# Refinement Types for Incremental Computational Complexity

Ezgi Çiçek[1], Deepak Garg[1], and Umut Acar[2]

[1] Max Planck Institute for Software Systems
[2] Carnegie Mellon University

**Abstract.** With recent advances, programs can be compiled to efficiently respond to incremental input changes. However, there is no language-level support for reasoning about the time complexity of incremental updates. Motivated by this gap, we present CostIt, a higher-order functional language with a lightweight refinement type system for proving asymptotic bounds on incremental computation time. Type refinements specify which parts of inputs and outputs may change, as well as dynamic stability, a measure of time required to propagate changes to a program's execution trace, given modified inputs. We prove our type system sound using a new step-indexed cost semantics for change propagation and demonstrate the precision and generality of our technique through examples.

## 1 Introduction

Many applications operate on data that change over time: compilers respond to source code changes by recompiling as necessary, robots interact with the physical world as it naturally changes over time, and scientific simulations compute with objects whose properties change over time. Although it is possible to develop such applications using ad hoc algorithms and data structures to handle changing data, such algorithms can be challenging to design even for problems that are simple in the batch/static setting where changes to data are not allowed. A striking example is the two-dimensional convex hull problem, whose dynamic (incremental) version required decades of more research [30, 7] than its static version (e.g., [17]). The field of incremental computation aims at deriving software that can respond automatically and efficiently to changing data. Earlier work investigated techniques based on static dependency graphs [14, 38] and memoization [32, 21]. More recent work on self-adjusting computation introduced dynamic dependency graphs [3] and a way to integrate them with a form of memoization [2, 4]. Several flavors of self-adjusting computation have been implemented in programming languages such as C [19], Haskell [8], Java [34] and Standard ML [27, 10].

However, in all prior work on incremental computation, the programmer must reason about the time complexity of incremental execution, which we call *dynamic stability*, by direct analysis of the cost semantics of programs [26]. While this analytical technique makes efficient design possible, dynamic stability is a difficult property to establish because it requires reasoning about execution traces, which can be viewed as graphs of computations and their (run-time) data and control dependencies.

Therefore, we are interested in designing static techniques to help a programmer reason about the dynamic stability of programs. As a first step in this direction, we equip a higher-order functional programming language with a refinement type system for establishing the dynamic stability of programs. Our type system, called CostIt, soundly approximates the dynamic stability of a program as an effect. CostIt builds on index refinement types [37] and type annotations to track which program values may change after an update and which may not [11]. To improve precision, we add subtyping rules motivated by co-monadic types [28]. Together, these give enough expressive power to perform non-trivial, asymptotically tight analysis of dynamic stability of programs.

We provide an overview of CostIt's design, highlighting some challenges and our solutions. First, dynamic stability is a function of changes to a program's inputs and, hence, analysis of dynamic stability requires knowing which of its free variables and, more generally, which of its subexpressions' result values may change after an update. To differentiate changeable and unchangeable values statically, we rely on refinement type annotations from Chen *et al.*'s work on implicit self-adjusting computation [11]:[3] $(\tau)^{\mathbb{S}}$ ascribes values of type $\tau$ which cannot change whereas $(\tau)^{\mathbb{C}}$ ascribes all values of type $\tau$. Second, the dynamic stability of a program is often a function of the length of an input list or the number of elements of the list that may change. To track such attributes of inputs in the type system, we add standard index refinement types in the style of Xi and Pfenning's DML [37] or Gaboardi *et al.*'s DFuzz [16].

Centrally, our type system treats dynamic stability as an effect [29]. Expression typing has the form $e :_{\kappa} \tau$, where $\kappa$ is an upper bound on the cost of propagating changes through any trace of $e$. Similarly, if changes to any trace of a function can be propagated in time at most $\kappa$, we give the function a type of the form $\tau_1 \xrightarrow{\kappa} \tau_2$. The cost $\kappa$ may depend on refinement parameters (e.g., list lengths) that are shared with $\tau_1$. For example, the usual higher-order list mapping function $\texttt{map} : (\tau_1 \to \tau_2) \to \texttt{list}\ \tau_1 \to \texttt{list}\ \tau_2$ can be given the following refined type: $(\tau_1 \xrightarrow{\kappa} \tau_2) \xrightarrow{0} \forall n, \alpha.\ \texttt{list}\,[n]^{\alpha}\ \tau_1 \xrightarrow{\alpha\cdot\kappa} \texttt{list}\,[n]^{\alpha}\ \tau_2$. Roughly, the type says that if each application of the mapping function can be updated in time $\kappa$ and at most $\alpha$ elements of the mapped list change, then the entire map can be updated in time $\alpha \cdot \kappa$. (This refined type is approximate; the exact type is shown later.)

Change propagation has the inherent property that if the inputs to a computation do not change, then propagation on the trace of the computation is bypassed and, hence, incurs zero cost. Often, this property must be taken into account in reasoning about dynamic stability. A key insight in CostIt is that this property corresponds to a *co-monadic* reasoning principle in the type system: If all free variables of an expression have types of the form $(\cdot)^{\mathbb{S}}$, then that ex-

[3] Nearly identical annotations are also used for other purposes, e.g., binding-time analysis [29] and information flow analysis [31].

pression's dynamic stability is 0 and its result type can also be annotated $(\cdot)^{\mathbb{S}}$, irrespective of what or how the expression computes. Thus, $(\tau)^{\mathbb{S}}$ can be treated like the co-monadic type $\Box\tau$ [28]. A novelty in CostIt is that whether a type's label is $(\cdot)^{\mathbb{S}}$ or $(\cdot)^{\mathbb{C}}$ may depend on index refinements (this flexibility is essential for inductive proofs of dynamic stability in many of our examples). Hence, co-monadic rules are represented in an expanded subtyping relation, which, as usual, takes index refinements into account.

We prove that any dynamic stability derived in our type system is an upper bound on the actual cost of trace update (i.e., that our type system is sound). To do this, we develop an abstract cost semantics for trace update. The cost semantics is formalized using a novel syntactic class called *bi-expression*, which simultaneously represents the original expression and the modified expression, and indicates (syntactically) where the two differ. We interpret types using a step-indexed logical relation over bi-expressions (i.e. relationally) with a stipulated change propagation semantics. Bi-expressions are motivated by largely unrelated work in analysis of security protocols [6].

In summary, we make the following contributions. 1) We develop the first type system for establishing dynamic stability of programs. 2) We combine lightweight dependent types, immutability annotations and co-monadic reasoning principles to facilitate static proofs of dynamic stability. 3) We prove the type system sound relative to a new cost semantics for change propagation. 4) We demonstrate the precision and generality of our technique on several examples. An online appendix, available from the authors' homepages, includes parametric polymorphism, many additional examples, higher-order sorts that are needed to type some of the additional examples, proofs of theorems and several inference rules that are omitted from this paper.

*Scope.* This paper focuses on laying the foundations of type-based analysis of dynamic stability. The issue of implementing CostIt's type system is beyond the scope of this paper. We comment on an ongoing implementation in Section 7.

## 2 Types for Dynamic Stability by Example

*Dynamic stability* Suppose a program $e$ has been executed with some input $v$ and, subsequently, we want to re-run the program with a slightly different input $v'$. Dynamic stability measures the amount of time needed for the second execution, given the entire trace of the first execution. The advantage of using the first trace for the second execution is that the runtime can reuse parts of the first trace that are not affected by changes to the input; for parts that are affected, it can selectively *propagate changes* [2]. This can be considerably faster than a from-scratch evaluation. Consider the program $(1+(2+\ldots+10))+x$. Suppose the input $x$ is 0 in the first run and 1 in the second. A naive evaluation of the second run requires 10 additions. However, if a trace of the first execution is available, then the runtime can reuse the result of the first 9 of these additions, which involve unchanged constants. Assuming that an addition takes exactly 1 unit of time (and, for simplicity, that no other runtime operation incurs a cost), the cost of the re-run or the dynamic stability of this program would be 1 unit of time. Abstractly, dynamic stability is a property of two executions of a program and is dependent on a specification of the language's change propagation semantics. For instance, our conclusion that $(1 + (2 + \ldots + 10)) + x$ has dynamic stability 1 assumes that change propagation directly reuses the result of the first 9 additions during the second run. If change propagation is naive, the program might be re-run in its entirety, resulting in a dynamic stability of 10, not 1.

*Change propagation* We assume a simple, standard change propagation semantics. We formalize the semantics in Section 4, but explain it here intuitively. During the first execution of a program expression, we record the expression's execution trace. The trace is a tree, a reification of the big-step derivation of the expression's execution. For the second execution, we allow updates to some of the values embedded in the expression (some of the trace's leaves). Change propagation recomputes the result of the modified expression by propagating changes upward through the trace, starting at the modified leaves. Pointers to modified leaves are an input to change propagation and finding them incurs zero cost. Primitive functions (like $+$, $-$, etc.) on the trace whose arguments change are recomputed, but large parts of the trace may *not* be recomputed from scratch, which makes change propagation asymptotically faster than from-scratch evaluation in many cases. The maximum amount of work done in change propagation of an expression's trace (given assumptions on allowed changes to the expression's leaves) is called the expression's dynamic stability. CostIt helps establish this dynamic stability statically.

If the shape of the execution trace of an updated expression is different from the shape of the trace of the original expression (i.e., if the control flow of the execution changes), then change propagation must, in general, construct some parts of the new trace by evaluating subexpressions from scratch. Analysis of dynamic stability in such cases requires also an analysis of worse-case execution time complexity. In this paper, we disallow (through our type system) control flow dependence on data that may change. This simplifying choice mirrors prior work like DFuzz [16] and still allows us to type several interesting programs like sorting and matrix algorithms. In Section 7, we comment on a CostIt extension that can handle control flow changes.

During change propagation, only re-execution of primitive functions incurs a non-zero cost. Although this may sound counter-intuitive, prior work has shown that by storing values in modifiable reference cells and updating them in-place during change propagation, the cost for structural operations like pairing, projection and list consing can be avoided during change propagation [2, 11]. The details of such implementations are not important here; readers only need to be aware that our change propagation incurs a cost only for re-executing primitive functions of the language.

*Type system overview* We build on a $\lambda$-calculus with lists. The simple types of our language are $\texttt{real}, \texttt{unit}, \tau_1 \times \tau_2, \texttt{list}\ \tau$ and $\tau_1 \to \tau_2$. Since the dynamic stability of an expression depends on sizes of input lists as well as knowledge of which of its free variables (inputs) may change, we add type refinements. First, we refine the type $\texttt{list}\ \tau$ to $\texttt{list}\,[n]^{\alpha}\ \tau$, which specifies lists of length exactly $n$, of which *at most* $\alpha$ elements are allowed to change before the second execution. Technically, $n$ and $\alpha$ are natural numbers in an index domain, over which types may quantify. Second, any type $\tau$ may be refined to $(\tau)^{\mu}$ where $\mu$ belongs to an index sort with two values, $\mathbb{S}$ and $\mathbb{C}$. $(\tau)^{\mathbb{S}}$ specifies those values of type $\tau$ that will not change in the second execution ($\mathbb{S}$ is read "stable"). $(\tau)^{\mathbb{C}}$ specifies all values of type $\tau$ ($\mathbb{C}$ is read "potentially changeable"). $\tau$ and $(\tau)^{\mathbb{C}}$ are subtypes of each other. Our typing judgment takes the form $\Gamma \vdash e :_{\kappa} \tau$. Here, $\kappa$ is an upper bound on the dynamic stability of the expression $e$. (For simplicity, we omit several contexts from the typing judgment in this section.)

*Example 1 (Warm-up)* Assume that computing a primitive operation like addition from scratch costs 1 unit of time. Consider the expression $x + 1$ with one input $x$. This expression can be typed in at least two ways: $x : (\texttt{real})^{\mathbb{S}} \vdash x+1 :_0 (\texttt{real})^{\mathbb{S}}$ and $x : (\texttt{real})^{\mathbb{C}} \vdash x+1 :_1 (\texttt{real})^{\mathbb{C}}$. When $x : (\texttt{real})^{\mathbb{S}}$, $x$ cannot change. So change propagation bypasses the expression $x+1$ and its cost is 0. Moreover, the value of $x+1$ does not change. This justifies the first typing judgment. When $x : (\texttt{real})^{\mathbb{C}}$, change propagation may incur a cost of 1 to recompute the addition in $x + 1$ and the value of $x + 1$ may change. This justifies the second judgment.

