A Type Theory for Incremental Computational Complexity with Control Flow Changes

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

Zoe Paraskevopoulou
Princeton University, USA
zoe.paraskevopoulou@princeton.edu

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

Abstract

Incremental computation aims to speed up re-runs of a program after its inputs have been modified slightly. It works by recording a trace of the program's first run and propagating changes through the trace in incremental runs, trying to re-use as much of the original trace as possible. The recent work CostIt is a type and effect system to establish the time complexity of incremental runs of a program, as a function of input changes. However, CostIt is limited in two ways. First, it prohibits input changes that influence control flow. This makes it impossible to type programs that, for instance, branch on inputs that may change. Second, the soundness of CostIt is proved relative to an abstract cost semantics, but it is unclear how the semantics can be realized.

In this paper, we address both these limitations. We present DuCostIt, a re-design of CostIt, that combines reasoning about costs of change propagation and costs of from-scratch evaluation. The latter lifts the restriction on control flow changes. To obtain the type system, we refine Flow Caml, a type system for information flow analysis, with cost effects. Additionally, we inherit from CostIt index refinements to track data structure sizes and a co-monadic type. Using a combination of binary and unary step-indexed logical relations, we prove DuCostIt's cost analysis sound relative to not only an abstract cost semantics, but also a concrete semantics, which is obtained by translation to an ML-like language.

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 Complexity analysis, incremental computation, type and effect systems

1. Introduction

Programs are often optimized under the implicit assumption that they will execute only once. However, in practice, many programs are executed again and again on slightly different inputs: spreadsheets compute the same formulas with modifications to some of their cells, search engines periodically crawl the web and software build processes respond to small source code changes. In such settings, it is not enough to design a program that is efficient for the first (from-scratch) execution; the program must also be efficient for the subsequent incremental executions (ideally much more efficient than the from-scratch execution). *Incremental computation* is a promising approach to this problem that aims to design software that can automatically and efficiently respond to changing inputs. The potential for efficient incremental updates comes from the fact that, in practice, large parts of the computation repeat between the first and the incremental run. As shown by prior work on self-adjusting computation [3, 4], by storing intermediate results in a trace in the first run, it is possible to re-execute only those parts that depend on the input changes during the incremental run, and to reuse the parts that didn't change free of cost.

Although previous work has investigated incremental computation from different aspects (demand-driven settings [18], compiler-driven automatic incrementalization [8], etc.), until recently, the programmer had to reason about the asymptotic time complexity of incremental execution, the *dynamic stability*, by direct analysis of the cost semantics of programs [23]. Such reasoning is usually difficult as the programmer has to reason about two executions—the first run and the incremental run—and their relation (dynamic stability is inherently a relational property of two runs of a program). Moreover, dynamic stability analysis heavily relies on a variety of parameters such as the underlying incremental execution technique, the input size, the nature of the input change, etc. Although many specific benchmark programs have been analyzed manually, establishing the dynamic stability of a program can be both difficult and tedious.

In our prior work, CostIt, we took the first steps towards addressing this problem by providing a refinement type and effect system for establishing upper bounds on the asymptotic update complexities of incremental programs [11]. This approach is attractive because programmers can reason about the dynamic stability of their programs without worrying about the semantics of traces and incremental computation algorithms, from which the type system abstracts away. Furthermore, the analysis is compositional: Large programs are analyzed by composing the results of the analysis of subprograms. CostIt can establish precise bounds on the dynamic stability of many examples, including list programs like map, append and reverse, matrix programs like dot products and multiplication and, divide-and-conquer algorithms like balanced list folds.

However, CostIt suffers from two serious limitations. First, CostIt assumes that changes to inputs do not change control flow—closures executed in the incremental run must match those executed in the first run. The type system imposes stringent restrictions to ensure this and cannot analyze many programs. For instance, CostIt 's analysis of merge sort has to assume that the merge function, which merges two sorted sub-lists, has been analyzed by external means, since this function's control flow depends on the values in the input lists. Second, the soundness of CostIt is established relative to an *abstract* change propagation semantics based on previous work on self-adjusting computation [3, 4], but beyond empirical analysis for *specific programs*, there is no evidence that the semantics are realizable.

In this paper, we address both these limitations. To address the first limitation, we re-design the type system, properly accounting for the fact that during incremental run, some closures, which were not executed during the first run, may have to be evaluated from scratch. Accordingly, our type system, called DuCostIt, has two typing judgments—one counts costs of incremental updates (change propagation) and the other counts costs of from-scratch evaluation. Switches between the two modes are mediated by type refinements. To address the second limitation, we show that our language, a λ-calculus with lists, can be translated (type-directed) to a low-level language similar to ML, preserving both incremental and from-scratch costs estimated by the type system. This translation significantly improves upon existing work [9], which provides a related translation but only shows that the translation preserves the cost of the first run (the more important cost here is the cost of the incremental run). We briefly summarize the key insights in DuCostIt's design.

First, dynamic stability is a function of input changes, so to analyze dynamic stability precisely, the type system must track which values may change. For this, we use type refinements that trace lineage to types for information flow analysis [26]: The type $(A)^{\mathbb{S}}$ contains values of type A that cannot change structurally, while $(A)^{\mathbb{C}}$ contains values of type A that may change arbitrarily. Second, dynamic stability depends on sizes of input data structures like lists. To track these sizes, we use index refinements in the style of DML [28] and DFuzz [15].

Third, like CostIt, DuCostIt's type system treats costs as an effect on the typing judgment. However, unlike CostIt, where the only possible effect is the cost of incremental update, in DuCostIt there are two possible costs, which are manifest in two different typing judgments. The judgment $\vdash_{\mathbb{S}} e : \tau \mid \kappa$ means that e (of type τ) has incremental update cost at most κ, while $\vdash_{\mathbb{C}} e : \tau \mid \kappa$ means that e's from-scratch execution cost is at most κ. For example, if $x : (\texttt{real})^{\mathbb{S}}$, i.e., x is a real number that cannot change, then $\vdash_{\mathbb{S}} x + 1 : (\texttt{real})^{\mathbb{S}} \mid 0$ (since x cannot change, there is nothing to incrementally update in $x + 1$, so the cost is zero), but $\vdash_{\mathbb{C}} x + 1 : (\texttt{real})^{\mathbb{S}} \mid 1$ (executing $x + 1$ from scratch requires unit time to compute the addition). By adding the from-scratch cost judgment, DuCostIt allows dynamic stability analysis of programs whose executed closures depend on inputs that may change. CostIt rejects such programs upfront. Examples of such programs are $(\texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2)$ when $x : (\texttt{bool})^{\mathbb{C}}$, and $(y\ 0)$ when $y : (\texttt{real} \to \texttt{real})^{\mathbb{C}}$ is a function which may change completely (e.g., from $\lambda y.\, y + 1$ to $\lambda y.\, y$). Overall, our type system can be viewed as a cost-effect refinement of the pure fragment of [26].

Finally, incremental update has the inherent property that any subcomputation whose dependencies don't change incurs zero cost. This property is needed in the analysis of many recursive programs like merge sort, the fast Fourier transform, etc. Much like CostIt, we internalize this property into the type system using a co-monadic type $\square(\tau)$, which contains values that cannot depend on values that may change (transitively). This type is stronger than $(A)^{\mathbb{S}}$ since $(A)^{\mathbb{S}}$ includes values that do not change structurally, but whose contained closures capture variables that may change, while $\square(\tau)$ excludes such values. In contrast, the annotations $\square(\cdot)$ and $(\cdot)^{\mathbb{S}}$ coincide in CostIt due to its syntactic restrictions.

In addition to showing how to type several examples with DuCostIt, we prove DuCostIt's type system sound relative to two semantics. Here, soundness means that costs established by typing are upper bounds on the costs of incremental update and from-scratch evaluation in a model of the runtime. The first semantics is an *abstract* cost semantics for evaluation from-scratch and for incremental update. The incremental update part extends similar semantics in CostIt with the possibility of switching to from-scratch evaluation when a closure not executed in the first run is reached. The second semantics is a *concrete* semantics that translates our source language to an ML-like target language with references and a cost semantics. The translation explicitly records dependencies (a trace) in the first run. We provide a concrete algorithm for incremental update and prove soundness relative to it. This shows that the costs estimated by our type system can be realized algorithmically. The reader may wonder why we present the abstract semantics when we also have a concrete semantics. The answer is that the abstract semantics are very easy to understand and they specify what the concrete semantics must do.

To prove soundness of the type system with respect to the two semantics, we develop logical relation models of types and typing judgments. These models are interesting in themselves: They combine a binary relation for the judgment $\vdash_{\mathbb{S}} e : \tau \mid \kappa$ with a unary relation for the judgment $\vdash_{\mathbb{C}} e : \tau \mid \kappa$, with an interesting interaction in the step-indices.

In summary, we make the following contributions:

- We develop a type system, DuCostIt, for dynamic stability that combines analyses of costs of incremental update and of from-scratch evaluation. The type system combines index refinements, changeability refinements, co-monadic reasoning and two kinds of cost effects. Our type system significantly extends prior work. (Section 3)
- We show that the type system can precisely type several interesting examples. (Section 2)
- We develop an abstract cost semantics and a concrete cost semantics and prove soundness with respect to both using models that mix binary and unary step-indexed logical relations. The soundness with respect to the concrete cost semantics is completely new and covers a gap in prior work. (Sections 4 and 5)

Omitted inference rules and proofs of theorems are included in an appendix available from the authors' webpages [1].

Implementation We have also designed and implemented an algorithmic version of DuCostIt's type system. Our bidirectional type-checker reduces the problem of type-checking to constraint solving over a first-order theory of integers and reals which, although undecidable, can be handled by SMT solvers with some manual intervention. All but one example in this paper were type-checked on this implementation but due to space limitations the implementation's details are deferred to a separate paper.

2. Typing for Dynamic Stability

This section introduces DuCostIt through examples. The main idea behind incremental computational complexity analysis is *dynamic stability* [11]. Assume that a program e is initially executed with input v and then the program is re-run with a slightly different input v'. Dynamic stability measures the amount of time it takes to re-run the program with the modified input v' using *incremental computation*. In incremental computation [3, 4], all intermediate values are stored in a trace during the initial run. During the re-run (also called the incremental or second run), a special algorithm, called *incremental update* or *change propagation* tries to re-use as many values from the trace as possible, and re-computes from-scratch only when a completely new closure is encountered, or a primitive function is reached and the function's arguments have changed. Concretely, change propagation is implemented by storing all values in reference cells, representing the trace as a dynamic dependence graph over those references, and updating the references by traversing the graph starting from changed leaves (inputs) and re-computing all references that depend on the changed references. This is a bottom-up procedure, that incurs cost only for the parts of the trace that have changed. The graph can be traversed using many different strategies [2]. We explain one such strategy in Section 5, but this intuition suffices for now. It should be clear that dynamic stability is a relational property of two runs of a program.

Like CostIt [11], our broad goal is to build a type and effect system to establish (upper bounds on) dynamic stability. In general, change propagation may have to recompute an intermediate value if either (a) that value was obtained as the result of a primitive function, whose inputs have changed, or (b) that value was obtained from a closure, but the closure has now changed, either due to a change in control flow or due to a non-trivial change to an input function (our setting is higher-order). CostIt only considers possibility (a); restrictions in CostIt's type system immediately discard any program that might afford possibility (b). Our primary goal is to re-design CostIt to lift this restriction.

Example 1a (Warm-up) Consider the boolean expression $x \leq 5$ with one input x of type `real`. Assuming that computing $\leq$ from-scratch costs 1 unit of time, what is the the dynamic stability of this expression? While one may instinctively answer 1, the precise answer depends on whether x may change in the incremental run or not: If x may change, then change propagation may recompute $\leq$, so the dynamic stability would be 1. If x cannot change, then change propagation will simply bypass this expression, and the cost will be 0. To track statically whether a value may change, we use type refinements $(A)^{\mathbb{S}}$ and $(A)^{\mathbb{C}}$ inspired by similar refinements in CostIt. $(A)^{\mathbb{S}}$ ascribes values of type A that may not change structurally, while $(A)^{\mathbb{C}}$ ascribes values of type A that may or may not change.[1] In words, $\mathbb{S}$ is read "stable" and $\mathbb{C}$ is read "changeable". The cost is written as an effect over the turnstile in typing. Hence, our program can be typed in two different ways: $x : (\texttt{real})^{\mathbb{S}} \vdash x \leq 5 : (\texttt{bool})^{\mathbb{S}} \mid 0$ and $x : (\texttt{real})^{\mathbb{C}} \vdash x \leq 5 : (\texttt{bool})^{\mathbb{C}} \mid 1$.

Dual-mode typing The typing judgment described above suffices for typing programs under CostIt's restrictions, where only primitive functions are re-executed during change propagation. However, in general, change propagation may execute fresh closures from-scratch. To count the costs of these closures, we need a second "mode" of typing, that upper-bounds the *from-scratch* execution cost of an expression. Accordingly, we use two typing judgments: $\vdash_{\mathbb{S}} e : \tau \mid \kappa$, which means that the cost of change propagating through e is at most κ and $\vdash_{\mathbb{C}} e : \tau \mid \kappa$, which means that the cost of evaluating e from-scratch is at most κ. As a rule, the from-scratch cost always dominates the change propagation cost. We often write the judgments generically as $\vdash_{\epsilon} e : \tau \mid \kappa$ for $\epsilon \in \{\mathbb{S}, \mathbb{C}\}$.

Remark on notation When used on typing judgments $\vdash_{\mathbb{S}} e : \tau \mid \kappa$ an $\vdash_{\mathbb{C}} e : \tau \mid \kappa$, the annotations $\mathbb{S}$ and $\mathbb{C}$ stand for *change-propagation* and *from-scratch execution* respectively, whereas on types $(A)^{\mathbb{S}}$ and $(A)^{\mathbb{C}}$, the annotations $\mathbb{S}$ and $\mathbb{C}$ stand for *stable* and *changeable*.

Example 1b (From-scratch cost) The program $x \leq 5$ can be given a from-scratch execution cost using the $\mathbb{C}$-mode typing judgment: $x : (\texttt{real})^{\mu} \vdash_{\mathbb{C}} x \leq 5 : (\texttt{bool})^{\mu} \mid 4$. The cost 4 counts unit costs for each of the following: applying the comparison function, reading from the variable x, (immediately) evaluating the constant 5, and executing the body of the comparison. Note that the from-scratch cost is independent of whether or not x may change. Hence, it holds for both $\mu = \mathbb{C}$ and $\mu = \mathbb{S}$.

