Relational Cost Analysis

Ezgi Çiçek
MPI-SWS, Germany
ecicek@mpi-sws.org

Gilles Barthe
IMDEA Software Institute, Spain
gilles.barthe@imdea.org

Marco Gaboardi
University at Buffalo,
The State University of New York, USA
gaboardi@buffalo.edu

Deepak Garg
MPI-SWS, Germany
dg@mpi-sws.org

Jan Hoffmann
Carnegie Mellon University, USA
jhoffmann@cmu.edu

Abstract

Establishing quantitative bounds on the execution cost of programs is essential in many areas of computer science such as complexity analysis, compiler optimizations, security and privacy. Techniques based on program analysis, type systems and abstract interpretation are well-studied, but methods for analyzing how the execution costs of *two programs* compare to each other have not received attention. Naively combining the worst and best case execution costs of the two programs does not work well in many cases because such analysis forgets the similarities between the programs or the inputs.

In this work, we propose a *relational* cost analysis technique that is capable of establishing precise bounds on the difference in the execution cost of two programs by making use of relational properties of programs and inputs. We develop RelCost, a refinement type and effect system for a higher-order functional language with recursion and subtyping. The key novelty of our technique is the combination of relational refinements with two modes of typing—relational typing for reasoning about similar computations/inputs and unary typing for reasoning about unrelated computations/inputs. This combination allows us to analyze the execution cost difference of two programs more precisely than a naive non-relational approach.

We prove our type system sound using a semantic model based on step-indexed unary and binary logical relations accounting for non-relational and relational reasoning principles with their respective costs. We demonstrate the precision and generality of our technique through examples.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Specifying and verifying and reasoning about programs; F.3.2 [*Logics and meanings of programs*]: Semantics of programming languages

General Terms Verification

Keywords Relational reasoning, complexity analysis, type and effect systems

1. Introduction

Statically analyzing the amount of resources needed to run a program is an active field of research that has many applications in security, privacy, embedded and real-time systems, and compiler optimizations. Formal techniques for performing such static execution-cost analysis usually focus on worst-case bounds and build on extensions of classical techniques for statically reasoning about functional properties of programs. For instance, CostIt [17], dℓPCF [20, 21], and Resource Aware ML [28, 30] are based on expressive type systems that rely on refinement or dependent types; Costa [2], the method developed by Sinn et al. [44], and KoAt [11] use techniques from term rewriting and abstract interpretation; other approaches [5, 15] are based on program logics such as Hoare or separation logic.

Many recent innovations for reasoning about properties of programs are based on *relational* or *differential* reasoning [9, 25, 31, 47]. The focus of these approaches is on proving relationships between two executions as opposed to properties of a single execution. Depending on the specific application considered, the two programs or their inputs may coincide. For 2-safety properties [19, 45], the executions are of the same program, but the initial states need not be equal. For translation validation and (some notions of) program equivalence [6, 13, 32, 34, 48], the programs are different, but the initial states are the same. For more general notions of program equivalence, including representation independence, the two programs and the initial states are different.

In this paper, we focus on the problem of *relational cost analysis*: the problem of statically analyzing the *difference* in the execution costs of two similar programs or of two runs of the same program with two different inputs. Relational cost analysis has applications in many different settings: in resource-aware compiler verification, where we want to prove that optimized code runs at least as fast as the original code; in side-channel analysis, where we want to prove that resource consumption does not leak any confidential information; in approximate computation, where we want to prove that an approximate algorithm achieves the desired level of efficiency.

The motivation behind studying relational cost analysis as a separate problem is that, often, naive non-relational cost analysis is intractable or imprecise where relational cost analysis becomes tractable or precise. For example, consider a developer updating a distributed cloud application which uses almost all available hardware resources such as memory on a single machine. Since every patch to the application potentially increases memory requirements, she has to ensure that the updated application does not run out of memory. One solution would be to derive a global memory bound for the updated application. However, this may be difficult, impre-

cise or even impossible in many situations. On the other hand, a formal relational analysis might be able to show that *the updated application does not use more memory than the original one*. Such an analysis could be local—if, e.g., changes have been made to the body of only one loop—and may match the intuitive soundness reasoning in the mind of the developer.

Concretely, we present RelCost, the first type theory for reasoning about the difference in the execution cost of two programs. RelCost is a refinement type and effect system for higher-order functional programs that provides two typing judgments: one relational and the other non-relational. The non-relational typing judgment $\vdash_k^t e : A$ can be read as asserting that the expression e is of (refinement) type A and has execution cost which is lower and upper bounded by the non-negative real numbers k and t, respectively. The relational typing judgment $\vdash e_1 \ominus e_2 \lesssim t : \tau$ can be read as: e_1 and e_2 are two expressions of relational (refinement) type τ, expressing their relation, and the difference of executions costs of e_1 and e_2 is upper bounded by t, a positive or negative real number. The non-relational typing allows us to reason about individual executions of programs, whereas the relational typing allows us to reason instead about the executions of two programs.

The key insight of our work is that a relational indexed refinement type system with effects can model exactly the intuition that we have when reasoning about how changes in program structure or inputs affect the difference of costs. Relational typing strives to relate two structurally similar programs as much as possible, performing synchronous steps on both sides, and only switches to the asynchronous mode, i.e., the comparison of best- and worst-case execution costs independently, whenever two programs structurally diverge. This enables our analysis to provide more precise bounds than computing the best- and worst-case execution cost difference.

The main technical innovation of our work is a semantic model based on step-indexed unary and binary logical relations accounting for the soundness of relational and non-relational reasoning about cost, respectively. The design of the type system closely reflects this semantic model: the typing judgment $\vdash e_1 \ominus e_2 \lesssim t : \tau$ provides an abstraction to reason about the binary relation, while the typing judgment $\vdash_k^t e : A$ provides an abstraction to reason about the unary relation. The step-indexes and the two modes of reasoning create an interesting interaction in the semantic model, explained in detail in Section 4.

Summarizing, we make the following contributions:

- We develop a type theory for relational execution cost analysis that combines relational and non-relational reasoning to provide precise bounds on execution cost differences of programs. The type system combines ingredients from lightweight dependent types, co-monadic reasoning and effect systems.
- We develop an abstract semantic model combining step-indexed binary and unary logical relations for relational and non-relational reasoning about cost. We prove our type system sound relative to this abstract model.
- We demonstrate the applicability of our analysis by typing a set of example programs ranging over optimizations, security and algorithmic analysis.

We start with an informal, example-driven overview of RelCost in Section 2. Sections 3 and 4 present the type system and semantic model, respectively. Section 5 discusses possible extensions to RelCost. Omitted typing rules and proofs of theorems are included in an appendix available from the authors' homepages.

2. RelCost by Examples

In this section, we first present some of the features of RelCost's type system and then demonstrate our relational cost analysis through examples.

Relational cost analysis, concretely Suppose we have two programs e_1 and e_2 that execute with costs c_1 and c_2, respectively. Relational cost analysis establishes an upper bound on the difference in the executions costs of e_1 and e_2, that is $c_1 - c_2$. We refer to this cost as the *relative cost* of e_1 with respect to e_2. In general, the cost could refer to the number of evaluation steps, abstract units of execution time, or to some measure of consumption of another resource. Moreover, the cost of each program may depend on some input values that are known only at runtime.

A naive way to statically establish an upper bound on the relative cost of e_1 with respect to e_2 would be to first establish an upper bound on e_1's cost and a lower bound on e_2's cost, i.e., find t, k such that $c_1 \leq t$ and $k \leq c_2$. Then, the relative cost of e_1 with respect to e_2 is upper bounded by the difference in these upper and lower bounds, i.e., $c_1 - c_2 \leq t - k$. However, such *non-relational* reasoning is often approximate and *imprecise*: it merely makes use of the lower and upper bounds on e_1 and e_2's execution costs in isolation without taking into account their relations/similarities. For instance, the two programs could share a lot of code/structure, or they could be run with similar inputs. The main idea behind RelCost's relational cost analysis is to benefit from such similarities as much as possible and to switch to non-relational reasoning only when necessary.

Example 1 (Conditionals) To see how imprecise the naive non-relational method is, consider a simple program "if $n \geq 0$ then $f(n)$ else 1", with a single input n, where f is a closed function with equal maximum and minimum execution costs $c(n)$ that is linear in n. Assuming that conditional evaluation takes 1 unit of cost, the program runs slowest with cost $c(n) + 1$ when n is non-negative and it runs fastest with cost 1 when n is negative. What can we say about the maximum execution cost difference in the two runs of this program? Although one may naturally answer that the relative cost is simply bounded by the difference in the worst- and best-case executions costs of the two runs, i.e. $c(n)$, the precise answer depends on the values assigned to n in the two runs of the program. If the two values assigned to n cannot differ in the two runs, then the two executions follow the same path and the difference in their execution cost would be 0, not $c(n)$. A relational analysis can establish this 0 cost by taking into account the fact that n is the same in the two runs, whereas a non-relational analysis based on best- and worst-case execution times cannot.

Two-layered typing To enable a more precise relational cost analysis like the one needed in the above example, we designed RelCost's type syntax (Figure 3) and typing judgments to be two-layered. Relational types τ type a pair of values (expressions) relationally, capturing the similarities between them, whereas unary or non-relational types A type individual values (expressions) in isolation. For instance, the unary type int represents integer values whereas, the relational type int_r represents pairs of identical integer values. In general, any unary type can be made relational by encapsulating it with the weakest relation U so that the type $U\,A$ relates two arbitrary values of type A.

Corresponding to these two layers of types, there are two typing judgments in RelCost. The unary typing judgment has the form $\Omega \vdash_k^t e : A$, where k and t are lower and upper bounds on the execution cost of e under the unary (non-relational) typing environment Ω. The relational typing judgment has the form $\Gamma \vdash e_1 \ominus e_2 \lesssim t : \tau$, where t is an upper bound on the relative cost of e_1 with respect to e_2 under the relational typing environment Γ. Relational typing aims to benefit from the similarities between the inputs and the

programs, whereas unary typing considers a single program and a single input in isolation. Unlike unary typing which tracks both lower and upper bounds, relational typing only tracks upper bounds on the execution cost difference: the lower bound on the relative cost of e_1 with respect to e_2 can be obtained by establishing an upper bound on the relative cost of e_2 with respect to e_1 and taking its negative ($c_2 - c_1 \leq k \Rightarrow -k \leq c_1 - c_2$).

Example 1 (Conditional reconsidered) Coming back to the example above, we explain how the relative cost of e = "if $n \geq 0$ then $f(n)$ else 1" can be established in RelCost. If n is not allowed to differ in the two runs, i.e., it has type int_r, then the two runs of e can be typed relationally with relative cost 0:

$$n : \text{int}_r \vdash e \ominus e \lesssim 0 : \text{int}_r \tag{1}$$

The intuition behind this typing is that since the two runs take the same execution path, it suffices to relationally type the two branches $f(n) \ominus f(n)$ and $1 \ominus 1$ component-wise, i.e. *synchronously*. Both of these branches have 0 relative cost and int_r result type, so the two runs of e can be typed as shown above in (1).

In contrast, if the value of n may differ between the two runs, i.e. $n : U\,\text{int}$, then these programs can be typed with cost $c(n)$:

$$n : U\,\text{int} \vdash e \ominus e \lesssim c(n) : U\,\text{int} \tag{2}$$

In this case, since the two executions might take different paths, we lose the relational reasoning. In order to establish an upper bound on their relative cost, we need to switch to the worst- and best-case execution cost comparison. In the type system, this is achieved by using the following switch rule:

$$\frac{|\Gamma| \vdash_{-}^{t_1} e_1 : A \qquad |\Gamma| \vdash_{\dot{-}k_2} e_2 : A}{\Gamma \vdash e_1 \ominus e_2 \lesssim t_1 - k_2 : U\,A}\ \textbf{switch}$$

where e_1 and e_2 are two arbitrary programs that are typed independently with maximum execution cost t_1 and minimum execution cost k_2, respectively. Then the relative cost of e_1 with respect to e_2 is upper bounded by $t_1 - k_2$. Since the execution costs of e_1 and e_2 are independent of their relation, we can type them with a non-relational environment $|\Gamma|$ obtained from Γ by ignoring the relations for each type, e.g., $|\text{int}_r| = |U\,\text{int}| = \text{int}$.

Using this rule, we can type e independently once with maximum execution cost $c(n) + 1$ and once with minimum execution cost 1 and obtain the above typing (2). Note that because n is unrelated in the two runs, any computation that depends on it must be unrelated as well. Hence, the result type is also unrelated, i.e. $U\,\text{int}$.