*Example 2 (List map)* The CostIt type $\tau_1 \xrightarrow{\kappa} \tau_2$ specifies a function whose body has a change propagation cost upper-bounded by $\kappa$ and whose type is $\tau_1 \to \tau_2$. For instance, based on Example 1, the function $\lambda x.(x + 1)$ can be given either of the types $(\texttt{real})^{\mathbb{S}} \xrightarrow{0} (\texttt{real})^{\mathbb{S}}$ and $(\texttt{real})^{\mathbb{C}} \xrightarrow{1} (\texttt{real})^{\mathbb{C}}$. Consider the standard list map function of simple type $(\tau_1 \to \tau_2) \to \texttt{list}\ \tau_1 \to \texttt{list}\ \tau_2$.

$\texttt{fix map}(f).\ \lambda l.\ \texttt{case}_{\texttt{L}}\ l\ \texttt{of}\ \ \texttt{nil}\ \to \texttt{nil}\ \ |\ \texttt{cons}(h,\ tl)\ \to\ \texttt{cons}(f\ h, \texttt{map}\ f\ tl)$

Suppose that the mapping function $f$ has dynamic stability $\kappa$, i.e., its type is $\tau_1 \xrightarrow{\kappa} \tau_2$ and that the list $l$ has type $\texttt{list}\,[n]^{\alpha}\ \tau_1$ (exactly $n$ elements of which at most $\alpha$ may change). What can we say about the type of the result and the dynamic stability of $\texttt{map}$? If we know that $f$ *will not change*, then change propagation will reapply $f$ at most $\alpha$ times (because at most $\alpha$ list elements will change), so the total cost can be bounded by $\alpha \cdot \kappa$. Moreover, at most $\alpha$ elements of the output will change, so we can give map the following type.[4]

$$\texttt{map} : (\tau_1 \xrightarrow{\kappa} \tau_2)^{\mathbb{S}} \to \forall n.\ \forall \alpha.\ \texttt{list}\,[n]^{\alpha}\ \tau_1 \xrightarrow{\alpha \cdot \kappa} \texttt{list}\,[n]^{\alpha}\ \tau_2 \tag{1}$$

If $f$ may change, then change propagation may have to remap every element of the list and all elements in the output may change. This yields a dynamic stability of $n \cdot \kappa$ and the following type.

$$\texttt{map} : (\tau_1 \xrightarrow{\kappa} \tau_2)^{\mathbb{C}} \to \forall n.\ \forall \alpha.\ \texttt{list}\,[n]^{\alpha}\ \tau_1 \xrightarrow{n \cdot \kappa} \texttt{list}\,[n]^{n}\ \tau_2 \tag{2}$$

[4] If $\kappa$ is omitted from $\tau_1 \xrightarrow{\kappa} \tau_2$, then it is treated as 0. Our expressions (Section 3) have explicit annotations for introducing and eliminating universal and existential quantifiers ($\Lambda.\ e, e[\,], \texttt{pack}\ e, \texttt{unpack}\ e_1\ \texttt{as}\ x\ \texttt{in}\ e_2$). We omit those annotations from our examples for better readability.

We explain how the type in (1) is derived as it highlights our co-monadic reasoning principle. The interesting part of the typing is establishing the change propagation cost of the $\texttt{cons}(h, tl)$ branch in the definition of `map`. We are trying to bound this cost by $\alpha \cdot \kappa$. We know from $l$'s type that at most $\alpha$ elements in $\texttt{cons}(h, tl)$ will change in the second run. However, we do not know whether $h$ is one of those elements. So, our case analysis rule (Section 3, Figure 4) has *two premises for the* `cons` *branch* (a total of three premises, including the premise for `nil`). In the first of these two premises, we assume that $h$ may change, so $h : \tau_1$ and $tl : \texttt{list}\,[n-1]^{\alpha-1}\ \tau_2$. In the second premise, we assume that $h$ cannot change, so $h : (\tau_1)^{\mathbb{S}}$ and $tl : \texttt{list}\,[n-1]^{\alpha}\ \tau_2$. Analysis of the first premise is straightforward: $(f\ h)$ incurs cost $\kappa$ (from $f$'s type $(\tau_1 \xrightarrow{\kappa} \tau_2)^{\mathbb{S}}$) and, inductively, $(\texttt{map}\ f\ tl)$ incurs cost $(\alpha-1)\cdot\kappa$, for a total cost $\kappa + (\alpha-1)\cdot\kappa = \alpha\cdot\kappa$. Analysis of the second premise requires nonstandard reasoning. Here, $tl : \texttt{list}\,[n-1]^{\alpha}\ \tau_2$, so the inductive cost of $(\texttt{map}\ f\ tl)$ is already $\alpha\cdot\kappa$. Hence, we must show that $(f\ h)$ has 0 change propagation cost. For this, we rely on our co-monadic reasoning principle: If all of an expression's free variables have types of the form $(\cdot)^{\mathbb{S}}$ (i.e., their substitutions will not change), then the expression's change propagation cost is 0. Since we know that $f : (\tau_1 \xrightarrow{\kappa} \tau_2)^{\mathbb{S}}$ and $h : (\tau_1)^{\mathbb{S}}$, we can immediately conclude that $(f\ h)$ has 0 change propagation cost.

The same reasoning cannot be applied to the second premise in type (2), where $f : (\tau_1 \xrightarrow{\kappa} \tau_2)^{\mathbb{C}}$. Instead, we can show only that $(f\ h)$ incurs cost $\kappa$. This results in a dynamic stability of $n \cdot \kappa$. Note that in both the types above, the dynamic stability depends on attributes of the input list ($\alpha$ and $n$, respectively). This demonstrates the importance of index refinements in CostIt.

*Example 3 (Balanced list fold)* Standard list fold operations (`foldl` and `foldr`) can be typed easily in CostIt but are uninteresting for incremental computation because they have linear traces and, hence, have $O(n)$ dynamic stability even for single element changes to the input list ($n$ is the list's length). A more interesting operation is what we call the balanced fold. Given an *associative and commutative* binary function $f$ of simple type $\tau \times \tau \to \tau$, a list of simple type $(\texttt{list}\ \tau)$ can be folded by splitting it into two nearly equal sized lists, folding the sublists recursively and then applying $f$ to the two results. This results in a balanced tree-like trace, whose depth is $\lceil \texttt{log}_2(n) \rceil$. A single change to the list causes $\lceil \texttt{log}_2(n) \rceil$ recomputations of $f$. So, if $f$ has dynamic stability $\kappa$, the dynamic stability with one change to the list is $O(\kappa \cdot \texttt{log}_2(n))$. More generally, it can be shown that if $\alpha$ changes are allowed to the list, then the dynamic stability is $O(\kappa \cdot (\alpha + \alpha \cdot \texttt{log}_2(n/\alpha)))$. This simplifies to $O(\kappa \cdot n)$ when $\alpha = n$ (entire list may change) and $O(\kappa \cdot \texttt{log}_2(n))$ when $\alpha = 1$. In the following we implement such a balanced fold operation, `bfold`, and derive its dynamic stability in CostIt.

Our first ingredient is the function `bsplit`, which splits a list of length $n$ into two lists of lengths $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$. This function is completely standard. Its CostIt type, although easily established, is somewhat interesting because it uses an existential quantifier to split the allowed number of changes $\alpha$ into the two split lists. The dynamic stability of `bsplit` is 0 because `bsplit` uses no primitive functions (*cf.* discussion earlier in this section).

```
bsplit : $\forall n.\ \forall\alpha.\ \texttt{list}\,[n]^{\alpha}\ \tau \xrightarrow{0} \exists\beta.\ (\texttt{list}\,[\lceil\frac{n}{2}\rceil]^{\beta}\ \tau \times \texttt{list}\,[\lfloor\frac{n}{2}\rfloor]^{\alpha-\beta}\ \tau)$
fix bsplit($l$). case$_{\mathrm{L}}$ $l$ of
 nil $\to$ (nil, nil)
| cons($h_1$, $tl_1$) $\to$ case$_{\mathrm{L}}$ $tl_1$ of nil $\to$ ([$h_1$], nil)
                                            | cons($h_2$, $tl_2$) $\to$ let ($z_1, z_2$) = bsplit $tl_2$ in
                                                                 (cons($h_1$, $z_1$), cons($h_2$, $z_2$))
```

Using `bsplit` we define the balanced fold function, `bfold`. The function applies only to non-empty lists (reflected in its type later), so the `nil` case is omitted.

```
fix bfold($f$). $\lambda l$. case$_{\mathrm{L}}$ $l$ of
   nil $\to \ldots$
 | cons($h_1$, $tl_1$) $\to$ case$_{\mathrm{L}}$ $tl_1$ of
                       nil $\to h_1$
                     | cons(_, _) $\to$ let ($z_1, z_2$) = (bsplit $l$) in
                                      $f$ (bfold $f$ $z_1$, bfold $f$ $z_2$)
```

We first derive a type for `bfold` informally, and then show how the type is established in CostIt. Assume that the argument $l$ has type $\texttt{list}\,[n]^{\alpha}\ \tau$. We count how many times change propagation may have to reapply $f$ in updating `bfold`'s trace, which is a nearly balanced tree of height $H = \lceil \texttt{log}_2(n) \rceil$. Counting levels from the deepest leaves upward (leaves have level 0), the number of applications of $f$ at level $k$ in the trace is at most $2^{H-k}$. If $\alpha$ leaves change, at most $\alpha$ of these applications must be recomputed. Consequently, the maximum number of recomputations of $f$ at level $k$ is $\texttt{min}(\alpha, 2^{H-k})$. If the dynamic stability of $f$ is $\kappa$, the dynamic stability of `bfold` is $P(n,\alpha,\kappa) = \sum_{k=0}^{\lceil \texttt{log}_2(n)\rceil} \kappa \cdot \texttt{min}(\alpha, 2^{\lceil \texttt{log}_2(n)\rceil - k})$. So, in principle, we should be able to give `bfold` the following type.

$$\texttt{bfold} : (\tau \times \tau \xrightarrow{\kappa} \tau)^{\mathbb{S}} \to \forall n > 0.\ \forall\alpha.\ \texttt{list}\,[n]^{\alpha}\ \tau \xrightarrow{P(n,\alpha,\kappa)} \tau$$

The expression $P(n,\alpha,\kappa)$ may look complex, but it is in $O(\kappa\cdot(\alpha+\alpha\cdot\texttt{log}_2(n/\alpha)))$. (To prove this, split the summation in $P(n,\alpha,\kappa)$ into two: one for $k \leq \lceil \texttt{log}_2(n)\rceil - \lceil \texttt{log}_2(\alpha)\rceil$ and the other for $k > \lceil \texttt{log}_2(n)\rceil - \lceil \texttt{log}_2(\alpha)\rceil$. Our appendix has the details.) Although the type above is correct, we will see soon that in typing the recursive calls in `bfold`, we need to know that `bfold`'s type is annotated $(\cdot)^{\mathbb{S}}$. Hence, the actual type we assign to `bfold` is stronger.

$$\texttt{bfold} : ((\tau \times \tau \xrightarrow{\kappa} \tau)^{\mathbb{S}} \to \forall n > 0.\ \forall\alpha.\ \texttt{list}\,[n]^{\alpha}\ \tau \xrightarrow{P(n,\alpha,\kappa)} \tau)^{\mathbb{S}} \qquad (3)$$

We explain how `bfold`'s type is established in CostIt. The interesting case starts where `bsplit` is invoked. From the type of `bsplit`, we know that variables $z_1$ and $z_2$ in the body of `bfold` have types $\texttt{list}\,[\lceil\frac{n}{2}\rceil]^{\beta}\ \tau$ and $\texttt{list}\,[\lfloor\frac{n}{2}\rfloor]^{\alpha-\beta}\ \tau$, respectively for some $\beta$. Inductively, the change propagation costs of (`bfold` $f$ $z_1$)

and (`bfold` $f$ $z_2$) are $P(\lceil \frac{n}{2} \rceil, \beta, \kappa)$ and $P(\lfloor \frac{n}{2} \rfloor, \alpha - \beta, \kappa)$, respectively. Hence, the change propagation cost of the whole body of `bfold` is $\kappa + P(\lceil \frac{n}{2} \rceil, \beta, \kappa) + P(\lfloor \frac{n}{2} \rfloor, \alpha - \beta, \kappa)$. The additional $\kappa$ accounts for the only application of $f$ in the body of `bfold` (non-primitive operations have zero cost and `bsplit` also has zero cost). Hence, to complete the typing, we must establish the following inequality.

$$\kappa + P(\left\lceil \frac{n}{2} \right\rceil, \beta, \kappa) + P(\left\lfloor \frac{n}{2} \right\rfloor, \alpha - \beta, \kappa) \leq P(n, \alpha, \kappa) \tag{4}$$

This is an easily established arithmetic tautology (our online appendix has a proof), *except* when $\alpha \doteq 0$. When $\alpha \doteq 0$, the right side of the inequality is 0 but we don't necessarily have $\kappa \leq 0$. So, in order to proceed, we consider the cases $\alpha \doteq 0$ and $\alpha > 0$ separately. This requires a typing rule for case analysis on the index domain, which poses no theoretical difficulty. The $\alpha > 0$ case succeeds as described above. For $\alpha \doteq 0$, we use our co-monadic reasoning principle. With $\alpha \doteq 0$, the types of $z_1$ and $z_2$ are equivalent (formally, via subtyping) to $\texttt{list}\left[\lceil \frac{n}{2} \rceil\right]^0 \tau$ and $\texttt{list}\left[\lfloor \frac{n}{2} \rfloor\right]^0 \tau$, respectively. Since, no elements in these lists can change, we use another subtyping rule to promote the types to $(\texttt{list}\left[\lceil \frac{n}{2} \rceil\right]^0 \tau)^{\mathbb{S}}$ and $(\texttt{list}\left[\lfloor \frac{n}{2} \rfloor\right]^0 \tau)^{\mathbb{S}}$, respectively. At this point, the type of every variable occurring in the expression $f$ (`bfold` $f$ $z_1$, `bfold` $f$ $z_2$), including the variable `bfold`, has annotation $(\cdot)^{\mathbb{S}}$. By our co-monadic reasoning principle, the change propagation cost of this expression and, hence, the body of `bfold`, must be 0, which is trivially no more than $P(n, \alpha, \kappa)$. This completes our argument.