Example 2 (Mode-switching) To understand how the two modes of typing interact with each other, consider $(\texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2)$. How do we establish the change propagation cost of this expression when x has types $(\texttt{bool})^{\mathbb{S}}$ and $(\texttt{bool})^{\mathbb{C}}$? If $x : (\texttt{bool})^{\mathbb{S}}$, we know that x will not change. So, the incremental run will execute the same branch (e_1 or e_2) as the initial run. This means that change propagation can be continued in the branch. Consequently, in this case, we only need to establish change propagation costs of the two branches e_i, not their from-scratch evaluation costs. In the type system, this means that the branches can be typed in $\mathbb{S}$ mode, as in the following derivation.

$$\frac{x : (\texttt{bool})^{\mathbb{S}} \vdash_{\mathbb{S}} x : (\texttt{bool})^{\mathbb{S}} \mid 0 \qquad x : (\texttt{bool})^{\mathbb{S}} \vdash_{\mathbb{S}} e_1 : \tau \mid \kappa \qquad x : (\texttt{bool})^{\mathbb{S}} \vdash_{\mathbb{S}} e_2 : \tau \mid \kappa}{x : (\texttt{bool})^{\mathbb{S}} \vdash_{\mathbb{S}} \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \mid \kappa}$$

If $x : (\texttt{bool})^{\mathbb{C}}$ then x may change. Consequently, the initial and incremental runs may execute different branches. If the branches end up being different, change propagation must execute the new branch from-scratch. Hence, we must establish the from-scratch costs of the two branches. (If the branch doesn't actually change, change propagation will not evaluate from-scratch, but in that case the cost will only be lower, so our established cost would be conservative.)

$$\frac{x : (\texttt{bool})^{\mathbb{C}} \vdash_{\mathbb{S}} x : (\texttt{bool})^{\mathbb{C}} \mid 0 \qquad x : (\texttt{bool})^{\mathbb{C}} \vdash_{\mathbb{C}} e_1 : \tau \mid \kappa' \qquad x : (\texttt{bool})^{\mathbb{C}} \vdash_{\mathbb{C}} e_2 : \tau \mid \kappa'}{x : (\texttt{bool})^{\mathbb{C}} \vdash_{\mathbb{S}} \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \mid \kappa' + 1}$$

In the second premise, κ' is not the cost for change-propagation, but from-scratch execution ($\epsilon = \mathbb{C}$, not $\mathbb{S}$). We also add a cost of 1 for determining which branch must be taken in the incremental run. CostIt cannot type-check this example when $x : (\texttt{bool})^{\mathbb{C}}$. The pattern illustrated by this example is general: Whenever we eliminate a boolean, sum, list or existential type labeled $\mathbb{C}$, we switch to the $\mathbb{C}$ (from-scratch) mode in typing the branches. We do not switch to the $\mathbb{C}$ mode when the eliminated type is labeled $\mathbb{S}$.

Example 3 (Map) Branch points are not the only reason why change propagation may end up executing a completely fresh expression. A second reason is that a function provided as input to another function may change non-trivially. To illustrate this, we type the standard list function `map`. We need two additional type refinements. First, dynamic stability usually depends on the sizes of input data structures, so we introduce index refinements as in CostIt. In particular, list types are refined to the form $\texttt{list}\,[n]^{\alpha}\ \tau$. Here, τ is the type of the elements of the list, n is the exact length of the list and α is an upper bound on the number of elements that may change. Second, the function type $\tau_1 \rightarrow \tau_2$ is refined to $\tau_1 \xrightarrow{\mathbb{S}(\kappa)} \tau_2$ and $\tau_1 \xrightarrow{\mathbb{C}(\kappa)} \tau_2$. The former type says that the cost of change propagating through the body of the function is κ, whereas the latter type says that the cost of executing the function's body from scratch is κ. For instance, based on Example 1a, the function $\lambda x.\, (x \leq 5)$ can be given the types $(\texttt{real})^{\mathbb{S}} \xrightarrow{\mathbb{S}(0)} (\texttt{bool})^{\mathbb{S}}$ and $(\texttt{real})^{\mathbb{C}} \xrightarrow{\mathbb{S}(1)} (\texttt{bool})^{\mathbb{C}}$ and based on Example 1b, it can be given the type $(\texttt{real})^{\mu} \xrightarrow{\mathbb{C}(4)} (\texttt{bool})^{\mu}$ for $\mu \in \{\mathbb{S}, \mathbb{C}\}$.

Consider the standard `map` function that applies an input function f to every element of an input list l.

$$\begin{aligned}&\texttt{fix map}(f).\, \lambda l.\, \texttt{case}_{\texttt{L}}\ l\ \texttt{of nil} \rightarrow \texttt{nil}\\ &\qquad\qquad\qquad\qquad \mid \texttt{cons}(h,\ tl)\ \rightarrow\ \texttt{cons}(f\ h,\ \texttt{map}\ f\ tl)\end{aligned}$$

To type `map`, we introduce a new co-monadic type $\Box(\tau)$, which ascribes values of type τ that do not depend on anything that may change. Suppose that the input list l has type $(\texttt{list}\,[n]^{\alpha}\ \tau)^{\mathbb{S}}$. Assume that f has type $\Box(\tau \xrightarrow{\mathbb{S}(\kappa)} \tau')$, i.e., f does not depend on anything that may change and its body change-propagates with cost at most κ. In this case, to change propagate `map`'s body, we

[1] Values of type $(A)^{\mathbb{S}}$ may admit indirect changes in nested sub-values. This is explained in Section 3. Also, our refinements $\mathbb{S}$ and $\mathbb{C}$ do not coincide semantically with their homonyms in CostIt. CostIt's refinement $\mathbb{S}$ is semantically equal to a third annotation that we write $\Box$ (see Examples 4 and 5), and CostIt's refinement $\mathbb{C}$ mixes the semantics of our $\mathbb{S}$ and $\mathbb{C}$ refinements. Owing to restrictions in CostIt's type system that we don't want, we do not believe it possible to build a semantically conservative extension of CostIt's refinements.

must only change propagate through f on changed elements of l, of which there are at most α. Hence, the cost is $O(\alpha \cdot \kappa)$ and, indeed, map can be given the following type in CostIt (and our type system) for a suitable linear function h.

$$\begin{aligned}\texttt{map} : \square(\tau \xrightarrow{\mathbb{S}(\kappa)} \tau') \xrightarrow{\mathbb{S}(0)} \forall n, \alpha :: \mathbb{N}. \\ (\texttt{list}\,[n]^{\alpha}\ \tau)^{\mathbb{S}} \xrightarrow{\mathbb{S}(h(\alpha \cdot \kappa))} (\texttt{list}\,[n]^{\alpha}\ \tau')^{\mathbb{S}}\end{aligned}$$

The more interesting question is what happens if we allow f to change, i.e., f has type $(\tau \xrightarrow{\mathbb{C}(\kappa)} \tau')^{\mathbb{C}}$. In this case, change propagation may have to re-execute the function on all list elements from scratch, so the cost of map is $O(n \cdot \kappa)$. This yields the following second type for map for a suitable linear function g.

$$\begin{aligned}\texttt{map} : (\tau \xrightarrow{\mathbb{C}(\kappa)} \tau')^{\mathbb{C}} \xrightarrow{\mathbb{S}(0)} \forall n, \alpha :: \mathbb{N}. \\ (\texttt{list}\,[n]^{\alpha}\ \tau)^{\mathbb{S}} \xrightarrow{\mathbb{S}(g(n \cdot \kappa))} (\texttt{list}\,[n]^{n}\ \tau')^{\mathbb{S}}\end{aligned}$$

Note that even though CostIt can express a similar second type for map, that type is actually more restrictive: In CostIt's interpretation of types, a function labeled $\mathbb{C}$ cannot change arbitrarily, e.g., it cannot change from $\lambda x.\,(x \leq 5)$ to $\lambda x.\,(x \geq 5)$, whereas in our interpretation it can. Finally, if the type of f were $(\tau \xrightarrow{\mathbb{S}(\kappa)} \tau')^{\mathbb{C}}$ (changeable function, whose body change-propagates with cost κ), then we would not be able to type map, because we would not be able to conservatively estimate the cost of change propagation: Since the function may change, change propagation may have to re-execute its body from scratch, but the cost of doing that would be unknown. In fact, our type system prohibits application of functions whose outer annotation is $\mathbb{C}$ and whose arrow has annotation $\mathbb{S}$ (Section 3).

Example 4 (Merge) The following example demonstrates the effect of mode-switching on the typing of recursive functions. Consider the following standard function that merges two sorted (ascending) lists x and y to produce a sorted list.

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

Suppose that the lists x and y have types $\texttt{list}\,[n]^{\alpha}\ (\texttt{real})^{\mathbb{C}}$ and $\texttt{list}\,[m]^{\beta}\ (\texttt{real})^{\mathbb{C}}$, respectively. How can we type merge? Since the list heads $a, b : (\texttt{real})^{\mathbb{C}}$ (they may change), $(a \leq b) : (\texttt{bool})^{\mathbb{C}}$, so as explained in Example 2, the two branches of the if-then-else must be typed in $\mathbb{C}$-mode, i.e., we must establish their from-scratch evaluation costs. This immediately means that for merge, which is recursively called in those branches, we must also calculate the from-scratch cost. Hence, the arrows in merge's type must be annotated $\mathbb{C}$. The rest of the typing is not surprising: It is easy to see that the from-scratch cost is $O(n + m)$ and that even a single change to an input list can cause the entire output to change. This yields the following type for merge, for a suitable linear function h.

$$\begin{aligned}&\texttt{merge} : \square(\forall n, m, \alpha, \beta :: \mathbb{N}.\ \texttt{list}\,[n]^{\alpha}\ (\texttt{real})^{\mathbb{C}} \xrightarrow{\mathbb{C}(1)} \\ &\texttt{list}\,[m]^{\beta}\ (\texttt{real})^{\mathbb{C}} \xrightarrow{\mathbb{C}(h(n+m))} (\texttt{list}\,[n+m]^{n+m}\ (\texttt{real})^{\mathbb{C}})^{\mathbb{C}})\end{aligned}$$

The outer annotation $\square$ on the type of merge states that merge does not depend on anything that can change (this is trivially true because merge is a closed expression). Its significance will become clear in the next example, where we use merge to type merge sort. We note that because merge's type establishes its from-scratch cost, CostIt cannot type merge. (In CostIt, merge is treated as a primitive function.)

Example 5 (Merge sort) This example is taken from CostIt and its analysis is similar to that in CostIt. We present the example here because it highlights central type system features that we inherit from CostIt. The standard function merge sort (msort) divides a given list into two nearly equal sized lists, sorts the two lists recursively and then merges the two sorted lists using the function merge from Example 4.

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

The function bsplit used to split the list has type

$$\begin{aligned}&\texttt{bsplit} : \square(\forall n, \alpha :: \mathbb{N}.\ \texttt{list}\,[n]^{\alpha}\ (\texttt{real})^{\mathbb{C}} \xrightarrow{\mathbb{S}(0)} \\ &\quad \exists \beta.(\texttt{list}\,[\lceil \tfrac{n}{2} \rceil]^{\beta}\ (\texttt{real})^{\mathbb{C}} \times \texttt{list}\,[\lfloor \tfrac{n}{2} \rfloor]^{\alpha-\beta}\ (\texttt{real})^{\mathbb{C}}))\end{aligned}$$

bsplit alternates elements of the input list to the two output lists. Its code is unimportant, but it is important that the function's change propagation cost is 0 (this holds because the function only rearranges the input list's elements, without inspecting them). What is the dynamic stability of msort? Suppose that the input list l has type $\texttt{list}\,[n]^{\alpha}\ (\texttt{real})^{\mathbb{C}}$. msort is a divide-and-conquer algorithm and its trace is a balanced binary tree of height $H = \lceil \log_2(n) \rceil$. At each node of the tree, msort splits the local list, makes two recursive calls to sort the resulting sublists and merges the sorted sublists. Counting tree levels starting from leaves (leaves have level 0), the cost of re-applying merge at a node at level k is at most $h(2^k)$, where h is the linear cost function from Example 4. If α changes occur, then the number of nodes at level k re-evaluated by change propagation is upper bounded by both α, which is the total number of changed leaves, and 2^{H-k}, which is the maximum number of nodes at level k. Hence, the total cost incurred at level k is at most $h(2^k) \cdot \texttt{min}(\alpha, 2^{\lceil \log_2(n) \rceil - k})$. The dynamic stability is, therefore, $Q(n, \alpha) = \sum_{k=0}^{\lceil \log_2(n) \rceil} h(2^k) \cdot \texttt{min}(\alpha, 2^{\lceil \log_2(n) \rceil - k})$. It can be easily shown that for linear h, this is in $O(n \cdot (1 + \log_2(\alpha)))$. For $\alpha = 1$, this yields $O(n)$ and for $\alpha = n$, this yields $O(n \cdot \log n)$, which is the standard from-scratch cost of merge sort.

We now explain briefly how this cost is established in DuCostIt and CostIt. We wish to type msort as follows:

$$\begin{aligned}\texttt{msort} : \square(\forall n, \alpha :: \mathbb{N}.\ \texttt{list}\,[n]^{\alpha}\ (\texttt{real})^{\mathbb{C}} \xrightarrow{\mathbb{S}(Q(n,\alpha))} \\ \texttt{list}\,[n]^{n}\ (\texttt{real})^{\mathbb{C}})\end{aligned}$$

Consider the most interesting case, where the list has at least two elements. Then, inductively, the two recursive calls to msort on the sublists z_1 and z_2 have change propagation costs $Q(\lceil \frac{n}{2} \rceil, \beta)$ and $Q(\lfloor \frac{n}{2} \rfloor, \alpha - \beta)$. Splitting incurs zero cost (for change propagation) and merge has cost $h(\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor) = h(n)$ from Example 4. Consequently, to complete the typing, we must show that

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

This inequality is an arithmetic tautology for $\alpha > 0$ (it is established as a constraint outside our type system). For $\alpha = 0$, this inequality does not hold: The left side is at least $h(n)$, while the right side is 0. To proceed, we observe that when $\alpha = 0$, the list does not change at all, so (dynamically) change propagation has nothing to do. Hence, its cost must be 0. To reflect this observation into the static type system and complete our proof, we introduce a typing rule (called **nochange** in Section 3), which essentially says that if all free variables of an expression are labeled $\square$, then the change propagation cost of the expression is 0. We use this rule on the subexpression starting $\texttt{let}\ (z_1, z_2) = \ldots$. This subexpres-

$$
\begin{array}{lll}
\text{Base types} & B & ::= \ \mathtt{real} \mid \mathtt{unit} \\
\text{Unann. types} & A & ::= \ B \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mathtt{list}\,[n]^{\alpha}\ \tau \mid \\
& & \quad \tau_1 \xrightarrow{\delta(\kappa)} \tau_2 \mid \forall i \overset{\delta(\kappa)}{::} S.\,\tau \mid \exists i{::}S.\,\tau \mid \\
& & \quad C \supset \tau \mid C \,\&\, \tau \\
\text{Types} & \tau & ::= \ (A)^{\mu} \mid \Box(\tau) \\
\text{Modes} & \mu, \epsilon, \delta & ::= \ \mathbb{S} \mid \mathbb{C} \\
\text{Sorts} & S & ::= \ \mathbb{N} \mid \mathbb{R}^{+} \mid \mathbb{V} \\
\text{Index terms} & I, \kappa, & ::= \ i \mid \mu \mid 0 \mid I+1 \mid I_1 + I_2 \mid I_1 - I_2 \mid \\
& n, \alpha & \quad I_1 \cdot I_2 \mid \frac{I_1}{I_2} \mid \lceil I \rceil \mid \lfloor I \rfloor \mid \mathtt{log}_2(I) \mid \\
& & \quad I_1^{I_2} \mid \mathtt{min}(I_1, I_2) \mid \mathtt{max}(I_1, I_2) \mid \\
& & \quad \sum_{i=I_1}^{I_n} I \\
\text{Constraints} & C & ::= \ I_1 \doteq I_2 \mid I_1 < I_2 \mid \neg C \\
\text{Constraint env.} & \Phi & ::= \ \top \mid C \wedge \Phi \\
\text{Sort env.} & \Delta & ::= \ \emptyset \mid \Delta, i :: S \\
\text{Type env.} & \Gamma & ::= \ \emptyset \mid \Gamma, x : \tau \\
\text{Primitive env.} & \Upsilon & ::= \ \emptyset \mid \Upsilon, \zeta : (B_1 \ldots B_n) \xrightarrow{\kappa'} B
\end{array}
$$

Figure 1: Syntax of types

sion has four free variables: `bsplit`, `msort`, `merge` and l. `bsplit` and `merge` are already labeled $\Box$ from their types. `msort` is inductively labeled $\Box$ (this requires a new rule for typing recursive functions). l can be labeled $\Box$ because $\alpha = 0$ and we can always coerce $\mathtt{list}\,[n]^{\alpha}\ \tau$ to $\Box(\mathtt{list}\,[n]^{\alpha}\ \tau)$ when $\alpha = 0$ (no changes are allowed). It follows from the new typing rule that the entire subexpression has cost 0. That completes the proof of `msort`'s type.