Example 2 (Constant-time comparison) In cryptographic applications, it is often necessary to prove that a program is *constant-time*, i.e., its execution time is independent of secret inputs, to prevent an attacker from inferring the secret inputs by measuring the execution time. Using relational cost analysis, we can prove that a program is constant time without separately proving that its worst- and best-case execution costs are equal (as would be necessary if we used non-relational cost analysis). For example, consider the following constant-time comparison function `comp` that checks the equality of two passwords represented as equal-length lists of bits.

```
fix comp(l1, l2).case l1 of
  nil  → true
| h1 :: tl1  →  case l2 of
                  nil  → false
                | h2 :: tl2  →  boolAnd (comp (tl1, tl2), eq (h1, h2))
```

To show that `comp` is constant-time, we first need a type to specify how close the input lists l_1 and l_2 are. Accordingly, we refine relational list types to the form $\text{list}[n]^{\alpha}\,\tau$, which ascribes a pair of lists, both of length exactly n and that differ in at most α positions. Similarly, unary list types are refined with exact length: $\text{list}[n]\,A$. Second, we refine the standard function type $\tau_1 \to \tau_2$ to the relational type $\tau_1 \xrightarrow{\text{diff}(t)} \tau_2$, which carries t, the maximum relative cost of the bodies of the two functions (given related arguments of type τ_1).

Suppose that the function `boolAnd` returns the conjunction of the two boolean values in constant-time; it has type `boolAnd` : $(U\,\text{bool} \times U\,\text{bool}) \xrightarrow{\text{diff}(0)} U\,\text{bool}$ and that the function `eq` checks integer equality in constant-time; it has type `eq` : $(U\,\text{int} \times U\,\text{int}) \xrightarrow{\text{diff}(0)} U\,\text{bool}$.[1]

We can now show that the function `comp` is *constant-time* by typing it as follows. The annotation on $\xrightarrow{\text{diff}(0)}$ means that the relative cost of two runs of the function is 0 and, here, the universal quantification over α, β means that this relative cost holds no matter how much the lists differ.[2]

$$\vdash \texttt{comp} \ominus \texttt{comp} \lesssim 0 : \forall n, \alpha, \beta{::}\mathbb{N}.$$
$$(\text{list}[n]^{\alpha}\,U\,\text{int} \times \text{list}[n]^{\beta}\,U\,\text{int}) \xrightarrow{\text{diff}(0)} U\,\text{bool}$$

The proof of this judgment proceeds by induction on the input lists (via a typing rule for fixpoints). The interesting case is when the two lists have at least one element each. Inductively, we know that the relative cost of `comp` $tl_1\ tl_2$ is 0. Furthermore, we assumed that `eq` and `boolAnd` are constant-time. Therefore, we can easily conclude that `comp` is constant-time. This proof of the relative cost of `comp` is trivial compared to a proof through a non-relational analysis that would have to establish best- and worst-case execution costs (taking into account constant factors carefully) and show that they are equal.

Example 3 (Square-and-multiply) This example demonstrates how we can combine RelCost's relational and non-relational reasoning principles to obtain precise bounds on the relative cost of programs. Consider the square-and-multiply algorithm, a fast way of computing the positive powers of a number based on the observation that $x^m = x \cdot (x^2)^{\frac{m-1}{2}}$ when m is odd, and $x^m = (x^2)^{\frac{m}{2}}$ when m is even. The following function, `sam`, implements this idea, assuming that m is stored in binary form in a list l of 0s and 1s, with the least significant bit at the head.

```
fix sam(x).λl.case l of
  nil  → contra
| b :: bs  →  case bs of
                 nil  → if x = 0 then 1 else x
               | _ :: _  →  let r= sam x bs in
                              if b = 0 then r^2 else x · r^2
```

Assume that multiplication (as in $x \cdot r^2$) has a fixed cost t. Consider two executions of `sam` on the same base ($x : \text{int}_r$) and two exponents that differ in at most α bit positions ($l : \text{list}[n]^{\alpha}\,(U\,\text{int})$). What is the maximum relative cost of one run with respect to the other? Intuitively, the relative cost is in $O(\alpha \cdot t)$ since the two runs may enter the two different branches of the **if** in at most α recursive calls and, the difference in the cost of the two branches is exactly one multiplication (r^2 vs $x \cdot r^2$). Hence, `sam` can be given the following type for a suitable linear function P.

$$\vdash \texttt{sam} \ominus \texttt{sam} \lesssim 0 : \text{int}_r \xrightarrow{\text{diff}(0)} \forall n > 0, \alpha{::}\mathbb{N}.$$
$$\text{list}[n]^{\alpha}\,U\,\text{int} \xrightarrow{\text{diff}(P(\alpha \cdot t))} U\,\text{int}.$$

We explain how `sam`'s type is derived in RelCost, focusing on the branch of the case analysis that recursively calls `sam`. From l's type,

[1] The function `boolAnd` can be defined and typed in our language, but we assume `eq` to be a primitive function.

[2] The expression-level introduction and elimination forms for universal and existential quantifiers such as those over n, α, β are omitted from all examples for better readability.

we know that at most α bits differ in the two runs. However, we do not know whether b is among these α bits. Accordingly, our case analysis rule for lists, rule **r-caseL** in Figure 5, requires two sub-cases for the cons branch: either the head b differs in the two runs or it does not. In the first case, we assume that b may have different values in the two runs and $bs : \text{list}[n-1]^{\alpha-1}\ (U\ \text{int})$. The total cost $P(\alpha \cdot t)$ suffices for the recursive call's cost $P((\alpha-1)\cdot t)$ as well as t, the relative cost of the two branches of the expression (if $b = 0$ then r^2 else $x \cdot r^2$), which is established through unary analysis of the expression and the rule **switch**. In the second case, we assume that b has the same value in the two runs and $bs : \text{list}[n-1]^{\alpha}\ (U\ \text{int})$. In this case, the two runs can differ only in the recursive call, which has an (inductive) cost of $P(\alpha \cdot t)$. Technically, the assumption that b has the same value in the two runs is represented using the relational type constructor $\square\,\tau$, which is the *diagonal* sub-relation of τ, i.e., the subset of τ containing equal values in the left and right components. Here, $b : \square\,(U\ \text{int})$ in the second sub-case.

Note that the relative cost of `sam` obtained by taking the difference of worst- and best-case costs would be linear in n, not in α. Thus, direct relational analysis makes the reasoning more precise.

Example 4 (Two-dimensional count) This example demonstrates that RelCost's relational analysis can establish that one program is faster than another when a unary analysis cannot. Consider the following function `2Dcount` that counts the number of rows of a matrix M (represented as a list of lists in row-major form) that both contain a key x and satisfy a predicate p. The function takes as argument another function `find` that returns 1 when a given row l contains x, else returns 0.

$$
\begin{array}{l}
\text{fix } \texttt{2Dcount}(\texttt{find}).\lambda x.\lambda M.\texttt{case } M \texttt{ of}\\
\quad \text{nil } \to 0\\
\mid l :: M' \ \to \ \text{let } r = \texttt{2Dcount find } x\ M' \text{ in}\\
\qquad\qquad\qquad \text{if } p\ l \text{ then } r + (\texttt{find } x\ l) \text{ else } r
\end{array}
$$

Consider the following two different implementations of `find`.

$$
\begin{array}{l}
\text{fix } \texttt{find1}(x).\lambda l.\texttt{case } l \texttt{ of}\\
\quad \text{nil } \to 0\\
\mid h :: tl \ \to \ \text{if } h = x \text{ then } 1 \text{ else } \texttt{find1 } x\ tl
\end{array}
$$

$$
\begin{array}{l}
\text{fix } \texttt{find2}(x).\lambda l.\texttt{case } l \texttt{ of}\\
\quad \text{nil } \to 0\\
\mid h :: tl \ \to \ \text{if } (\texttt{find2 } x\ tl) = 1 \text{ then } 1\\
\qquad\qquad\qquad\qquad\qquad \text{else if } (h = x) \text{ then } 1 \text{ else } 0
\end{array}
$$

The function `find1` scans the row l from head to tail and returns 1 when an element matches x, whereas the function `find2` recurs to the end of l and scans it from tail to head, looking for a match. For simplicity, assume that applications cost a unit and all other operations cost nothing. We can establish that on input lists of length n, the unary cost of `find1` lies in the interval $[1, 3n]$ and that of `find2` lies in the interval $[3n, 4n]$. Hence, `find1` is never slower than `find2` and, so, the relative cost of (`2Dcount find1`) with respect to (`2Dcount find2`) is upper-bounded by 0 (assuming that the same matrix M is given to the two expressions, i.e., M has type $\text{list}[m]^0\ (\text{list}[n]^0\ \text{int})$ for some m and n). In RelCost, this cost can be established in three steps. First, we type `2Dcount`.

$$
\begin{array}{l}
\vdash \texttt{2Dcount} \ominus \texttt{2Dcount} \lesssim 0 :\\
(U\ \text{int} \to \forall n{::}\mathbb{N}.\, U\ (\text{list}[n]\ \text{int}) \xrightarrow{\text{diff}(0)} U\ \text{int}) \to \text{int}_r \to\\
\qquad\qquad \forall m, n{::}\mathbb{N}.\, \text{list}[m]^0\ (\text{list}[n]^0\ \text{int}_r) \xrightarrow{\text{diff}(0)} U\ \text{int}
\end{array}
$$

This type means that, given two `find` functions with relative cost 0 (first $\xrightarrow{\text{diff}(0)}$ in the type above), the relative cost of `2Dcount` with those `find` functions is 0. This type is easily established by induction on M's outer list. Then, we show that the relative cost of `find1` with respect to `find2` is 0, i.e.,

$$
\begin{array}{l}
\vdash \texttt{find1} \ominus \texttt{find2} \lesssim 0 :\\
\qquad\qquad U\ \text{int} \to \forall n{::}\mathbb{N}.\, U\ (\text{list}[n]\ \text{int}) \xrightarrow{\text{diff}(0)} U\ \text{int}
\end{array}
$$

This is done by establishing the best- and worst-case costs of `find1` and `find2` as outlined above (we omit the technical details). Using these two types we can immediately prove that for a fixed matrix $M : \text{list}[m]^0\ (\text{list}[n]^0\ \text{int})$, we have

$$\vdash (\texttt{2Dcount find1 } M) \ominus (\texttt{2Dcount find2 } M) \lesssim 0 : U\ \text{int}$$

Importantly, this relational cost cannot be established using a naive best- and worst-case analysis. This is because the cost of the function (`2Dcount find1` M) is as high as $3nm + 7m$ when the predicate p is true on all rows of M and the element x does not appear anywhere, and the cost of (`2Dcount find2` M) is as low as $4m$ when the predicate p is false on every row. Clearly, $3nm + 7m$ is more than $4m$, so a unary cost analysis cannot establish that (`2Dcount find1` M) is always faster than (`2Dcount find2` M).

Example 5 (Mergesort) Consider the following standard merge-sort function, `msort`, that splits a list into two nearly equal-sized sublists, recursively sorts each sublist and then merges the two sorted sublists. We are interested in establishing the relative cost of two runs of `msort` with two input lists of length n that differ in at most α positions, i.e., when $l : \text{list}[n]^{\alpha}\ (U\ \text{int})$. For simplicity, we count a unit cost for all applications and no cost for the other operations. A naive non-relational analysis establishes this cost at $O(n \cdot \log(n))$. This is imprecise and we show here how a precise cost can be established in RelCost.

$$
\begin{array}{l}
\text{fix } \texttt{msort}(l).\texttt{case } l \texttt{ of}\\
\quad \text{nil } \to \text{nil}\\
\mid h_1 :: tl_1 \ \to \ \texttt{case } tl_1 \texttt{ of}\\
\qquad \text{nil } \to \text{cons}(h_1, \text{nil})\\
\quad \mid \_ :: \_ \ \to \ \text{let } (z_1, z_2) = (\texttt{bsplit } l) \text{ in}\\
\qquad\qquad\qquad \texttt{merge } (\texttt{msort } z_1, \texttt{msort } z_2)
\end{array}
$$

The helper function `bsplit` splits an input list into two nearly equal lists by alternating the input's elements to the two outputs. Its code is unimportant but its relational type is shown below (we omit a description of `bsplit`'s type derivation for brevity). The relative cost of `bsplit` is 0 because `bsplit` rearranges the items in the input list without looking at their values.

$$
\begin{array}{l}
\texttt{bsplit} : (\forall n, \alpha{::}\mathbb{N}.\, \text{list}[n]^{\alpha}\ \tau \xrightarrow{\text{diff}(0)}\\
\qquad\qquad\qquad \exists \beta{::}\mathbb{N}.\, (\text{list}[\lceil \tfrac{n}{2} \rceil]^{\beta}\ \tau \times \text{list}[\lfloor \tfrac{n}{2} \rfloor]^{\alpha-\beta}\ \tau))
\end{array}
$$

The function `merge` takes two sorted lists as input and merges them to produce a sorted list.

$$
\begin{array}{l}
\text{fix } \texttt{merge}(x).\lambda y\ \texttt{case } x \texttt{ of}\\
\quad \text{nil } \to y\\
\mid a :: as \ \to \ \texttt{case } y \texttt{ of}\\
\qquad\qquad\qquad \text{nil } \to x\\
\qquad\qquad \mid b :: bs \ \to \ \text{if } a \le b \text{ then cons}(a, \texttt{merge } as\ y)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else cons}(b, \texttt{merge } x\ bs)
\end{array}
$$

Assuming that the lists input to `merge` may differ in the two runs, the two runs might take different branches of $a \le b$, so a relational analysis of `merge` does not pay off. Instead, we establish its (unary) best- and worst-case execution costs. Given input lists of lengths n and m, `merge`'s best-case cost is $h(\texttt{min}(n, m))$ (for a linear function h), when all the elements in the shorter list are less than or equal to all elements in the longer list. Its worst-case cost is $h(n + m)$, when both lists must be traversed completely. This results in the following type. ($\xrightarrow{\text{exec}(k,t)}$ means that the function

body's cost has lower and upper bounds k and t, respectively.)

