non-parametric parametricity. *J. Funct. Program.* **21**(4–5), 497–562.

Ngo, V. C., Dehesa-Azuara, M., Fredrikson, M. & Hoffmann, J. (2017) Verifying and synthesizing constant-resource implementations with types. In 2017 IEEE Symposium on Security & Privacy.

Nielson, F. & Nielson, H. (1999) Type and effect systems. In *Correct System Design*. Lecture Notes in Computer Science, vol. 1710, pp. 114–136.

Pierce, B. C. & Turner, D. N. (2000) Local type inference. *ACM Trans. Program. Lang. Syst.* **22**(1), 1–44.

Radicek, I., Barthe, G., Gaboardi, M., Garg, D. & Zuleger, F. (2018) Monadic refinements for relational cost analysis. *PACMPL* **2**(POPL), 36–1.

Reistad, B. & Gifford, D. K. (1994) Static dependent costs for estimating execution time. In Proceedings of the 1994 ACM Conference on LISP and Functional Programming. LFP'94, pp. 65–78.

Sinn, M., Zuleger, F. & Veith, H. (2014) A simple and scalable approach to bound analysis and amortized complexity analysis. In Computer Aided Verification - 26th International Conference, CAV.

Turon, A. J., Thamsborg, J., Ahmed, A., Birkedal, L. & Dreyer, D. (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, pp. 343–356.

Unno, H., Torii, S. & Sakamoto, H. (2017) Automating induction for solving horn clauses. In Computer Aided Verification - 29th International Conference, CAV.

Wang, P., Wang, D. & Chlipala, A. (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).

Xi, H. & Pfenning, F. (1999) Dependent types in practical programming. In Proceedings of the Symposium on Principles of Programming Languages (POPL).

Zhang, D., Wang, Y., Suh, G. E. & Myers, A. C. (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.