POPL: G: Refinement Types for Incremental Computational Complexity

Ezgi Çiçek
PhD student at MPI-SWS, Saarbrücken, Germany
PhD advisor: Deepak Garg, MPI-SWS

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 incremental computational complexity, 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 changes to source code by recompiling as necessary, robots must interact with the physical world as it naturally changes over time, and scientific simulations compute with objects whose properties change over time.Rather than re-executing these programs each time there is a small input change, the field of incremental computation aims at building software that can respond automatically and efficiently to changing data by making use of the computation performed during the initial run to speed up the incremental run. Earlier work investigated ad-hoc techniques based on static dependency graphs [7] and memoization [12]. More recent work on self-adjusting computation improved upon this earlier work by introducing dynamic dependency graphs [1] and presenting a way to integrate them with a form of memoization [2] using language-based techniques. Using this technique, conventional programs can be automatically converted to their incremental counterparts which can handle incremental changes efficiently [2]. Several flavors of self-adjusting computation have been implemented in programming languages such as C [10], Haskell [3] and Standard ML [5].

However, in all prior work on incremental computation, the programmer must analyze the cost of incremental execution by direct analysis of the cost semantics of programs [16]. While this analytical technique makes efficient design possible, the behavior of the program under input changes is very difficult to establish because it requires reasoning about execution traces, which can be viewed as graphs of suspended computations and their (run-time) data and control dependencies.

Therefore, we are interested in designing static techniques to help a programmer reason about the update times of incremental programs. As a first step in this direction, we equip a higher-order polymorphic functional programming language with a refinement type system for establishing upper bounds on the update times of incremental programs. Our type system, called CostIt, soundly approximates the update times of a program. CostIt builds on index refinement types [19] and type annotations to track which program values may change after an update and which may not [5]. To enable precise analysis, we add subtyping rules motivated by co-monadic types [17]. Together, these give enough expressive power to perform non-trivial incremental complexity analysis of programs.

We provide an overview of CostIt's design, highlighting some challenges and our solutions. First, the update time of a program is a function of its input changes and, hence, incremental computational complexity analysis requires knowing which parts of a program's input (which of its free variables) may change. More generally, for a compositional analysis, we need to know for every subexpression in a program, whether or not the subexpression's value may change after an update. To differentiate changeable and unchangeable expressions, we rely on refinement type annotations from Chen *et al.*'s work on implicit self-adjusting computation [5]:[1] the refined type $(\tau)^{\mathbb{S}}$ ascribes values of type τ which cannot change, whereas the refined type $(\tau)^{\mathbb{C}}$ ascribes all values of type τ. Second, the update time 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 [19] or Gaboardi *et al.*'s DFuzz [8].

Centrally, our type system treats update time of an incremental program as an effect. Expression typing has the form $e :_\kappa \tau$, where κ 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 κ, we give the function a type of the form $\tau_1 \xrightarrow{\kappa} \tau_2$. The cost κ may depend on refinement parameters (e.g., list lengths) that are shared with τ_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 in CostIt:

$$(\tau_1 \xrightarrow{\kappa} \tau_2) \xrightarrow{0} \forall n, \alpha.\ \texttt{list}\,[n]^{\alpha}\ \tau_1 \xrightarrow{\alpha\cdot\kappa} \texttt{list}\,[n]^{\alpha}\ \tau_2$$

Roughly, the type says that if each application of the mapping function can be updated in time κ and at most α elements of the mapped list change, then the entire map can be updated in time $\alpha \cdot \kappa$ (since the mapping of the unchanged $(n - \alpha)$ elements can be re-used from the previous run).

Change propagation has the inherent property that if the inputs of 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 the update times of a program. A key insight in CostIt is that this property cor-

[1] Nearly identical annotations are also used for other purposes, e.g. information flow analysis [18].

responds to a *co-monadic* reasoning principle in the type system: If all free variables of an expression have types of the form $(\cdot)^{\mathbb{S}}$ (i.e. they don't change), then that expression's change propagation cost 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 $\Box\tau$. 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 deriving upper bounds on the update times of 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 incremental computational complexity derived in our type system is correct (i.e., that our type system is sound) relative to an abstract *cost semantics for trace update*. The cost semantics is formalized using a novel syntactic class called *bi-values*, a pair of values, one representing the original input value and the other representing the modified input value. We interpret types as sets of bi-values (i.e., relationally) with a stipulated change propagation cost and prove our type system sound using step-indexing.