$\vdash_0^0 \texttt{merge} : \text{int} \to \forall n, m{::}\mathbb{N}.$
$(\text{list}[n]\,\text{int} \times \text{list}[m]\,\text{int}) \xrightarrow{\text{exec}(h(\texttt{min}(n,m)),h(n+m))} \text{list}[n+m]\,\text{int}$

Next, we turn to `msort`'s relational analysis. Suppose that the input list has length n. Since `msort` divides the input list into two almost equally-sized lists at each step, its recursive calls form a balanced binary tree of height $H = \lceil \texttt{log}_2(n) \rceil$. To calculate the relative cost of the two runs of `msort`, we calculate the relative cost at each level of the recursion tree, counting from the leaves at level 0 up to the root of the tree at level H. The relative cost of `bsplit` is 0 everywhere (from its relational type above) and the relative cost of `merge` is at most $h(n+m) - h(\texttt{min}(n,m)) = h(\texttt{max}(n,m))$ (from its unary type above). At tree level i, n and m in the call to `merge` are $\left\lceil \frac{2^i}{2} \right\rceil$ and $\left\lfloor \frac{2^i}{2} \right\rfloor$, respectively, so the relative cost of `merge` at level i is $h(\left\lceil \frac{2^i}{2} \right\rceil)$. Further, the number of nodes at level i of the recursion tree is at most 2^{H-i}. Since there are at most α differences between the two lists, at most α applications to `merge` can differ in the two runs. Therefore, at level i, the maximum number of `merge` applications that may result in non-zero relative cost is $\texttt{min}(\alpha, 2^{H-i})$. Therefore, the total relative cost of `merge` is $Q(n,\alpha) = \sum_{i=0}^{H} h(\left\lceil \frac{2^i}{2} \right\rceil) \cdot \texttt{min}(\alpha, 2^{H-i})$. Although $Q(n,\alpha)$ looks complicated in this "open" form, it is easily shown to be in $O(n \cdot (1 + \texttt{log}_2(\alpha)))$, which is asymptotically better than the bound $O(n \cdot \texttt{log}_2(n))$ than can be established non-relationally. In fact, for $\alpha = 1$, the precise bound is only $O(n)$.

Let us examine briefly how the relative cost $Q(n,\alpha)$ can be established in RelCost. Our aim is to type `msort` as follows.

$$\texttt{msort} : \Box\,(\forall n, \alpha{::}\mathbb{N}.\, \text{list}[n]^{\alpha}\, U\,\text{int} \xrightarrow{\text{diff}(Q(n,\alpha))} U\,(\text{list}[n]\,\text{int}))$$

(Note the $\Box$ outside the type; its significance will be clear soon.) The most interesting case is when we recursively call `msort`. Splitting the input list into two lists $z_1 : \text{list}[\lceil \frac{n}{2} \rceil]^{\beta}\,(U\,\text{int})$ and $z_2 : \text{list}[\lfloor \frac{n}{2} \rfloor]^{\alpha-\beta}\,(U\,\text{int})$ incurs no relative cost (from the type of `bsplit`). Inductively, we know that the relative costs of the two recursive calls on z_1 and z_2 are $Q(\lceil \frac{n}{2} \rceil, \beta)$ and $Q(\lfloor \frac{n}{2} \rfloor, \alpha - \beta)$, respectively. Merging the two sorted lists has $h(\lceil \frac{n}{2} \rceil)$ relative cost, as established above. Therefore, to complete the typing, we must show that the following inequality holds.

$$h(\left\lceil \frac{n}{2} \right\rceil) + Q(\left\lceil \frac{n}{2} \right\rceil, \beta) + Q(\left\lfloor \frac{n}{2} \right\rfloor, \alpha - \beta) \leq Q(n,\alpha)$$

For $\alpha > 0$, this is an arithmetic tautology. However, for $\alpha = 0$, the right side is 0, but the left side is at least $h(\lceil \frac{n}{2} \rceil)$. Nevertheless, notice that when $\alpha = 0$, the input lists in the two runs do not differ at all, so the relative cost of `msort` should be 0 trivially. To reflect this intuition into RelCost, we add two typing rules. The first typing rule (called **split** in Section 3) permits a case analysis on the index domain, allowing us to type the cases $\alpha = 0$ and $\alpha > 0$ separately. The second typing rule (called **nochange** in Section 3) allows us to establish 0 relative cost whenever we relate an expression to itself and all variables in the environment are in the diagonal relation, i.e., their types are labeled $\Box$. We apply this rule to the body of `msort` in the case $\alpha = 0$. In the subexpression starting `let` $(z_1, z_2) = \ldots$, there are four free variables: `bsplit`, `msort`, `merge` and l. The first two variables, `bsplit` and `merge`, are closed functions, so they cannot change across the two runs and, hence, their types can be (trivially) pre-pended with $\Box$. `msort` is inductively annotated with $\Box$ (inductively typing recursive functions with a $\Box$ annotation requires a separate typing rule called **r-fixNC** in Section 3). Finally, the input list l can be annotated with $\Box$ because when $\alpha = 0$, its type $\text{list}[n]^{\alpha}\,\tau$ means that all list elements will be the same in the two runs, so this is a subtype of $\Box\,\text{list}[n]^{\alpha}\,\tau$. By the **nochange** rule, we immediately derive that the entire subexpression has 0 relative cost. This completes the proof of `msort`'s type.

Unary types	A	$::=$	$\text{unit} \mid \text{int} \mid A_1 \times A_2 \mid A_1 + A_2$ $\mid \text{list}[n]\,A \mid A_1 \xrightarrow{\text{exec}(k,t)} A_2$ $\mid \forall i \overset{\text{exec}(k,t)}{::} S.\, A \mid \exists i{::}S.\, A$ $\mid C \,\&\, A$
Relational types	τ	$::=$	$\text{unit}_r \mid \text{int}_r \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2$ $\mid \text{list}[n]^{\alpha}\,\tau \mid \tau_1 \xrightarrow{\text{diff}(t)} \tau_2$ $\mid \forall i \overset{\text{diff}(t)}{::} S.\, \tau \mid \exists i{::}S.\, \tau$ $\mid C \,\&\, \tau \mid U\,A \mid \Box\tau$
Sorts	S	$::=$	$\mathbb{N} \mid \mathbb{R}$
Index terms	$I, k,$ t, α	$::=$	$i \mid 0 \mid \infty \mid I+1 \mid I_1 + I_2 \mid$ $\mid I_1 - I_2 \mid \frac{I_1}{I_2} \mid I_1 \cdot I_2 \mid \lceil I \rceil \mid$ $\mid \lfloor I \rfloor \mid \texttt{log}_2(I) \mid I_1^{I_2} \mid \sum_{i=I_1}^{I_n} I$ $\mid \texttt{min}(I_1, I_2) \mid \texttt{max}(I_1, I_2)$
Constraints	C	$::=$	$I_1 \doteq I_2 \mid I_1 < I_2 \mid \neg C$
Constraint env.	Φ	$::=$	$\top \mid C \wedge \Phi$
Sort env.	Δ	$::=$	$\emptyset \mid \Delta, i :: S$
Unary type env.	Ω	$::=$	$\emptyset \mid \Omega, x : A$
Rel. type env.	Γ	$::=$	$\emptyset \mid \Gamma, x : \tau$
Primitive env.	Υ	$::=$	$\emptyset \mid \Upsilon, \zeta : \tau_1 \xrightarrow{\text{diff}(t)} \tau_2 \mid$ $\Upsilon, \zeta : A_1 \xrightarrow{\text{exec}(k,t)} A_2$

Figure 1. Syntax of types

Other examples The appendix contains three additional examples: selection sort, an instance of approximate computation and loop unswitching (an optimizing program transformation).

3. The RelCost Type System

In this section, we present the technical ideas behind RelCost. We first describe RelCost's type grammar and expression syntax, then we present the underlying abstract cost semantics and explain the typing and subtyping rules. The design of the type system reflects the underlying semantic model, explained in Section 4.

Types RelCost's type syntax is shown in Figure 1. We have two kinds of types. Unary or non-relational types, denoted A, are ascribed to single expressions, whereas relational types, denoted τ, are ascribed to pairs of expressions. Both types contain familiar type constructors with some refinements. The relational base types int_r and unit_r are distinguished from their unary counterparts int and unit syntactically; for the remaining type constructors such as sums and products, we rely on the context to make this distinction clear. Both relational and non-relational list types are indexed with n, the exact length of the list. Relational list types $\text{list}[n]^{\alpha}\,\tau$ are further refined with α, the maximum number of elements that differ between the two lists.

Unary function types $A_1 \xrightarrow{\text{exec}(k,t)} A_2$ are refined with two effects k and t, the best- and worst-case costs of the body of the function, respectively, whereas the relational function types

$\tau_1 \xrightarrow{\text{diff}(t)} \tau_2$ are refined with a single effect t, an upper bound on the relative cost of the bodies of the two functions. Additionally, we could refine relational function types with best- and worst-case costs of each of the related function bodies but this is not necessary for any of our examples.

Universally quantified types are also refined with costs—similar to function types—for the body of the closure. The relational type $C \mathbin{\&} \tau$ reads "τ and the constraint C is true" (a similar comment applies to the unary type $C \mathbin{\&} A$). Constraints, C, are predicates over index terms as explained below. They are often useful for restricting the set of values in the interpretation of a type, e.g., the type $n > 0 \mathbin{\&} \text{list}[n]\, A$ specifies non-empty lists.

Relational types are interpreted as sets of pairs of values whereas unary types are interpreted–as usual–as sets of values (explained in Section 4). Any unary type can be trivially made relational using the full (weakest) relation $U\ A$ (read "unrelated"), that contains all pairs of values of type A. For instance, the type U int specifies two arbitrary values of type int. In contrast, for base types like integers, the relational type int_r ascribes only those pairs of integers where the two components are equal. The relational type $\tau_1 + \tau_2$ represents two values with the same tag: either both inl or both inr. Pairs of values of a sum type with different tags can be typed at $U\ (A_1 + A_2)$.[3]

Finally, the stronger type $\Box\, \tau$ specifies two values of type τ that are *syntactically* equal. This is best understood by looking at sum types. For example, $(\text{int}_r + U\,\text{int})$ contains pairs of tagged values which have the same tag but whose content can differ if the tag is inr. The stronger type $\Box\,(\text{int}_r + U\,\text{int})$ forces both values to be syntactically equal and is, in fact, semantically equal to $(\text{int}_r + \text{int}_r)$. The $\Box$ annotation is used mainly in typing list expressions, e.g., in typing related lists of type $\text{list}[n]^{\alpha}\, \tau$, where at most α elements of the two related lists are allowed to differ whereas at least $n - \alpha$ elements are assumed to be identical, i.e., of type $\Box\, \tau$. Technically, $\Box\, \tau$ is a co-monadic type.