Observe that the inference of the annotation $(\cdot)^{\mathbb{S}}$ on the types of $z_1$ and $z_2$ is conditional on the constraint $\alpha \doteq 0$. Subtyping, which is aware of constraints, plays an essential role in determining these annotations and in making our co-monadic reasoning principle useful. Also, the fact that we have to consider the cases $\alpha \doteq 0$ and $\alpha > 0$ separately is not as surprising as it may seem. The case $\alpha \doteq 0$ corresponds to a sub-trace whose leaves have not changed. Since change propagation is a bottom-up procedure, it will bypass this sub-trace completely, incurring no cost. This is exactly what our analysis for $\alpha \doteq 0$ establishes.

Using the type (3) of `bfold`, we can show that for $f : (\tau \times \tau \xrightarrow{\kappa} \tau)^{\mathbb{S}}$ and $l : \texttt{list}\,[n]^{\alpha}\ \tau$, the dynamic stability of (`bfold` $f$ $l$) is in $O(\texttt{log}_2(n))$ when $\alpha \in O(1)$ and in $O(n)$ when $\alpha \in O(n)$, assuming $\kappa$ constant. This dynamic stability is asymptotically tight.

*Example 4 (Merge sort)* The analysis of Example 3 generalizes to other divide-and-conquer algorithms. We illustrate this generalization using merge sort as a second example; our appendix describes a generic template for establishing the dynamic stability of divide-and-conquer algorithms. Abstractly, the trace of merge sort on a list of length $n$ is a tree of height $\lceil \texttt{log}_2(n) \rceil$, where each node receives a list (a sublist of the original list) as input, partitions the list into two nearly equal length sublists, recursively sorts the sublists and then merges the sorted sublists. During change propagation, cost is incurred at a node only in merging the sorted sublists. In the worst case, this cost is $O(m)$, where $m$ is the length of the list being sorted at that node because merging is a linear-time operation. Counting levels from the deepest leaves upward to the root, at level $k$, $m \leq 2^k$. If a single element of the list changes, change propagation might re-merge at each node on the path from this changed element to the root. Hence, the cost is upper-bounded by $1+2+4+\ldots+2^{\lceil \mathtt{log}_2(n) \rceil} \in O(n)$. If all elements of the list may change, the change propagation cost is $O(n \cdot \mathtt{log}_2(n))$. More generally, as we prove below, if $\alpha$ elements of the list change, then change propagation cost is bounded by $O(n \cdot (1+\mathtt{log}_2(\alpha)))$. Importantly, this calculation does not require an analysis of the change propagation cost of the merge function: A completely pessimistic assumption that all merges on any path from a changed element to the root must be re-executed from scratch yields these bounds. Accordingly, we assume that we have a merge function with the most pessimistic bounds. Using this function, we can define the merge sort function, `msort`.

$$\mathtt{merge} : (\forall n, m, \alpha, \beta.\ (\mathtt{list}\,[n]^{\alpha}\ \mathtt{real} \times \mathtt{list}\,[m]^{\beta}\ \mathtt{real}) \xrightarrow{n+m} \mathtt{list}\,[n+m]^{n+m}\ \mathtt{real})^{\mathbb{S}}$$

```
fix msort(l). case_L l of
  nil  → nil
| cons(h_1, tl_1) → case_L tl_1 of
                      nil  → cons(h_1, nil)
                    | cons(_, _) → let (z_1, z_2) = (bsplit l) in
                                       merge (msort z_1, msort z_2)
```

Almost exactly as for `bfold`, `msort` can be given the following type:

$$\mathtt{msort} : (\forall n.\ \forall \alpha.\ \mathtt{list}\,[n]^{\alpha}\ \tau \xrightarrow{Q(n,\alpha)} \mathtt{list}\,[n]^{n}\ \tau)^{\mathbb{S}}$$

where for $Q(n, \alpha) = \sum_{k=0}^{\lceil \mathtt{log}_2(n) \rceil} 2^k \cdot \mathtt{min}(\alpha, 2^{\lceil \mathtt{log}_2(n) \rceil - k})$. $Q(n, \alpha)$ is in $O(n \cdot (1 + \mathtt{log}_2(\alpha)))$. Using this type for `msort`, we can show that for $l : \mathtt{list}\,[n]^{\alpha}\ \tau$, $(\mathtt{msort}\ l)$ has dynamic stability in $O(n)$ for $\alpha \in O(1)$ and in $O(n \cdot \mathtt{log}_2(n))$ for $\alpha \in O(n)$. This dynamic stability is asymptotically tight.

Note that the syntactic cumbersomeness of the expressions $P(n, \alpha, \kappa)$ (Example 3, `bfold`) and $Q(n, \alpha)$ (Example 4, `msort`) is inherent to the dynamic stability of the two algorithms. It is not an artifact of CostIt. We tried to find simpler expressions that would support inductive proofs. For `bfold`, the simpler form $P(n, \alpha, \kappa) = \kappa \cdot (\alpha - 1 + \alpha \cdot (\lceil \mathtt{log}_2(n) \rceil - \mathtt{log}_2(\alpha)))$ for $\alpha > 0$ can be used, but the constraint corresponding to (4) is more difficult to establish and requires real analysis. We do not know of a useful simpler form for $Q(n, \alpha)$.

*Other examples* Our appendix contains several other examples. We briefly list some of these with their asymptotic CostIt-established dynamic stability for single element changes in parenthesis: list append (1), list pair zip (1), matrix transpose (1), dot product ($\mathtt{log}_2(n)$) and matrix multiplication ($n \cdot \mathtt{log}_2(n)$). The matrix examples demonstrate that CostIt can establish asymptotically tight bounds on dynamic stability even when the latter depends on the sizes of nested inner lists.

We note that the dynamic stability proved using CostIt is asymptotically tight for all the examples in this section and our appendix. Nonetheless, like

| | | | |
|---|---|---|---|
| Types | $\tau$ | $::=$ | $\mathtt{real} \mid \tau_1 \times \tau_2 \mid \mathtt{list}\,[n]^{\alpha}\ \tau \mid \tau_1 \xrightarrow{\kappa} \tau_2 \mid \forall i \overset{\kappa}{::} S.\ \tau \mid \exists i.\ \tau \mid$ |
| | | | $\mathtt{unit} \mid C \to \tau \mid C \wedge \tau \mid (\tau)^{\mu}$ |
| Sorts | $S$ | $::=$ | $\mathbb{N} \mid \mathbb{R}^{+} \mid \mathbb{V}$ |
| Index terms | $I, \mu, \kappa,$ | $::=$ | $i \mid \mathbb{S} \mid \mathbb{C} \mid 0 \mid I+1 \mid I_1 + I_2 \mid I_1 - I_2 \mid \frac{I_1}{I_2} \mid I_1 \cdot I_2 \mid$ |
| | $n, \alpha$ | | $\lceil I \rceil \mid \lfloor I \rfloor \mid \mathtt{log}_2(I) \mid I_1^{I_2} \mid \mathtt{min}(I_1, I_2) \mid \mathtt{max}(I_1, I_2) \mid \sum\limits_{k=I_1}^{I_n} I$ |
| Constraints | $C$ | $::=$ | $I_1 \doteq I_2 \mid I_1 < I_2 \mid \neg C$ |
| Constraint env. | $\Phi$ | $::=$ | $\emptyset \mid C \mid \Phi_1 \wedge \Phi_2$ |
| Sort env. | $\Delta$ | $::=$ | $\emptyset \mid \Delta, i :: S$ |
| Type env. | $\Gamma$ | $::=$ | $\emptyset \mid \Gamma, x : \tau$ |
| Primitive env. | $\Upsilon$ | $::=$ | $\emptyset \mid \Upsilon, \zeta : \forall \overline{t_i}.\ \tau_1 \xrightarrow{\kappa} \tau_2$ |

**Fig. 1.** Syntax of types

other type systems, CostIt abstracts over concrete program values and, hence, we cannot expect CostIt's analysis to be asymptotically tight on all programs.

## 3 Syntax and Type System

This section describes CostIt's language, types and type system. Section 4 defines CostIt's dynamic semantics. CostIt is a refinement type system on a call-by-value $\lambda$-calculus with lists, similar to DFuzz [16]. The syntax of CostIt's types and type refinements is listed in Figure 1.

*Index terms and constraints* CostIt's types are refined by index terms, denoted $I, \mu, \kappa, n, \alpha$, etc. Index terms are sorted as follows: (a) natural numbers, $\mathbb{N}$, which are used to specify list lengths and number of changes allowed in a list, (b) non-negative real numbers, $\mathbb{R}^{+}$, that show up in logarithmic expressions in change propagation costs, and (c) the two-valued sort *variation*, $\mathbb{V} = \{\mathbb{S}, \mathbb{C}\}$, used as a type refinement to specify whether a value may change or not from the first to the second execution. The syntax of index terms includes various arithmetic operators, with their usual meanings. Most operators are overloaded for the sorts $\mathbb{R}^{+}$ and $\mathbb{N}$ and there is an implicit coercion from $\mathbb{N}$ to $\mathbb{R}^{+}$. A standard sorting judgment $\Delta \vdash I :: S$ assigns sort $S$ to index term $I$. The sort environment $\Delta$, assigns sorts to index variables, $i, t$. We use different letters for index terms in different roles: $n$ for list lengths, $\alpha$ for the number of allowed changes in a list, $\mu$ for terms of sort $\mathbb{V}$, $\kappa$ for change propagation costs and $I$ for generic terms.

Propositions over index terms are called constraints, denoted $C$. For our examples, we only need comparison and negation. Constraints are collected in a context called the constraint environment, denoted $\Phi$. As usual, logical entailment over constraints is defined by the black-box judgment $\Delta; \Phi \models C$, which is assumed to embody the usual algebraic laws of arithmetic. Constraints are also subject to standard syntactic sorting rules, which we omit.

| | | | |
|---|---|---|---|
| Values | $v$ | $::=$ | $\mathtt{r} \mid (v_1, v_2) \mid \mathtt{nil} \mid \mathtt{cons}(v_1,\, v_2) \mid \mathtt{fix}\ f(x).\, e \mid \Lambda.\, e \mid \mathtt{pack}\ v \mid ()$ |
| Expressions | $e, f$ | $::=$ | $x \mid \mathtt{r} \mid (e_1, e_2) \mid \mathtt{fst}\ e \mid \mathtt{snd}\ e \mid \mathtt{nil} \mid \mathtt{cons}(e_1,\, e_2) \mid$ $\mathtt{case}_{\mathtt{L}}\ e\ \mathtt{of}\ \mathtt{nil}\ \rightarrow\ e_1 \mid \mathtt{cons}(h, tl)\ \rightarrow\ e_2 \mid$ $\mathtt{fix}\ f(x).\, e \mid e_1\ e_2 \mid \Lambda.\, e \mid e[] \mid \mathtt{pack}\ e \mid \mathtt{unpack}\ e\ \mathtt{as}\ x\ \mathtt{in}\ e' \mid$ $\mathtt{let}\ x\ =\ e_1\ \mathtt{in}\ e_2 \mid () \mid \zeta\ e$ |

**Fig. 2.** Syntax of expressions and values

*Types* CostIt types refine types of the simply typed $\lambda$-calculus. The list type $\mathtt{list}\,[n]^{\alpha}\ \tau$ contains two index refinements — $n$ and $\alpha$ — which specify, respectively, the precise length of the list and the maximum number of elements of the list that may be updated before change propagation. The cost annotation $\kappa$ in function types $\tau_1 \xrightarrow{\kappa} \tau_2$ and universally quantified types $\forall i \overset{\kappa}{::} S.\ \tau$ is an upper bound on the change propagation cost of closures contained in the type. The type $C \rightarrow \tau$ reads "$\tau$ if constraint $C$ is true, else every expression". Any type $\tau$ may be annotated with a variation term $\mu$, written $(\tau)^{\mu}$. $(\tau)^{\mathbb{S}}$ specifies values of type $\tau$ that cannot change (in our relational interpretation, $(\tau)^{\mathbb{S}}$ is the diagonal relation on $\tau$). $(\tau)^{\mathbb{C}}$ is equivalent (via subtyping) to $\tau$. There is one representative, unrefined base type $\mathtt{real}$. Other refined and unrefined base types can be added, as in our appendix. We note that it is not obvious how refinements may be extended to algebraic datatypes beyond lists, because needed refinements vary by application. In the case of lists, the refinements length and the number of allowed changes suffice for many applications, so we adopt them. CostIt supports standard type quantification (parametric polymorphism). Type quantification does not interact with our technical development in any significant way, so we defer its details to the appendix.

