Incremental λ-Calculus in Cache-Transfer Style

Static Memoization by Program Transformation

Paolo G. Giarrusso[1(✉)], Yann Régis-Gianas[2], and Philipp Schuster[3]

[1] LAMP—EPFL, Lausanne, Switzerland
[2] IRIF, University of Paris Diderot, Inria, Paris, France
[3] University of Tübingen, Tübingen, Germany

Abstract. Incremental computation requires propagating changes and reusing intermediate results of base computations. Derivatives, as produced by static differentiation [7], propagate changes but do not reuse intermediate results, leading to wasteful recomputation. As a solution, we introduce conversion to *Cache-Transfer-Style*, an additional program transformations producing purely incremental functional programs that create and maintain nested tuples of intermediate results. To prove CTS conversion correct, we extend the correctness proof of static differentiation from STLC to untyped λ-calculus via *step-indexed logical relations*, and prove sound the additional transformation via simulation theorems.

To show ILC-based languages can improve performance relative to from-scratch recomputation, and that CTS conversion can extend its applicability, we perform an initial performance case study. We provide derivatives of primitives for operations on collections and incrementalize selected example programs using those primitives, confirming expected asymptotic speedups.

1 Introduction

After computing a base output from some base input, we often need to produce updated outputs corresponding to updated inputs. Instead of rerunning the same *base program* on the updated input, incremental computation transforms the input change to an output change, potentially reducing asymptotic time complexity and significantly improving efficiency, especially for computations running on large data sets.

Incremental λ-Calculus (ILC) [7] is a recent framework for *higher-order* incremental computation. ILC represents changes from a base value v_1 to an updated value v_2 as a first-class *change value dv*. Since functions are first-class values, change values include *function changes*.

ILC also statically transforms *base programs* to *incremental programs* or *derivatives*, that are functions mapping input changes to output changes. Incremental language designers can then provide their language with (higher-order) primitives (with their derivatives) that efficiently encapsulate incrementalizable

© The Author(s) 2019
L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 553–580, 2019.
https://doi.org/10.1007/978-3-030-17184-1_20

computation skeletons (such as tree-shaped folds), and ILC will incrementalize higher-order programs written in terms of these primitives.

Alas, ILC only incrementalizes efficiently *self-maintainable computations* [7, Sect. 4.3], that is, computations whose output changes can be computed using only input changes, but not the inputs themselves [11]. Few computations are self-maintainable: for instance, mapping self-maintainable functions on a sequence is self-maintainable, but dividing numbers is not! We elaborate on this problem in Sect. 2.1. In this paper, we extend ILC to non-self-maintainable computations. To this end, we must enable derivatives to reuse intermediate results created by the base computation.

Many incrementalization approaches remember intermediate results through dynamic memoization: they typically use hashtables to memoize function results, or dynamic dependence graphs [1] to remember a computation trace. However, looking up intermediate results in such dynamic data structure has a runtime cost that is hard to optimize; and reasoning on dynamic dependence graphs and computation traces is often complex. Instead, ILC produces purely functional programs, suitable for further optimizations and equational reasoning.

To that end, we replace dynamic memoization with *static memoization*: following Liu and Teitelbaum [20], we transform programs to *cache-transfer style (CTS)*. A CTS function outputs their primary result along with *caches* of intermediate results. These caches are just nested tuples whose structure is derived from code, and accessing them does not involve looking up keys depending on inputs. Instead, intermediate results can be fetched from these tuples using statically known locations. To integrate CTS with ILC, we extend differentiation to produce *CTS derivatives*: these can extract from caches any intermediate results they need, and produce updated caches for the next computation step.

The correctness proof of static differentiation in CTS is challenging. First, we must show a forward simulation relation between two triples of reduction traces (the first triple being made of the source base evaluation, the source updated evaluation and the source derivative evaluation; the second triple being made of the corresponding CTS-translated evaluations). Dealing with six distinct evaluation environments at the same time was error prone on paper and for this reason, we conducted the proof using Coq [26]. Second, the simulation relation must not only track values but also caches, which are only partially updated while in the middle of the evaluation of derivatives. Finally, we study the translation for an untyped λ-calculus, while previous ILC correctness proofs were restricted to simply-typed λ-calculus. Hence, we define which changes are valid via a *logical relation* and show its *fundamental property*. Being in an untyped setting, our logical relation is not indexed by types, but *step-indexed*. We study an untyped language, but our work also applies to the erasure of typed languages. Formalizing a type-preserving translation is left for future work because giving a type to CTS programs is challenging, as we shall explain.