Other examples DuCostIt is a conservative extension of CostIt and can type all of CostIt's examples including list append and reverse, matrix transpose, dot products, matrix multiplication, list fold and balanced list fold. Additionally, we have typed several examples where control flow depends on changing data, e.g., `merge` (and, hence, `msort`) shown above and other divide-and-conquer algorithms like the fast Fourier transform (FFT) and list inversion count. In all these examples, change propagation and from-scratch costs established in DuCostIt are asymptotically tight.

3. Syntax and Type System

DuCostIt is a higher-order, call-by-value functional language with recursive functions, lists, sums, products and base types like `reals`. DuCostIt's type system has two kinds of *refinements*—index refinements to track sizes of lists and mode refinements $\mathbb{S}, \mathbb{C}$ to track which values may change. The main novelty in DuCostIt is two different *type effects*—the cost of from-scratch evaluation and the cost of change propagation. Only the latter is a relational effect. The two effects are established using two different typing judgments that interact with each other in the typing rules.

Types The syntax of types is shown in Figure 1. Unannotated types contain most familiar types with some refinements. The list type $\mathtt{list}\,[n]^{\alpha}\ \tau$ is refined with n, the exact length of the list and α, the maximum number of allowed changes ($\alpha \leq n$, even though we don't write this for brevity). Function types $\tau_1 \xrightarrow{\delta(\kappa)} \tau_2$ are refined with an effect κ and a mode δ. If $\delta = \mathbb{S}$, then κ is the cost of change propagating through the body of the function, and if $\delta = \mathbb{C}$, then κ is the cost of executing the body from-scratch. Universally quantified types $\forall i \overset{\delta(\kappa)}{::} S.\,\tau$ are also refined with a cost κ and a mode δ for the closure's body. Constraints C are predicates over index terms (described below). The type $C \supset \tau$ reads "τ if constraint C is true, else every expression" and $C \,\&\, \tau$ reads "τ and constraint C is true".

Unannotated types are refined with annotations $\mathbb{S}$ (stable) and $\mathbb{C}$ (changeable) to obtain annotated types or, simply, types, τ. $(A)^{\mathbb{C}}$ specifies values of unannotated type A that may change arbitrarily between the initial and incremental run, whereas $(A)^{\mathbb{S}}$ specifies values of ground type A whose values cannot change structurally. For base types like `real`, $(\cdot)^{\mathbb{S}}$ specifies values that cannot change at all. For lists and products, the annotation has no specific meaning (it is present only for technical convenience in writing typing rules). On sums, the annotation $\mathbb{S}$ means that the value is not allowed to change from `inl _` to `inr _` or vice versa (whether the value within `inl` or `inr` may change is determined by the nested annotations in the two components of the sum type). On function types, the annotation $\mathbb{S}$ means that the function's body cannot change syntactically, but it may capture free changeable variables from outer contexts. Thus, if $y : (\mathtt{real})^{\mathbb{C}}$, then both functions $\lambda x.\,x$ and $\lambda x.\,(y+1;\,x)$ have type $(\tau \xrightarrow{\mathbb{S}(\kappa)} \tau)^{\mathbb{S}}$ for an appropriate κ. The stronger annotation $\Box(\tau)$ represents values of τ that cannot even *depend* on changeable variables from outer contexts and, hence, cannot change at all. Thus, $\lambda x.\,(y+1;\,x)$ does not have type $\Box((\tau \xrightarrow{\mathbb{S}(\kappa)} \tau)^{\mathbb{S}})$, but $\lambda x.\,x$ does. Technically, $\Box(\tau)$ is a co-monadic type (see subtyping later in this section).

$$
\begin{array}{lll}
\text{Values} & v ::= & \mathtt{r} \mid (v_1, v_2) \mid \mathtt{inl}\ v \mid \mathtt{inr}\ v \mid \mathtt{nil} \mid \\
& & \mathtt{cons}(v_1,\, v_2) \mid \mathtt{fix}\ f(x).\,e \mid \Lambda.\,e \mid \\
& & \mathtt{pack}\ v \mid () \\
\text{Expr.} & e ::= & x \mid \mathtt{r} \mid (e_1, e_2) \mid \mathtt{fst}\ e \mid \mathtt{snd}\ e \mid \mathtt{inl}\ e \mid \mathtt{inr}\ e \mid \\
& & \mathtt{case}(e, x.e_1, y.e_2) \mid \mathtt{nil} \mid \mathtt{cons}(e_1,\, e_2) \mid \\
& & (\mathtt{case_L}\ e\ \mathtt{of\ nil}\ \rightarrow\ e_1 \mid \mathtt{cons}(h, tl)\ \rightarrow\ e_2) \mid \\
& & \mathtt{fix}\ f(x).\,e \mid e_1\ e_2 \mid \zeta\ e \mid \Lambda.\,e \mid e[] \mid \\
& & \mathtt{pack}\ e \mid \mathtt{unpack}\ e\ \mathtt{as}\ x\ \mathtt{in}\ e' \mid \\
& & \mathtt{let}\ x\ =\ e_1\ \mathtt{in}\ e_2 \mid e.\mathtt{_c} \mid \mathtt{clet}\ e_1\ \mathtt{as}\ x\ \mathtt{in}\ e_2 \mid ()
\end{array}
$$

Figure 2: Syntax of expressions and values

Index terms Static index terms I, κ, n, α that refine DuCostIt's types are classified into the following sorts: (a) natural numbers, $\mathbb{N}$, which are used to specify list sizes and the number of allowed changes in lists, (b) non-negative real numbers, $\mathbb{R}^{+}$, that appear in logarithmic expressions in costs and (c) the two-valued sort $\mathbb{V} = \{\mathbb{S}, \mathbb{C}\}$, whose primary purpose has been explained above. Most operators are overloaded for the sorts $\mathbb{R}^{+}$ and $\mathbb{N}$ and there is an implicit coercion from $\mathbb{N}$ to $\mathbb{R}^{+}$. Sorts are assigned to index terms via a sorting judgment $\Delta \vdash I :: S$, whose details we omit. Δ is a sort environment that maps index variables (denoted i, t) to their sorts.

Expressions The grammar of DuCostIt's values and expressions is shown in Figure 2. Most of the syntax is standard. `r` denotes constants of type `real`. ζ denotes a primitive function and $\zeta\ e$ is application of the function to e. The construct $\mathtt{case_L}$ is case analysis on lists. Quantification and instantiation over index variables are written $\Lambda.\,e$ and $e[]$, respectively. Elimination forms for constrained types $C \supset \tau$ and $C \,\&\, \tau$ are written $e.\mathtt{_c}$ and $\mathtt{clet}\ x\ \mathtt{as}\ e_1\ \mathtt{in}\ e_2$, respectively. Expressions do not mention any index terms. This is done to simplify the code of programs that case analyze lists, as in CostIt.

Constraints and assumptions Constraints C are predicates over index terms. Our subtyping rules critically rely on constraint entail-

$\boxed{\Delta; \Phi; \Gamma \vdash_\epsilon e : \tau \mid \kappa}$ expression e has type τ. We use ϵ to stand for $\mathbb{S}$ or $\mathbb{C}$. Every rule should be read separately as its instantiation for both possible values of ϵ. If $\epsilon = \mathbb{S}$, then the change propagation cost of e is at most κ. If $\epsilon = \mathbb{C}$, then the from-scratch cost of e is at most κ.

$$\frac{\kappa = ((\epsilon \doteq \mathbb{C})\ ?\ c_{var}()\ :\ 0)}{\Delta; \Phi; \Gamma, x : \tau \vdash_\epsilon x : \tau \mid \kappa}\ \textbf{var} \qquad \frac{\kappa = ((\epsilon \doteq \mathbb{C})\ ?\ c_{real}()\ :\ 0)}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{r} : (\mathtt{real})^{\mathbb{S}} \mid \kappa}\ \textbf{real} \qquad \frac{\Delta; \Phi \vdash \tau\ \mathtt{wf} \qquad \kappa = ((\epsilon \doteq \mathbb{C})\ ?\ c_{nil}()\ :\ 0)}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{nil} : (\mathtt{list}\,[0]^{0}\ \tau)^{\mathbb{S}} \mid \kappa}\ \textbf{nil}$$

$$\frac{\begin{array}{c}\Delta; \Phi \vdash (\tau_1 \xrightarrow{\delta(\kappa)} \tau_2)^{\mathbb{S}}\ \mathtt{wf} \\ \Delta; \Phi; f : (\tau_1 \xrightarrow{\delta(\kappa)} \tau_2)^{\mathbb{S}}, x : \tau_1, \Gamma \vdash_\delta e : \tau_2 \mid \kappa' \\ \kappa = ((\epsilon \doteq \mathbb{C})\ ?\ c_{fix}()\ :\ 0)\end{array}}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{fix}\, f(x).\, e : (\tau_1 \xrightarrow{\delta(\kappa')} \tau_2)^{\mathbb{S}} \mid \kappa}\ \textbf{fix1} \qquad \frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash_\epsilon e_1 : (\tau_1 \xrightarrow{\delta(\kappa')} \tau_2)^{\mu} \mid \kappa_1 \qquad \Delta; \Phi; \Gamma \vdash_\epsilon e_2 : \tau_1 \mid \kappa_2 \\ \models (\epsilon \sqcup \mu) \leq \delta \qquad \models \mu \trianglelefteq \tau_2 \\ \kappa = \kappa_1 + \kappa_2 + \kappa' + (((\epsilon \sqcup \mu) \doteq \mathbb{C})\ ?\ c_{app}(\epsilon, \mu)\ :\ 0)\end{array}}{\Delta; \Phi; \Gamma \vdash_\epsilon e_1\ e_2 : \tau_2 \mid \kappa}\ \textbf{app}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash_\epsilon e : \tau_1 \mid \kappa' \qquad \Delta; \Phi \vdash \tau_2\ \mathtt{wf} \\ \kappa = \kappa' + ((\epsilon \doteq \mathbb{C})\ ?\ c_{inl}()\ :\ 0)\end{array}}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{inl}\, e : (\tau_1 + \tau_2)^{\mathbb{S}} \mid \kappa}\ \textbf{inl} \qquad \frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash_\epsilon e : (\tau_1 + \tau_2)^{\mu} \mid \kappa_e \qquad \Delta; \Phi; \Gamma, x : \tau_1 \vdash_{\epsilon \sqcup \mu} e_1 : \tau \mid \kappa' \\ \Delta; \Phi; \Gamma, y : \tau_2 \vdash_{\epsilon \sqcup \mu} e_2 : \tau \mid \kappa' \qquad \models \mu \trianglelefteq \tau \\ \kappa = \kappa_e + \kappa' + (((\epsilon \sqcup \mu) \doteq \mathbb{C})\ ?\ c_{case}(\epsilon, \mu)\ :\ 0)\end{array}}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{case}(e, x.e_1, y.e_2) : \tau \mid \kappa}\ \textbf{case}$$

$$\frac{\Delta; \Phi; \Gamma \vdash_\epsilon e_1 : \square(\tau) \mid \kappa_1 \qquad \Delta; \Phi; \Gamma \vdash_\epsilon e_2 : (\mathtt{list}\,[n]^{\alpha}\ \tau)^{\mu} \mid \kappa_2 \qquad \kappa = \kappa_1 + \kappa_2 + (((\epsilon \sqcup \mu) \doteq \mathbb{C})\ ?\ c_{cons}()\ :\ 0)}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{cons}(e_1,\, e_2) : (\mathtt{list}\,[n{+}1]^{\alpha}\ \tau)^{\mu} \mid \kappa}\ \textbf{cons1}$$

$$\frac{\Delta; \Phi; \Gamma \vdash_\epsilon e_1 : \tau \mid \kappa_1 \qquad \Delta; \Phi; \Gamma \vdash_\epsilon e_2 : (\mathtt{list}\,[n]^{\alpha - 1}\ \tau)^{\mu} \mid \kappa_2 \qquad \Delta; \Phi \models \alpha > 0 \qquad \kappa = \kappa_1 + \kappa_2 + (((\epsilon \sqcup \mu) \doteq \mathbb{C})\ ?\ c_{cons}()\ :\ 0)}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{cons}(e_1,\, e_2) : (\mathtt{list}\,[n{+}1]^{\alpha}\ \tau)^{\mu} \mid \kappa}\ \textbf{cons2}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash_\epsilon e : (\mathtt{list}\,[n]^{\alpha}\ \tau)^{\mu} \mid \kappa_e \\ \Delta; \Phi \wedge n \doteq 0 \wedge \alpha \doteq 0; \Gamma \vdash_{\epsilon \sqcup \mu} e_1 : \tau' \mid \kappa' \qquad i :: \iota, \Delta; \Phi \wedge n \doteq i + 1 \wedge \alpha \leq i; h : \square(\tau), tl : (\mathtt{list}\,[i]^{\alpha}\ \tau)^{\mu}, \Gamma \vdash_{\epsilon \sqcup \mu} e_2 : \tau' \mid \kappa' \\ i :: \iota, \beta :: \iota, \Delta; \Phi \wedge n \doteq i + 1 \wedge \beta \leq i \wedge \alpha \doteq \beta + 1; h : \tau, tl : (\mathtt{list}\,[i]^{\beta}\ \tau)^{\mu}, \Gamma \vdash_{\epsilon \sqcup \mu} e_2 : \tau' \mid \kappa' \\ \models \mu \trianglelefteq \tau' \qquad \kappa = \kappa_e + \kappa' + (((\epsilon \sqcup \mu) \doteq \mathbb{C})\ ?\ c_{caseL}(\epsilon, \mu)\ :\ 0)\end{array}}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{case_L}\ e\ \mathtt{of}\ \mathtt{nil}\ \rightarrow\ e_1 \mid \mathtt{cons}(h, tl)\ \rightarrow\ e_2 : \tau' \mid \kappa}\ \textbf{caseL}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash_\epsilon e : \tau \mid \kappa' \qquad \forall x \in dom(\Gamma).\ \Delta; \Phi \models \Gamma(x) \sqsubseteq \square\,\Gamma(x) \\ \kappa = ((\epsilon \doteq \mathbb{S}\ ?\ 0\ :\ \kappa'))\end{array}}{\Delta; \Phi; \Gamma, \Gamma' \vdash_\epsilon e : \square(\tau) \mid \kappa}\ \textbf{nochange}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash_\epsilon e : \tau \mid \kappa' \qquad \forall x \in \Gamma\ \ \Delta; \Phi \wedge C \models \Gamma(x) \sqsubseteq \square(\Gamma(x)) \\ \Delta; \Phi \wedge \neg C \models \kappa' \leq \kappa\end{array}}{\Delta; \Phi; \Gamma \vdash_\epsilon e : \tau \mid \kappa}\ \textbf{r-split}$$

Figure 3: Selected typing rules. The context that carries types of primitive functions is omitted from all rules.

ment, which is represented as $\Delta; \Phi \models C$ ("for any substitution of index variables in Δ, the constraints Φ entail the constraint C"). We do not stipulate syntactic rules for this judgment, but assume that it codifies the standard rules of arithmetic. Constraints are also subject to a standard syntactic well-formedness judgment $\Delta \vdash C\ \mathtt{wf}$, which we omit entirely.