In summary, we make the following contributions.

- We develop the first type system for establishing bounds on the update times of incremental programs.
- We combine lightweight dependent types, immutability annotations and co-monadic reasoning principles to facilitate static proofs of incremental computational complexity bounds.
- We prove the type system sound relative to a new cost semantics for change propagation.
- We demonstrate the precision and generality of our technique on several examples.

2. Main Ideas

Incremental computational complexity 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'. Incremental computational complexity 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. 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 update of this program would be 1 unit of time. Abstractly, incremental computational complexity 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 update cost 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 an update cost of 10, not 1.

Change propagation We assume a simple, standard change propagation semantics which consists of two parts. 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). During the second execution, 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 incremental computational complexity. CostIt helps establish this incremental computational complexity statically.

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, 5]. 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.

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 the update time in such cases requires also an analysis of worst-case execution time complexity. For the sake of simplicity, in this paper, we disallow (through our type system) control flow dependence on data that may change. In the conclusion, we comment on an extension that can handle control flow changes.

Type system overview We build on a λ-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 change propagation cost 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* α elements are allowed to change before the second execution. Technically, n and α are natural numbers in an index domain, over which types may quantify. Second, any type τ may be refined to $(\tau)^{\mu}$ where μ belongs to an index sort with two values, $\mathbb{S}$ and $\mathbb{C}$. $(\tau)^{\mathbb{S}}$ specifies those values of type τ that will not change in the second execution ($\mathbb{S}$ is read "stable"). $(\tau)^{\mathbb{C}}$ specifies all values of type τ ($\mathbb{C}$ is read "potentially changeable"). τ and $(\tau)^{\mathbb{C}}$ are subtypes of each other. Our typing judgment takes the form $\Gamma \vdash e :_{\kappa} \tau$. Here, κ is an upper bound on the update time of the expression e. (For simplicity, we omit several contexts from the typing judgment in this section.)

A few selected typing rules are shown in Figure 1. First, if an expression contains subexpressions, then the change propagation costs (κ'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). Third, variables represent values, so they have zero change propagation cost (rule **var**). In the next subsection, we explain the rule **shortcut** with an example.

The rule **primApp**, although complex looking, merely says that the application of a primitive function (ζ) incurs a change propagation cost specified by the semantics of the function (the context Υ specifies the change propagation cost for all primitive functions).

$\boxed{\Delta;\Phi;\Gamma \vdash e :_\kappa \tau}$ expression e has type τ and incremental computational complexity at most κ

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

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

Figure 1: Selected typing rules. The context Υ carrying types of primitive functions is omitted from all rules.

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 two ways: $x : (\mathtt{real})^{\mathbb{S}} \vdash x+1 :_0 (\mathtt{real})^{\mathbb{S}}$ and $x : (\mathtt{real})^{\mathbb{C}} \vdash x+1 :_1 (\mathtt{real})^{\mathbb{C}}$. When $x : (\mathtt{real})^{\mathbb{S}}$, x cannot change. So change propagation bypasses the expression $x+1$ without re-execution and since the result is re-used and its cost is 0. Moreover, the value of $x+1$ does not change. This justifies the first typing judgment. When $x : (\mathtt{real})^{\mathbb{C}}$, 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 (Balanced list fold) The CostIt type $\tau_1 \xrightarrow{\kappa} \tau_2$ specifies a function whose body has a change propagation cost upper-bounded by κ 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 $(\mathtt{real})^{\mathbb{S}} \xrightarrow{0} (\mathtt{real})^{\mathbb{S}}$ and $(\mathtt{real})^{\mathbb{C}} \xrightarrow{1} (\mathtt{real})^{\mathbb{C}}$.

Standard list fold operations (`foldl` and `foldr` which iterate over a list from left-to-right or right-to-left, computing partial intermediate results) can be typed easily in CostIt. However, these functions are uninteresting for incremental computation. The reason is that each partial result computed by either of these functions has a dependency on all earlier partial results and, hence, the incremental computational complexity is $O(n)$ even for single element changes to the input list (we use n to denote the list's length).