In addition to the correctness proof, we present preliminary experimental results from three case studies. We obtain efficient incremental programs even on non self-maintainable functions.

We present our contributions as follows. First, we summarize ILC and illustrate the need to extend it to remember intermediate results via CTS (Sect. 2). Second, in our mechanized formalization (Sect. 3), we give a novel proof of correctness for ILC differentiation for untyped λ-calculus, based on step-indexed logical relations (Sect. 3.4). Third, building on top of ILC differentiation, we show how to transform untyped higher-order programs to CTS (Sect. 3.5) and we show that CTS functions and derivatives *simulate* correctly their non-CTS counterparts (Sect. 3.7). Finally, in our case studies (Sect. 4), we compare the performance of the generated code to the base programs. Section 4.4 discusses limitations and future work. Section 5 discusses related work and Sect. 6 concludes. Our mechanized proof in Coq, the case study material, and the extended version of this paper with appendixes are available online at https://github.com/yurug/cts.

2 ILC and CTS Primer

In this section we exemplify ILC by applying it on an average function, show why the resulting incremental program is asymptotically inefficient, and use CTS conversion and differentiation to incrementalize our example efficiently and speed it up asymptotically (as confirmed by benchmarks in Sect. 4.1). Further examples in Sect. 4 apply CTS to higher-order programs and suggest that CTS enables incrementalizing efficiently some core database primitives such as joins.

2.1 Incrementalizing *average* via ILC

Our example computes the average of a bag of numbers. After computing the *base output* y_1 of the average function on the *base input* bag xs_1, we want to update the output in response to a stream of updates to the input bag. Here and throughout the paper, we contrast *base* vs *updated* inputs, outputs, values, computations, and so on. For simplicity, we assume we have two *updated inputs* xs_2 and xs_3 and want to compute two *updated outputs* y_2 and y_3. We express this program in Haskell as follows:

$$\begin{aligned}
&average && :: Bag\ \mathbb{Z} \rightarrow \mathbb{Z} \\
&average\ xs && = \textbf{let}\ s = sum\ xs; n = length\ xs; r = div\ s\ n\ \textbf{in}\ r \\
&average_3 && = \textbf{let}\ y_1 = average\ xs_1; y_2 = average\ xs_2; y_3 = average\ xs_3 \\
& && \quad\ \textbf{in}\ (y_1, y_2, y_3)
\end{aligned}$$

To compute the updated outputs y_2 and y_3 in $average_3$ faster, we try using ILC. For that, we assume that we receive not only updated inputs xs_2 and xs_3 but also *input change* dxs_1 from xs_1 to xs_2 and input change dxs_2 from xs_2 to xs_3. A change dx from x_1 to x_2 describes the changes from base value x_1 to updated value x_2, so that x_2 can be computed via the *update operator* $\oplus$ as $x_1 \oplus dx$. A nil change $\mathbf{0}_x$ is a change from base value x to updated value x itself.

ILC differentiation automatically transforms the *average* function to its derivative $daverage :: Bag\ \mathbb{Z} \to \Delta(Bag\ \mathbb{Z}) \to \Delta\mathbb{Z}$. A derivative maps input changes to output changes: here, $dy_1 = daverage\ xs_1\ dxs_1$ is a change from base output $y_1 = average\ xs_1$ to updated output $y_2 = average\ xs_2$, hence $y_2 = y_1 \oplus dy_1$.

Thanks to *daverage*'s correctness, we can rewrite $average_3$ to avoid expensive calls to *average* on updated inputs and use *daverage* instead:

$$
\begin{array}{l}
incrementalAverage_3 :: (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}) \\
incrementalAverage_3 = \\
\quad \mathbf{let}\ y_1 = average\ xs_1; dy_1 = daverage\ xs_1\ dxs_1 \\
\qquad\ \ y_2 = y_1 \oplus dy_1; dy_2 = daverage\ xs_2\ dxs_2 \\
\qquad\ \ y_3 = y_2 \oplus dy_2 \\
\quad \mathbf{in}\ \ (y_1, y_2, y_3)
\end{array}
$$

In general, also the value of a function $f :: A \to B$ can change from a base value f_1 to an updated value f_2, mainly when f is a closure over changing data. In that case, the change from base output $f_1\ x_1$ to updated output $f_2\ x_2$ is given by $df\ x_1\ dx$, where $df :: A \to \Delta A \to \Delta B$ is now a *function change* from f_1 to f_2. Above, *average* exemplifies the special case where $f_1 = f_2 = f$: then the function change df is a nil change, and $df\ x_1\ dx$ is a change from $f_1\ x_1 = f\ x_1$ and $f_2\ x_2 = f\ x_2$. That is, a nil function change for f is a derivative of f.