Typing rules DuCostIt relies on two typing judgments — $\Delta; \Phi; \Gamma \vdash_{\mathbb{S}} e : \tau \mid \kappa$ and $\Delta; \Phi; \Gamma \vdash_{\mathbb{C}} e : \tau \mid \kappa$, which say that κ is an upper bound on the cost of change propagating through e and evaluating e from-scratch, respectively. However, the cost of change propagation is no more than the cost of evaluating from-scratch, so the second judgment implies the first semantically and, hence, it is perfectly sound to change propagate through expressions typed with either judgment. We rely on this property heavily in our semantics. For brevity in writing rules, we often use the unified notation $\Delta; \Phi; \Gamma \vdash_\epsilon e : \tau \mid \kappa$ with $\epsilon \in \{\mathbb{S}, \mathbb{C}\}$ being the *mode of the typing judgment*.[2] Besides the typing environments Γ and Δ for type and index variables, respectively, an extra environment Φ records assumed constraints. The judgments also include a fourth context that specifies the types of primitive functions ζ, but this context does not change in the rules, so we exclude it from the presentation.

Important typing rules are shown in Figure 3. If an expression contains subexpressions, then the costs (κ's) of subexpressions are added to obtain the total cost of the expression. In addition, each language construct incurs a runtime execution cost that depends on the mode of evaluation, ϵ, and, in the case of elimination constructs, the annotation on the type of the eliminated value (which we uniformly denote with μ). For instance, if we are change propagating ($\epsilon = \mathbb{S}$) and we encounter a case analysis over a sum annotated $\mathbb{C}$ (i.e., $\mu = \mathbb{C}$), then because the sum may have been `inl _` in the first run and may be `inr _` now, we must incur some cost to determine the current tag. If, on the other hand, the sum were annotated $\mathbb{S}$ (i.e., $\mu = \mathbb{S}$), we know statically that this tag would not change, so there would be no need to read it during change propagation. In the type system, we use meta-symbols like $c_{case}(\epsilon, \mu)$ to denote such ϵ- and μ-dependent costs. The concrete semantics (Section 5) determine the exact definitions of these meta-symbols. For asymptotic cost analysis, the exact constants represented by these meta-symbols are irrelevant, so we defer their details to our appendix. However, even for asymptotic analysis, we need to know whether a cost is zero or non-zero, so when the cost of a construct is zero, we write this explicitly in the typing rule.

We explain some of the rules here. In a call-by-value language like ours, variables are substituted by values and the cost of updating (change propagating) the substitution for a variable is paid by the context that provides the substitution. So a variable incurs zero

[2] The annotation ϵ on typing judgments is based on a similar annotation called pc in type systems for information flow analysis [26]. Our entire type system can be viewed as a substantial extension of the pure fragment of [26] with index refinements and costs. Our models of types (Sections 4 and 5) are tailored to incremental computation and substantially differ from models of types needed for information flow analysis.

cost during change propagation ($\epsilon = \mathbb{S}$). On the other hand, during from-scratch evaluation ($\epsilon = \mathbb{C}$), a variable incurs a constant cost, denoted $c_{var}()$, to copy the variable's value to a place where it can be used by its context (during change propagation, the value is updated in-situ, so this cost of copying does not have to be paid). This explains the premise $\kappa = ((\epsilon \doteq \mathbb{C})\ ?\ c_{var}()\ :\ 0)$ of the rule **var**. A similar premise appears for all value forms, including functions (rule **fix1**) and constants (rule **real**).

Rules **fix1** and **app** type recursive functions and function applications, respectively. In rule **fix1**, the body of the function is typed in the same mode as the annotation δ in the function's type $(\tau_1 \xrightarrow{\delta(\kappa')} \tau_2)^{\mathbb{S}}$. The annotation on the function's type is $\mathbb{S}$ because the function is constructed within the program, so it will not change syntactically across runs. This principle also applies to all value introduction forms (for instance, the rules **inl** and **real**).

In the rule **app**, the meta-function $\epsilon \sqcup \mu$ returns $\mathbb{C}$ when either $\epsilon = \mathbb{C}$ or $\mu = \mathbb{C}$, else it returns $\mathbb{S}$. The partial order $\leq$ on annotations is defined by $\mathbb{S} \leq \mathbb{C}$ and $\mu \leq \mu$. The rule is best understood separately for $\epsilon = \mathbb{C}$ and $\epsilon = \mathbb{S}$. When $\epsilon = \mathbb{C}$, we may be executing from-scratch so the function can be applied only if its body was typed in mode $\mathbb{C}$, i.e., only if $\delta = \mathbb{C}$ (else we cannot count the cost of executing the body soundly). This is forced by the premise $\epsilon \sqcup \mu \leq \delta$. When $\epsilon = \mathbb{S}$, we are change propagating, so the function can be applied independent of the mode in which its body was typed, but if the function itself may change completely ($\mu = \mathbb{C}$), then we may have to run the function's body from-scratch, so the function's body must have been typed with $\delta = \mathbb{C}$. Again, this is forced by $\epsilon \sqcup \mu \leq \delta$. Finally, if the function may change ($\mu = \mathbb{C}$), then the result type τ_2 must also have annotation $\mathbb{C}$, as the result may change completely. This is checked by the premise $\models \mu \trianglelefteq \tau_2$, which holds when $\tau_2 = (A)^{\mu'}$ and $\mu \leq \mu'$.

The rule **case** eliminates a sum type and is similar to **app**: Whenever the scrutinee is $\mathbb{C}$-annotated, the branches are typed in with $\epsilon = \mathbb{C}$ because the branch executed in the incremental run may be different from that executed in the first run. The rules **cons1** and **cons2** type non-empty lists of type $(\mathtt{list}\,[n+1]^{\alpha}\ \tau)^{\mu}$. Depending on whether the head may change or not, the tail expression is permitted either α or $\alpha - 1$ changes. Correspondingly, the list elimination rule **caseL** considers three cases. If the list is empty, then the number of changes and the size of the list are both 0 (second premise). If the list is not empty, then there are two possibilities: the head of the list does not change and the tail has up to α changes (third premise) or the head of the list may change and the tail has up to $\beta = \alpha - 1$ changes (fourth premise). In all cases, if the eliminated list is changeable, i.e. $\mu = \mathbb{C}$, then we switch to $\epsilon = \mathbb{C}$ for typing the case branches.

Note that the premises of the rules **app**, **case** and **caseL** may be typed with $\epsilon = \mathbb{C}$, even when the conclusion is typed with $\epsilon = \mathbb{S}$. Thus, we may switch from $\mathbb{S}$-mode to $\mathbb{C}$-mode in constructing a typing derivation bottom-up. The reverse transition may also occur but only in typing a closure's body, as in **fix1**.

The rule **nochange** captures the intuition that if no dependencies (substitutions for free variables) of an expression can change, then the expression's result cannot change and there is no need to change propagate through its trace (i.e., its change propagation cost is zero). The second premise of **nochange** checks that the types of all variables can be subtyped to the form $\square(\cdot)$, which ensures that the dependencies of the expression cannot change. The rule's conclusion allows the type to be annotated $\square(\cdot)$ and, additionally, if $\epsilon = \mathbb{S}$, then the cost κ is 0. For from-scratch evaluation ($\epsilon = \mathbb{C}$), the rule has no effect on the cost.

In typing many programs like merge sort, we case analyze whether or not a list has any allowed changes. For this, we need a case analysis rule for constraints, such as the following straightforward rule:

$$\frac{\Delta; \Phi \wedge C; \Gamma \vdash_{\epsilon} e : \tau \mid \kappa \qquad \Delta; \Phi \wedge \neg C; \Gamma \vdash_{\epsilon} e : \tau \mid \kappa}{\Delta; \Phi; \Gamma \vdash_{\epsilon} e : \tau \mid \kappa}\ \textbf{split}$$

However, this rule is incompatible with our concrete semantics, where the two premises may get translated in incompatible ways. Accordingly, we restrict the type system to a special case of this rule, where this rule is immediately preceded by the rule **nochange** in the first premise. The resulting rule, **r-split** in Figure 3, can be derived using **nochange** and **split** and also suffices for typing all examples we have encountered so far.

$\boxed{\Delta; \Phi \models \tau_1 \sqsubseteq \tau_2}$ 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

$$\frac{\Delta; \Phi \models^{\mathsf{A}} A_1 \sqsubseteq A_2}{\Delta; \Phi \models (A_1)^{\mu} \sqsubseteq (A_2)^{\mu}}\ \mathbf{C} \qquad \frac{\Delta; \Phi \models \mu_1 \leq \mu_2}{\Delta; \Phi \models (A)^{\mu_1} \sqsubseteq (A)^{\mu_2}}\ \boldsymbol{\mu}$$

$$\frac{\Delta; \Phi \models \tau_1' \sqsubseteq \tau_1 \qquad \Delta; \Phi \models \tau_2 \sqsubseteq \tau_2' \qquad \Delta; \Phi \models \kappa \leq \kappa'}{\Delta; \Phi \models^{\mathsf{A}} \tau_1 \xrightarrow{\delta(\kappa)} \tau_2 \sqsubseteq \tau_1' \xrightarrow{\delta(\kappa')} \tau_2'}\ \boldsymbol{\rightarrow}$$

$$\frac{}{\Delta; \Phi \models \square((\tau_1 \xrightarrow{\delta(\kappa)} \tau_2)^{\mu}) \sqsubseteq (\square(\tau_1) \xrightarrow{\delta(\kappa)} \square(\tau_2))^{\mathbb{S}}}\ \boldsymbol{\rightarrow\square}$$

$$\frac{\Delta; \Phi \models n_1 \doteq n_2 \qquad \Delta; \Phi \models \alpha_1 \leq \alpha_2 \leq n_2 \qquad \Delta; \Phi \models \tau_1 \sqsubseteq \tau_2}{\Delta; \Phi \models^{\mathsf{A}} \mathtt{list}\,[n_1]^{\alpha_1}\ \tau_1 \sqsubseteq \mathtt{list}\,[n_2]^{\alpha_2}\ \tau_2}\ \mathbf{l1}$$

$$\frac{\Delta; \Phi \models \alpha \doteq 0}{\Delta; \Phi \models^{\mathsf{A}} \mathtt{list}\,[n]^{\alpha}\ \tau \equiv \mathtt{list}\,[n]^{\alpha}\ \square(\tau)}\ \mathbf{l2}$$

$$\frac{}{\Delta; \Phi \models \square((\mathtt{list}\,[n]^{\alpha}\ \tau)^{\mu}) \equiv (\mathtt{list}\,[n]^{\alpha}\ \square(\tau))^{\mathbb{S}}}\ \mathbf{l\square}$$

$$\frac{}{\Delta; \Phi \models \square(\tau) \sqsubseteq \tau}\ \mathbf{T}$$

Figure 4: Selected subtyping rules

Subtyping Subtyping plays a crucial role in DuCostIt. Subtyping is constraint dependent. The subtyping judgment $\Delta; \Phi \models \tau_1 \sqsubseteq \tau_2$ states that τ_1 is a subtype of τ_2 under the index environment Δ and constraints Φ. We write $\tau_1 \equiv \tau_2$ for $\tau_1 \sqsubseteq \tau_2$ and $\tau_2 \sqsubseteq \tau_1$. Selected rules are shown Figure 4. The rule $\boldsymbol{\mu}$ allows weakening of annotations along the order $\leq$ on $\{\mathbb{S}, \mathbb{C}\}$. In particular, $(A)^{\mathbb{S}} \sqsubseteq (A)^{\mathbb{C}}$. This subtyping is immediately justified by the intuitive meanings of the annotations $(A)^{\mathbb{S}}$ and $(A)^{\mathbb{C}}$. The rule $\rightarrow$ is the subtyping rule for functions, contravariant in the argument and covariant in the result and cost (as expected). The rule **l1** allows the number of changes in a list to be weakened as long as the revised number does not exceed the size of the list. The rule **l2** allows a list with 0 changes to be re-typed as a list whose elements' type is labeled $\square(\cdot)$. In addition, the rule **l**$\square$ states that $\square((\mathtt{list}\,[n]^{\alpha}\ \tau)^{\mu}) \equiv (\mathtt{list}\,[n]^{\alpha}\ \square(\tau))^{\mathbb{S}}$: A list that is not allowed to change, represented by the outer $\square$ on the left side, is equivalent to a list whose elements cannot change, represented by the inner $\square$ on the right side. The rules **l2** and **l**$\square$ are critical for typing Example 5 of Section 2.

For readers familiar with co-monadic types, we note that the type $\square(\tau)$ is a co-monad: $\square(\tau) \sqsubseteq \tau$ (rule **T**) and $\square(\tau_1 \rightarrow \tau_2) \sqsubseteq \square(\tau_1) \rightarrow \square(\tau_2)$ (rule $\boldsymbol{\rightarrow\square}$). The rule **l**$\square$ for lists is analogous to the standard co-monadic property $\square(\tau_1 \times \tau_2) \equiv \square(\tau_1) \times \square(\tau_2)$.

4. Abstract Semantics and Soundness

In this section, we define abstract cost-counting semantics for change propagation and for from-scratch evaluation. We then prove our type system sound relative to this abstract semantics. Later, in

$\boxed{e \Downarrow \langle v, D\rangle, f}$ e evaluates to value v with derivation D in f steps

$$\frac{}{v \Downarrow \langle v, v\rangle, 0}\ \textbf{value} \qquad \frac{F = \mathtt{fix}\, f(x).\, e}{\mathtt{fix}\, f(x).\, e \Downarrow \langle F, F\rangle, c_{fix}()}\ \textbf{fix}$$

$$\frac{e_1 \Downarrow T_1, f_1 \qquad e_2 \Downarrow T_2, f_2 \qquad v_i = \mathtt{V}(T_i)}{\mathtt{cons}(e_1,\, e_2) \Downarrow \langle \mathtt{cons}(v_1,\, v_2), \mathtt{cons}(T_1, T_2)\rangle, f_1 + f_2 + c_{cons}(\mathbb{C}, \_)}\ \textbf{cons}$$

$$\frac{e \Downarrow T, f \qquad \mathtt{inl}\; v = \mathtt{V}(T) \qquad e_1[v/x] \Downarrow T_r, f_r \qquad v_r = \mathtt{V}(T_r)}{\mathtt{case}(e, x.e_1, y.e_2) \Downarrow \langle v_r, \mathtt{case}_{\mathtt{inl}}(T, T_r)\rangle, f + f_r + c_{case}(\mathbb{C}, \_)}\ \textbf{case-l}$$

Figure 5: Selected evaluation rules

Section 5, we show how these abstract semantics can be realized by translation to an ML-like language.