Indices Index terms I, k, t, α are a key ingredient of RelCost's relational cost analysis (shown in Figure 1). They serve two purposes: (i) as refinements on the typing judgments and function types, they specify relative or best- and worst-case costs and (ii) as refinements on the list types, they specify the lengths of lists and the maximum number of differences. We consider index terms to be *sorted*. Index terms over list types are always interpreted over the domain $\mathbb{N}$ of natural numbers, whereas the cost terms are interpreted over the domain $\mathbb{R}$ of real numbers. Most operations over index terms are overloaded for the sorts $\mathbb{N}$ and $\mathbb{R}$ and there is a natural subsorting from $\mathbb{N}$ to $\mathbb{R}$. The index term ∞ is often used to mean that there is no guaranteed bound on the (relative) cost. The sorting judgment $\Delta; \Phi \vdash I :: S$ assigns sort S to the index term I; its rules are shown in the appendix.

Expressions and values The syntax of expressions and values is shown in Figure 2. It includes the standard introduction and elimination forms for RelCost's types. Integer constants are written $\mathtt{n}$. Recursive functions are written $\text{fix}\ f(x).e$. This is also written $\lambda x.e$ when f doesn't occur in e. Primitive functions and their application are denoted ζ and $\zeta\ e$, respectively. Index variable quantification and instantiation are denoted $\Lambda.\, e$ and $e[\,]$, respectively. To simplify programs that case analyze lists, index terms do not appear in expressions. The elimination form for the constrained type $C \mathbin{\&} \tau$ is written $(\text{clet}\ x\ \text{as}\ e_1\ \text{in}\ e_2)$.

Constraints Constraints C represent predicates over index terms. They may appear in (a) constrained types like $C \mathbin{\&} \tau$, (b) assumptions Φ in typing judgments (explained below), and (c) constraint

[3] In the appendix, we generalize $U \cdot$ to the form $U\ (A_1, A_2)$ that relates pairs of arbitrary values of different types A_1 and A_2.

$$
\begin{array}{lrcl}
\text{Expr.} & e & ::= & x \mid \mathtt{n} \mid \text{fix}\ f(x).e \mid e_1\ e_2 \mid \zeta\ e \mid \langle e_1, e_2\rangle \\
& & & \mid \pi_1(e) \mid \pi_2(e) \mid \text{inl}\ e \mid \text{inr}\ e \\
& & & \mid\ \text{case}\ (e, x.e_1, y.e_2) \mid \text{nil} \mid \text{cons}(e_1, e_2) \\
& & & \mid (\ \text{case}\ e\ \text{of nil}\ \to e_1 \mid h :: tl \to e_2) \\
& & & \mid \Lambda.\, e \mid e[\,] \mid \text{pack}\ e \mid \text{unpack}\ e_1\ \text{as}\ x\ \text{in}\ e_2 \\
& & & \mid \text{let}\ x = e_1\ \text{in}\ e_2 \mid () \mid \text{clet}\ e_1\ \text{as}\ x\ \text{in}\ e_2 \\
\text{Values} & v & ::= & \mathtt{n} \mid \text{fix}\ f(x).v \mid \langle v_1, v_2\rangle \mid \text{inl}\ v \mid \text{inr} \mid \text{nil} \\
& & & \mid \text{cons}(v_1, v_2) \mid \Lambda.\, e \mid \text{pack}\ v \mid ()
\end{array}
$$

Figure 2. Syntax of terms and values

$\boxed{e \Downarrow^c v}$ Expression e evaluates to value v with cost c

$$\frac{}{\mathtt{n} \Downarrow^0 \mathtt{n}}\ \textbf{const} \qquad \frac{e \Downarrow^c v}{\text{inl}\ e \Downarrow^c \text{inl}\ v}\ \textbf{inl}$$

$$\frac{e \Downarrow^c \text{inl}\ v \qquad e_1[v/x] \Downarrow^{c_r} v_r}{\text{case}\ (e, x.e_1, y.e_2) \Downarrow^{c+c_r+c_{case}} v_r}\ \textbf{case-inl}$$

$$\frac{}{\text{fix}\ f(x).e \Downarrow^0 \text{fix}\ f(x).e}\ \textbf{fix}$$

$$\frac{e_1 \Downarrow^{c_1} v_1 \qquad e_2 \Downarrow^{c_2} v_2}{\text{cons}(e_1, e_2) \Downarrow^{c_1+c_2} \text{cons}(v_1, v_2)}\ \textbf{cons}$$

$$\frac{e_1 \Downarrow^{c_1} \text{fix}\ f(x).e \qquad e_2 \Downarrow^{c_2} v_2 \qquad e[v_2/x, (\text{fix}\ f(x).e)/f] \Downarrow^{c_r} v_r}{e_1\ e_2 \Downarrow^{c_1+c_2+c_r+c_{app}} v_r}\ \textbf{app}$$

Figure 3. Selected evaluation rules

entailment in subtyping, denoted $\Delta; \Phi \models C$, and read "for any substitution for the index variables in Δ, the constraint assumptions Φ entail the constraint C". We do not stipulate syntactic rules for constraint entailment, but they are assumed to embody the standard laws of arithmetic.

3.1 Abstract Cost Model

We consider a big-step call-by-value semantics for RelCost. The evaluation judgment $e \Downarrow^c v$ states that expression e evaluates to value v with evaluation cost c. The rules are standard and we only show a few representative cases in Figure 3. The total evaluation cost of an expression is the sum of the costs of its subexpressions, plus a distinct symbolic cost for the following elimination constructs: projections, pattern matches on lists and sum types, function applications and let-bindings. All other reduction rules, including the ones for values, are assigned zero additional cost. We use metavariables like c_{app} to denote such construct-dependent elimination costs. The advantage of this cost metric is that it is easy to understand and it captures asymptotic costs for recursive functions. Our analysis is sound for any values of these cost metavariables as long as they are all natural numbers and the cost of application (c_{app}) is at least 1.[4] Our effect system could be extended to more fine-grained metrics, if needed. Alternatively, it can be easily simplified to more coarse-grained metrics by setting the values of these meta-variables to zero, as in some examples of Section 2.

[4] This requirement is due to the step-indexing used in our semantic model.

3.2 Typing Judgments

RelCost's type system contains two typing judgments. The judgment

$$\Delta; \Phi; \Omega \vdash_k^t e : A$$

states that execution cost of e is lower bounded by k and upper bounded by t, and the expression e has the unary type A. The judgment

$$\Delta; \Phi; \Gamma \vdash e_1 \ominus e_2 \lesssim t : \tau$$

states that the relative cost of e_1 with respect to e_2 is upper bounded by t and the two expressions have the relational type τ. We refer to e_1 as the *left* and e_2 as the *right* expression. These typing judgments use two kinds of type environments: Ω is a type environment for the unary typing judgments, and Γ is a type environment for relational typing judgments. Beside these, both typing judgments have two other environments: Δ for index variables and Φ for assumed constraints. There is also an additional global environment with types for primitive functions, but this environment remains the same across the rules, so we don't write it explicitly. In the presentation of the typing rules, we omit premises concerning wellformedness of types, which clutter the presentation and do not provide any insights. Complete rules appear in the appendix.

Lower bounds on the relative cost RelCost's relational typing judgment can be extended to $\Delta; \Phi; \Gamma \vdash k \lesssim e_1 \ominus e_2 \lesssim t : \tau$, tracking a lower bound k and an upper bound t on the relative cost simultaneously. In addition, function types can be modified to internalize the lower bounds on the relative cost, as in $\tau_1 \xrightarrow{\text{diff}(k,t)} \tau_2$. However, doing so is redundant since the following **swap** rule is *admissible*.

$$\frac{\Delta; \Phi; \Gamma \vdash k \lesssim e_1 \ominus e_2 \lesssim t : \tau}{\Delta; \Phi; d(\Gamma) \vdash -t \lesssim e_2 \ominus e_1 \lesssim -k : d(\tau)} \textbf{ swap}$$

In essence, the rule states that if we can show that the relative cost of e_1 and e_2 is lower bounded by k and upper bounded by t, then we can also show that the relative cost of e_2 and e_1 is lower bounded by $-t$ and upper bounded by $-k$. Semantically, this rule follows trivially from the fact that $k \leq c_1 - c_2 \leq t$ iff $-t \leq c_2 - c_1 \leq -k$. Note that in the conclusion of the **swap** rule, the result type and the environment are also dualized using the type level operation $d(.)$. For instance, $d(\tau_1 \xrightarrow{\text{diff}(k,t)} \tau_2) = d(\tau_1) \xrightarrow{\text{diff}(-k,-t)} d(\tau_2)$.

Since this rule is admissible, adding the lower bound to the relational judgment is redundant: Whenever we are interested in a lower bound on $e_1 \ominus e_2$, we can instead derive an upper bound on $e_2 \ominus e_1$ and flip the sign of the bound. Hence, we do not consider the extended relational judgment with the lower bound any further.

Typing principles and design choices Before we explain the details of RelCost's type system, we review the general design principles behind the unary and relational typing rules.

- The total cost of an expression is obtained by summing the costs of its subexpressions. Moreover, for the unary typing, elimination constructs mentioned in Section 3.1 incur an additional cost. For relational typing, since we track the difference in the execution costs, these costs cancel out in all the rules that relate two structurally similar programs. For programs with different structure, e.g., relating an arbitrary expression to an application, the cost of the extra elimination is subtracted or added depending on the side on which it occurs.
- In all the *synchronous* typing rules that relate two structurally similar expressions, we only allow eliminating truly related expressions that are not of type $U\,A$. For instance, case-elimination on unrelated sum types cannot be typed *relationally*. All such cases are handled *uniformly*: If the eliminated expressions are unrelated, i.e., of type $U\,A$, we immediately switch to non-relational typing for the whole expression. Another possibility would be to duplicate all typing rules for elimination forms that have unrelated types so that continuations would switch to non-relational reasoning. This approach is taken in refinement type systems such as FlowCaml [40] or DuCostIt [18] but we believe our approach is cleaner (it results in fewer typing rules as well).
- Our index refinements are a form of lightweight dependent types that enable static reasoning about runtime properties of a program. In RelCost, we choose to keep the complexity of dependencies limited in comparison to full dependent types. Richer dependencies, such as allowing index terms to be different in two related expressions, would increase the number of programs that can be relationally analyzed. However, this would also make the metatheory more difficult. We discuss this further in Section 5.

$$\frac{\Omega(x) = A}{\Delta; \Phi; \Omega \vdash_0^0 x : A} \textbf{ var} \qquad \frac{}{\Delta; \Phi; \Omega \vdash_0^0 \mathtt{n} : \text{int}} \textbf{ const}$$

$$\frac{}{\Delta; \Phi; \Omega \vdash_0^0 \text{nil} : \text{list}[0]\, A} \textbf{ nil}$$

$$\frac{\Delta; \Phi; \Omega \vdash_{k_1}^{t_1} e_1 : A \qquad \Delta; \Phi; \Omega \vdash_{k_2}^{t_2} e_2 : \text{list}[n]\, A}{\Delta; \Phi; \Omega \vdash_{k_1+k_2}^{t_1+t_2} \text{cons}(e_1, e_2) : \text{list}[n+1]\, A} \textbf{ cons}$$

$$\frac{\Delta; \Phi; x : A_1, f : A_1 \xrightarrow{\text{exec}(k,t)} A_2, \Omega \vdash_k^t e : A_2}{\Delta; \Phi; \Omega \vdash_0^0 \text{fix } f(x).e : A_1 \xrightarrow{\text{exec}(k,t)} A_2} \textbf{ fix}$$

$$\frac{\Delta; \Phi; \Omega \vdash_{k_1}^{t_1} e_1 : A_1 \xrightarrow{\text{exec}(k,t)} A_2 \qquad \Delta; \Phi; \Omega \vdash_{k_2}^{t_2} e_2 : A_1}{\Delta; \Phi; \Omega \vdash_{k_1+k_2+k+c_{app}}^{t_1+t_2+t+c_{app}} e_1\ e_2 : A_2} \textbf{ app}$$

$$\frac{\Delta; \Phi; \Omega \vdash_k^t e : A_1}{\Delta; \Phi; \Omega \vdash_k^t \text{inl}\ e : A_1 + A_2} \textbf{ inl}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Omega \vdash_k^t e : A_1 + A_2 \\ \Delta; \Phi; x : A_1, \Omega \vdash_{k'}^{t'} e_1 : A \qquad \Delta; \Phi; y : A_2, \Omega \vdash_{k'}^{t'} e_2 : A\end{array}}{\Delta; \Phi; \Omega \vdash_{k+k'+c_{case}}^{t+t'+c_{case}} \text{case}\ (e, x.e_1, y.e_2) : A} \textbf{ case}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Omega \vdash_k^t e : A \\ \Delta; \Phi \models^{\mathsf{A}} A \sqsubseteq A' \qquad \Delta; \Phi \models k' \leq k \qquad \Delta; \Phi \models t \leq t'\end{array}}{\Delta; \Phi; \Omega \vdash_{k'}^{t'} e : A'} \sqsubseteq \text{ exec}$$

Figure 4. Selected unary typing rules

Important typing rules for the unary and relational typing judgments are shown in Figures 4, 5 and 6. Below, we explain selected rules for the two judgments separately.