*Expressions* Figure 2 shows the grammar of CostIt values and expressions. The syntax is mostly standard. $\mathtt{r}$ denotes constants of type $\mathtt{real}$. $\zeta$ denotes a primitive function and $\zeta\ e$ is application of the function to $e$. Primitive functions have a special role in our dynamic semantics because only they incur a non-zero cost during change propagation. The construct $\mathtt{case}_{\mathtt{L}}$ is case analysis on lists.

Our expressions do not mention index terms or index variables. For instance, the introduction and elimination forms for the universal quantifier are $\Lambda.\, e$ and $e[]$ instead of the more common and more elaborate forms $\Lambda i.\, e$ and $e\ [I]$. Index terms are absent from expressions for a reason. As explained in Section 2, the list case analysis rule has two premises for the $\mathtt{cons}(\cdot,\ \cdot)$ branch. Often, universally quantified terms must be instantiated differently in the two premises, which means that if index terms were included in expressions, we would have to write *two expressions* for the $\mathtt{cons}(\cdot,\ \cdot)$ branch. This would be cumbersome at best, so we do not include index terms in expressions. If necessary, the two separate fully annotated expressions can be created by elaboration after type-checking.

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

$$\frac{}{\Delta;\Phi \models (\tau_1 \xrightarrow{\kappa} \tau_2)^{\mu} \sqsubseteq (\tau_1)^{\mu} \xrightarrow{\kappa} (\tau_2)^{\mu}} \to \mathbf{1} \qquad \frac{\Delta;\Phi \models \tau_1' \sqsubseteq \tau_1 \quad \Delta;\Phi \models \tau_2 \sqsubseteq \tau_2' \quad \Delta;\Phi \models \kappa \leq \kappa'}{\Delta;\Phi \models \tau_1 \xrightarrow{\kappa} \tau_2 \sqsubseteq \tau_1' \xrightarrow{\kappa'} \tau_2'} \to \mathbf{2}$$

$$\frac{}{\Delta;\Phi \models (\tau_1 \times \tau_2)^{\mu} \equiv (\tau_1)^{\mu} \times (\tau_2)^{\mu}} \times\mathbf{1} \qquad \frac{}{\Delta;\Phi \models (\mathtt{list}\,[n]^{\alpha}\ \tau)^{\mu} \equiv \mathtt{list}\,[n]^{\alpha}\ (\tau)^{\mu}} \mathbf{l1}$$

$$\frac{\Delta;\Phi \models \mu \doteq \mathbb{S}}{\Delta;\Phi \models (\mathtt{list}\,[n]^{\alpha}\ \tau)^{\mu} \equiv \mathtt{list}\,[n]^{0}\ \tau} \mathbf{l2} \qquad \frac{\Delta;\Phi \models n \doteq n' \quad \Delta;\Phi \models \alpha \leq \alpha' \quad \Delta;\Phi \models \tau \sqsubseteq \tau'}{\Delta;\Phi \models \mathtt{list}\,[n]^{\alpha}\ \tau \sqsubseteq \mathtt{list}\,\left[n'\right]^{\alpha'}\ \tau'} \mathbf{l4}$$

$$\frac{}{\Delta;\Phi \models (\forall t \overset{\kappa}{::} S.\ \tau)^{\mu} \equiv \forall t \overset{\kappa}{::} S.\ (\tau)^{\mu}} \forall\mathbf{2} \qquad \frac{}{\Delta;\Phi \models (\tau)^{\mu} \sqsubseteq \tau} \mathbf{T} \qquad \frac{}{\Delta;\Phi \models \tau \sqsubseteq (\tau)^{\mathbb{C}}} \mathbf{I}$$

**Fig. 3.** Selected subtyping rules

*Subtyping* Like all other index refinement type systems, CostIt relies heavily on subtyping. Selected rules of our subtyping judgment $\Delta;\Phi \vdash \tau_1 \sqsubseteq \tau_2$ are shown in Figure 3. The judgment $\tau_1 \sqsubseteq \tau_2$ means that $\tau_1$ is a subtype of $\tau_2$ and $\tau_1 \equiv \tau_2$ is shorthand for ($\tau_1 \sqsubseteq \tau_2$ and $\tau_2 \sqsubseteq \tau_1$). The rule $\to \mathbf{2}$ defines standard subtyping for function types, covariant in the result and contravariant in the argument. Additionally, function subtyping is covariant in the cost $\kappa$, because $\kappa$ is an upper bound on the dynamic stability. Rule **l4** makes list subtyping invariant in the list size $n$ and covariant in the number $\alpha$ of elements allowed to change (because the former is exact but the latter is an upper-bound).

The remaining subtyping rules shown in Figure 3 mention variation annotations $(\tau)^{\mu}$. These rules are best understood separately for the cases $\mu \doteq \mathbb{S}$ and $\mu \doteq \mathbb{C}$. Rules **T** and **I** imply that $(\tau)^{\mathbb{C}} \equiv \tau$ (expressions are allowed to change unless specified, so the annotation $(\cdot)^{\mathbb{C}}$ provides no additional information). Given this observation, the remaining rules state obvious identities for the case $\mu \doteq \mathbb{C}$.

We describe the rules for the case $\mu \doteq \mathbb{S}$. As expected, $(\tau)^{\mathbb{S}} \sqsubseteq \tau$ (rule **T**), but the converse is not true in general. Rule $\to \mathbf{1}$ says that $(\tau_1 \xrightarrow{\kappa} \tau_2)^{\mathbb{S}} \sqsubseteq (\tau_1)^{\mathbb{S}} \xrightarrow{\kappa} (\tau_2)^{\mathbb{S}}$. This can be read as follows: If a function will not change and it is given an argument that will not change, then the result will not change. The converse is not true: If given a non-changing argument, a function's result will not change, this does not imply that the function itself will not change (e.g., some dead code in the function may change). Rule **l2** implies that $(\mathtt{list}\,[n]^{\alpha}\ \tau)^{\mathbb{S}} \equiv \mathtt{list}\,[n]^{0}\ \tau$. This equivalence is justified as follows: The annotation 0 in the type on the right forbids changes to any elements of the list and its length is fixed at $n$, so the list cannot change. This rule is critical to typing Examples 3 and 4 of Section 2.

Readers familiar with co-monadic types or constructive modal logic will notice that our subtyping rules for $(\tau)^{\mathbb{S}}$ mirror rules for a co-monad $\Box\tau$: $\Box\tau \sqsubseteq \tau$ (but not the converse), $\Box(\tau_1 \to \tau_2) \sqsubseteq (\Box\tau_1 \to \Box\tau_2)$ and $\Box(\tau_1 \times \tau_2) \equiv (\Box\tau_1 \times \Box\tau_2)$.

$\boxed{\Delta;\Phi;\Gamma \vdash e :_{\kappa} \tau}$ expression $e$ has type $\tau$ and dynamic stability at most $\kappa$

$$\frac{}{\Delta;\Phi;\Gamma, x:\tau \vdash x :_0 \tau}\ \textbf{var} \qquad \frac{}{\Delta;\Phi;\Gamma \vdash \mathtt{r} :_0 (\mathtt{real})^{\mathbb{S}}}\ \textbf{real}$$

$$\frac{}{\Delta;\Phi;\Gamma \vdash \mathtt{nil} :_0 \mathtt{list}\,[0]^{0}\ \tau}\ \textbf{nil}$$

$$\frac{\Delta;\Phi;\Gamma \vdash e_1 :_{\kappa_1} (\tau)^{\mathbb{S}} \qquad \Delta;\Phi;\Gamma \vdash e_2 :_{\kappa_2} \mathtt{list}\,[n]^{\alpha}\ \tau}{\Delta;\Phi;\Gamma \vdash \mathtt{cons}(e_1,\ e_2) :_{\kappa_1+\kappa_2} \mathtt{list}\,[n{+}1]^{\alpha}\ \tau}\ \textbf{cons1}$$

$$\frac{\Delta;\Phi;\Gamma \vdash e_1 :_{\kappa_1} \tau \qquad \Delta;\Phi;\Gamma \vdash e_2 :_{\kappa_2} \mathtt{list}\,[n]^{\alpha-1}\ \tau \qquad \Delta;\Phi \models \alpha > 0}{\Delta;\Phi;\Gamma \vdash \mathtt{cons}(e_1,\ e_2) :_{\kappa_1+\kappa_2} \mathtt{list}\,[n+1]^{\alpha}\ \tau}\ \textbf{cons2}$$

$$\frac{\begin{array}{c} \Delta;\Phi;\Gamma \vdash e :_{\kappa} \mathtt{list}\,[n]^{\alpha}\ \tau \qquad \Delta;\Phi \wedge n \doteq 0;\Gamma \vdash e_1 :_{\kappa'} \tau' \\ i :: \iota, \Delta;\Phi \wedge n \doteq i+1; h : (\tau)^{\mathbb{S}}, tl : \mathtt{list}\,[i]^{\alpha}\ \tau, \Gamma \vdash e_2 :_{\kappa'} \tau' \\ i :: \iota, \beta :: \iota, \Delta;\Phi \wedge n \doteq i+1 \wedge \alpha \doteq \beta + 1; h : (\tau)^{\mathbb{C}}, tl : \mathtt{list}\,[i]^{\beta}\ \tau, \Gamma \vdash e_2 :_{\kappa'} \tau' \end{array}}{\Delta;\Phi;\Gamma \vdash \mathtt{case}_{\mathtt{L}}\ e\ \mathtt{of}\ \mathtt{nil}\ \rightarrow\ e_1\ \mid\ \mathtt{cons}(h, tl)\ \rightarrow\ e_2 :_{\kappa+\kappa'} \tau'}\ \textbf{caseL}$$

$$\frac{\Delta;\Phi;x:\tau_1, f:\tau_1 \xrightarrow{\kappa} \tau_2, \Gamma \vdash e :_{\kappa} \tau_2}{\Delta;\Phi;\Gamma \vdash \mathtt{fix}\ f(x).\,e :_0 \tau_1 \xrightarrow{\kappa} \tau_2}\ \textbf{fix1} \qquad \frac{\begin{array}{c} \Delta;\Phi;\Gamma \vdash e_1 :_{\kappa_1} \tau_1 \xrightarrow{\kappa} \tau_2 \\ \Delta;\Phi;\Gamma \vdash e_2 :_{\kappa_2} \tau_1 \end{array}}{\Delta;\Phi;\Gamma \vdash e_1\ e_2 :_{(\kappa_1+\kappa_2+\kappa)} \tau_2}\ \textbf{app}$$

$$\frac{t :: S, \Delta;\Phi;\Gamma \vdash e :_{\kappa} \tau}{\Delta;\Phi;\Gamma \vdash \Lambda.\,e :_0 \forall t \overset{\kappa}{::} S.\ \tau}\ \forall\textbf{I} \qquad \frac{\Delta;\Phi;\Gamma \vdash e :_{\kappa} \forall t \overset{\kappa'}{::} S.\ \tau \qquad \Delta \vdash I :: S}{\Delta;\Phi;\Gamma \vdash e[] :_{\kappa+\kappa'\{I/t\}} \tau\{I/t\}}\ \forall\textbf{E}$$

$$\frac{\Delta;\Phi;\Gamma \vdash e :_{\kappa} \tau \qquad \Delta;\Phi \models \tau \sqsubseteq \tau' \qquad \Delta;\Phi \models \kappa \leq \kappa'}{\Delta;\Phi;\Gamma \vdash e :_{\kappa'} \tau'}\ \sqsubseteq$$

$$\frac{\varUpsilon(\zeta) = \zeta : \forall \overline{t_i}{::}\overline{S_i}.\ \tau_1 \xrightarrow{\kappa} \tau_2 \qquad \Delta \vdash \overline{I_i} :: \overline{S_i} \qquad \Delta;\Phi;\Gamma \vdash e :_{\kappa_e} \tau_1[\overline{I_i/t_i}]}{\Delta;\Phi;\Gamma \vdash \zeta\ e :_{\kappa_e+\kappa[\overline{I_i/t_i}]} \tau_2[\overline{I_i/t_i}]}\ \textbf{primApp}$$

$$\frac{\Delta;\Phi;\Gamma \vdash e :_{\kappa} \tau \qquad \forall y \in \Gamma.\ \ \Delta;\Phi \models \Gamma(y) \sqsubseteq (\Gamma(y))^{\mathbb{S}}}{\Delta;\Phi;\Gamma,\Gamma' \vdash e :_0 (\tau)^{\mathbb{S}}}\ \textbf{nochange}$$

$$\frac{\Delta;\Phi;x:\tau_1, f:(\tau_1 \xrightarrow{\kappa} \tau_2)^{\mathbb{S}}, \Gamma \vdash e :_{\kappa} \tau_2 \qquad \forall y \in \Gamma.\ \ \Delta;\Phi \models \Gamma(y) \sqsubseteq (\Gamma(y))^{\mathbb{S}}}{\Delta;\Phi;\Gamma,\Gamma' \vdash \mathtt{fix}\ f(x).\,e :_0 (\tau_1 \xrightarrow{\kappa} \tau_2)^{\mathbb{S}}}\ \textbf{fix2}$$

**Fig. 4.** Selected typing rules. The context $\varUpsilon$ carrying types of primitive functions is omitted from all rules.