Evaluation semantics and traces Our big-step call-by-value evaluation judgment $e \Downarrow T, f$ states that expression e evaluates to a trace T with evaluation cost f. The trace T is a representation of the entire big-step derivation and explicitly includes the final and all intermediate values. It is a pair $\langle v, D\rangle$, where v is the result of the evaluation and D is a derivation, which recursively contains subtraces. For every big-step evaluation rule, there is one derivation constructor. The syntax below shows only some of the constructors for brevity. The constructors for case analysis record which branch was taken using subscripts like `inl` or `inr`. Selected evaluation rules are shown in Figure 5. The rules are obtained by adding costs to standard big-step evaluation rules. The costs are based on meta-symbols like c_{app} that the type system also uses. The meta function $\mathtt{V}(\cdot)$ returns the final value contained in a trace: $\mathtt{V}(\langle v, D\rangle) = v$.

$$\begin{array}{lll} \text{Traces} & T ::= & \langle v, D\rangle \\ \text{Derivations} & D ::= & v \mid \mathtt{r} \mid (T_1, T_2) \mid \mathtt{fst}\; T \mid \mathtt{snd}\; T \mid \mathtt{inl}\; T \mid \\ & & \mathtt{inr}\; T \mid \mathtt{case}_{\mathtt{inl}}(T, T_r) \mid \mathtt{case}_{\mathtt{inr}}(T, T_r) \mid \ldots \end{array}$$

Changes and biexpressions Change propagation takes an expression's evaluation trace and a modified expression and produces an updated trace and a new output result (along with a cost of doing all the updates). Change propagation is bottom up: It starts from modified leaves of the expression (trace) and moves towards the root, updating values in place. When there is a change to control flow, e.g., an updated value causes a fresh branch or a fresh closure to be executed, change propagation switches to from-scratch evaluation. Importantly, change propagation bypasses parts of the trace whose leaves have not changed and is thus faster than evaluating the whole expression from scratch.

To formalize change propagation, we first need notation to specify where an expression has changed. For this we adapt CostIt's *biexpressions*. A biexpression, denoted $\mathbb{e}$, represents in a single syntax two expressions—the original one and the updated one—that share most structure, but may differ at some leaves. To represent differing leaves, we use the biexpression constructor $\mathtt{new}(v_1, v_2)$, which represents v_1 in the first run and v_2 in the second run. v_1 and v_2 do not have to be related to each other. CostIt restricted this constructor to the case where v_1 and v_2 are primitive values (like `reals`). Although this change is syntactically small, it has deep implications. In particular, by allowing sums and functions to change arbitrarily, we allow for changes to control flow during change propagation and, hence, we need to switch to from-scratch evaluation during change propagation.

The syntax of bivalues and biexpressions is shown below. The syntax mirrors the syntax of values and expressions. The construct $\mathtt{keep}(\mathtt{r})$ represents a real number $\mathtt{r}$ that has not changed. As explained above, $\mathtt{new}(v_1, v_2)$ represents v_1 in the first run and an unrelated valued v_2 in the second run. The remaining constructors are interpreted homomorphically over pairs. For instance, $\mathtt{fix}\; f(x).(x + \mathtt{new}(1, 2))$ represents $\mathtt{fix}\; f(x).\, x + 1$ in the first run and $\mathtt{fix}\; f(x).\, x + 2$ in the second run. More generally, we define the functions $\mathrm{L}(\mathbb{e})$ and $\mathrm{R}(\mathbb{e})$ that project the first-run ("left") and second-run ("right") expressions from $\mathbb{e}$ as the homomorphic lifts of the following rules: $\mathrm{L}(\mathtt{keep}(\mathtt{r})) = \mathrm{R}(\mathtt{keep}(\mathtt{r})) = \mathtt{r}$, $\mathrm{L}(\mathtt{new}(v_1, v_2)) = v_1$ and $\mathrm{R}(\mathtt{new}(v_1, v_2)) = v_2$.

$$\begin{array}{lll} \text{Bival.} & \mathbb{w} ::= & \mathtt{keep}(\mathtt{r}) \mid \mathtt{new}(v, v') \mid (\mathbb{w}_1, \mathbb{w}_2) \mid \\ & & \mathtt{inl}\; \mathbb{w} \mid \mathtt{inr}\; \mathbb{w} \mid \mathtt{nil} \mid \mathtt{cons}(\mathbb{w}_1, \mathbb{w}_2) \mid \\ & & \mathtt{fix}\; f(x).\mathbb{e} \mid \Lambda.\mathbb{e} \mid \mathtt{pack}\; \mathbb{w} \mid () \\ \text{Biexpr.} & \mathbb{e} ::= & x \mid \mathtt{keep}(\mathtt{r}) \mid \mathtt{new}(v, v') \mid (\mathbb{e}_1, \mathbb{e}_2) \mid \\ & & \mathtt{fst}\; \mathbb{e} \mid \mathtt{snd}\; \mathbb{e} \mid \mathtt{inl}\; \mathbb{e} \mid \mathtt{inr}\; \mathbb{e} \mid \\ & & \mathtt{case}(\mathbb{e}, x.\mathbb{e}_1, y, \mathbb{e}_2) \mid \ldots \end{array}$$

$\boxed{\Delta; \Phi; \Gamma \vdash_\epsilon \mathbb{w} \gg \tau \text{ and } \Delta; \Phi; \Gamma \vdash_\epsilon \mathbb{e} \gg \tau \mid \kappa}$ Bivalue and biexpression typing

$$\frac{}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{keep}(\mathtt{r}) \gg (\mathtt{real})^{\mathbb{S}}}\ \textbf{keep-r}$$

$$\frac{\Delta; \Phi; \cdot \vdash_{\mathbb{C}} v : \tau \mid \kappa \qquad \Delta; \Phi; \cdot \vdash_{\mathbb{C}} v' : \tau \mid \kappa' \qquad \models \mathbb{C} \trianglelefteq \tau}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{new}(v, v') \gg \tau}\ \textbf{new}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash_\epsilon \mathbb{w} \gg \tau \\ \forall x \in \Gamma.\ \Delta; \Phi \models \Gamma(x) \sqsubseteq \Box(\Gamma(x)) \qquad \mathtt{stable}(\mathbb{w})\end{array}}{\Delta; \Phi; \Gamma, \Gamma' \vdash_\epsilon \mathbb{w} \gg \Box(\tau)}\ \textbf{nochange}$$

$$\frac{\Delta; \Phi; \Gamma \vdash_\epsilon \mathbb{w}_i \gg \tau_i \qquad \Delta; \Phi; \overline{x_i : \tau_i}, \Gamma \vdash_\epsilon e : \tau \mid \kappa}{\Delta; \Phi; \Gamma \vdash_\epsilon \ulcorner e \urcorner [\overline{\mathbb{w}_i / x_i}] \gg \tau \mid \kappa}\ \textbf{exp}$$

Figure 6: Selected typing rules for bivalues and biexpressions

Both bivalues and biexpressions are typed. Selected typing rules are shown in Figure 6. The judgment $\Delta; \Phi; \Gamma \vdash_\epsilon \mathbb{w} \gg \tau$ states that the bivalue $\mathbb{w}$ represents a valid change from an initial value $\mathrm{L}(\mathbb{w})$ of type τ to the modified value $\mathrm{R}(\mathbb{w})$ of type τ. The typing rules for bivalues mirror those for values. The construct $\mathtt{keep}(\mathtt{r})$ is typed at $(\mathtt{real})^{\mathbb{S}}$ since it represents a real number that did not change. The construct $\mathtt{new}(v_1, v_2)$ can be typed at τ only if τ is labeled $\mathbb{C}$ (premise $\models \mathbb{C} \trianglelefteq \tau$ in rule **new**). There is only one rule, **exp**, for typing biexpressions. This rule uses explicit substitutions for technical convenience. We could also have written equivalent syntax-directed rules for typing biexpressions. The notation $\ulcorner e \urcorner$ denotes the biexpression that represents e in both the first and second runs. It is obtained by replacing every primitive constant like $\mathtt{r}$ in e with $\mathtt{keep}(\mathtt{r})$.

Change propagation Change propagation is formalized abstractly by the judgment $\langle T, \mathbb{e}\rangle \curvearrowright \mathbb{w}', T', c'$. It takes as inputs the trace T and the biexpression $\mathbb{e}$ and it returns $\mathbb{w}', T'$ and c'. The input T must be the trace that is obtained from executing the original expression $\mathrm{L}(\mathbb{e})$. The bivalue $\mathbb{w}'$ resulting from change propagation represents two values, $\mathrm{L}(\mathbb{w}')$ and $\mathrm{R}(\mathbb{w}')$, which are the results of evaluating the original and modified expressions, respectively. The output T' is the trace of the modified expression. The non-negative number c' represents the total cost incurred in change propagation.

Selected rules for change propagation are shown in Figure 7. The rules case analyze the syntax of $\mathbb{e}$. The most important rule is **r-nochange**. Its premise, $\mathtt{stable}(\mathbb{e})$ holds when $\mathbb{e}$ does not contain $\mathtt{new}(\cdot, \cdot)$ anywhere, i.e., when $\mathbb{e}$ represents an expression that has not changed. In this case, the value v stored in the original trace is output immediately (technically, it must be cast into the bivalue $\ulcorner v \urcorner$) and the cost of change propagation is 0.

To change propagate $\mathtt{case}(\mathbb{e}, x.\mathbb{e}_1, y, \mathbb{e}_2)$, we first change propagate through the scrutinee $\mathbb{e}$. If the initial and incremental

$\boxed{\langle D, \mathtt{ee}\rangle \curvearrowright \mathtt{w}', D', c'}$ Change propagation with cost-counting

$$\frac{\mathtt{stable}(\mathtt{ee})}{\langle\langle v, D\rangle, \mathtt{ee}\rangle \curvearrowright \ulcorner v \urcorner, \langle v, D\rangle, 0}\ \textbf{r-nochange}$$

$$\frac{}{\langle\langle v, D\rangle, \mathtt{new}(\_, v')\rangle \curvearrowright \mathtt{new}(v, v'), \langle v', v'\rangle, 0}\ \textbf{r-new}$$

$$\frac{\begin{array}{c}\langle T, \mathtt{ee}\rangle \curvearrowright \mathtt{inl}\ \mathtt{w}, T', c' \\ \langle T_r, \mathtt{ee}_1[\mathtt{w}/x]\rangle \curvearrowright \mathtt{w}'_r, T'_r, c'_r \qquad v'_r = \mathtt{V}(T'_r)\end{array}}{\begin{array}{c}\langle\langle \_, \mathtt{case}_{\mathtt{inl}}(T, T_r)\rangle, \mathtt{case}(\mathtt{ee}, x.\mathtt{ee}_1, y.\mathtt{ee}_2)\rangle \curvearrowright \\ \mathtt{w}'_r, \langle v'_r, \mathtt{case}_{\mathtt{inl}}(T', T'_r)\rangle, c' + c'_r\end{array}}\ \textbf{r-case-inl1}$$

$$\frac{\begin{array}{c}\langle T, \mathtt{ee}\rangle \curvearrowright \mathtt{new}(\_, \mathtt{inr}\ v'), T', c' \\ \mathrm{R}(\mathtt{ee}_2)[v'/y] \Downarrow T'_r, f'_r \qquad v'_r = \mathtt{V}(C'_r)\end{array}}{\begin{array}{c}\langle\langle v_r, \mathtt{case}_{\mathtt{inl}}(T, T_r)\rangle, \mathtt{case}(\mathtt{ee}, x.\mathtt{ee}_1, y.\mathtt{ee}_2)\rangle \curvearrowright \\ \mathtt{new}(v_r, v'_r), \langle v'_r, \mathtt{case}_{\mathtt{inr}}(T', T'_r)\rangle, c' + f'_r + c_{case}(\mathbb{S}, \mathbb{C})\end{array}}\ \textbf{r-case-inl2}$$

$$\frac{\begin{array}{c}\langle T_1, \mathtt{ee}_1\rangle \curvearrowright \mathtt{fix}\ f(x).\mathtt{ee}, T'_1, c'_1 \qquad \langle T_2, \mathtt{ee}_2\rangle \curvearrowright \mathtt{w}'_2, T'_2, c'_2 \\ \langle T_r, \mathtt{ee}[\mathtt{w}'_2/x, (\mathtt{fix}\ f(x).\mathtt{ee})/f]\rangle \curvearrowright \mathtt{w}'_r, T'_r, c'_r \\ v'_r = \mathtt{V}(T'_r) \qquad c' = c'_1 + c'_2 + c'_r\end{array}}{\langle\langle \_, \mathtt{app}(T_1, T_2, T_r)\rangle, \mathtt{ee}_1\ \mathtt{ee}_2\rangle \curvearrowright \mathtt{w}'_r, \langle v'_r, \mathtt{app}(T'_1, T'_2, T'_r)\rangle, c'}\ \textbf{r-app1}$$

$$\frac{\begin{array}{c}\langle T_1, \mathtt{ee}_1\rangle \curvearrowright \mathtt{new}(\_, \mathtt{fix}\ f(x).\ e'), T'_1, c'_1 \\ \langle T_2, \mathtt{ee}_2\rangle \curvearrowright \mathtt{w}'_2, T'_2, c'_2 \\ \mathtt{e}'[\mathrm{R}(\mathtt{w}'_2)/x, (\mathtt{fix}\ f(x).\ \mathtt{e}')/f] \Downarrow T'_r, f'_r \\ v'_r = \mathtt{V}(T'_r) \qquad c' = c'_1 + c'_2 + f'_r + c_{app(\mathbb{S},\mathbb{C})}\end{array}}{\begin{array}{c}\langle\langle v_r, \mathtt{app}(T_1, T_2, T_r)\rangle, \mathtt{ee}_1\ \mathtt{ee}_2\rangle \curvearrowright \\ \mathtt{new}(v_r, v'_r), \langle v'_r, \mathtt{app}(T'_1, T'_2, T'_r)\rangle, c'\end{array}}\ \textbf{r-app2}$$

Figure 7: Selected Replay Rules

runs both took the same branch, e.g. the bivalue resulting from $\mathtt{ee}$ is $\mathtt{inl}\ \mathtt{w}$, we keep change propagating through that branch (rule **r-case-inl1**). However, if $\mathtt{ee}$'s result has changed from $\mathtt{inl}\ \_$ to $\mathtt{inr}\ \_$ (detected by a bivalue of the form $\mathtt{new}(\_, \mathtt{inr}\ v)$), then we execute the right branch from-scratch, as in rule **r-case-inl2**. In addition, we incur an extra cost, $c_{case}(\mathbb{S}, \mathbb{C})$, for switching to the from-scratch mode. This pattern of switching to from-scratch evaluation repeats in all rules that apply closures. To change propagate a function application $\mathtt{ee}_1\ \mathtt{ee}_2$, we first change propagate through the function $\mathtt{ee}_1$. If the resulting function does not differ from the original one structurally, i.e., the resulting bivalue has the form $\mathtt{fix}\ f(x).\mathtt{ee}$, then we keep change propagating through the body (rule **r-app1**). However, if the resulting function is structurally different from the original one (bivalue $\mathtt{new}(\_, \mathtt{fix}\ f(x).\ e')$), then we switch to from-scratch execution and incur an additional cost $c_{app}(\mathbb{S}, \mathbb{C})$ (rule **r-app2**).