A more interesting operation is what we call the balanced fold. Given an *associative and commutative* binary function f of type $\tau \times \tau \xrightarrow{\kappa} \tau$, a list of type $\mathtt{list}\,[n]^\alpha\ \tau$ (at most α of the list's n elements might change) 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 $\mathtt{log}_2(n)$. For simplicity, we assume that the list size is power of two, even though our analysis can handle lists of arbitrary sizes.

A single change to the list causes $\mathtt{log}_2(n)$ recomputations of f. So, if f has change propagation cost κ, the total incremental computational complexity under single change to the list is $O(\kappa \cdot \mathtt{log}_2(n))$. More generally, it can be shown that if α changes are allowed to the list, then the incremental computational complexity is $O(\kappa \cdot (\alpha + \alpha \cdot \mathtt{log}_2(n/\alpha)))$. This simplifies to $O(\kappa \cdot n)$ when $\alpha = n$ (entire list may change) and $O(\kappa \cdot \mathtt{log}_2(n))$ when $\alpha = 1$. In the following we implement such a balanced fold operation, `bfold`, and derive a bound on its update time in CostIt.

Our first ingredient is the function `split`, which splits a list of length n into two lists of lengths $\frac{n}{2}$ each. 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 α into the two split lists. The incremental computational complexity of `split` is 0 because `split` uses no primitive functions (*cf.* discussion earlier in this section).

$$\mathtt{split} : \forall n.\ \forall \alpha.\ \mathtt{list}\,[n]^\alpha\ \tau \xrightarrow{0} \exists \beta.\ (\mathtt{list}\,\left[\tfrac{n}{2}\right]^\beta\ \tau \times \mathtt{list}\,\left[\tfrac{n}{2}\right]^{\alpha-\beta}\ \tau)$$

```
fix split(l). case_L l of
  nil → (nil, nil)
| (h1 :: tl1) → case_L tl1 of
                   nil → ([h1], nil)
                 | (h2 :: tl2) → let (z1, z2) = split tl2 in
                                 ((h1 :: z1), (h2 :: z2))
```