*Typing rules* Our typing judgment has the form $\Delta; \Phi; \Gamma \vdash e :_\kappa \tau$. Here, $\kappa$ is an upper bound on the dynamic stability of $e$. It is treated as an effect. Important typing rules are shown in Figure 4. Technically, all rules include a fourth context $\Upsilon$ that specifies the types of primitive functions $\zeta$, but this context does not change in the rules, so we exclude it from the presentation. The rules follow some general principles. First, if an expression contains subexpressions, then the change propagation costs ($\kappa$'s) of subexpressions are added to obtain the change propagation cost of the expression. This is akin to accumulation of effects in a type and effect system. Second, values incur 0 change propagation cost because they are either updated before change propagation starts or by earlier steps of change propagation (which account for the cost of their update).

Variables represent values, so they have $\kappa = 0$ (rule **var**). All primitive constants like $\mathtt{r}$ can be given the type annotation $(\cdot)^{\mathbb{S}}$ as in the rule **real**. (Modifiable constants can be modeled as variables with types without the $(\cdot)^{\mathbb{S}}$ annotation and given two different substitutions in the two runs. This is standard in relational semantics and should be clear in Section 5.) This also applies to the empty list `nil`, but in its typing (rule **nil**) we do not explicitly write the annotation $(\cdot)^{\mathbb{S}}$ because this annotation can be established through the subtyping rule **l2**. The term $\mathtt{cons}(e_1, e_2)$ can be typed at $\mathtt{list}\,[n+1]^{\alpha}\ \tau$ using one of two rules (**cons1** and **cons2**) depending on whether $e_1$ may change or not. If $e_1$ cannot change (it has type $(\tau)^{\mathbb{S}}$), then $e_2$ is allowed $\alpha$ changes (rule **cons1**). If $e_1$ may change, then $e_2$ is allowed $\alpha - 1$ changes (rule **cons2**). The elimination rule for a list expression $e : \mathtt{list}\,[n]^{\alpha}\ \tau$ has three premises for the case branches (rule **caseL**). The first of these premises applies when $e$ evaluates to `nil`. In this premise, we assume that the size of the list $n$ and the number of allowed changes $\alpha$ are both 0. The remaining two premises correspond to the two typing rules for `cons`. In one premise, we assume that the head of the list (variable $h$) cannot change, so it has type $(\tau)^{\mathbb{S}}$ and the tail may have $\alpha$ changes. In the other premise, we assume that the head may change, so it has type $\tau$, but the tail may have only $\alpha - 1$ changes ($\alpha - 1$ is denoted by a new index variable $\beta$ in the rule).

Rules **fix1** and **app** type recursive functions and function applications, respectively. A function is a value, so $\kappa = 0$ in rule **fix1**. In rule **app**, we add the function's change propagation cost $\kappa$ to the cost of the application, as expected. Rule $\sqsubseteq$ allows weakening an ascribed type to any supertype and also allows weakening the change propagation cost upper-bound $\kappa$. Rule **primApp** types primitive function applications. This rule eliminates both $\forall$ and $\rightarrow$ from the type of the primitive function. $\overline{I_i}$ denotes a vector of index terms.

The rule **nochange** embodies our co-monadic reasoning principle. It says: If $e :_\kappa \tau$ in some context $\Gamma$ (first premise) and the type of every variable in type $\Gamma$ is a *subtype* of the same type annotated with $(\cdot)^{\mathbb{S}}$ (second premise), then we can also give $e$ the type $(\tau)^{\mathbb{S}}$ and change propagation cost 0. In other words, if an expression depends only on unchanging variables, then its result cannot change and no change propagation is required. This rule is a strict generalization of the introduction rule for the type $\Box\tau$ in co-monadic type systems like [28]: If $e : \tau$ and all of $e$'s free variables have types of the form $\Box\tau'$, then $e : \Box\tau$. The generalization here is that whether or not a variable in context has annotation $(\cdot)^{\mathbb{S}}$ can depend on the constraints in $\Phi$ (via subtyping). We showed an application of this general rule in Example 3 of Section 2. Finally, we need an additional rule to type some recursive functions with annotation $(\cdot)^{\mathbb{S}}$ (an example is the function `bfold` of Section 2). This rule, **fix2**, has the same condition on the function's free variables as the rule **nochange**. In typing the body of the recursive function, **fix2** allows us to assume that the function itself has a type annotated $(\cdot)^{\mathbb{S}}$. This rule cannot be derived using the rules **fix1** and **nochange**.

## 4 Dynamic Semantics

We define a tracing evaluation semantics and a cost-counting change propagation semantics for our language (Sections 4.1 and 4.2, respectively). We then prove our type system sound relative to the change propagation semantics (Section 5).

### 4.1 Evaluation Semantics and Traces

Our big-step, call-by-value evaluation judgment has the form $e \Downarrow v, T$ where $e$ is the evaluated program, value $v$ is the result of evaluating $e$ and $T$ is a reification of the big-step derivation tree, called a trace. The trace is used for change propagation after $e$ has been modified. Traces have the following syntax.

$$\begin{array}{lll}\text{Traces} & T ::= & \mathtt{r} \mid () \mid (T_1, T_2) \mid \mathtt{fst}\ T \mid \mathtt{snd}\ T \mid \mathtt{nil} \mid \mathtt{cons}(T_1, T_2) \mid \\ & & \mathtt{case}_{\mathtt{nil}}(T, T') \mid \mathtt{case}_{\mathtt{cons}}(T, T') \mid \mathtt{fix}\, f(x).e \mid \mathtt{app}(T_1, T_2, T_r) \mid \\ & & \Lambda.e \mid \mathtt{iApp}(T, T_r) \mid \mathtt{pack}\ T \mid \mathtt{unpack}\ T\ \mathtt{as}\ x\ \mathtt{in}\ T' \mid \\ & & \mathtt{let}\ x\ =\ T_1\ \mathtt{in}\ T_2 \mid \mathtt{primApp}(T, v_r, \zeta)\end{array}$$

This syntax has one constructor for every evaluation rule and is largely self-explanatory. The trace of a value is the value itself. The trace of a primitive function application $\zeta\ e$ has the form $\mathtt{primApp}(T, v_r, \zeta)$, where $T$ is the trace of $e$ and $v_r$ is the result of the application. Recording $v_r$ is important: During change propagation, if the argument to the primitive function has not changed, then we simply reuse $v_r$, without re-computing the primitive function.

Selected evaluation rules are shown in Figure 5. The rules are self-explanatory, given the description of traces above. In the rule **primapp**, $\widehat{\zeta}$ denotes the semantic interpretation of the primitive $\zeta$. For every value $v$, $\widehat{\zeta}(v)$ is a pair $(c_r, v_r)$, where $v_r$ is the result of evaluating the primitive $\zeta$ with argument $v$ and $c_r$ is the cost of this primitive evaluation.

### 4.2 Cost-counting Change Propagation Semantics

Change propagation takes as input the trace of an expression and a modified expression, and computes the trace of the modified expression by propagating changes through the original trace. This begs two questions: First, what kinds of expression modifications we allow and, second, how do we specify the modifications. The answer to the first question is that changes *stem* from replacing

$\boxed{e \Downarrow v, T}$ Expression $e$ evaluates to value $v$ with trace $T$

$$\frac{}{\mathtt{r} \Downarrow \mathtt{r}, \mathtt{r}}\ \textbf{r} \qquad \frac{e_1 \Downarrow v_1, T_1 \qquad e_2 \Downarrow v_2, T_2}{\mathtt{cons}(e_1,\, e_2) \Downarrow \mathtt{cons}(v_1,\, v_2), \mathtt{cons}(T_1, T_2)}\ \textbf{cons}$$

$$\frac{}{\mathtt{fix}\ f(x).\, e \Downarrow \mathtt{fix}\ f(x).\, e, \mathtt{fix}\, f(x).e}\ \textbf{fix} \qquad \frac{e \Downarrow v, T \qquad \widehat{\zeta}(v) = (c_r, v_r)}{\zeta\ e \Downarrow v_r, \mathtt{primApp}(T, v_r, \zeta)}\ \textbf{primapp}$$

$$\frac{e_1 \Downarrow \mathtt{fix}\ f(x).\, e, T_1 \qquad e_2 \Downarrow v_2, T_2 \qquad e[v_2/x, (\mathtt{fix}\ f(x).\, e)/f] \Downarrow v_r, T_r}{e_1\ e_2 \Downarrow v_r, \mathtt{app}(T_1, T_2, T_r)}\ \textbf{app}$$

**Fig. 5.** Selected evaluation rules

$$\begin{array}{lrl}
\text{Bi-values} & \mathtt{w} ::= & \mathtt{keep(r)} \mid \mathtt{repl(r, r')} \mid (\mathtt{w}_1, \mathtt{w}_2) \mid \mathtt{nil} \mid \mathtt{cons}(\mathtt{w}_1, \mathtt{w}_2) \mid \\
 & & \mathtt{fix}\ f(x).\mathtt{ee} \mid \Lambda.\mathtt{ee} \mid \mathtt{pack}\ \mathtt{w} \mid () \\
\text{Bi-expr.} & \mathtt{ee} ::= & x \mid \mathtt{keep(r)} \mid \mathtt{repl(r, r')} \mid (\mathtt{ee}_1, \mathtt{ee}_2) \mid \mathtt{fst}\ \mathtt{ee} \mid \mathtt{snd}\ \mathtt{ee} \mid \\
 & & \mathtt{nil} \mid \mathtt{fix}\ f(x).\mathtt{ee} \mid \mathtt{ee}_1\ \mathtt{ee}_2 \mid \Lambda.\mathtt{ee} \mid \mathtt{ee}[] \mid \mathtt{pack}\ \mathtt{ee} \mid \\
 & & \mathtt{unpack}\ \mathtt{ee}\ \mathtt{as}\ x\ \mathtt{in}\ \mathtt{ee}' \mid \mathtt{let}\ x\ =\ \mathtt{ee}_1\ \mathtt{in}\ \mathtt{ee}_2 \mid \zeta\ \mathtt{ee} \mid () \mid \\
 & & \mathtt{cons}(\mathtt{ee}_1, \mathtt{ee}_2) \mid (\mathtt{case}_{\mathrm{L}}\ \mathtt{ee}\ \mathtt{of}\ \mathtt{nil}\ \rightarrow\ \mathtt{ee}_1 \mid \mathtt{cons}(h, tl)\ \rightarrow\ \mathtt{ee}_2)
\end{array}$$

**Fig. 6.** Syntax of bi-values and bi-expressions

primitive values of a base type like `real` with other primitive values of the same type. Because our language contains closures and lists, changes lift to higher types, e.g., if a function receives the function $\lambda x.(x+1)$ as argument in the original execution, it may receive $\lambda x.(x+2)$ after modification. However, it is not possible to receive $\lambda x.(x+1)$ as argument in the original execution and $\lambda x.(x+x)$ after modification. Similarly, the list $[1, 2, 3]$ may be modified to $[2, 2, 4]$, but because the refined list type mentions a statically determined length, it is not possible to modify the length of a list.