Soundness We prove our type system sound with respect to the abstract evaluation and change propagation semantics. First, we show that on well-typed expressions, evaluation and abstract change propagation (formalized by $\Downarrow$ and $\curvearrowright$ respectively) are total and the latter produces correct results. Second, we show that the costs κ estimated by expression typing for $\epsilon = \mathbb{C}$ and $\epsilon = \mathbb{S}$ are upper bounds on the costs of from-scratch evaluation and change propagation, respectively. These three statements are formalized in the following two theorems. For readability, we only state the theorems with a single input x, but the generalized versions with any number of inputs hold as well.

Theorem 1 (Soundness for from-scratch execution)
Suppose that (a) $x : \tau \vdash_{\mathbb{C}} e : \tau' \mid \kappa$; (b) $\vdash_{\mathbb{C}} v : \tau \mid -$. Then the following hold for some v', D and f: (1) $e[v/x] \Downarrow \langle v', D\rangle, f$ and (2) $f \leq \kappa$.

Theorem 2 (Soundness for change propagation)
Suppose that (a) $x : \tau \vdash_{\mathbb{S}} e : \tau' \mid \kappa$; (b) $\vdash_{\epsilon} \mathtt{w} \gg \tau$; and (c) $e[\mathrm{L}(\mathtt{w})/x] \Downarrow T, f$. Then the following hold for some T', $\mathtt{w}'$ and c: (1) $\langle T, \ulcorner e \urcorner[\mathtt{w}/x]\rangle \curvearrowright \mathtt{w}', T', c$; (2) $e[\mathrm{R}(\mathtt{w})/x] \Downarrow T', f'$; (3) $\mathtt{V}(T') = \mathrm{R}(\mathtt{w}')$; and (4) $c \leq \kappa$.

To prove these theorems, we build two cost-annotated models of types: a *relational* (binary) one for change propagation ($\epsilon = \mathbb{S}$) and a *unary* one for from-scratch execution ($\epsilon = \mathbb{C}$). The relational model depends on the unary model. The unary model is a standard logical relation. To handle recursive functions, we step-index the relation [5]. Each type τ has a value and an expression interpretation. The value interpretation, written $(\!|\tau|\!)_v$, contains pairs (m, v) of step indices and values. The expression interpretation, written $(\!|\tau|\!)^{\kappa}_{\varepsilon}$, contains pairs (m, e) of step indices and expressions, with the proviso that if $\kappa < m$, then e evaluates to a value with cost no more than κ. Selected clauses of this relation are shown in Figure 9. The relation is agnostic to almost all "relational" refinements such as the annotations $\mathbb{C}$ and $\mathbb{S}$ and the annotation α on list types. The only exception is that $(\!|\tau_1 \xrightarrow{\mathbb{S}(\kappa)} \tau_2|\!)_v$ contains all functions, since a function of this type cannot be applied during from-scratch evaluation, i.e., when $\epsilon = \mathbb{C}$ (see rule **app** in Figure 3).

The relational model is based on bivalues and biexpressions. The relational value interpretation of a type, written $[\![\tau]\!]_v$, contains pairs $(m, \mathtt{w})$ of a step-index and a bivalue. The relation relates the original value $\mathrm{L}(\mathtt{w})$ to the updated value $\mathrm{R}(\mathtt{w})$. The expression interpretation $[\![\tau]\!]^{\kappa}_{\varepsilon}$ is a set of pairs of the form $(m, \mathtt{ee})$. It forces that change propagating $\mathtt{ee}$ (using the rules of $\curvearrowright$) cost no more than κ. The relation is defined in Figure 8. We note some salient points. First, the expression interpretation is asymmetric in the left and right components of $\mathtt{ee}$. Second, $[\![\square((A)^{\mu})]\!]_v \subseteq [\![(A)^{\mathbb{S}}]\!]_v \subseteq [\![(A)^{\mathbb{C}}]\!]_v$. $\square((A)^{\mu})$ contains only those bivalues whose two projections are *identical*, $(A)^{\mathbb{S}}$ contains bivalues whose projections are related (in the logical sense), whereas $(A)^{\mathbb{C}}$ contains all bivalues, even those of the form $\mathtt{new}(v, v')$ that contain completely unrelated values. Third, $(m, \mathtt{new}(v, v'))$ is in $[\![(A)^{\mathbb{C}}]\!]_v$ only if (k, v) and (k, v') are in the unary relation $(\!|A|\!)_v$ *for any step index* k. When reasoning with the relational step-index m, we can call out to any unary step-index k. This shows up in our proofs and works because the unary relation does not depend on the binary relation. Fourth, $[\![\tau_1 \xrightarrow{\mathbb{C}(\kappa)} \tau_2]\!]_v \subseteq [\![\tau_1 \xrightarrow{\mathbb{S}(\kappa)} \tau_2]\!]_v$. This is needed because we may change propagate through the body of a function even if that body was typed in $\mathbb{C}$-mode. It also allows us to show that the judgment $\Delta; \Phi; \Gamma \vdash_{\mathbb{C}} e : \tau \mid \kappa$ entails the judgment $\Delta; \Phi; \Gamma \vdash_{\mathbb{S}} e : \tau \mid \kappa$ semantically. Finally, on list types, the relational interpretation uses both the length and the number of allowed changes meaningfully.

We prove the fundamental theorem for our typing judgments, which roughly says that an expression typed with $\epsilon = \mathbb{S}$ ($\epsilon = \mathbb{C}$) lies in the binary (unary) relation for any bivalue (value) substitution that respects the binary (unary) relation. Technically, the theorem consists of six mutually inductive statements, one for each of the three syntactic categories expressions, bivalues and biexpressions, in each of the two modes change propagation and from-scratch evaluation. Here, we show only the statements for expressions. Theorems 1 and 2 are immediate corollaries of this theorem.

Theorem 3 (Fundamental Theorem)

1. If $\Delta; \Phi; \Gamma \vdash_{\mathbb{S}} e : \tau \mid \kappa$ and $\sigma \in \mathcal{D}[\![\Delta]\!]$ and $(m, \theta) \in \mathcal{G}[\![\sigma\Gamma]\!]$ and $\models \sigma\Phi$, then $(m, \theta\ulcorner e \urcorner) \in [\![\sigma\tau]\!]^{\sigma\kappa}_{\varepsilon}$.
2. If $\Delta; \Phi; \Gamma \vdash_{\mathbb{C}} e : \tau \mid \kappa$ and $\sigma \in \mathcal{D}[\![\Delta]\!]$ and $(m, \mathcal{U}) \in \mathcal{G}(\!|\sigma\Gamma|\!)$ and $\models \sigma\Phi$, then $(m, \mathcal{U}e) \in (\!|\sigma\tau|\!)^{\sigma\kappa}_{\varepsilon}$.

5. Concrete Semantics and Soundness

In order to show the realizability of the from-scratch and change propagation costs estimated by our type system, we present a translation from our source language to an ML-like language with cost

$\llbracket\tau\rrbracket_v \subseteq \text{Step index} \times \text{Bivalue}$ and $\llbracket\tau\rrbracket^{\kappa}_{\varepsilon} \subseteq \text{Step index} \times \text{Biexpression}$

$$
\begin{array}{lcl}
\llbracket (A)^{\mathbb{S}} \rrbracket_v & = & \llbracket A \rrbracket_v \\
\llbracket (A)^{\mathbb{C}} \rrbracket_v & = & \llbracket A \rrbracket_v \cup \{(m, \mathtt{new}(v, v')) \mid \forall k.\ (k, \mathtt{v}) \in \llparenthesis A \rrparenthesis_v \wedge (k, \mathtt{v}') \in \llparenthesis A \rrparenthesis_v\} \\
\llbracket \square(\tau) \rrbracket_v & = & \{(m, \mathtt{w}) \mid \mathtt{stable}(\mathtt{w}) \wedge (m, \mathtt{w}) \in \llbracket \tau \rrbracket_v\} \\
\llbracket \mathtt{real} \rrbracket_v & = & \{(m, \mathtt{keep}(\mathtt{r})) \mid \top\} \\
\llbracket \tau_1 + \tau_2 \rrbracket_v & = & \{(m, \mathtt{inl}\ \mathtt{w}) \mid (m, \mathtt{w}) \in \llbracket \tau_1 \rrbracket_v\} \cup \{(m, \mathtt{inr}\ \mathtt{w}) \mid (m, \mathtt{w}) \in \llbracket \tau_2 \rrbracket_v\} \\
\llbracket \mathtt{list}\, [0]^0\ \tau \rrbracket_v & = & \{(m, \mathtt{nil}) \mid \top\} \\
\llbracket \mathtt{list}\, [n+1]^\alpha\ \tau \rrbracket_v & = & \{(m, \mathtt{cons}(\mathtt{w}_1, \mathtt{w}_2)) \mid ((m, \mathtt{w}_1) \in \llbracket \tau \rrbracket_v \wedge (m, \mathtt{w}_2) \in \llbracket \mathtt{list}\, [n]^{\alpha-1}\ \tau \rrbracket_v \wedge \alpha > 0) \vee \\
& & \qquad ((m, \mathtt{w}_1) \in \llbracket \square(\tau) \rrbracket_v \wedge (m, \mathtt{w}_2) \in \llbracket \mathtt{list}\, [n]^{\alpha}\ \tau \rrbracket_v)\} \\
\llbracket \tau_1 \xrightarrow{\mathbb{S}(\kappa)} \tau_2 \rrbracket_v & = & \{(m, \mathtt{fix}\, f(x).\mathtt{ee}) \mid \forall j < m.\ \forall \mathtt{w}.\ (j, \mathtt{w}) \in \llbracket \tau_1 \rrbracket_v \Rightarrow (j, \mathtt{ee}[\mathtt{fix}\, f(x).\mathtt{ee}/f][\mathtt{w}/x]) \in \llbracket \tau_2 \rrbracket^{\kappa}_{\varepsilon}\} \\
\llbracket \tau_1 \xrightarrow{\mathbb{C}(\eta)} \tau_2 \rrbracket_v & = & \{(m, \mathtt{fix}\, f(x).\mathtt{ee}) \mid (\forall k.\ (k, \mathtt{fix}\, f(x).\, \mathrm{L}(\mathtt{ee})) \in \llparenthesis \tau_1 \xrightarrow{\mathbb{C}(\eta)} \tau_2 \rrparenthesis_v \ \wedge\ (k, \mathtt{fix}\, f(x).\, \mathrm{R}(\mathtt{ee})) \in \llparenthesis \tau_1 \xrightarrow{\mathbb{C}(\eta)} \tau_2 \rrparenthesis_v) \wedge \\
& & \qquad (\forall j < m.\ \forall \mathtt{w}.\ (j, \mathtt{w}) \in \llbracket \tau_1 \rrbracket_v \Rightarrow (j, \mathtt{ee}[\mathtt{fix}\, f(x).\mathtt{ee}/f][\mathtt{w}/x]) \in \llbracket \tau_2 \rrbracket^{\kappa}_{\varepsilon})\} \\
\llbracket \forall t \overset{\mathbb{S}(\kappa)}{::} S.\ \tau \rrbracket_v & = & \{(m, \Lambda.\mathtt{ee}) \mid \forall I.\ \vdash I :: S \Rightarrow (m, \mathtt{ee}) \in \llbracket \tau[I/t] \rrbracket^{\kappa[I/t]}_{\varepsilon}\} \\
\llbracket \forall t \overset{\mathbb{C}(\kappa)}{::} S.\ \tau \rrbracket_v & = & \{(m, \Lambda.\mathtt{ee}) \mid (\forall k.\ (k, \mathrm{L}(\Lambda.\mathtt{ee})) \in \llparenthesis \forall t \overset{\mathbb{C}(\kappa)}{::} S.\ \tau \rrparenthesis_v \wedge (k, \mathrm{R}(\Lambda.\mathtt{ee})) \in \llparenthesis \forall t \overset{\mathbb{C}(\kappa)}{::} S.\ \tau \rrparenthesis_v) \wedge \\
& & \qquad (\forall I.\ \vdash I :: S \Rightarrow (m, \mathtt{ee}) \in \llbracket \tau[I/t] \rrbracket^{\kappa[I/t]}_{\varepsilon})\} \\
\llbracket \exists t{::}S.\ \tau \rrbracket_v & = & \{(m, \mathtt{pack}\ \mathtt{w}) \mid \exists I. \vdash I :: S \ \wedge\ (m, \mathtt{w}) \in \llbracket \tau[I/t] \rrbracket_v\} \\
\\
\llbracket \tau \rrbracket^{\kappa}_{\varepsilon} & = & \{(m, \mathtt{ee}) \mid \forall f, D, v.\ \mathrm{L}(\mathtt{ee}) \Downarrow \langle v, D \rangle, f \ \wedge\ f < m \ \Rightarrow \exists\, v', D', \mathtt{w}', c', f' \text{ such that} \\
& & \quad 1.\ \langle \langle v, D \rangle, \mathtt{ee} \rangle \curvearrowright \mathtt{w}', \langle v', D' \rangle, c' \\
& & \quad 2.\ \mathrm{R}(\mathtt{ee}) \Downarrow \langle v', D' \rangle, f' \\
& & \quad 3.\ v' = \mathrm{R}(\mathtt{w}') \ \wedge\ v = \mathrm{L}(\mathtt{w}') \\
& & \quad 4.\ c' \leq \kappa \\
& & \quad 5.\ (m - f, \mathtt{w}') \in \llbracket \tau \rrbracket_v\} \\
\\
\mathcal{G}\llbracket \cdot \rrbracket & = & \{(k, \emptyset)\} \\
\mathcal{G}\llbracket \Gamma, x : \tau \rrbracket & = & \{(m, \theta[x \mapsto \mathtt{w}]) \mid (m, \theta) \in \mathcal{G}\llbracket \Gamma \rrbracket \ \wedge (m, \mathtt{w}) \in \llbracket \tau \rrbracket_v\}
\end{array}
$$