Using `split` 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). λl. case_L l of
  nil → ...
| (h1 :: tl1) → case_L tl1 of
                   nil → h1
                 | (_ :: _) → let (z1, z2) = (split l) in
                              f (bfold f z1, bfold f z2)
```

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 $\mathtt{list}\,[n]^\alpha\ \tau$. 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 $H = \mathtt{log}_2(n)$. Counting levels from the root downward (the root has level 0 and the leaves have level H), the number of applications of f at level k in the trace is at most 2^k. If α leaves change, at most α of these applications must be recomputed. Consequently, the maximum number of recomputations of f at level k is $\mathtt{min}(\alpha, 2^k)$. If the incremental computational complexity of f is κ, the incremental computational complexity of `bfold` is $P(n,\alpha,\kappa) = \sum_{k=0}^{\mathtt{log}_2(n)} \kappa \cdot \mathtt{min}(\alpha, 2^k)$. So, in principle, we should be able to give `bfold` the following type.

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

The expression $P(n,\alpha,\kappa)$ may look complex, but it is in $O(\kappa \cdot (\alpha + \alpha \cdot \mathtt{log}_2(n/\alpha)))$. (To prove this, split the summation in $P(n,\alpha,\kappa)$ into two: one for $k \le \mathtt{log}_2(n) - \alpha$ and the other for $k > \mathtt{log}_2(n) - \mathtt{log}_2(\alpha)$). 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 $(\cdot)^{\mathbb{S}}$ since it is a closed function. Hence, the actual type we assign to `bfold` is stronger.

$$\mathtt{bfold} : ((\tau \times \tau \xrightarrow{\kappa} \tau)^{\mathbb{S}} \to \forall n > 0.\ \forall \alpha.\ \mathtt{list}\,[n]^\alpha\ \tau \xrightarrow{P(n,\alpha,\kappa)} \tau)^{\mathbb{S}} \tag{1}$$

We explain how `bfold`'s type is established in CostIt. The interesting case starts where `split` is invoked. From the type of `split`, we know that variables z_1 and z_2 in the body of `bfold` have types $\mathtt{list}\,\left[\frac{n}{2}\right]^\beta\ \tau$ and $\mathtt{list}\,\left[\frac{n}{2}\right]^{\alpha-\beta}\ \tau$, respectively for some β. Inductively, the change propagation costs of $(\mathtt{bfold}\ f\ z_1)$ and $(\mathtt{bfold}\ f\ z_2)$ are $P(\frac{n}{2},\beta,\kappa)$ and $P(\frac{n}{2},\alpha-\beta,\kappa)$, respectively. Hence, the change propagation cost of the whole body of `bfold` is $\kappa + P(\frac{n}{2},\beta,\kappa) + P(\frac{n}{2},\alpha-\beta,\kappa)$. The additional κ accounts for the only application of f in the body of `bfold` (non-primitive operations have zero cost and `split` also has zero cost). Hence, to

Examples	From-scratch complexity	Incremental comp. complexity
List map	$O(n)$	$O(\alpha)$ or $O(n)$
List foldl or foldr	$O(n)$	$O(n)$
Mergesort	$O(n \log n)$	$O(n(1+\log \alpha))$
List append	$O(n)$	$O(1)$
Matrix multiply	$O(n^3)$	$O(n \log n)$ (single change)
Matrix transpose	$O(n^2)$	$O(1)$

Table 1: From-scratch (initial run) complexity and incremental computational complexity of example programs

complete the typing, we must establish the following inequality.

$$\kappa + P(\frac{n}{2}, \beta, \kappa) + P(\frac{n}{2}, \alpha - \beta, \kappa) \leq P(n, \alpha, \kappa) \qquad (2)$$

This is an easily established arithmetic tautology, *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 case $\alpha > 0$ succeeds as described above. For $\alpha \doteq 0$, we use our co-monadic reasoning principle: if all free variables of an expression have types of the form $(\cdot)^{\mathbb{S}}$ (i.e. they don't change), then that expression's change propagation cost is 0. This principle is embodied by rule **shortcut** in Figure 1. With $\alpha \doteq 0$, the types of z_1 and z_2 are equivalent (formally, via subtyping) to $\texttt{list}\left[\frac{n}{2}\right]^0\ \tau$ and $\texttt{list}\left[\frac{n}{2}\right]^0\ \tau$, respectively. Since, no elements in these lists can change, by subtyping, we can promote the types to $(\texttt{list}\left[\frac{n}{2}\right]^0\ \tau)^{\mathbb{S}}$ and $(\texttt{list}\left[\frac{n}{2}\right]^0\ \tau)^{\mathbb{S}}$, respectively. At this point, the type of every variable occurring in the expression $f\ (\texttt{bfold}\ f\ z_1, \texttt{bfold}\ f\ z_2)$, 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.

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

Other examples Table 1 contains several other examples that can be typed in CostIt with their from-scratch and incremental computational complexities. The first example (list map) has linear from-scratch complexity and two possible incremental computational complexities ($O(\alpha)$ when only the input list may change but the mapping function cannot change, and $O(n)$ when the mapping function itself may change). The analysis of balanced fold above generalizes to arbitrary divide-and-conquer algorithms. One such algorithm, mergesort, is listed in the table. Our method also works when the inputs and outputs contain nested lists, as for example in matrix operations (for readability, Table 1 lists the incremental complexity of matrix multiplication with a single input change only).

We note that the incremental computational complexities proved using CostIt is asymptotically tight for all the examples in Table 1. Nonetheless, like other type systems, CostIt abstracts over concrete program values and, hence, we cannot expect CostIt's analysis to be asymptotically tight on all programs.

Soundness We prove that our type system is sound, i.e., the incremental complexity of a program estimated by our type system is an upper-bound on the actual change propagation cost of the program. We formalize the "actual change propagation cost of a program" in the form of a novel *cost-counting change propagation semantics*. In the following we describe these semantics and the soundness theorem very briefly and in greatly simplified form.[2]

Consider a program e with one free variable (input) x. Suppose that e has already been executed with some substitution for x, resulting in trace T. We wish to recompute e with a new value v' for the input x. We capture this recomputation with the judgment $\langle T, e[v'/x] \rangle \curvearrowright v'', T', c'$, which means that change propagation through T results in the new output v'', the new trace T' and, most importantly, incurs cost c'.