To *specify* expression changes and to prove soundness of our type system, we find it convenient to define a new syntactic category called a *bi-expression*, denoted $\mathtt{ee}$. A bi-expression represents two nearly identical expressions (the original and the modified) that differ only in some primitive constants. The functions $\mathrm{L}(\mathtt{ee})$ and $\mathrm{R}(\mathtt{ee})$ project out the left and right (original and modified) expressions from $\mathtt{ee}$. The syntax of bi-expressions, shown in Figure 6, is identical to that of expressions, except that instead of primitive constants $\mathtt{r}$, we have the forms $\mathtt{keep(r)}$ and $\mathtt{repl(r, r')}$. Roughly, $\mathtt{keep(r)}$ means that the original constant $\mathtt{r}$ has not been modified, whereas $\mathtt{repl(r, r')}$ means that the original constant $\mathtt{r}$ has been replaced by the constant $\mathtt{r}'$. Analogous to bi-expressions, we define bi-values, denoted $\mathtt{w}$, that represent pairs of values differing only in primitive constants. As an example, if the original value $\mathtt{fix}\ f(x).\,(x+1)$ is modified to $\mathtt{fix}\ f(x).\,(x+2)$, then the two values can be represented together as the bi-value $\mathtt{fix}\ f(x).\,(x + \mathtt{repl}(1, 2))$. The left and right projections of a bi-expression/bi-

$\boxed{\Delta; \Phi; \Gamma \vdash \mathbb{w} \gg \tau \text{ and } \Delta; \Phi; \Gamma \vdash \mathbb{ee} \gg_\kappa \tau}$ Bi-value and bi-expression typing

$$\frac{}{\Delta; \Phi; \Gamma \vdash \texttt{keep}(\texttt{r}) \gg (\texttt{real})^{\mathbb{S}}}\ \textbf{keep} \qquad \frac{}{\Delta; \Phi; \Gamma \vdash \texttt{repl}(\texttt{r}, \texttt{r}') \gg (\texttt{real})^{\mathbb{C}}}\ \textbf{repl}$$

$$\frac{\Delta; \Phi; x : \tau_1, f : \tau_1 \xrightarrow{\kappa} \tau_2, \Gamma \vdash \mathbb{ee} \gg_\kappa \tau_2}{\Delta; \Phi; \Gamma \vdash \texttt{fix}\ f(x). \mathbb{ee} \gg \tau_1 \xrightarrow{\kappa} \tau_2}\ \textbf{fix1}$$

$$\frac{\Delta; \Phi; \Gamma \vdash \mathbb{w} \gg \tau \qquad \forall z \in \Gamma.\ \ \Delta; \Phi \models \Gamma(z) \sqsubseteq (\Gamma(z))^{\mathbb{S}} \qquad \texttt{stable}(\mathbb{w})}{\Delta; \Phi; \Gamma, \Gamma' \vdash \mathbb{w} \gg (\tau)^{\mathbb{S}}}\ \textbf{nochange}$$

$$\frac{\begin{array}{c}\Delta; \Phi; x : \tau_1, f : (\tau_1 \xrightarrow{\kappa} \tau_2)^{\mathbb{S}}, \Gamma \vdash \mathbb{ee} \gg_\kappa \tau_2 \\ \forall z \in \Gamma.\ \ \Delta; \Phi \models \Gamma(z) \sqsubseteq (\Gamma(z))^{\mathbb{S}} \qquad \texttt{stable}(\mathbb{ee})\end{array}}{\Delta; \Phi; \Gamma, \Gamma' \vdash \texttt{fix}\ f(x). \mathbb{ee} \gg (\tau_1 \xrightarrow{\kappa} \tau_2)^{\mathbb{S}}}\ \textbf{fix2} \qquad \frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash \mathbb{w} \gg \tau \\ \Delta; \Phi \models \tau \sqsubseteq \tau'\end{array}}{\Delta; \Phi; \Gamma \vdash \mathbb{w} \gg \tau'}\ \sqsubseteq$$

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

**Fig. 7.** Selected typing rules for bi-values and bi-expressions

value are defined as the homomorphic lifting of the following definitions.

$$\begin{array}{r||l} \mathrm{L}(\texttt{keep}(\texttt{r})) = \texttt{r} & \mathrm{R}(\texttt{keep}(\texttt{r})) = \texttt{r} \\ \mathrm{L}(\texttt{repl}(\texttt{r}, \texttt{r}')) = \texttt{r} & \mathrm{R}(\texttt{repl}(\texttt{r}, \texttt{r}')) = \texttt{r}' \end{array}$$

Bi-values and bi-expressions are typed as shown in Figure 7. The judgment $\Delta; \Phi; \Gamma \vdash \mathbb{w} \gg \tau$ means that the bi-value $\mathbb{w}$ represents two (related) values of type $\tau$. Its rules mirror those of value typing, mostly. The bi-value $\texttt{keep}(\texttt{r})$ has the type $(\texttt{real})^{\mathbb{S}}$, whereas the bi-value $\texttt{repl}(\texttt{r}, \texttt{r}')$ has the type $(\texttt{real})^{\mathbb{C}}$, reflecting the difference between the refinements $(\cdot)^{\mathbb{S}}$ and $(\cdot)^{\mathbb{C}}$. Rules **fix2** and **nochange** are analogous to their homonyms from expression typing and introduce the annotation $(\cdot)^{\mathbb{S}}$. In these rules, we have to additionally check that the bi-value being typed contains no syntactic occurrences of $\texttt{repl}(\cdot, \cdot)$ because the annotation $(\cdot)^{\mathbb{S}}$ means absence of syntactic change. This is formalized by the proposition $\texttt{stable}(\mathbb{ee})$, which means that $\mathbb{ee}$ has no occurrences of $\texttt{repl}(\cdot, \cdot)$.

The judgment $\Delta; \Phi; \Gamma \vdash \mathbb{ee} \gg_\kappa \tau$ means that $\mathbb{ee}$ represents two related expressions of type $\tau$ and that the trace of any one of those expressions can be change propagated for the other expression, incurring cost at most $\kappa$. This judgment is defined by only one rule, **exp**, that relies on the typing judgments for expressions and bi-values. Let $\ulcorner e \urcorner$ denote the bi-expression obtained by replacing all occurrences of $\texttt{r}$ in $e$ with $\texttt{keep}(\texttt{r})$. It is easy to see that every bi-expression $\mathbb{ee}$ can be written as $\ulcorner e \urcorner [\overline{\mathbb{w}_i / x_i}]$ for some expression $e$ and some sequence of bi-values $\overline{\mathbb{w}_i}$. The rule **exp** types $\mathbb{ee}$ by typing $e$ (using the expression typing rules) and $\overline{\mathbb{w}_i}$ (using the bi-value typing rules). Setting up bi-expression typing this way is primarily for technical convenience in proving the soundness of our type

$\boxed{\langle T, \mathbb{ee}\rangle \curvearrowright \mathtt{w}', T', c'}$ Change propagation with cost-counting

$$\frac{\mathtt{stable}(\mathbb{ee})}{\langle \mathtt{primApp}(T, v_r, \zeta), \zeta\ \mathbb{ee}\rangle \curvearrowright \ulcorner v_r \urcorner, \mathtt{primApp}(T, v_r, \zeta), 0}\ \textbf{r-prim-s}$$

$$\frac{\neg\mathtt{stable}(\mathbb{ee}) \qquad \langle T, \mathbb{ee}\rangle \curvearrowright \mathtt{w}', T', c' \qquad \widehat{\zeta}(\mathrm{R}(\mathtt{w}')) = (c'_r, v'_r)}{\langle \mathtt{primApp}(T, v_r, \zeta), \zeta\ \mathbb{ee}\rangle \curvearrowright \mathtt{merge}(v_r, v'_r), \mathtt{primApp}(T', v'_r, \zeta),\ c' + c'_r}\ \textbf{r-prim}$$

$$\frac{}{\langle \mathtt{r}, \mathtt{keep}(\_)\rangle \curvearrowright \mathtt{keep}(\mathtt{r}), \mathtt{r}, 0}\ \textbf{r-keep} \qquad \frac{}{\langle \mathtt{r}, \mathtt{repl}(\_, \mathtt{r}')\rangle \curvearrowright \mathtt{repl}(\mathtt{r}, \mathtt{r}'), \mathtt{r}', 0}\ \textbf{r-repl}$$

$$\frac{\langle T_1, \mathbb{ee}_1\rangle \curvearrowright \mathtt{w}'_1, T'_1, c'_1 \qquad \langle T_2, \mathbb{ee}_2\rangle \curvearrowright \mathtt{w}'_2, T'_2, c'_2}{\langle \mathtt{cons}(T_1, T_2), \mathtt{cons}(\mathbb{ee}_1, \mathbb{ee}_2)\rangle \curvearrowright \quad \mathtt{cons}(\mathtt{w}'_1, \mathtt{w}'_2), \mathtt{cons}(T'_1, T'_2), c'_1 + c'_2}\ \textbf{r-cons}$$

$$\frac{\langle T, \mathbb{ee}\rangle \curvearrowright \mathtt{nil}, T', c' \qquad \langle T_1, \mathbb{ee}_1\rangle \curvearrowright \mathtt{w}'_1, T'_1, c'_1}{\begin{array}{c}\langle \mathtt{case}_{\mathtt{nil}}(T, T_1), \mathtt{case}_{\mathtt{L}}\ \mathbb{ee}\ \mathtt{of}\ \mathtt{nil}\ \rightarrow\ \mathbb{ee}_1 \mid \mathtt{cons}(h, tl)\ \rightarrow\ \mathbb{ee}_2\rangle \curvearrowright \\ \mathtt{w}'_1, \mathtt{case}_{\mathtt{nil}}(T', T'_1), c' + c'_1\end{array}}\ \textbf{r-case-nil}$$

$$\frac{\langle T, \mathbb{ee}\rangle \curvearrowright \mathtt{cons}(\mathtt{w}_h, \mathtt{w}_{tl}), T', c' \qquad \langle T_2, \mathbb{ee}_2[\mathtt{w}_h/h, \mathtt{w}_{tl}/tl]\rangle \curvearrowright \mathtt{w}'_2, T'_2, c'_2}{\begin{array}{c}\langle \mathtt{case}_{\mathtt{cons}}(T, T_1), \mathtt{case}_{\mathtt{L}}\ \mathbb{ee}\ \mathtt{of}\ \mathtt{nil}\ \rightarrow\ \mathbb{ee}_1 \mid \mathtt{cons}(h, tl)\ \rightarrow\ \mathbb{ee}_2\rangle \curvearrowright \\ \mathtt{w}'_2, \mathtt{case}_{\mathtt{cons}}(T', T'_2), c' + c'_2\end{array}}\ \textbf{r-case-cons}$$

**Fig. 8.** Selected Replay Rules

system. An equivalent type system is obtained by mirroring all the expression typing rules for bi-expressions.

*Change Propagation* We formalize change propagation abstractly by the judgment $\langle T, \mathbb{ee}\rangle \curvearrowright \mathtt{w}', T', c'$, which has inputs $T$ and $\mathbb{ee}$ and outputs $\mathtt{w}'$, $T'$ and $c'$. The input $T$ must be the trace of the original expression $\mathrm{L}(\mathbb{ee})$. The output $\mathtt{w}'$ represents two values, $\mathrm{L}(\mathtt{w}')$ and $\mathrm{R}(\mathtt{w}')$, which are the results of evaluating the original and modified expressions, respectively. The output $T'$ is the trace of the modified expression. Most importantly, $c'$ is the total cost incurred in change propagation. The output $\mathtt{w}'$ is an artifact of our formalization and important for an inductive proof of our soundness theorem. Actual implementations of change propagation never construct it and, hence, we do not count any cost for constructing or analyzing it during change propagation. As part of our soundness theorem, we show that $\curvearrowright$ is a total function on well-typed programs.

Rules defining the judgment $\curvearrowright$ case analyze the input trace $T$. Representative rules are shown in Figure 8. To change propagate the trace $\mathtt{primApp}(T, v_r, \zeta)$ for the primitive function application bi-expression $\zeta\ \mathbb{ee}$, we case analyze whether the original expression in $\mathbb{ee}$ changed or not. If $\mathtt{stable}(\mathbb{ee})$, then the argument to $\zeta$ has not changed ($\mathtt{stable}(\mathbb{ee})$ implies $\mathrm{L}(\mathbb{ee}) = \mathrm{R}(\mathbb{ee})$). So we simply reuse the result $v_r$ stored in the original trace. The output bi-value is $\ulcorner v_r \urcorner$ (which represents $v_r$ paired with itself). The output trace is the same as the input trace and the cost is 0. This is summarized in the rule **r-prim-s**. If, on the other hand, $\neg\mathtt{stable}(\mathbb{ee})$ (rule **r-prim**), then the argument to $\zeta$ has changed, so we change propagate through the argument (second premise) and reapply the primitive function $\zeta$ to the updated argument (third premise). The bi-value in the output is obtained by *merging* the original result $v_r$ with the new result $v'_r$. Merge is defined as follows: If $\mathrm{L}(\mathtt{w}) = v_r$ and $\mathrm{R}(\mathtt{w}) = v'_r$, then $\mathtt{merge}(v_r, v'_r) = \mathtt{w}$. In general, merge is a partial function. But, if the primitive function's interpretation lies in the semantic interpretation of its type (semantic interpretations are defined in the next section), then the merge must be defined. The cost of change propagation is the sum of the cost $c'$ of change propagating the argument of $\zeta$ and the cost $c'_r$ of evaluating $\zeta$ on the new argument. This rule is the only source of non-zero costs during change propagation. All other rules either incur zero cost, or simply aggregate costs from the premises.