3.3 Unary Typing

The unary typing rules treat lower and upper bounds similarly. We assume that values evaluate with zero cost. So, variables (rule **var**), as well as all introduction forms including functions and index abstractions incur zero cost. For functions, the minimum and maximum costs of the body, denoted k and t respectively, are internalized into the type $A_1 \xrightarrow{\text{exec}(k,t)} A_2$ (rule **fix**). In the rule **app**, these internalized costs k and t are added to the total minimum and maximum execution costs of the application. The rule $\sqsubseteq$ exec allows weakening of the result type as well as the costs: An expression with minimum execution cost k and maximum execution cost t can be typed with a lower cost $k' \leq k$ and a higher cost $t' \geq t$. As usual,

weakening is needed when typing a case construct whose branches have different static costs.

3.4 Relational Typing

Relational typing establishes the relative cost of a pair of expressions and gives the pair a relational type. Relational typing rules can be divided into two categories: (a) *synchronous rules* that relate two structurally similar expressions and (b) *asynchronous rules* that relate two expressions with different structures but possibly similar subcomputations.

Synchronous rules We first explain some of the relational synchronous rules shown in Figure 5. All synchronous rules relate two structurally similar expressions, e.g., a pair of cons constructs or a pair of functions. If the two expressions contain subexpressions, the corresponding subexpressions are related component-wise. The rule **r-var** relates a variable to itself with zero relative cost. Similarly, all other axioms like **r-const** and **r-nil** relate an expression to itself. The rules **r-cons1** and **r-cons2** type non-empty lists of size $n+1$. If the tails have the relational type $\text{list}[n]^{\alpha}\,\tau$, then the two cons'ed lists can be typed at either $\text{list}[n+1]^{\alpha+1}\,\tau$ or $\text{list}[n+1]^{\alpha}\,\tau$ depending on whether the heads may differ or not. The corresponding elimination rule **caseL** has four premises. The first premise establishes the type $\text{list}[n]^{\alpha}\,\tau$ for the pair of lists being eliminated. The second premise types the nil branches, which are taken only when the two lists are empty and, hence, the constraint assumption $n = 0$ is added in this premise. If the lists are not empty, then there are two cases corresponding to the two cons rules. In the first case, the heads of the lists are the same and the tails differ in at most α elements (third premise). In this case, we assume that the heads have type $\Box\,\tau$. In the second case, the heads of the lists may differ (they have type τ, without a $\Box$) and the tails differ in at most $\alpha - 1$ elements (fourth premise). The value $\alpha - 1$ is represented by a fresh variable β that satisfies the constraint $\alpha = \beta + 1$.

Like all other values, recursive functions are relationally typed with zero cost. The relative cost t of the two related bodies is internalized into the function type $\tau_1 \xrightarrow{\text{diff}(t)} \tau_2$ (rule **r-fix**). In the rule **r-app**, this internalized cost is added to the total cost of the application. The rule **r-inl** introduces a sum type with tag inl on both expressions (the rule for tag inr is similar, hence omitted). The **case** rule eliminates a sum type and assumes synchronous execution: The same branch must be taken in the left and right expressions. This is ensured by the interpretation of the type $\tau_1 + \tau_2$ that only contains pairs of values with the same tag. If the case analyzed values have different tags, i.e., they are related at type $U\,(A_1 + A_2)$, then the analysis must switch to unary reasoning via the **switch** rule that is explained below.

The rule **nochange** relates an expression to itself at the (diagonal) type $\Box\,\tau$ and assigns a relative cost of 0, if the expression depends only on variables that are also labeled $\Box$. The latter condition ensures that at runtime, the two expressions being compared are syntactically equal. Statically, the rule applies when for all variables $x \in \Gamma$, the assumed type of x, i.e. $\Gamma(x)$, is a subtype of the same type annotated with $\Box$, i.e. of $\Box\,\Gamma(x)$. In addition, the rule **r-fixNC** allows inductively typing a recursive function with $\Box$ annotation. In typing the function's body, the function itself is assumed to be $\Box$-annotated. This rule cannot be derived using the rules **nochange** and **r-fix**.

Asynchronous rules In addition to the synchronous rules that require the two related expressions to have the same structure, we have several asynchronous rules, some of which are shown in Figure 6. Our appendix shows an example of an optimizing program transformation–loop unswitching–that heavily relies on these asynchronous rules.

The most generic asynchronous rule is the **switch** rule that allows two arbitrary expressions of type A to be related at the weakest relation with type $U\,A$. It switches from *relational* reasoning on two expressions to *unary* reasoning that types the two expressions independently in an erased environment $|\Gamma|$, and takes a difference of the left expression's maximum cost and the right expression's minimum cost. The type erasure operation $|.|$ is a function from relational types to unary types that forgets the relational refinements: $|\text{list}[n]^{\alpha}\,\tau| = \text{list}[n]\,|\tau|$ and $|U\,A| = A$. For function types $\tau_1 \xrightarrow{\text{diff}(t)} \tau_2$, erasure constructs the weakest non-relational type $|\tau_1| \xrightarrow{\text{exec}(0,\infty)} |\tau_2|$, providing no meaningful guarantees on minimum and maximum cost. The definition of $|.|$ extends pointwise to relational environments: $|\Gamma, x : \tau| = |\Gamma|, x : |\tau|$.

The remaining asynchronous rules apply when the left expression is related to a subexpression of the right expression, or vice-versa. Every asynchronous rule has a corresponding inverse/symmetric rule. For instance, the rule **r-let-e** relates $\text{let}\ x = e_1\ \text{in}\ e_2$ to an arbitrary expression e by relating e_2 to e. The symmetric rule **r-e-let** dually relates e to $\text{let}\ x = e_1\ \text{in}\ e_2$. We explain only the rule **r-let-e** here. In the first premise, we type the subexpression e_1 non-relationally with maximum execution cost t_1. In the second premise, we relate the left subexpression e_2 to the right expression e with relative cost t_2 under the assumption that the variable x is unrelated in the two runs ($x : U\,A$). Since x occurs only in e_2, this is sound. The total relative cost is the sum of the costs t_1 and t_2, plus an additional cost c_{let} for the extra let elimination performed on the left side. In the rule **r-e-case**, we relate an arbitrary expression e to a case construct. In the first premise, we type the guard e' of the case construct non-relationally with minimum cost k'. In the second and third premises, we relate the left expression e to the branches with relative cost t. The total cost is the difference of the relative costs t and k', minus the cost of the case elimination, since it is performed only on the right side. The rule **r-app-e** relates $e_1\ e_2$ to an arbitrary expression e by non-relationally typing e_1 at a unary function type $A \xrightarrow{\text{exec}(k,t)} A$ with maximum cost t_1, and relationally typing the argument e_2 with e with relative cost t_2. The relative cost of $e_1\ e_2$ with respect to e is the total cost of evaluating e_1 to a function, the cost of evaluating the function's body, the relative cost of the argument with respect to e and the cost of a function application. These four cost components are t_1, t, t_2 and c_{app}, respectively. (The restriction in **r-app-e** that the function's argument and result have the same type is eliminated in the appendix.)

3.5 Subtyping

Subtyping is essential for both unary and relational typing. There are two subtyping judgments: $\Delta; \Phi \models^{\mathsf{A}} A_1 \sqsubseteq A_2$ for unary types and $\Delta; \Phi \models \tau_1 \sqsubseteq \tau_2$ for relational types. Subtyping is constraint-dependent and the judgments state that A_1 (τ_1) is a subtype of A_2 (τ_2) under the index environment Δ and assumptions Φ. In terms of subsumption, subtyping means that related values of type τ_1 may be used where related values of type τ_2 are expected (similarly for unary types). Figure 7 shows selected subtyping rules. The rule **U** allows lifting subtyping from unary types to relational types at the weakest relation U. The rule **W** allows weakening the type τ to $U\,|\tau|$. The rules $\rightarrow$ **exec** and $\rightarrow$ **diff** are subtyping rules for unary and relational function types. Beyond the usual contravariance for arguments and covariance for results, upper bounds on costs are covariant whereas lower bounds are contavariant. We have two additional subtyping rules for function types. The rule $\rightarrow$ **execdiff** allows converting two unrelated functions—with minimum and maximum execution costs k and t, respectively—to related functions with execution cost $t - k$, but with unrelated arguments and results. The rule $\rightarrow \Box$**diff** captures the idea that syntactically equal functions, when applied to equal arguments, produce equal results and have relative cost 0.

$$\frac{\Gamma(x) = \tau}{\Delta;\Phi;\Gamma \vdash x \ominus x \lesssim 0 : \tau}\ \textbf{r-var} \qquad \frac{}{\Delta;\Phi;\Gamma \vdash \mathtt{n} \ominus \mathtt{n} \lesssim 0 : \text{int}_r}\ \textbf{r-const} \qquad \frac{\Delta;\Phi \vdash \tau\ \mathtt{wf}}{\Delta;\Phi;\Gamma \vdash \text{nil} \ominus \text{nil} \lesssim 0 : \text{list}[0]^{\alpha}\,\tau}\ \textbf{r-nil}$$

$$\frac{\Delta;\Phi;\Gamma \vdash e_1 \ominus e_1' \lesssim t_1 : \tau \qquad \Delta;\Phi;\Gamma \vdash e_2 \ominus e_2' \lesssim t_2 : \text{list}[n]^{\alpha}\,\tau}{\Delta;\Phi;\Gamma \vdash \text{cons}(e_1,e_2) \ominus \text{cons}(e_1',e_2') \lesssim t_1 + t_2 : \text{list}[n+1]^{\alpha+1}\,\tau}\ \textbf{r-cons1}$$

$$\frac{\Delta;\Phi;\Gamma \vdash e_1 \ominus e_1' \lesssim t_1 : \Box\tau \qquad \Delta;\Phi;\Gamma \vdash e_2 \ominus e_2' \lesssim t_2 : \text{list}[n]^{\alpha}\,\tau}{\Delta;\Phi;\Gamma \vdash \text{cons}(e_1,e_2) \ominus \text{cons}(e_1',e_2') \lesssim t_1 + t_2 : \text{list}[n+1]^{\alpha}\,\tau}\ \textbf{r-cons2}$$

$$\frac{\begin{array}{c}\Delta;\Phi;\Gamma \vdash e \ominus e' \lesssim t : \text{list}[n]^{\alpha}\,\tau \\ \Delta;\Phi \wedge n = 0;\Gamma \vdash e_1 \ominus e_1' \lesssim t' : \tau' \qquad i,\Delta;\Phi \wedge n = i+1;h : \Box\tau, tl : \text{list}[i]^{\alpha}\,\tau,\Gamma \vdash e_2 \ominus e_2' \lesssim t' : \tau' \\ i,\beta,\Delta;\Phi \wedge n = i+1 \wedge \alpha = \beta+1;h : \tau, tl : \text{list}[i]^{\beta}\,\tau,\Gamma \vdash e_2 \ominus e_2' \lesssim t' : \tau'\end{array}}{\Delta;\Phi;\Gamma \vdash \text{ case } e \text{ of nil } \to e_1 \mid h :: tl \to e_2 \ominus \text{ case } e' \text{ of nil } \to e_1' \mid h :: tl \to e_2' \lesssim t + t' : \tau'}\ \textbf{r-caseL}$$

$$\frac{\Delta;\Phi;x : \tau_1, f : \tau_1 \xrightarrow{\text{diff}(t)} \tau_2,\Gamma \vdash e_1 \ominus e_2 \lesssim t : \tau_2}{\Delta;\Phi;\Gamma \vdash \text{fix } f(x).e_1 \ominus \text{fix } f(x).e_2 \lesssim 0 : \tau_1 \xrightarrow{\text{diff}(t)} \tau_2}\ \textbf{r-fix} \qquad \frac{\begin{array}{c}\Delta;\Phi;\Gamma \vdash e_1 \ominus e_1' \lesssim t_1 : \tau_1 \xrightarrow{\text{diff}(t)} \tau_2 \\ \Delta;\Phi;\Gamma \vdash e_2 \ominus e_2' \lesssim t_2 : \tau_1\end{array}}{\Delta;\Phi;\Gamma \vdash e_1\ e_2 \ominus e_1'\ e_2' \lesssim t_1 + t_2 + t : \tau_2}\ \textbf{r-app}$$