Figure 8: Step-indexed binary interpretation of selected types

$\llparenthesis\tau\rrparenthesis_v \subseteq \text{Step index} \times \text{Value}$ and $\llparenthesis\tau\rrparenthesis^{\kappa}_{\varepsilon} \subseteq \text{Step index} \times \text{Expression}$

$$
\begin{array}{lcl}
\llparenthesis (A)^{\mu} \rrparenthesis_v & = & \llparenthesis A \rrparenthesis_v \\
\llparenthesis \square(\tau) \rrparenthesis_v & = & \llparenthesis \tau \rrparenthesis_v \\
\llparenthesis \mathtt{list}\, [0]^0\ \tau \rrparenthesis_v & = & \{(m, \mathtt{nil}) \mid \top\} \\
\llparenthesis \mathtt{list}\, [n+1]^\alpha\ \tau \rrparenthesis_v & = & \{(m, \mathtt{cons}(v_1,\ v_2)) \mid (m, v_1) \in \llparenthesis \tau \rrparenthesis_v \wedge \\
& & ((m, v_2) \in \llparenthesis \mathtt{list}\, [n]^{\alpha}\ \tau \rrparenthesis_v \vee \\
& & \ (m, v_2) \in \llparenthesis \mathtt{list}\, [n]^{\alpha-1}\ \tau \rrparenthesis_v)\} \\
\llparenthesis \tau_1 \xrightarrow{\mathbb{S}(\kappa)} \tau_2 \rrparenthesis_v & = & \{(m, \mathtt{fix}\, f(x).\, e) \mid \top\} \\
\llparenthesis \tau_1 \xrightarrow{\mathbb{C}(\kappa)} \tau_2 \rrparenthesis_v & = & \{(m, \mathtt{fix}\, f(x).\, e) \mid \forall j < m.\ \forall v.\ (j, v) \in \llparenthesis \tau_1 \rrparenthesis_v \Rightarrow (j, e[\mathtt{fix}\, f(x).\, e/f][v/x]) \in \llparenthesis \tau_2 \rrparenthesis^{\kappa}_{\varepsilon}\} \\
\llparenthesis \forall t \overset{\mathbb{S}(\kappa)}{::} S.\ \tau \rrparenthesis_v & = & \{(m, \Lambda.\, e) \mid \top\} \\
\llparenthesis \forall t \overset{\mathbb{C}(\kappa)}{::} S.\ \tau \rrparenthesis_v & = & \{(m, \Lambda.\, e) \mid \forall I.\ \vdash I :: S \Rightarrow (m, e) \in \llparenthesis \tau[I/t] \rrparenthesis^{\kappa[I/t]}_{\varepsilon}\} \\
\llparenthesis \exists t{::}S.\ \tau \rrparenthesis_v & = & \{(m, \mathtt{pack}\ v) \mid \exists I. \vdash I :: S \ \wedge (m, v) \in \llparenthesis \tau[I/t] \rrparenthesis_v\} \\
\\
\llparenthesis \tau \rrparenthesis^{\kappa}_{\varepsilon} & = & \{(m, e) \mid \kappa < m \Rightarrow \exists\, v, D, f \text{ such that} \\
& & \ 1.\ e \Downarrow \langle v, D \rangle, f \\
& & \ 2.\ f \leq \kappa \\
& & \ 3.\ (m - f, v) \in \llparenthesis \tau \rrparenthesis_v\} \\
\\
\mathcal{G}\llparenthesis \cdot \rrparenthesis & = & \{(m, \emptyset)\} \\
\mathcal{G}\llparenthesis \Gamma, x : \tau \rrparenthesis & = & \{(m, \mathcal{U}[x \mapsto v]) \mid (m, \mathcal{U}) \in \mathcal{G}\llparenthesis \Gamma \rrparenthesis \ \wedge (m, v) \in \llparenthesis \tau \rrparenthesis_v\}
\end{array}
$$

Figure 9: Step-indexed unary interpretation of selected types

semantics and we prove that the translations of typed programs respect statically established costs. To support change propagation, the translation stores all values in mutable references in the first run to allow in-place update during the incremental run. Additionally, during the first run the translation constructs a dependency graph that indicates which locations depend on any given location. Change propagation then proceeds bottom up: It starts from input locations that have changed, then updates their dependents and so on.

The target language, ML^C, is inspired by AFL [3]. It extends (simply typed) ML with a single primitive, $\mathtt{read}(e_1,\ x.\ e_2)$, that evaluates e_1 to a reference l, binds the value in that reference to x and then evaluates e_2. On the side, the construct automatically records a *dependency edge* from l to the location l' that holds the result of evaluating e_2. The edge is labeled $\lambda x.\, e_2$. If l changes, then change propagation will re-execute e_2 to obtain a new updated value for l'. The syntax, and typing rule of $\mathtt{read}(e_1,\ x.\ e_2)$ are shown below.

$$e ::= \ \dots \mid \mathtt{read}(e_1,\ x.\ e_2)$$

$$\frac{\Gamma \vdash e_1 : \mathtt{ref}\ \tau' \qquad \Gamma,\ x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \mathtt{read}(e_1,\ x.\ e_2) : \mathtt{ref}\ \tau}$$

The semantics of ML^C are a slight variant of ML, formalized by two judgments: $e_1,\ \sigma_1,\ t_1 \Downarrow^{\mathbb{S}} l,\ \sigma_2,\ t_2,\ c$ and $e_1,\ \sigma_1,\ t_1 \Downarrow^{\mathbb{C}(l)} v,\ \sigma_2,\ t_2,\ c$. Both judgments say that expression e_1, when executed in store σ_1, results in store σ_2 and the cost of this evaluation is c. The two judgments differ in the final result: $e_1,\ \sigma_1,\ t_1 \Downarrow^{\mathbb{S}} l,\ \sigma_2,\ t_2,\ c$ allocates a fresh location l, stores the final result in it and returns l whereas $e_1,\ \sigma_1,\ t_1 \Downarrow^{\mathbb{C}(l)} v,\ \sigma_2,\ t_2,\ c$ directly returns the computed value v and, additionally, records dependency edges to the output location l, which is passed as an input. In addition, the judgments also say that the execution happens during the *logical* time interval $[t_1, t_2]$. Logical time points t are necessary during change propagation to decide which closures are nested inside others, as in prior work [3]. We increment the time counter by 1 whenever we start or end a new dependency. The last point to note here is that the store maps a location l to not just a value, but a pair of a value and a list of dependents, together written $(v, \vec{e})$. Each dependent in $\vec{e}$ is a tuple $(l', \lambda x.e', t_1', t_2')$ and means that, in the first run, the value in l' was obtained as a result of the execution of $\mathtt{read}(l,\ x.\ e')$ and that $e'[v/x]$ evaluated during the interval $[t_1', t_2']$. It is these dependencies that change propagation traverses to update results. As examples, the evaluation rules for the construct $\mathtt{read}(e_1,\ x.\ e_2)$ are shown in Figure 10. We note that the time stamps and dependencies that are baked into the semantics here could readily be implemented in an actual language or via a library.

Type translation Our translation is type-directed and preserves typing. At the type level, the translation is essentially an erasure of refinements, that additionally puts every value (and sub-value) in a reference. The type translation is shown below. Importantly, note that both $(A)^{\mathbb{S}}$ and $(A)^{\mathbb{C}}$ map to $\mathtt{ref}\ \|A\|_A$, which allows us to subtype from $(A)^{\mathbb{S}}$ to $(A)^{\mathbb{C}}$ without paying any coercion cost. A list of type $\mathtt{list}\,[n]^{\alpha}\ \tau$ translates to $\mathtt{reflist}\ (\|\tau\| + \|\tau\|)$. The sum type is used to distinguish $\square(\tau)$-typed elements from τ-typed elements, corresponding to the two typing rules **cons1** and **cons2**: Any element tagged $\mathtt{inl}$ corresponds to a source element of type $\square(\tau)$ and any element tagged $\mathtt{inr}$ corresponds to a source element of type τ.

$$\begin{array}{lcl}
\|(A)^{\mu}\| & = & \mathtt{ref}\ \|A\|_A \\
\|\square(\tau)\| & = & \|\tau\| \\
\|\tau_1 + \tau_2\|_A & = & \|\tau_1\| + \|\tau_2\| \\
\|\mathtt{list}\,[n]^{\alpha}\ \tau\|_A & = & \mathtt{reflist}\ (\|\tau\| + \|\tau\|) \\
\|\tau_1 \xrightarrow{\kappa(\mu)} \tau_2\|_A & = & \|\tau_1\| \to \|\tau_2\| \\
\|\forall i \overset{\mu(\kappa)}{::} S.\ \tau\|_A & = & \mathtt{unit} \to \|\tau\| \\
\|\exists i{::}S.\ \tau\| & = & \|\tau\|
\end{array}$$

where $\mathtt{reflist}\ \tau = \mathtt{nil} \mid \mathtt{cons}\ \tau \times \mathtt{ref}\ (\mathtt{reflist}\ \tau)$

Expression translation Interesting cases of the translation are shown in Figure 11. The translation is type-directed, but is independent of costs, so we omit several constraints related to costs from the rules. The main idea is twofold. First, every introduction form puts the result in a reference (e.g., rules **real** and **fix1**). Second, to translate the elimination of an expression with a type labeled $\mathbb{C}$, we introduce a $\mathtt{read}$ on the expression's translation to force the addition of a dependence edge, as in rule $\mathbf{app}_{\mathbb{C}}$. This ensures that if the expression changes, then the elimination form is re-executed during change propagation. In contrast, in eliminating an expression with a type labeled $\mathbb{S}$, we do not add a $\mathtt{read}$, as in rule $\mathbf{app}_{\mathbb{S}}$. The rule $\mathbf{caseL}_{\mathbb{S}}$ for list case is quite interesting. Recall that the typing rule **caseL** has two premises for the cons case: One where the head may not change and one where it may. In the translation, these cases are distinguished by the tags $\mathtt{inl}$ and $\mathtt{inr}$ on the head. Consequently, the translation of list case also immediately case analyzes the head to decide which premise to use. Our translation is total on typed expressions and it generates well-typed target expressions.

Theorem 4 (Totality of the translation and type soundness) If $\Delta; \Phi; \Gamma \vdash_{\epsilon} e : \tau \mid \kappa$, then $\Delta; \Phi; \Gamma \vdash_{\epsilon} e : \tau \mid \kappa \hookrightarrow \ulcorner e \urcorner$ and $\|\Gamma\| \vdash \ulcorner e \urcorner : \|\tau\|$

Change propagation During the first run of a translated program, dependencies generated by $\mathtt{read}$ are recorded in the store. These dependencies constitute an acyclic graph on references, whose edges are labeled by closures and pairs of starting and ending timestamps. An input change is manifest by (externally) updating some of the initial references in this graph. To change propagate, we need to re-run all closures that are reachable from these changes in a topologically sorted order (else, we run the risk of evaluating a closure before its dependencies have been updated). To do this, we first do a bit of *one time* pre-processing on the dependency graph of the first run. We restrict the graph to references and edges reachable from inputs that *may* change (these are all clearly marked using types annotated $\mathbb{C}$ and, in the case of list elements, using the tag $\mathtt{inr}$). Then, we sort all edges in this restricted graph in order of their starting timestamps. We then delete any edge whose two timestamps are contained in another selected edge's two timestamps—in this case, the first edge represents a subcomputation of the second edge and since the second edge's closure will be re-executed from scratch, there is no need to evaluate the first edge's closure separately. We then throw away the timestamps. This yields a topologically sorted list D of tuples (edges) of the form $(l_s, l_d, \lambda x.\, e)$. Such a tuple says that the updated value of l_d should be obtained by executing $e[v/x]$, where v is the updated value in l_s.

Change propagation is then an extremely simple algorithm, that just evaluates $e[v/x]$ in sequence for all tuples in the list D in order *after* inputs have been updated externally. We formalize change propagation using the judgment $D,\ \sigma \rightsquigarrow \sigma',\ c$, which means that list D change propagates store σ (which contains updated inputs) to store σ' (which contains the entire updated computation) with cost c. The judgment has only two rules, which are shown below. Saliently, the second rule adds the cost of evaluating closures from-scratch (denoted c) to the cost of change propagation.

$$\frac{}{[],\ \sigma \rightsquigarrow \sigma,\ 0}\ \textbf{stop}$$

$$\frac{\sigma(l_s) = (v,\ \_) \qquad e[v/x],\ \sigma \Downarrow^{\mathbb{C}(l_d)} v,\ \sigma',\ c \qquad D,\ \sigma'[l_d \mapsto (v,\ [])] \rightsquigarrow \sigma'',\ c'}{(l_s,\ l_d,\ \lambda x.\, e) :: D,\ \sigma \rightsquigarrow \sigma'',\ c + c' + 1}\ \textbf{eval}$$

We note that this algorithm is simpler than prior work on adaptive change propagation as in AFL [3], where the goal is to change propagate only from the inputs that have *actually* changed. AFL's algorithm uses a priority queue, whose overhead is difficult to estimate statically. To avoid this overhead, our algorithm updates all loca-

$$\frac{e_1,\ \sigma,\ t_1 \Downarrow^{\mathbb{S}} l,\ \sigma',\ t_2,\ c_1 \qquad l_n \notin \mathtt{dom}(\sigma') \qquad \sigma'(l) = (v',\ \vec{e}) \qquad e_2[v'/x],\ \sigma'[l_n \mapsto \Box],\ t_2+1 \Downarrow^{\mathbb{C}(l_n)} v,\ \sigma'',\ t_3,\ c_2}{\mathtt{read}(e_1,\ x.\, e_2),\ \sigma,\ t_1 \Downarrow^{\mathbb{S}} l_n,\ \sigma''[l \mapsto (v',\ (l_n,\ \lambda x.\, e_2,\ t_2,\ t_3) :: \vec{e}),\ l_n \mapsto (v,\ [])],\ t_3+1,\ c_1+c_2+1}\ \textbf{read}_{\mathbb{S}}$$

$$\frac{e_1,\ \sigma,\ t_1 \Downarrow^{\mathbb{S}} l,\ \sigma',\ t_2,\ c_1 \qquad \sigma'(l) = (v',\ \vec{e}) \qquad e_2[v'/x],\ \sigma',\ t_2+1 \Downarrow^{\mathbb{C}(l_n)} v,\ \sigma'',\ t_3,\ c_2}{\mathtt{read}(e_1,\ x.\, e_2),\ \sigma,\ t_1 \Downarrow^{\mathbb{C}(l_n)} v,\ \sigma''[l \mapsto (v',\ (l_n,\ \lambda x.\, e_2,\ t_2,\ t_3) :: \vec{e})],\ t_3+1,\ c_1+c_2+1}\ \textbf{read}_{\mathbb{C}}$$