The trace of a primitive constant $\mathtt{r}$ is change propagated using rules **r-keep** and **r-repl**. If the constant has not changed (rule **r-keep**) then the trace does not change and no cost is incurred. If the constant has changed, the resulting trace is the new value of the constant (rule **r-repl**). Even in this case, no cost is incurred, because in an implementation of change propagation, the trace and the expression can share a pointer to the constant so the update to the expression (which happens before change propagation starts) implicitly updates the trace [11]. At constructors like `cons`, change propagation simply recurses on argument sub-traces and adds the costs (rule **cons**). Elimination forms like $\mathtt{case}_{\mathrm{L}}$ are handled similarly. Because control flow changes are forbidden, the original trace determines the branch of the case analysis to which changes must be propagated (rules **r-case-nil** and **r-case-cons**).

*Implementation.* The relation $\curvearrowright$ formalizes change propagation and its cost *abstractly*. An obvious question is whether change propagation can be *implemented* with the costs stipulated by $\curvearrowright$. The answer is affirmative. Prior work on libraries and compilers for self-adjusting computation already shows how to implement change propagation with these costs using imperative traces, leaf-to-root traversals and in-place update of values [1, 10]. Since values are updated in-place, no cost is incurred for structural operations like pairing, projection, consing, etc; cost is incurred only for re-evaluating primitive functions on paths starting in updated leaves, exactly as in the judgment $\curvearrowright$. To double-check, we implemented most of our examples on an existing library, AFL [1], and observed exactly the costs stipulated by $\curvearrowright$. Due to lack of space, we omit the experimental results.

## 5 Soundness

We prove our type system sound in two ways: (a) Trace propagation is total and produces correct results on typed expressions, and (b) The cost of change propagation (determined by $\curvearrowright$) on a typed expression is no more than the cost $\kappa$ estimated in the expression's typing judgment. We combine these two statements together in the following theorem. This theorem considers an expression $e$ with one free variable $x$, which receives two potentially different substitutions (the two projections of a bi-value $\mathtt{w}$) in the original and modified execution. A more general theorem with any number of free variables (and, hence, any number of independent changes) holds as well, but we skip it here to improve readability.

**Theorem 1** (Type soundness). *Suppose that (a) $x : \tau \vdash e :_\kappa \tau'$; (b) $\vdash \mathtt{w} \gg \tau$; and (c) $e[L(\mathtt{w})/x] \Downarrow v', T$. 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) $e[R(\mathtt{w})/x] \Downarrow R(\mathtt{w}'), T'$; and (3) $c \leq \kappa$.*

In words, the theorem says that if expression $e$ types with dynamic stability $\kappa$ and we execute $e$ with an initial substitution $\mathrm{L}(\mathtt{w})/x$ to obtain a trace $T$, then we can successfully change propagate $T$ with a new substitution $\mathrm{R}(\mathtt{w})/x$ in $e$ (statement 1) to obtain the correct new output and trace (statement 2) with cost $c$ of change propagation no more than the statically estimated dynamic stability $\kappa$ (statement 3). Briefly, (1) states totality of change propagation for typed programs, (2) states its functional correctness, and (3) shows that our type system estimates dynamic stability conservatively. Note that this theorem models changes to expressions as different substitutions to the expression's free variable. Syntactic constants in $e$ cannot change, which explains why we can type constants with annotation $(\cdot)^{\mathbb{S}}$ in Figure 4.

$$\boxed{\llbracket\tau\rrbracket_v \subseteq \text{Step index} \times \text{Bi-values and } \llbracket\tau\rrbracket_\varepsilon^\kappa \subseteq \text{Step index} \times \text{Bi-expressions}}$$

$$\begin{array}{ll}
\llbracket(\tau)^{\mathbb{S}}\rrbracket_v & = \{(m, \mathtt{w}) \mid (m, \mathtt{w}) \in \llbracket\tau\rrbracket_v \wedge \mathtt{stable}(\mathtt{w})\} \\
\llbracket(\tau)^{\mathbb{C}}\rrbracket_v & = \llbracket\tau\rrbracket_v \\
\llbracket\mathtt{real}\rrbracket_v & = \{(m, \mathtt{keep}(\mathtt{r})) \mid \top\} \ \cup\ \{(m, \mathtt{repl}(\mathtt{r}, \mathtt{r}')) \mid \top\} \\
\llbracket\mathtt{list}\,[0]^\alpha\ \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)^{\mathbb{S}}\rrbracket_v \wedge (m, \mathtt{w}_2) \in \llbracket\mathtt{list}\,[n]^\alpha\ \tau\rrbracket_v) \\
& \qquad\qquad\qquad \vee ((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)\} \\
\llbracket\tau_1 \xrightarrow{\kappa} \tau_2\rrbracket_v & = \{(m, \mathtt{fix}\ f(x).\mathtt{ee}) \mid \\
& \qquad \forall j < n,\ \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_\varepsilon^\kappa\} \\
\llbracket\forall t \overset{\kappa}{::} S.\ \tau\rrbracket_v & = \{(m, \Lambda.\mathtt{ee}) \mid \forall I\ I :: S\ \ (m, \mathtt{ee}) \in \llbracket\tau[I/t]\rrbracket_\varepsilon^{\kappa[I/t]}\} \\
\llbracket\exists t.\ \tau\rrbracket_v & = \{(m, \mathtt{pack}\ \mathtt{w}) \mid \exists I. I :: S\ \wedge (m, \mathtt{w}) \in \llbracket\tau[I/t]\rrbracket_v\} \\
\llbracket\tau_1 \times \tau_2\rrbracket_v & = \{(m, (\mathtt{w}_1, \mathtt{w}_2)) \mid (m, \mathtt{w}_1) \in \llbracket\tau_1\rrbracket_v \wedge (m, \mathtt{w}_2) \in \llbracket\tau_2\rrbracket_v\} \\
& \\
\llbracket\tau\rrbracket_\varepsilon^\kappa & = \{(m, \mathtt{ee}) \mid \forall j < n.\ \mathrm{L}(\mathtt{ee}) \Downarrow v, T\ \wedge\ j = |T| \ \ \Rightarrow \exists\ v', T', \mathtt{ee}', c' : \\
& \qquad\qquad 1.\ \ \langle T, \mathtt{ee}\rangle \curvearrowright \mathtt{w}', T', c' \\
& \qquad\qquad 2.\ \ c' \leq \kappa \\
& \qquad\qquad 3.\ \ (m - j, \mathtt{w}') \in \llbracket\tau\rrbracket_v \\
& \qquad\qquad 4.\ \ \mathrm{R}(\mathtt{ee}) \Downarrow v', T' \\
& \qquad\qquad 5.\ \ v' = \mathrm{R}(\mathtt{w}')\ \wedge\ v = \mathrm{L}(\mathtt{w}')\} \\
& \\
\mathcal{D}\llbracket\cdot\rrbracket, \mathcal{G}\llbracket\cdot\rrbracket & = \{\emptyset\} \\
\mathcal{D}\llbracket\Delta, t :: S\rrbracket & = \{\sigma[t \mapsto I] \mid \sigma \in \mathcal{D}\llbracket\Delta\rrbracket\ \wedge\ I :: S\} \\
\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}$$

**Fig. 9.** Step-indexed interpretation of selected types

To prove this theorem, we build a *relational* model of types interpreted as sets of bi-values and bi-expressions. To handle recursive functions, we step-index our model [5]. The index counts trace size in our model. Trace size is proportional to the number of steps in complete reductions of small-step semantics. The size $|T|$ of a trace $T$ is defined as follows: Primitive constants and functions have size 0 and each trace constructor adds 1 to the size.

For every closed type $\tau$ we define a value interpretation $[\![\tau]\!]_v$ and an expression interpretation $[\![\tau]\!]_\varepsilon^\kappa$. The value interpretation $[\![\tau]\!]_v$ is a set of pairs of the form $(m, \mathtt{w})$, where $m$ is a step index. The expression interpretation $[\![\tau]\!]_\varepsilon^\kappa$ is a set of pairs of the form $(m, \mathbb{ee})$, where change propagating the trace of $\mathrm{L}(\mathbb{ee})$ with $\mathbb{ee}$ costs no more than $\kappa$ if the size of that trace is less than $m$. The two interpretations of types, shown in Figure 9, are defined simultaneously by induction on $\tau$. In the definition of the value interpretation of the list type $\mathtt{list}\,[n]^\alpha\ \tau$, we subinduct on $n$. Our definitions are unsurprising but we mention a few salient points. First, $[\![(\tau)^{\mathbb{C}}]\!]_v = [\![\tau]\!]_v$ and $[\![(\tau)^{\mathbb{S}}]\!]_v \subseteq [\![\tau]\!]_v$. Moreover, $(m, \mathtt{w}) \in [\![(\tau)^{\mathbb{S}}]\!]_v$ implies $\mathtt{stable}(\mathtt{w})$, as expected. The value interpretation of $\mathtt{list}\,[n{+}1]^\alpha\ \tau$ has two clauses corresponding to the two typing rules for `cons`. Most importantly, the expression interpretation $[\![\tau]\!]_\varepsilon^\kappa$ captures enough invariants about change propagation to enable us to prove the soundness theorem above. Figure 9 also shows the definitions of semantic substitutions $\sigma$ and $\theta$ for the contexts $\Delta$ and $\Gamma$, respectively. As usual, the substitution for each variable in $\Gamma$ must lie in the value interpretation of the variable's type.

We prove the following fundamental theorem for our type interpretations. The theorem consists of three statements for three different syntactic classes: expressions, bi-values and bi-expressions (in that order). The statement for expressions is established by an induction on expression typing, with a subinduction on step-indices for recursive functions. The other two statements follow by simultaneous induction on bi-value and bi-expression typing. The theorem relies on the assumption that the interpretation of every primitive function lies in the interpretation of the function's type. The formal statement of this assumption and the proof of the theorem are in our online appendix. Type soundness, Theorem 1, is an immediate corollary of the first two statements of this theorem.

**Theorem 2** (Fundamental Theorem)**.** *1. If $\Delta; \Phi; \Gamma \vdash e :_\kappa \tau$ and $\sigma \in \mathcal{D}[\![\Delta]\!]$ and $(m, \theta) \in \mathcal{G}[\![\sigma\Gamma]\!]$ and $\models \sigma\Phi$, then $(m, \theta\ulcorner e \urcorner) \in [\![\sigma\tau]\!]_\varepsilon^{\sigma\kappa}$.*

*2. If $\Delta; \Phi; \Gamma \vdash \mathtt{w} \gg \tau$ and $\sigma \in \mathcal{D}[\![\Delta]\!]$ and $(m, \theta) \in \mathcal{G}[\![\sigma\Gamma]\!]$ and $\models \sigma\Phi$, then $(m, \theta\mathtt{w}) \in [\![\sigma\tau]\!]_v$.*

*3. If $\Delta, \Phi, \Gamma \vdash \mathbb{ee} \gg_\kappa \tau$ and $\sigma \in \mathcal{D}[\![\Delta]\!]$ and $(m, \theta) \in \mathcal{G}[\![\sigma\Gamma]\!]$ and $\models \sigma\Phi$, then $(m, \theta(\mathbb{ee})) \in [\![\sigma\tau]\!]_\varepsilon^{\sigma\kappa}$.*

## 6 Related Work

*Incremental and self-adjusting computation.* Incremental computation has been studied extensively in the last three decades (reduction in the lambda calculus [15], graph algorithms [24], attribute grammars [14], programming languages [8] etc.). While most work focuses on efficient data-structures and memoization techniques for incremental computation, recent work develops type-directed techniques for automatic incrementalization of batch programs [10]. Ley-Wild *et al.* propose a cost semantics for program execution and bound the change propagation time of self-adjusting programs using a metric of trace distances [26]. Their analysis only yields that change propagation is no slower than from-scratch evaluation, asymptotically. Although they are able to prove tight bounds for some benchmark programs, this analysis requires comparing trace distances by hand for each change. Unlike our work, no existing approach provides a general, static technique for establishing tight asymptotic dynamic stability.