Our soundness theorem is listed below. In particular, clause (3) of the theorem says that if our type system establishes the incremental complexity κ for a program e, under some assumptions about changes to the program's input x (embodied in the annotations on the input's type τ), then as long as changes to the input respect the assumptions, the actual change propagation cost c' is upper-bounded by κ.

Theorem 1 (Type soundness)**.** *Suppose:*

$x : \tau \vdash e :_{\kappa} \tau'$
$\vdash (v_1, v_2) \gg \tau$
$e[v_1/x] \Downarrow v_1', T$

Then, there exist T', c and v_2' such that the following hold:

1. $\langle T, e[v_2/x] \rangle \curvearrowright v_2', T', c'$;
2. $e[v_2/x] \Downarrow v_2', T'$; *and*
3. $c' \leq \kappa$.

The proof of the theorem is reasonably involved, and is based on an asymmetric relational model with step-indexing.

3. Related Work

Incremental and self-adjusting computation. Incremental computation has been studied extensively in the last three decades (graph algorithms [14], attribute grammars [7], programming languages [1] 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 [5]. Unlike our work, no existing approach provides a general, static technique for establishing tight incremental computational complexities.

Perhaps the closest attempt for bounding the execution time of change propagation is the work of Ley-Wild *et al.* that proposes cost semantics for program execution using a metric of trace distances [16]. Although the analysis proves tight bounds for some benchmark programs, it requires comparing trace distances by hand for each change. In general, their analysis only yields that change propagation is asymptotically no slower than from-scratch evaluation.

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 incremental computational complexity. Our type system also proves a limited form of program continuity, as an intermediate step in establishing bounds on the update times.

[2] Complete details can be found in our full paper [4].

Gaboardi *et al.* present a linear lightweight dependent type system called DFuzz for proving continuity [8], as an intermediate step in verifying differential privacy properties. DFuzz's syntax and use of lightweight dependent types influenced our work significantly. A technical difference from DFuzz 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.

Static computation of resource bounds/complexity analysis. The programming languages community is rife with work on static computation of resource bounds, particularly asymptotic execution time complexity, using different techniques such as abstract interpretation [9], linear dependent types [6], amortized resource analysis [13] and sized types [15]. A common denominator of these techniques is that they all reason about a single execution of a program. In contrast, our focus — incremental computational complexity — 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.

Hoffmann *et al.* [13] 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 incremental computational complexity may be possible although the compatibility of logarithmic functions (which are necessary to state the incremental computational complexity of interesting programs) with Hoffmann *et al.*'s approach remains an open problem.

4. 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 bounds on the update times of incremental programs. Our work, CostIt, takes a first step in this direction by equipping a higher-order functional language with a type system to analyze incremental computational complexity of programs. We find that index refinements, immutability annotations, co-monadic reasoning and constraint-aware subtyping are useful in analyzing incremental computational complexity. Our type system is sound relative to a cost semantics for change propagation. We demonstrate the expressiveness and precision of CostIt on several examples.

We have also extended the analysis to cover situations where program control flow may change with input changes. This is a nontrivial extension, beyond what has been described here. Briefly, we extend the type system with a standard worst-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 [18] .

Our ongoing work builds on the content of this paper in two 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 [19]. We are also exploring the possibility of automation to the extent possible inspired by [13]. Second, motivated by recent work on demand-driven incremental computation [11], we are planning to work on a version of CostIt for lazy evaluation semantics.

References

[1] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, 2006.

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

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

[4] E. Çiçek, D. Garg, and U. Acar. Refinement types for incremental computational complexity. In J. Vitek, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 406–431. Springer Berlin Heidelberg, 2015.

[5] Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar. Implicit self-adjusting computation for purely functional programs. In *International Conference on Functional Programming*, ICFP '11, pages 129–141, 2011.

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

[7] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the 8th Symposium on Principles of Programming Languages*, POPL '81, pages 105–116. ACM, 1981.

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

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

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

[11] M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of the 35th Conference on Programming Language Design and Implementation*, PLDI '14, pages 156–166. ACM, 2014. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594324.

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

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

[14] J. Holm and K. de Lichtenberg. Top-trees and dynamic graph algorithms. Technical Report DIKU-TR-98/17, Department of Computer Science, University of Copenhagen, 1998.

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

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

[17] A. Nanevski and F. Pfenning. Staged computation with names and necessity. *J. Funct. Program.*, 15(6):893–939, 2005.

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

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