$$\frac{i :: S,\Delta;\Phi;\Gamma \vdash e \ominus e' \lesssim t : \tau \qquad i \notin \text{FIV}(\Phi;\Gamma)}{\Delta;\Phi;\Gamma \vdash \Lambda.\,e \ominus \Lambda.\,e' \lesssim 0 : \forall i \overset{\text{diff}(t)}{::} S.\,\tau}\ \textbf{r-iLam} \qquad \frac{\Delta;\Phi;\Gamma \vdash e \ominus e' \lesssim t : \forall i \overset{\text{diff}(t')}{::} S.\,\tau \qquad \Delta \vdash I : S}{\Delta;\Phi;\Gamma \vdash e[\,] \ominus e'[\,] \lesssim t + t'[I/i] : \tau\{I/i\}}\ \textbf{r-iApp}$$

$$\frac{\Delta;\Phi;\Gamma \vdash e \ominus e' \lesssim t : \tau_1}{\Delta;\Phi;\Gamma \vdash \text{inl } e \ominus \text{inl } e' \lesssim t : \tau_1 + \tau_2}\ \textbf{r-inl}$$

$$\frac{\Delta;\Phi;\Gamma \vdash e \ominus e' \lesssim t : \tau_1 + \tau_2 \qquad \Delta;\Phi;x : \tau_1,\Gamma \vdash e_1 \ominus e_1' \lesssim t' : \tau \qquad \Delta;\Phi;y : \tau_2,\Gamma \vdash e_2 \ominus e_2' \lesssim t' : \tau}{\Delta;\Phi;\Gamma \vdash \text{ case } (e,x.e_1,y.e_2) \ominus \text{ case } (e',x.e_1',y.e_2') \lesssim t + t' : \tau}\ \textbf{r-case}$$

$$\frac{\begin{array}{c}\Delta;\Phi;\Gamma \vdash e \ominus e \lesssim t : \tau \\ \forall x \in dom(\Gamma).\ \Delta;\Phi \models \Gamma(x) \sqsubseteq \Box\Gamma(x)\end{array}}{\Delta;\Phi;\Gamma,\Gamma' \vdash e \ominus e \lesssim 0 : \Box\tau}\ \textbf{nochange} \qquad \frac{\Delta;\Phi \wedge C;\Gamma \vdash e_1 \ominus e_2 \lesssim t : \tau \qquad \Delta;\Phi \wedge \neg C;\Gamma \vdash e_1 \ominus e_2 \lesssim t : \tau}{\Delta;\Phi;\Gamma \vdash e_1 \ominus e_2 \lesssim t : \tau}\ \textbf{r-split}$$

$$\frac{\Delta;\Phi;x : \tau_1, f : \Box(\tau_1 \xrightarrow{\text{diff}(t)} \tau_2),\Gamma \vdash e \ominus e \lesssim t : \tau_2 \qquad \forall x \in dom(\Gamma).\ \Delta;\Phi \models \Gamma(x) \sqsubseteq \Box\Gamma(x)}{\Delta;\Phi;\Gamma \vdash \text{fix } f(x).e \ominus \text{fix } f(x).e \lesssim 0 : \Box(\tau_1 \xrightarrow{\text{diff}(t)} \tau_2)}\ \textbf{r-fixNC}$$

Figure 5. Selected synchronous typing rules

$$\frac{\begin{array}{c}|\Gamma| \vdash^{t_1}_{\cdot} e_1 : A \\ |\Gamma| \vdash^{\cdot}_{k_2} e_2 : A\end{array}}{\Gamma \vdash e_1 \ominus e_2 \lesssim t_1 - k_2 : U\,A}\ \textbf{switch}$$

$$\frac{\Delta;\Phi;|\Gamma| \vdash^{t_1}_{k_1} e_1 : A_1 \qquad \Delta;\Phi;x : U\,A_1,\Gamma \vdash e_2 \ominus e \lesssim t_2 : \tau_2}{\Delta;\Phi;\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \ominus e \lesssim t_1 + t_2 + c_{let} : \tau_2}\ \textbf{r-let-e}$$

$$\frac{\Delta;\Phi;|\Gamma| \vdash^{t_1}_{k_1} e_1 : A_1 \qquad \Delta;\Phi;x : U\,A_1,\Gamma \vdash e \ominus e_2 \lesssim t_2 : \tau_2}{\Delta;\Phi;\Gamma \vdash e \ominus \text{let } x = e_1 \text{ in } e_2 \lesssim t_2 - k_1 - c_{let} : \tau_2}\ \textbf{r-e-let}$$

$$\frac{\begin{array}{c}\Delta;\Phi;|\Gamma| \vdash^{\cdot}_{k'} e' : A_1 + A_2 \\ \Delta;\Phi;x : U\,A_1,\Gamma \vdash e \ominus e_1' \lesssim t : \tau \\ \Delta;\Phi;y : U\,A_2,\Gamma \vdash e \ominus e_2' \lesssim t : \tau\end{array}}{\Delta;\Phi;\Gamma \vdash e \ominus \text{ case } (e',x.e_1',y.e_2') \lesssim t - k' - c_{case} : \tau}\ \textbf{r-e-case}$$

$$\frac{\begin{array}{c}\Delta;\Phi;|\Gamma| \vdash^{t_1}_{k_1} e_1 : A \xrightarrow{\text{exec}(k,t)} A \\ \Delta;\Phi;\Gamma \vdash e_2 \ominus e \lesssim t_2 : U\,A\end{array}}{\Delta;\Phi;\Gamma \vdash e_1\ e_2 \ominus e \lesssim t_1 + t + t_2 + c_{app} : U\,A}\ \textbf{r-app-e}$$

Figure 6. Selected asynchronous typing rules

The rule **l1** allows the number of elements that differ in a list to be weakened covariantly. The rule **l2** allows two related lists with zero differences to be retyped as two related lists whose elements are in the diagonal relation. The rule **l** $\Box$ allows two related lists whose elements are equal to be retyped as two equal lists, represented by the outer $\Box$. We note that the type $\Box\tau$ follows the standard comonadic rules: $\Box\tau \sqsubseteq \tau$, $\Box(\tau_1 \xrightarrow{\text{diff}(t)} \tau_2) \sqsubseteq \Box\tau_1 \xrightarrow{\text{diff}(0)} \Box\tau_2$ and $\Box(\tau_1 \times \tau_2) \sqsubseteq \Box\tau_1 \times \Box\tau_2$ (the rule for the last subtyping is not shown here).

4. Metatheory and Soundness

RelCost's type system has several standard structural properties like weakening and strengthening of environments. We elide these properties here. Instead, in this section, we present a logical relations model for our type and effect system and use it to prove RelCost sound relative to the abstract cost semantics presented in Section 3.1.

Concretely, we build two cost-annotated models of types: a *non-relational* (unary) one for unary types and unary execution and a *relational* (binary) one for relational types and relational execution. Both models are step-indexed to handle recursive functions [1, 4]. The binary model depends on the unary one and a key novelty is how unary and relational step indices interact. Below, we explain the models in detail.

$\boxed{\Delta;\Phi \models \tau_1 \sqsubseteq \tau_2}$ τ_1 is a subtype of τ_2

$\boxed{\Delta;\Phi \models^{\mathsf{A}} A_1 \sqsubseteq A_2}$ A_1 is a subtype of A_2

$$\frac{\Delta;\Phi \models^{\mathsf{A}} A_1 \sqsubseteq A_2}{\Delta;\Phi \models U\,A_1 \sqsubseteq U\,A_2}\ \mathbf{U} \qquad \frac{}{\Delta;\Phi \models \tau \sqsubseteq U\,|\tau|}\ \mathbf{W}$$

$$\frac{\Delta;\Phi \models^{\mathsf{A}} A_2 \sqsubseteq A_2' \qquad \Delta;\Phi \models^{\mathsf{A}} A_1' \sqsubseteq A_1 \qquad \Delta;\Phi \models k' \le k \qquad \Delta;\Phi \models t \le t'}{\Delta;\Phi \models^{\mathsf{A}} A_1 \xrightarrow{\mathrm{exec}(k,t)} A_2 \sqsubseteq A_1' \xrightarrow{\mathrm{exec}(k',t')} A_2'}\ \to \mathbf{exec}$$

$$\frac{\Delta;\Phi \models \tau_1' \sqsubseteq \tau_1 \qquad \Delta;\Phi \models \tau_2 \sqsubseteq \tau_2' \qquad \Delta;\Phi \models t \le t'}{\Delta;\Phi \models \tau_1 \xrightarrow{\mathrm{diff}(t)} \tau_2 \sqsubseteq \tau_1' \xrightarrow{\mathrm{diff}(t')} \tau_2'}\ \to \mathbf{diff}$$

$$\frac{}{\Delta;\Phi \models U\,(A_1 \xrightarrow{\mathrm{exec}(k,t)} A_2) \sqsubseteq U\,A_1 \xrightarrow{\mathrm{diff}(t-k)} U\,A_2}\ \to \mathbf{execdiff}$$

$$\frac{}{\Delta;\Phi \models \Box\,(\tau_1 \xrightarrow{\mathrm{diff}(t)} \tau_2) \sqsubseteq \Box\,\tau_1 \xrightarrow{\mathrm{diff}(0)} \Box\,\tau_2}\ \to \Box\mathbf{diff}$$

$$\frac{\Delta;\Phi \models n \doteq n' \qquad \Delta;\Phi \models \alpha \le \alpha' \qquad \Delta;\Phi \models \tau \sqsubseteq \tau'}{\Delta;\Phi \models \mathrm{list}[n]^{\alpha}\,\tau \sqsubseteq \mathrm{list}[n']^{\alpha'}\,\tau'}\ \mathbf{l1}$$

$$\frac{\Delta;\Phi \models \alpha \doteq 0}{\Delta;\Phi \models \mathrm{list}[n]^{\alpha}\,\tau \sqsubseteq \mathrm{list}[n]^{\alpha}\,\Box\,\tau}\ \mathbf{l2}$$

$$\frac{}{\Delta;\Phi \models \mathrm{list}[n]^{\alpha}\,\Box\,\tau \sqsubseteq \Box\,(\mathrm{list}[n]^{\alpha}\,\tau)}\ \mathbf{l}\Box$$

$$\frac{}{\Delta;\Phi \models \mathrm{int}_r \sqsubseteq \Box\,\mathrm{int}_r}\ \mathbf{int}\text{-}\Box \qquad \frac{}{\Delta;\Phi \models \Box\,\tau \sqsubseteq \tau}\ \mathbf{T}$$

Figure 7. Selected subtyping rules

4.1 Unary Interpretation of Types

The value interpretation $[\![A]\!]_v$ of unary types A is inductively defined in Figure 8. For each A, $[\![A]\!]_v$ is a set, containing pairs (m, v) of step indices and values. The interpretation is fairly standard, modulo the cost annotations. The expression interpretation $[\![A]\!]_{\varepsilon}^{k,t}$ is shown below and contains pairs (m, e) of step indices and expressions.

$$[\![A]\!]_{\varepsilon}^{k,t} = \left\{ (m,e) \;\middle|\; \begin{array}{l} t < m \implies \begin{pmatrix} e \Downarrow^c v \ \wedge \\ c \le t \ \ \wedge \\ (m-c, v) \in [\![A]\!]_v \end{pmatrix} \wedge \\ \begin{pmatrix} e \Downarrow^c v \ \wedge \\ c < m \end{pmatrix} \implies \begin{pmatrix} k \le c \ \wedge \\ (m-c, v) \in [\![A]\!]_v \end{pmatrix} \end{array} \right\}$$