Chen *et al.* [11] use variation annotations similar to CostIt's, but do not address the problem of estimating dynamic stability. Instead, they focus on compiling a higher-order functional language to AFL, a language with change propagation semantics. Their translation is facilitated by types annotated $(\cdot)^{\mathbb{S}}$ and $(\cdot)^{\mathbb{C}}$, which CostIt uses for a different purpose. In turn, Chen *et al.* borrow these type annotations from Simonet and Pottier's work on type inference for information flow analysis [31].

In contrast to our co-monadic interpretation of $(\tau)^{\mathbb{S}}$ and identification of $(\tau)^{\mathbb{C}}$ with $\tau$, a significant amount of prior work on implementation of incremental programs equates $(\tau)^{\mathbb{S}}$ to $\tau$ and gives a monadic interpretation to the type $(\tau)^{\mathbb{C}}$ [1, 8, 11]. Although a deeper study of the connection between these two approaches is necessary, the choice so far seems to be motivated by the task at hand. For executing programs, it is natural to confine changes (and change propagation) to a monad, whereas for reasoning about dynamic stability it is often necessary to conclude by looking at an expression's inputs that the expression's result cannot change, which is easier in our co-monadic interpretation.

*Continuity and program sensitivity.* Also closely related to our work in concept, but not in the end-goal, is work on analysis of program continuity. There, the goal is to prove that the outputs of two runs of a program are closely related if the inputs are. Program continuity does not account for dynamic stability. Our type system also proves a limited form of program continuity, as an intermediate step in establishing dynamic stability. Reed and Pierce present a linear type system called Fuzz for proving continuity [33], as an intermediate step in verifying differential privacy properties. Gaboardi *et al.* extend Fuzz with lightweight dependent types in a type system called DFuzz [16]. DFuzz's syntax and use of lightweight dependent types influenced our work significantly. A technical difference from DFuzz (and Fuzz) is that our types capture where two values differ whereas in DFuzz, the "distance" between related values is not explicit in the type, but only in the relational model. As a result, our type system does not need linearity, which DFuzz does. Unlike CostIt and DFuzz, Chaudhuri *et al.*'s static analysis can prove program continuity even with control flow changes as long as perturbations to the input result in branches that are close to each other [9].

*Static computation of resource bounds/complexity analysis.* The programming languages community is rife with work on static computation of resource bounds, particularly worse-case execution time complexity, using different techniques such as abstract interpretation [18, 35], linear dependent types [13], amortized resource analysis [22] and sized types [12, 25, 36]. A common denominator of these techniques is that they all reason about a single execution of a program. In contrast, our focus — dynamic stability — is a two-trace property. It requires a relational model of execution which accounts for change propagation, as well as a relational model of types to track what parts of values can change across the executions, both of which we develop in this paper.

We mention some type-theoretic approaches to inferring and verifying resource usage bounds in programs. Dal Lago *et al.* present a complete time complexity analysis for PCF [13]. They use linear types to statically limit the number of times a function may be applied by the context. This allows reasoning about the time complexity of recursive functions precisely. We could adopt a similar approach in our work, although we have not found this necessary so far. Hoffmann *et al.* [23, 22] infer polynomial-shaped bounds on resource usage of RAML (Resource Aware ML) programs. A significant advantage of their technique is automation. A similar analysis for dynamic stability may be possible although the compatibility of logarithmic functions (which are necessary to state the dynamic stability of interesting programs) with Hoffmann *et al.*'s approach remains an open problem.

We use sized types [25] for lists. Sized types are often used in termination checking and analysis of heap and stack space [35]. Our types are precise on list lengths, unlike conventional uses where the size in the type is an upper-bound. For the number of allowed changes, our types specify upper-bounds.

## 7 Conclusion and Future Work

Existing work on incremental computation has been very successful at improving efficiency of incremental runs of a program, but does not consider the equally important question of developing static tools to analyze dynamic stability. Our work, CostIt, takes a first step in this direction by equipping a higher-order functional language with a type system to analyze dynamic stability of programs. We find that index refinements, immutability annotations, co-monadic reasoning and constraint-aware subtyping are useful in analyzing dynamic stability. Our type system is sound relative to a cost semantics for change propagation. We demonstrate the expressiveness and precision of CostIt on several examples.

Our ongoing work builds on the content of this paper in three ways. First, we are working on a prototype implementation of CostIt using bidirectional type-checking. We reduce type-checking and type inference to constraint satisfiability as in Dependent ML [37]. There is no new conceptual difficulty, but the constraint domain is largely intractable, as demonstrated by the occurrence of logarithmic and exponential functions in Examples 3 and 4. Consequently, we are exploring the possibility of using a combination of automatic and semiautomatic constraint solving (Dal Lago *et al.* use a similar approach in the context of worse-case execution time complexity analysis [13]).

Second, in work done after the review of this paper, we have extended CostIt's type system, relational model and soundness theorem to cover situations where program control flow may change with input changes. This is a nontrivial extension, beyond the scope of this paper. Briefly, we extend the type system with a standard worse-case execution time complexity analysis for branches which might execute from scratch during change propagation. The resulting type system is a significant refinement of the pure fragment of Pottier and Simonet's (simple) information flow type system for ML [31] (in contrast, the work in this paper corresponds to the special case where Pottier and Simonet's program counter or *pc* is always "low" or unchanging).

Finally, motivated by recent work on demand-driven incremental computation [20], we are planning to work on a version of CostIt for lazy evaluation semantics.

*Acknowledgments* The research of Umut Acar is partially supported by the European Research Council under grant number ERC-2012-StG-308246 and by the National Science Foundation under grant numbers CCF-1320563 and CCF-1408940.

## References

1. Acar, U., Blelloch, G., Blume, M., Harper, R., Tangwongsan, K.: A library for self-adjusting computation. Elec. Notes in Theor. Comp. Sci. 148(2), 127–154 (2006)
2. Acar, U.A., Blelloch, G.E., Blume, M., Harper, R., Tangwongsan, K.: An experimental analysis of self-adjusting computation. ACM Trans. Program. Lang. Syst. 32(1), 3:1–3:53 (2009)
3. Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. ACM Trans. Program. Lang. Syst. 28(6), 990–1034 (2006)
4. Acar, U.A., Blume, M., Donham, J.: A consistent semantics of self-adjusting computation. The Journal of Functional Programming (2013)
5. Ahmed, A.: Step-indexed syntactic logical relations for recursive and quantified types. In: Proceedings of the 15th European Conference on Programming Languages and Systems. pp. 69–83. ESOP'06, Springer-Verlag (2006)
6. Blanchet, B., Abadi, M., Fournet, C.: Automated verification of selected equivalences for security protocols. The Journal of Logic and Algebraic Programming 75(1), 3–51 (2008)
7. Brodal, G.S., Jacob, R.: Dynamic planar convex hull. In: Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science. pp. 617–626 (2002)
8. Carlsson, M.: Monads for incremental computing. In: Proceedings of the 7th International Conference on Functional Programming. pp. 26–35. ICFP '02, ACM (2002)
9. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity and robustness of programs. Communications of the ACM 55(8), 107–115 (2012)
10. Chen, Y., Dunfield, J., Acar, U.A.: Type-directed automatic incrementalization. In: Proceedings of the 33rd Conference on Programming Language Design and Implementation. pp. 299–310. PLDI '12, ACM (2012)

11. Chen, Y., Dunfield, J., Hammer, M.A., Acar, U.A.: Implicit self-adjusting computation for purely functional programs. In: International Conference on Functional Programming. pp. 129–141. ICFP '11 (2011)
12. Chin, W.N., Khoo, S.C.: Calculating sized types. In: Proceedings of the Workshop on Partial Evaluation and Semantics-based Program Manipulation. pp. 62–72. PEPM '00, ACM (1999)
13. Dal lago, U., Petit, B.: The geometry of types. In: Proceedings of the 40th Annual Symposium on Principles of Programming Languages. pp. 167–178. POPL '13, ACM (2013)
14. Demers, A., Reps, T., Teitelbaum, T.: Incremental evaluation for attribute grammars with application to syntax-directed editors. In: Proceedings of the 8th Symposium on Principles of Programming Languages. pp. 105–116. POPL '81, ACM (1981)
15. Field, J.: Incremental Reduction in the Lambda Calculus and Related Reduction Systems. Ph.D. thesis, Department of Computer Science, Cornell University (1991)
16. Gaboardi, M., Haeberlen, A., Hsu, J., Narayan, A., Pierce, B.C.: Linear dependent types for differential privacy. In: Proceedings of the 40th Annual Symposium on Principles of Programming Languages. pp. 357–370. POPL '13, ACM (2013)
17. Graham, R.L.: An efficient algorithm for determining the convex hull of a finite planar set. Information Processing Letters 1(4), 132–133 (1972)
18. Gulwani, S., Mehra, K.K., Chilimbi, T.: Speed: Precise and efficient static estimation of program computational complexity. In: Proceedings of the 36th Annual Symposium on Principles of Programming Languages. pp. 127–139. POPL '09, ACM (2009)
19. Hammer, M.A., Acar, U.A., Chen, Y.: Ceal: A C-based language for self-adjusting computation. In: Proceedings of the 2009 Conference on Programming Language Design and Implementation. pp. 25–37. PLDI '09, ACM (2009)
20. Hammer, M.A., Phang, K.Y., Hicks, M., Foster, J.S.: Adapton: Composable, demand-driven incremental computation. In: Proceedings of the 35th Conference on Programming Language Design and Implementation. pp. 156–166. PLDI '14, ACM (2014)
21. Heydon, A., Levin, R., Yu, Y.: Caching function calls using precise dependencies. In: Proceedings of the Conference on Programming Language Design and Implementation. pp. 311–320. PLDI '00, ACM (2000)
22. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: Proceedings of the 38th Annual Symposium on Principles of Programming Languages. pp. 357–370. POPL '11, ACM (2011)
23. Hoffmann, J., Hofmann, M.: Amortized resource analysis with polynomial potential: A static inference of polynomial bounds for functional programs. In: Proceedings of the 19th European Conference on Programming Languages and Systems. pp. 287–306. ESOP'10, Springer-Verlag (2010)
24. Holm, J., de Lichtenberg, K.: Top-trees and dynamic graph algorithms. Tech. Rep. DIKU-TR-98/17, Department of Computer Science, University of Copenhagen (1998)
25. Hughes, J., Pareto, L.: Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In: Proceedings of the Fourth International Conference on Functional Programming. pp. 70–81. ICFP '99, ACM (1999)
26. Ley-Wild, R., Acar, U.A., Fluet, M.: A cost semantics for self-adjusting computation. In: Proceedings of the 36th Annual Symposium on Principles of Programming Languages. pp. 186–199. POPL '09, ACM (2009)

27. Ley-Wild, R., Fluet, M., Acar, U.A.: Compiling self-adjusting programs with continuations. In: Proceedings of the 13th International Conference on Functional Programming. pp. 321–334. ICFP '08, ACM (2008)
28. Nanevski, A., Pfenning, F.: Staged computation with names and necessity. J. Funct. Program. 15(6), 893–939 (2005)
29. Nielson, F., Nielson, H.: Type and effect systems. In: Correct System Design, Lecture Notes in Computer Science, vol. 1710, pp. 114–136. Springer-Verlag (1999)
30. Overmars, M.H., van Leeuwen, J.: Maintenance of configurations in the plane. Journal of Computer and System Sciences 23, 166–204 (1981)
31. Pottier, F., Simonet, V.: Information flow inference for ML. ACM Trans. Program. Lang. Syst. 25(1), 117–158 (2003)
32. Pugh, W., Teitelbaum, T.: Incremental computation via function caching. In: Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages. pp. 315–328. POPL'89, ACM (1989)
33. Reed, J., Pierce, B.C.: Distance makes the types grow stronger: A calculus for differential privacy. In: Proceedings of the 15th International Conference on Functional Programming. pp. 157–168. ICFP '10, ACM (2010)
34. Shankar, A., Bodík, R.: Ditto: Automatic incrementalization of data structure invariant checks (in Java). In: Proceedings of the Conference on Programming Language Design and Implementation. pp. 310–319. PLDI '07, ACM (2007)
35. Vasconcelos, P.: Space cost analysis using sized types. Ph.D. thesis, School of Computer Science, University of St Andrews (2008)
36. Vasconcelos, P.B., Hammond, K.: Inferring cost equations for recursive, polymorphic and higher-order functional programs. In: Implementation of Functional Languages, 15th International Workshop, IFL 2003, Edinburgh, UK, September 8-11, 2003, Revised Papers. pp. 86–101 (2003)
37. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proceedings of the 26th Symposium on Principles of Programming Languages. pp. 214–227. POPL '99, ACM (1999)
38. Yellin, D., Strom, R.: Inc: A language for incremental computations. In: Proceedings of the Conference on Programming Language Design and Implementation. pp. 115–124. PLDI '88, ACM (1988)