Figure 10: Selected rules of the target evaluation semantics

$\boxed{\Delta; \Phi; \Gamma \vdash_\epsilon e : \tau \mid \kappa \hookrightarrow \ulcorner e \urcorner}$

$$\frac{}{\Delta; \Phi; \Gamma, x : \tau \vdash_{\mathbb{S}} x : \tau \mid 0 \hookrightarrow x}\ \textbf{var}_{\mathbb{S}} \qquad \frac{}{\Delta; \Phi; \Gamma, x : \tau \vdash_{\mathbb{C}} x : \tau \mid 1 \hookrightarrow \mathtt{read}(x,\ x.\, x)}\ \textbf{var}_{\mathbb{C}} \qquad \frac{}{\Delta; \Phi; \Gamma \vdash_\epsilon r : (\mathtt{real})^{\mathbb{S}} \mid \kappa \hookrightarrow \mathtt{ref}\ r}\ \textbf{real}$$

$$\frac{\Delta; \Phi; \Gamma, f : (\tau_1 \xrightarrow{\delta(\kappa')} \tau_2)^{\mathbb{S}}, x : \tau_1 \vdash_\delta e : \tau_2 \mid \kappa' \hookrightarrow \ulcorner e \urcorner}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{fix}\ f(x).\, e : (\tau_1 \xrightarrow{\delta(\kappa')} \tau_2)^{\mathbb{S}} \mid \kappa \hookrightarrow \mathtt{ref}\ (\mathtt{fix}\ f(x).\ulcorner e \urcorner)}\ \textbf{fix1}$$

$$\frac{\Delta; \Phi; \Gamma \vdash_\epsilon e_1 : (\tau_1 \xrightarrow{\delta(\kappa')} \tau_2)^{\mathbb{S}} \mid \kappa_1 \hookrightarrow \ulcorner e_1 \urcorner \qquad \Delta; \Phi; \Gamma \vdash_\epsilon e_2 : \tau_1 \mid \kappa_2 \hookrightarrow \ulcorner e_2 \urcorner}{\Delta; \Phi; \Gamma \vdash_\epsilon e_1\ e_2 : \tau_2 \mid \kappa \hookrightarrow\ !\ulcorner e_1 \urcorner \ulcorner e_2 \urcorner}\ \textbf{app}_{\mathbb{S}}$$

$$\frac{\Delta; \Phi; \Gamma \vdash_\epsilon e_1 : (\tau_1 \xrightarrow{\mathbb{C}(\kappa')} \tau_2)^{\mathbb{C}} \mid \kappa_1 \hookrightarrow \ulcorner e_1 \urcorner \qquad \Delta; \Phi; \Gamma \vdash_\epsilon e_2 : \tau_1 \mid \kappa_2 \hookrightarrow \ulcorner e_2 \urcorner}{\Delta; \Phi; \Gamma \vdash_\epsilon e_1\ e_2 : \tau_2 \mid \kappa \hookrightarrow \mathtt{let}\ f\ =\ \ulcorner e_1 \urcorner \mathtt{in}\ \mathtt{let}\ x\ =\ \ulcorner e_2 \urcorner \mathtt{in}\ \mathtt{read}(f,\ f.\, f\ x)}\ \textbf{app}_{\mathbb{C}} \qquad \frac{}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{nil} : (\mathtt{list}\ [0]^0\ \tau)^{\mathbb{S}} \mid \kappa \hookrightarrow \mathtt{ref}\ \mathtt{nil}}\ \textbf{nil}$$

$$\frac{\Delta; \Phi; \Gamma \vdash_\epsilon e_1 : \Box(\tau) \mid \kappa_1 \hookrightarrow \ulcorner e_1 \urcorner \qquad \Delta; \Phi; \Gamma \vdash_\epsilon e_2 : (\mathtt{list}\ [n]^\alpha\ \tau)^\mu \mid \kappa_2 \hookrightarrow \ulcorner e_2 \urcorner}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{cons}(e_1,\ e_2) : (\mathtt{list}\ [n{+}1]^\alpha\ \tau)^\mu \mid \kappa \hookrightarrow \mathtt{ref}\ (\mathtt{cons}(\mathtt{inl}\ \ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner))}\ \textbf{cons1}$$

$$\frac{\Delta; \Phi; \Gamma \vdash_\epsilon e_1 : \tau \mid \kappa_1 \hookrightarrow \ulcorner e_1 \urcorner \qquad \Delta; \Phi; \Gamma \vdash_\epsilon e_2 : (\mathtt{list}\ [n]^{\alpha-1}\ \tau)^\mu \mid \kappa_2 \hookrightarrow \ulcorner e_2 \urcorner}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{cons}(e_1,\ e_2) : (\mathtt{list}\ [n{+}1]^\alpha\ \tau)^\mu \mid \kappa \hookrightarrow \mathtt{ref}\ (\mathtt{cons}(\mathtt{inr}\ \ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner))}\ \textbf{cons2}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash_\epsilon e : (\mathtt{list}\ [n]^\alpha\ \tau)^{\mathbb{S}} \mid \kappa_e \hookrightarrow \ulcorner e \urcorner \\ \Delta; \Phi \wedge n \doteq 0; \Gamma \vdash_\epsilon e_1 : \tau' \mid \kappa' \hookrightarrow \ulcorner e_1 \urcorner \qquad \Delta, i :: \iota; \Phi \wedge n \doteq i+1; \Gamma, h : \Box(\tau), tl : (\mathtt{list}\ [i]^\alpha\ \tau)^{\mathbb{S}} \vdash_\epsilon e_2 : \tau' \mid \kappa' \hookrightarrow \ulcorner e_{2_l} \urcorner \\ \Delta, i :: \iota, \beta :: \iota; \Phi \wedge n \doteq i+1 \wedge \alpha \doteq \beta+1; \Gamma, h : \tau, tl : (\mathtt{list}\ [i]^\beta\ \tau)^{\mathbb{S}} \vdash_\epsilon e_2 : \tau' \mid \kappa' \hookrightarrow \ulcorner e_{2_r} \urcorner\end{array}}{\Delta; \Phi; \Gamma \vdash_\epsilon \mathtt{case}_{\mathtt{L}}\ e\ \mathtt{of}\ \mathtt{nil}\ \to\ e_1 \mid \mathtt{cons}(h, tl)\ \to\ e_2 : \tau' \mid \kappa \hookrightarrow \mathtt{case}_{\mathtt{L}}\ !\ulcorner e \urcorner\ \mathtt{of}\ \mathtt{nil}\ \to\ \ulcorner e_1 \urcorner \mid \mathtt{cons}(h, tl)\ \to\ \mathtt{case}(h, h.\ulcorner e_{2_l} \urcorner, h.\ulcorner e_{2_r} \urcorner)}\ \textbf{caseL}_{\mathbb{S}}$$

Figure 11: Selected rules of the translation

tions that might *possibly* change by starting from all $\mathbb{C}$-annotated inputs. Another difference from AFL is that our algorithm does not update the dependency graph during the incremental run. So every incremental run must use the dependency graph of the first run, as opposed to AFL, where every incremental run uses the dependency graph of the previous run.

Soundness We show that our translation is sound, both functionally and with respect to costs established by the type system, for both from-scratch evaluation and change propagation. For the soundness proof, we design two new step-indexed logical relations—one unary and one binary. The unary relation, written $\mathcal{V}(\!|\tau|\!)$, relates one source value to one target value (the source value's translation) and a target store. The binary relation, written $\mathcal{V}[\![\tau]\!]$, relates two source values (obtained by applying a relational substitution to the same value) to a target value with two related stores (corresponding to the relational substitution). As expected, the corresponding expression relations capture costs. Due to lack of space, we defer details of the relations to our appendix, but the relations allow us to prove the following soundness theorems. In the second theorem, γ represents the subpart of σ that is updated due to input changes, $\gamma\sigma$ denotes the update of σ with γ wherever γ is defined, and $\mathcal{D}(\sigma_f,\ \mathtt{dom}(\gamma))$ denotes the result of pre-processing the dependency graph in store σ_f, starting from the locations in γ. The important clauses in these two theorems are (3) and (4), respectively, which say that the from-scratch and change propagation costs established in the type system are upper bounds on the corresponding costs of the result of the translation.

Theorem 5 (Soundness of from-scratch execution)
If $x : \tau' \vdash_{\mathbb{C}} e : \tau \mid \kappa \hookrightarrow \ulcorner e \urcorner$ and $(v_s',\ v_t',\ \sigma) \in \mathcal{V}(\!|\tau'|\!)$, then

1. $e[v_s'/x] \Downarrow v_s, j$
2. $\ulcorner e \urcorner[v_t'/x],\ \sigma_i \Downarrow^{\mathbb{S}} v_t,\ \sigma',\ cl$
3. $c \leq \kappa$
4. $(v_s,\ v_t, \sigma') \in \mathcal{V}(\!|\tau|\!)$

Theorem 6 (Soundness of change propagation)
If $\cdot; \cdot; x : \tau' \vdash_{\mathbb{S}} e : \tau \mid \kappa \hookrightarrow \ulcorner e \urcorner$ and $(v_i',\ v_c',\ v_t',\ \sigma,\ \gamma\sigma \cup \sigma') \in \mathcal{V}[\![\tau']\!]$ and $e[v_i'/x] \Downarrow v_i, \_$, then

1. $e[v_c'/x] \Downarrow v_c, \_$
2. $\ulcorner e \urcorner[v_t'/x],\ \sigma \Downarrow^{\mathbb{S}} v_t,\ \sigma_f,\ \_$
3. $\mathcal{D}(\sigma_f,\ \mathtt{dom}(\gamma)),\ \gamma\sigma_f \cup \sigma' \rightsquigarrow \sigma_f',\ c$
4. $c \leq \kappa$
5. $(v_i,\ v_s,\ v_t, \sigma_f, \sigma_f') \in \mathcal{V}[\![\tau]\!]$

6. Related Work

Incremental computation There is a vast amount of literature on incremental computation, ranging from algorithmic techniques like memoization [19, 24], to language-based approaches using dynamic dependence graphs [3, 7, 8] and static techniques like finite differencing [6, 22, 25]. To speed up incremental runs, approaches based on dynamic dependency graphs store intermediate results from the initial run. A prominent language-based technique that uses this approach is self-adjusting computations (AFL) [3], which has been subsequently expanded to Standard ML [4] and a dialect of C [17]. Our change propagation algorithm is inspired by AFL, but is different. AFL uses a priority queue ordered by timestamps to decide which closures to execute; we rely on not only timestamps but also static annotations to pre-process the dependency graph to determine which closures to execute. AFL's approach is more flexible but incurs higher bookkeeping cost for priority queue operations when the same inputs change in subsequent incremental runs.

Previous work by Chen *et al.* [8, 9] automatically translates purely functional programs to their incremental counterparts. Our translation (Section 5) is loosely inspired by this work, but the translation itself and the theorems differ significantly. In particular, we translate both $\mathbb{S}$- and $\mathbb{C}$-labeled expressions to reference types, while Chen *et al.* translate only $\mathbb{C}$-labeled types to reference types. Our approach allows cost-free coercion from $(A)^{\mathbb{S}}$ to $(A)^{\mathbb{C}}$, and also supports the **nochange** rule, which is essential to typing recursive functions with precise costs. A second significant difference is that Chen *et al.* only show that the initial run of the translated program is no slower (asymptotically) than the source program. They do not analyze costs for incremental runs. In contrast, we show that both incremental and from-scratch costs of translated programs are bounded by those estimated by our type system. (Chen *et al.*'s type system does not provide cost bounds.)

Approaches based on static transformations extract program derivatives, which can be executed in place of the original programs with only the updated inputs to produce updated results [6, 25]. Such techniques make use of the algebraic properties of a set of primitives and restrict the programmer to only those primitives. In contrast to these approaches, our work is based on dynamic dependence graphs and our static analysis only establishes the cost of incremental runs.

In general, in all prior work on incremental computation the efficiency of incremental updates is established either by empirical analysis of benchmark programs, algorithmic analysis or direct analysis of cost semantics [23]. CostIt [11] was the first proposal for statically analyzing dynamic stability. Our work directly builds on CostIt, but our type system is richer: CostIt cannot type programs where fresh closures may execute in the incremental runs. We do away with this restriction by introducing a second typing mode that analyzes from-scratch execution costs. This requires a re-design of the type system and substantially complicates the metatheory (we use both a binary and a unary logical relation, while CostIt needs only the former) but allows the analysis of many programs that CostIt cannot handle. Additionally, we prove soundness of the type system relative to a concrete change propagation algorithm, which CostIt does not.

Refinement types and information flow control Like CostIt, we rely on index refinements in the style of DML [28]. Index refinements are usually data structure-specific. Allowing programmer-defined size metrics and extending our analysis with algebraic datatypes is nontrivial. We believe that recent work by Danner *et al.* is a good starting point [14].

In addition, our type system can be considered as a cost-effect refinement of the pure fragment of Pottier and Simonet's information flow type system for ML [26]. The security levels L ("low") and H ("high") in information flow analysis correspond to $\mathbb{S}$ ("stable") and $\mathbb{C}$ ("changeable") respectively in DuCostIt. Our ϵ corresponds to what is often called the program counter or *pc* in information flow analysis. The *pc* tracks implicit influences due to control flow.

Type systems on resource bounds/complexity analysis Static analysis of resource bounds of one run of a program is well-studied. Specifically, many different techniques can be used to prove worse-case execution time (WCET) complexity. These include linear dependent types [12, 13], abstract interpretation [16], amortized resource analysis [20] and sized types [10, 21, 27]. In particular, amortized resource analysis [20] is an automatic technique that can *infer* WCET, but only for polynomial-time programs. However, all these techniques differ from our work (and CostIt) in a fundamental way: They all reason about a single execution of a program, whereas dynamic stability is a property of two executions of a program. The from-scratch cost analysis that we add in DuCostIt is closer to the aforementioned prior work. We could have used an existing technique for from-scratch cost analysis, but it was unclear how existing techniques could be extended to relational analysis, so we found it easier to build our own analysis.

7. Conclusion and Future Work

This paper introduces DuCostIt, a type system that combines unary- and relational-cost analysis using type refinements. Although the focus here is on a specific application—analysis of dynamic stability—the overall design of type system is quite general. It can be adapted to other kinds of relational cost analysis, e.g., to show that of two similar programs, one consumes less resources than the other (as may be relevant for compiler optimizations), or that a program's execution time is independent of certain input values (as may be relevant for showing the absence of timing-based leaks in security settings). In our ongoing work, we are examining such generalizations.

References

[1] A type theory for incremental computational complexity with control flow changes (technical appendix). Online at `http://www.mpi-sws.org/~ecicek/ducostit_appendix.pdf`.

[2] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 2005.

[3] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6), 2006.

[4] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Program. Lang. Syst.*, 32(1):3:1–3:53, 2009.

[5] 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.

[6] Y. Cai, P. G. Giarrusso, T. Rendel, and K. Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 145–155, 2014.

[7] M. Carlsson. Monads for incremental computing. In *Proceedings of the 7th International Conference on Functional Programming*, ICFP '02, pages 26–35, 2002.

[8] Y. Chen, J. Dunfield, and U. A. Acar. Type-directed automatic incrementalization. In *Proceedings of the 33rd Conference on Programming Language Design and Implementation*, PLDI '12, pages 299–310, 2012.

[9] Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar. Implicit self-adjusting computation for purely functional programs. *J. Functional Programming*, 24(1):56–112, 2014.

[10] W.-N. Chin and S.-C. Khoo. Calculating sized types. In *Proceedings of the Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '00, pages 62–72, 1999.

[11] E. Çiçek, D. Garg, and U. A. Acar. Refinement types for incremental computational complexity. In *Programming Languages and Systems -*

24th European Symposium on Programming Proceedings, pages 406–431, 2015.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] M. A. Hammer, U. A. Acar, and Y. Chen. Ceal: A C-based language for self-adjusting computation. In *Proceedings of the 2009 Conference on Programming Language Design and Implementation*, PLDI '09, pages 25–37, 2009.

[18] M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of the 35th Conference on Programming Language Design and Implementation*, PLDI '14, pages 156–166, 2014.

[19] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '00, pages 311–320, 2000.

[20] 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.

[21] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *Proceedings of the Fourth International Conference on Functional Programming*, ICFP '99, pages 70–81, 1999.

[22] C. Koch. Incremental query evaluation in a ring of databases. In *Proceedings of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, pages 87–98, 2010.

[23] R. Ley-Wild, U. A. Acar, and M. Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 36th Annual Symposium on Principles of Programming Languages*, POPL '09, pages 186–199, 2009.

[24] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, 1995.

[25] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, 1982.

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

[27] P. B. Vasconcelos and K. Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *Implementation of Functional Languages, 15th International Workshop, IFL 2003*, pages 86–101, 2003.

[28] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th Symposium on Principles of Programming Languages*, POPL '99, pages 214–227, 1999.