The interpretation of $[\![A]\!]_{\varepsilon}^{k,t}$ has two clauses, one for the soundness of the cost upper-bound t and the other for the soundness of the lower bound k. The first clause states that if $t < m$, then e evaluates to a value with cost c that is no more than t and the resulting value is in the value interpretation of A at step-index $m - c$. This clause implies finite normalization for all expressions that can be typed with a finite upper cost bound in the type system. The second clause states that if e evaluates to a value with cost $c < m$, then k is a lower bound on c and the resulting value is in the value interpretation with step-index $m - c$. The two clauses are dissimilar only because we want our logical relation to establish expression normalization when

$$\begin{array}{ll}
[\![\mathrm{int}]\!]_v & = \{(m, \mathtt{n})\} \\
[\![A_1 \times A_2]\!]_v & = \{(m, \langle v_1, v_2\rangle) \mid (m, v_1) \in [\![A_1]\!]_v \ \wedge \\
 & \qquad\qquad (m, v_2) \in [\![A_2]\!]_v\} \\
[\![A_1 + A_2]\!]_v & = \{(m, \mathrm{inl}\ v) \mid (m, v) \in [\![A_1]\!]_v\} \cup \\
 & \quad \{(m, \mathrm{inr}\ v) \mid (m, v) \in [\![A_2]\!]_v\} \\
[\![\mathrm{list}[0]\ A]\!]_v & = \{(m, \mathrm{nil}\ )\} \\
[\![\mathrm{list}[n+1]\ A]\!]_v & = \{(m, \mathrm{cons}(e_1, e_2)) \mid (m, e_1) \in [\![A]\!]_v \ \wedge \\
 & \qquad\qquad (m, e_2) \in [\![\mathrm{list}[n]\ A]\!]_v\} \\
[\![A_1 \xrightarrow{\mathrm{exec}(k,t)} A_2]\!]_v & = \{(m, \mathrm{fix}\, f(x).e) \mid \forall j < m. \forall v. (j, v) \in [\![A_1]\!]_v \\
 & \qquad \implies (j, e[v/x, \mathrm{fix}\ f(x).e]) \in [\![A_2]\!]_{\varepsilon}^{k,t}\} \\
[\![\forall i \overset{::}{} \overset{\mathrm{exec}(k,t)}{} S.\, A]\!]_v & = \{(m, \Lambda.\, e) \mid \forall I.\ \vdash I :: S. \\
 & \qquad (m, e) \in [\![A\{I/i\}]\!]_{\varepsilon}^{k[I/i], t[I/i]}\}
\end{array}$$

Figure 8. Unary interpretation of selected types

t is finite. However, this is not essential: We can drop normalization and write the first clause like the second one.[5]

As usual, we interpret open expressions under some semantic environment interpretation δ. We write $(m, \delta) \in \mathcal{G}[\![\Omega]\!]$ to mean that δ maps all variables in the domain of the environment Ω to appropriately-typed semantic value relations for m steps.

$$\begin{array}{l}
\mathcal{G}[\![\cdot]\!] = \{(m, \emptyset)\} \\
\mathcal{G}[\![\Omega, x : A]\!] = \{(m, \delta[x \mapsto v]) \mid (m, \delta) \in \mathcal{G}[\![\Omega]\!] \wedge (m, v) \in [\![A]\!]_v\}
\end{array}$$

We write $\sigma \in \mathcal{D}[\![\Delta]\!]$ to mean that σ is a valid (well-sorted) substitution for the index environment Δ.

We prove the following fundamental theorem for unary typing. Roughly, the theorem says that the expression e if typed in RelCost at unary type A, lies in the unary expression interpretation of A for any value substitution that respects the environment's types.

Theorem 1 (Fundamental Theorem for Unary Typing) Assume that $\Delta; \Phi; \Omega \vdash_k^t e : A$ and $\sigma \in \mathcal{D}[\![\Delta]\!]$ and $\models \sigma\Phi$ and there exists Ω' s.t. $\mathrm{FV}(e) \subseteq \mathrm{dom}(\Omega')$, $\Omega' \subseteq \Omega$ and $(m, \gamma) \in \mathcal{G}[\![\sigma\Omega']\!]$. Then, $(m, \gamma e) \in [\![\sigma A]\!]_{\varepsilon}^{\sigma k, \sigma t}$.

4.2 Relational Interpretation of Types

The value interpretation $(\!|\tau|\!)_v$ of a relational type τ is defined in Figure 9. The binary interpretation is a set, containing triples (m, v_1, v_2) consisting of a step index m and two related values v_1 and v_2. We briefly comment on some salient points about $(\!|\tau|\!)_v$. The interpretation of $\Box\,\tau$ forces the two related values to be identical. The interpretation of $U\,A$ contains unrelated pairs of values (v_1, v_2) in which the individual values v_1 and v_2 are in the unary interpretation $[\![A]\!]_v$ *at any step index* j, i.e., $(j, v_1) \in [\![A]\!]_v$ and $(j, v_2) \in [\![A]\!]_v$ for any j. Essentially, this means that when we switch from relational to unary reasoning, we can call out to any unary step index j. This works because the unary relation does not refer to the binary relation.

The interpretation of $\tau_1 \xrightarrow{\mathrm{diff}(t)} \tau_2$ relates a pair of functions that, given related arguments at $j < m$ steps, return related computations (in the expression relation $(\!|\tau|\!)_{\varepsilon}^{t}$ discussed below) at step-index j. In addition, the two functions are in the unary interpretation of

[5] In both this unary interpretation of expressions, as well as the binary one described later, step indices m interact with costs like t in formulas like $t < m$. This may seem strange. However, this interaction is merely due to the fact that, here, our costs count reduction steps. If we were to build a similar model for a different resource, then this interaction would not occur.

$$
\begin{aligned}
(\!|\Box\,\tau|\!)_v &= \{(m,v,v) \mid (m,v,v) \in (\!|\tau|\!)_v\}\\
(\!|U\ A|\!)_v &= \{(m,v_1,v_2) \mid \forall j.\ (j,v_1) \in [\![A]\!]_v \wedge (j,v_2) \in [\![A]\!]_v\}\\
(\!|\mathsf{int}_r|\!)_v &= \{(m,\mathtt{n},\mathtt{n})\}\\
(\!|\tau_1 \times \tau_2|\!)_v &= \{(m,\langle v_1,v_2\rangle,\langle v_1',v_2'\rangle) \mid (m,v_1,v_1') \in (\!|\tau_1|\!)_v \wedge (m,v_2,v_2') \in (\!|\tau_2|\!)_v\}\\
(\!|\tau_1 + \tau_2|\!)_v &= \{(m,\mathsf{inl}\ v,\mathsf{inl}\ v') \mid (m,v,v') \in (\!|\tau_1|\!)_v\} \cup \{(m,\mathsf{inr}\ v,\mathsf{inr}\ v') \mid (m,v,v') \in (\!|\tau_2|\!)_v\}\\
(\!|\mathsf{list}[0]^{\alpha}\ \tau|\!)_v &= \{(m,\mathsf{nil}\ ,\mathsf{nil}\ )\}\\
(\!|\mathsf{list}[n+1]^{\alpha}\ \tau|\!)_v &= \{(m,\mathsf{cons}(e_1,e_2),\mathsf{cons}(e_1',e_2')) \mid ((m,e_1,e_1') \in (\!|\Box\,\tau|\!)_v \ \wedge\ (m,e_2,e_2') \in (\!|\mathsf{list}[n]^{\alpha}\ \tau|\!)_v) \vee\\
&\qquad ((m,e_1,e_1') \in (\!|\tau|\!)_v \wedge (m,e_2,e_2') \in (\!|\mathsf{list}[n]^{\alpha-1}\ \tau|\!)_v \wedge \alpha > 0)\}\\
(\!|\tau_1 \xrightarrow{\mathrm{diff}(t)} \tau_2|\!)_v &= \{(m,\mathsf{fix}\ f(x).e_1,\mathsf{fix}\ f(x).e_2) \mid\\
&\qquad (\forall j < m.\ \forall v_1,v_2.\ (j,v_1,v_2) \in (\!|\tau_1|\!)_v \implies (j,e_1[v_1/x,\mathsf{fix}\ f(x).e_1/f],e_2[v_2/x,\mathsf{fix}\ f(x).e_2/f]) \in (\!|\tau_2|\!)_{\varepsilon}^{t}) \ \wedge\\
&\qquad (\forall j.\ (j,\mathsf{fix}\ f(x).e_1) \in [\![|\tau_1| \xrightarrow{\mathrm{exec}(0,\infty)} |\tau_2|]\!]_v \ \wedge\ (j,\mathsf{fix}\ f(x).e_2) \in [\![|\tau_1| \xrightarrow{\mathrm{exec}(0,\infty)} |\tau_2|]\!]_v)\}
\end{aligned}
$$

Figure 9. Relational interpretation of selected types

$|\tau_1| \xrightarrow{\mathrm{exec}(0,\infty)} |\tau_2|$ for any step-index j. The latter allows any pair of related functions to be used in a unary context with the weakest cost bounds, 0 and ∞. In essence, we can *semantically* show that the relational judgment $\Delta;\Phi;\Gamma \vdash e_1 \ominus e_2 \lesssim t : \tau$ entails the unary judgment $\Delta;\Phi;|\Gamma| \vdash_0^{\infty} e_i : |\tau|$ for $i \in \{1,2\}$.

In the relational interpretation of list types, the sizes and number of differences of lists are taken into account, encoding the rationale behind the typing rules **r-cons1** and **r-cons2**.

The expression interpretation $(\!|\tau|\!)_{\varepsilon}^{t}$ defines when two expressions are logically related.

$$
(\!|\tau|\!)_{\varepsilon}^{t} = \left\{ (m,e_1,e_2) \;\middle|\; \begin{array}{l} (e_1 \Downarrow^{c_1} v_1 \ \wedge\ e_2 \Downarrow^{c_2} v_2 \ \wedge\ c_1 < m)\\ \quad\implies \begin{pmatrix} c_1 - c_2 \le t \ \wedge \\ (m - c_1, v_1, v_2) \in (\!|\tau|\!)_v \end{pmatrix} \end{array} \right\}
$$

The definition states that if e_1 and e_2 evaluate to values in c_1 and c_2 steps, respectively, and $c_1 < m$, then t is an upper bound on the relative cost of e_1 with respect to e_2, i.e., $c_1 - c_2 \le t$ and the resulting values are related at step-index $m - c_1$. The relational step-index counts steps of the left expression but it could be set up to count steps of the right expression or both.

We interpret pairs of open expressions under a related pair of substitutions, (δ_1,δ_2). We write $(m,\delta_1,\delta_2) \in \mathcal{G}(\!|\Gamma|\!)$ to mean that δ_1 and δ_2 map all the variables in the domain of the environment Γ to appropriately-typed semantic relational values for m steps.

$$
\begin{aligned}
\mathcal{G}(\!|\cdot|\!) &= \{(m,\emptyset,\emptyset)\}\\
\mathcal{G}(\!|\Gamma, x:\tau|\!) &= \{(m,\delta[x \mapsto v_1],\delta[x \mapsto v_2]) \mid (m,\delta_1,\delta_2) \in \mathcal{G}(\!|\Gamma|\!)\\
&\qquad \wedge\ (m,v_1,v_2) \in (\!|\tau|\!)_v\}
\end{aligned}
$$

We prove the following fundamental theorem for our relational typing judgment. An immediate corollary of the theorem is that relative costs established in the type system are upper bounds on execution cost differences.

Theorem 2 (Fundamental Theorem for Relational Typing) Assume that $\Delta;\Phi;\Gamma \vdash e_1 \ominus e_2 \lesssim t : \tau$ and $\sigma \in \mathcal{D}[\![\Delta]\!]$ and $\models \sigma\Phi$ and $(m,\delta_1,\delta_2) \in \mathcal{G}(\!|\sigma\Gamma|\!)$. Then, $(m,\delta_1 e_1,\delta_2 e_2) \in (\!|\sigma\tau|\!)_{\varepsilon}^{\sigma t}$.

Finally, we prove that, semantically, relational typing is a refinement of unary typing with the weakest bounds–0 and ∞–on minimum and maximum costs, respectively. The erasure operation $|.|$ (explained in Section 3.4) forgets the relational refinements, and returns a unary (non-relational) type.

Theorem 3 (Fundamental Theorem for Weak Rel. Typing) Assume that $\Delta;\Phi;\Gamma \vdash e_1 \ominus e_2 \lesssim t : \tau$ and $\sigma \in \mathcal{D}[\![\Delta]\!]$ and $\models \sigma\Phi$. Then for $i \in \{1,2\}$, if there exists Γ_i' s.t. $\mathrm{FV}(e_i) \subseteq \mathrm{dom}(\Gamma_i')$, $\Gamma_i' \subseteq \Gamma$ and $(m,\gamma) \in \mathcal{G}[\![|\sigma\Gamma_i'|]\!]$, then $(m,\gamma e_i) \in [\![|\sigma\tau|]\!]_{\varepsilon}^{0,\infty}$.

5. Discussion and Future Work

Embedding functional equivalences We designed RelCost to reason about the execution cost differences of programs *relationally*. Although our relational analysis is powerful enough to analyze a rich set of examples from a variety of domains, there are programs whose analysis requires more involved reasoning such as the ability to benefit from functional equivalences or the ability to relationally reason about index terms. As an example, consider the sieve of Eratosthenes, a standard algorithm for finding all prime numbers up to a given integer n. There are several variations of this algorithm, but the main idea is to start with a list of all natural numbers that are less than or equal to n and then repeatedly drop all the composites until all the remaining numbers are the primes. Below, we only show the top level function `erat` that takes as input the list l containing the values $[2,3,\ldots,n]$. The function `drop` drops all multiples of its numerical first argument from its second argument, which is a list of natural numbers.

```
fix erat(drop).λl.case l of
  nil  → nil
| h :: tl  →  cons(h, erat drop (drop h tl))
```

Suppose that `drop` is implemented in two functionally equivalent ways but the execution costs of these two versions are different. To establish a precise bound on the relative cost of `erat` with respect to these two implementations of `drop`, we would need to show that two versions of `drop` are functionally equivalent. Such reasoning is not possible in RelCost. This is not an inherent limitation, but a design choice we made to simplify the type system. We believe our analysis can be extended to more fine-grained relations, by building on previous work on relational refinement types [7, 8], which can be used for capturing the necessary invariants. However, the more involved the relational invariants, the more difficult it is for non-experts to use our analysis and perhaps to automate the type-checking. In this work, we have chosen a lightweight form of relational reasoning that still yields a powerful analysis.

Possible extensions In RelCost's relational typing, size and cost refinements are assumed to be identical for the two related programs. For the examples we have considered, such an assumption is sufficient but allowing relational reasoning on index terms could enable more fine-grained analysis. Another possible extension in this direction is to allow tracking insertions and deletions in lists.

Size refinements are data structure-specific. We are planning to investigate a more generic framework in which the programmer can specify a particular size metric along with each algebraic data type. For instance, our appendix considers trees to be refined with the

number of nodes but there could be cases where the depth of a tree is a useful size metric. In the non-relational setting, existing work by Danner et al. considers such generalizations [22]. In RelCost's semantic model, we have anticipated generalization to recursive types—our step-indexed logical relations are capable of modeling recursive types—but we have not yet worked out the generalization to user-defined size or difference metrics.

Our relational cost model can also be adapted to different kinds of resources. For instance, our model can be modified to track the span or work of parallel programs. Alternatively, the resource model can be adapted to track resources other than execution time such as space or energy usage. We believe that adding state would also be useful in this regard.

Implementation We are currently implementing an algorithmic version of RelCost's type system. The implementation uses bidirectional type-checking á la DML in combination with constraint solving through an SMT solver. During the first phase of type-checking, we generate a set of constraints that would have to be satisfied for the program to be typed. Constraints are passed to an external SMT solver capable of handling integers and real numbers.

6. Related work

Our work represents a convergence of two main bodies of research: execution cost/complexity analysis and relational analysis. We first discuss related work in these two areas and then elaborate on recent work on cost analysis for incremental computation from which we borrow some technical ideas.

Static execution cost analysis There are many static techniques for analyzing the execution cost/complexity of programs ranging from semantic interpretation [10, 26] to type-based techniques such as linear dependent types [20, 21], amortized resource analysis [29], type and effect systems [36, 42], etc. In theory, one can simply combine best- and worst-case execution cost analysis from one of these techniques to reason about the relative costs of two programs. However, such combinations forget the relations between programs and inputs, leading to imprecision. RelCost distinguishes itself from prior work in its relational reasoning principles which provide the ability to establish precise bounds on relative cost by making use of similarities between inputs and programs. To achieve this, we build on type and effect systems [36, 42] and extend them to a relational setting.

Relational properties There is a large body of work on relational properties of programs. Traditionally, many of the techniques for relational reasoning have been semantic, but there is an increasing focus on developing practical approaches based on assertion checking [33], symbolic execution [38], static analysis [31], model checking [46], program logics [9, 47], and refinement types [7, 8]. Moreover, researchers have developed specialized approaches for many relational properties, e.g., information flow [3, 35, 40], continuity [16], determinism [12], differential privacy [24, 41], or quantitative reliability [13, 14]. Other important applications of relational verification include regression verification [23, 25], semantical differences [32, 37] and cross or relative verification [27, 34, 39]. One advantage of our work over many of these approaches is that we can freely switch from the relational world to the non-relational world when program executions diverge. In [43], Sands introduces *improvements*, a semantic notion which naturally embeds relational cost reasoning, and uses them as artefacts for proving equivalence between functional programs. However, improvements only offer a qualitative guarantee that one program is faster than another (in all contexts). In contrast, RelCost can establish quantitative bounds.

Types for incremental complexity analysis Our work is closely related to the type system CostIt [17] and its successor DuCostIt [18], which were developed to reason about update costs of incremental programs. Some aspects of our work, like the type $\Box\,\tau$, are based on DuCostIt, but the two differ significantly in the end-goal (relational cost analysis vs. incremental update times), the design of the type system and the semantic model.

DuCostIt aims at establishing upper bounds on the time it takes to update an incremental program. Hence, its cost model is geared towards incremental evaluation. Given an initial execution that stores intermediate results in a trace, a change propagation mechanism accounts for the cost of incremental update when the input changes. For cases where an input change causes a change in control flow, the cost model must also account for from-scratch execution costs of programs. In comparison, our cost model is somewhat simpler since we only account for execution costs and not change propagation. However, to provide precise bounds, we establish both upper and lower bounds on unary execution in the type system whereas only upper bounds are needed in DuCostIt. In particular, DuCostIt has no analogue of the rule **switch**, which is central to many of our examples.

A second difference is that DuCostIt's type system and semantic model are inherently limited to two runs of the same program with different inputs. In particular, asynchronous typing rules, which are often necessary to obtain precision when programs differ structurally are not present in DuCostIt and the binary relation in DuCostIt relates "biexpressions"—pairs of expressions that are structurally identical (but may have different, related substitutions).

Finally, unlike DuCostIt, we syntactically separate unary and relational types. This has two advantages. First, it simplifies the rules of the type system considerably. In DuCostIt, every type constructor has three elimination rules, one for use in unary reasoning and two for use in relational reasoning. In contrast, our type system has only two elimination rules for every type constructor, one for unary reasoning and the other for binary reasoning. Second, the separation reflects the unary and relational semantic interpretations *syntactically*. In contrast, in DuCostIt, the separation exists only in the model, with the unpleasant consequence that relational refinements like α in $(\text{list}[n]^{\alpha}\ \tau)$ that are meaningless in unary reasoning must nonetheless be carried through unary typing derivations.

7. Summary

This paper introduces the problem of relational cost analysis– establishing relative costs of programs, relationally. As a first step towards solving the problem, we have presented a refinement type and effect system, RelCost, that can establish precise relational costs in cases where the use of a unary cost analysis may be far less compact or infeasible. We have demonstrated the expressivity and generality of RelCost on several examples from different domains. Our semantic model combines unary and binary step-indexed logical relations and comes with strong meta-theoretic properties. In particular, we show that the relative cost of two programs estimated by our type system is indeed an upper bound on the actual relative cost of the programs.

Acknowledgments

We thank the anonymous referees for helpful comments and suggestions. This article is based on research that has been supported, in part, by AFRL under DARPA STAC award FA8750-15-C-0082, by NSF under grant 1319671 (VeriQ) and grant TWC-1565365, and by a Google Research Award. Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

References

[1] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 69–83, 2006.

[2] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. Ramírez, G. Román, and D. Zanardini. Termination and Cost Analysis with COSTA and its User Interfaces. *Electr. Notes Theor. Comput. Sci.*, 258(1):109–121, 2009.

[3] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In J. G. Morrisett and S. L. P. Jones, editors, *Proceedings of the 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'06*, pages 91–102, 2006.

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

[5] R. Atkey. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*, pages 85–103, 2010.

[6] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In M. J. Butler and W. Schulte, editors, *Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 200–214, 2011.

[7] G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Z. Béguelin. Probabilistic relational verification for cryptographic implementations. In S. Jagannathan and P. Sewell, editors, *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'14*, pages 193–206, 2014.

[8] G. Barthe, M. Gaboardi, E. J. Gallego Arias, J. Hsu, A. Roth, and P.-Y. Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 55–68, 2015.

[9] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'04*, pages 14–25, 2004.

[10] G. Bonfante, J. Marion, and J. Moyen. Quasi-interpretations a way to control resources. *Theor. Comput. Sci.*, 412(25):2776–2796, 2011.

[11] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Alg. for the Constr. and Anal. of Systems - 20th Int. Conf. (TACAS'14)*, pages 140–155, 2014.

[12] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In H. van Vliet and V. Issarny, editors, *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 3–12, 2009.

[13] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In J. Vitek, H. Lin, and F. Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 169–180, 2012.

[14] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 33–52, 2013.

[15] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional Certified Resource Bounds. In *36th Conference on Programming Language Design and Implementation (PLDI'15)*, 2015. Artifact submitted and approved.

[16] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 57–70, 2010. .

[17] E. Çiçek, D. Garg, and U. A. Acar. Refinement types for incremental computational complexity. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, London, UK, April 11-18, 2015. Proceedings*, pages 406–431, 2015.

[18] E. Çiçek, Z. Paraskevopoulou, and D. Garg. A type theory for incremental computational complexity with control flow changes. In *Proceedings of the 21st International Conference on Functional Programming*, ICFP '16, 2016.

[19] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *Proceedings of CSF'08*, pages 51–65, 2008.

[20] U. Dal Lago and M. Gaboardi. Linear dependent types and relative completeness. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*, LICS '11, pages 133–142, 2011.

[21] U. Dal lago and B. Petit. The geometry of types. In *Proceedings of the 40th Annual Symposium on Principles of Programming Languages*, POPL '13, pages 167–178, 2013.

[22] N. Danner, D. R. Licata, and R. Ramyaa. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 140–151, 2015.

[23] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In I. Crnkovic, M. Chechik, and P. Grünbacher, editors, *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 349–360, 2014. .

[24] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *Proceedings of the 40th Annual Symposium on Principles of Programming Languages*, POPL '13, pages 357–370, 2013.

[25] B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *Softw. Test., Verif. Reliab.*, 23(3): 241–258, 2013.

[26] S. Gulwani, K. K. Mehra, and T. Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th Annual Symposium on Principles of Programming Languages*, POPL '09, pages 127–139, 2009.

[27] C. Hawblitzel, S. K. Lahiri, K. Pawar, H. Hashmi, S. Gokbulut, L. Fernando, D. Detlefs, and S. Wadsworth. Will you still compile me tomorrow? static cross-version compiler validation. In B. Meyer, L. Baresi, and M. Mezini, editors, *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13*, pages 191–201, 2013.

[28] J. Hoffmann and Z. Shao. Automatic Static Cost Analysis for Parallel Programs. In *24th European Symposium on Programming (ESOP'15)*, 2015.

[29] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. In *Proceedings of the 38th Annual Symposium on Principles of Programming Languages*, POPL '11, pages 357–370, 2011.

[30] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 2012.

[31] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare. Differential static analysis: opportunities, applications, and challenges. In G. Roman and K. J. Sullivan, editors, *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 201–204, 2010. .

[32] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012*, volume 7358 of *Lecture Notes in Computer Science*, pages 712–717, 2012.

[33] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In B. Meyer, L. Baresi, and M. Mezini, editors,

Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, pages 345–355, 2013.

[34] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear. Verification modulo versions: towards usable verification. In M. F. P. O'Boyle and K. Pingali, editors, *Proceedings of 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14*, page 32, 2014.

[35] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 165–179, 2011.

[36] F. Nielson and H. Nielson. Type and effect systems. In *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. 1999.

[37] N. Partush and E. Yahav. Abstract semantic differencing via speculative correlation. In A. P. Black and T. D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 811–828, 2014.

[38] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In M. J. Harrold and G. C. Murphy, editors, *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 226–237, 2008.

[39] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 504–515, 2011.

[40] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. Prog. Lang. Sys.*, 25(1):117–158, Jan. 2003.

[41] J. Reed and B. C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 157–168, 2010.

[42] B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 65–78, 1994.

[43] D. Sands. Total correctness by local improvement in program transformation. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 221–232, 1995.

[44] M. Sinn, F. Zuleger, and H. Veith. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th Int. Conf. (CAV'14)*, page 743–759, 2014.

[45] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin and I. Siveroni, editors, *Static Analysis Symposium*, volume 3672, pages 352–367, 2005.

[46] G. Yang, M. B. Dwyer, and G. Rothermel. Regression model checking. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, pages 115–124, 2009.

[47] H. Yang. Relational separation logic. *Theoretical Comput. Sci.*, 375 (1-3):308–334, 2007.

[48] A. Zaks and A. Pnueli. CoVaC: Compiler Validation by Program Analysis of the Cross-Product. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 35–51, 2008.