2.2 Self-maintainability and Efficiency of Derivatives

Alas, derivatives are efficient only if they are *self-maintainable*, and *daverage* is not, so $incrementalAverage_3$ is no faster than $average_3$! Consider the result of differentiating *average*:

$$
\begin{array}{l}
daverage :: Bag\ \mathbb{Z} \to \Delta(Bag\ \mathbb{Z}) \to \Delta\mathbb{Z} \\
daverage\ xs\ dxs = \mathbf{let}\ s = sum\ xs; ds = dsum\ xs\ dxs; \\
\qquad\qquad\qquad\qquad\quad n = length\ xs; dn = dlength\ xs\ dxs; \\
\qquad\qquad\qquad\qquad\quad r = div\ s\ n; dr = ddiv\ s\ ds\ n\ dn \\
\qquad\qquad\qquad\quad\ \ \mathbf{in}\ dr
\end{array}
$$

Just like *average* combines *sum*, *length*, and *div*, its derivative *daverage* combines those functions and their derivatives. *daverage* recomputes base intermediate results s, n and r exactly as done in *average*, because they might be needed as base inputs of derivatives. Since r is unused, its recomputation can be dropped during later optimizations, but expensive intermediate results s and n are used by *ddiv*:

$$
\begin{array}{l}
ddiv :: \mathbb{Z} \to \Delta\mathbb{Z} \to \mathbb{Z} \to \Delta\mathbb{Z} \to \Delta\mathbb{Z} \\
ddiv\ a\ da\ b\ db = div\ (a \oplus da)\ (b \oplus db) - div\ a\ b
\end{array}
$$

Function *ddiv* computes the difference between the updated and the original result, so it needs its base inputs a and b. Hence, *daverage* must recompute s and n and will be slower than *average*!

Typically, ILC derivatives are only efficient if they are *self-maintainable*: a self-maintainable derivative does not inspect its base inputs, but only its change inputs, so recomputation of its base inputs can be elided. Cai et al. [7] leave efficient support for non-self-maintainable derivatives for future work.

But this problem is fixable: executing *daverage xs dxs* will compute exactly the same s and n as executing *average xs*, so to avoid recomputation we must simply save s and n and reuse them. Hence, we CTS-convert each function f to a *CTS function fC* and a *CTS derivative dfC*: CTS function *fC* produces, together with its final result, a *cache* containing intermediate results, that the caller must pass to CTS derivative *dfC*.

CTS-converting our example produces the following code, which requires no wasteful recomputation.

```
type AverageC = (ℤ, SumC, ℤ, LengthC, ℤ, DivC)
averageC :: Bag ℤ → (ℤ, AverageC)
averageC xs =
  let (s, cs₁) = sumC xs; (n, cn₁) = lengthC xs; (r, cr₁) = divC s n
  in  (r, (s, cs₁, n, cn₁, r, cr₁))
daverageC :: Bag ℤ → Δ(Bag ℤ) → AverageC → (Δℤ, AverageC)
daverageC xs dxs (s, cs₁, n, cn₁, r, cr₁) =
  let (ds, cs₂)  = dsumC xs dxs cs₁
      (dn, cn₂) = dlengthC xs dxs cn₁
      (dr, cr₂)  = ddivC s ds n dn cr₁
  in  (dr, ((s ⊕ ds), cs₂, (n ⊕ dn), cn₂, (r ⊕ dr), cr₂))
```

For each function f, we introduce a type FC for its cache, such that a CTS function fC has type $A \rightarrow (B, FC)$ and CTS derivative dfC has type $A \rightarrow \Delta A \rightarrow FC \rightarrow (\Delta B, FC)$. Crucially, CTS derivatives like *daverageC* must return an updated cache to ensure correct incrementalization, so that application of further changes works correctly. In general, if $(y_1, c_1) = fC\ x_1$ and $(dy, c_2) = dfC\ x_1\ dx\ c_1$, then $(y_1 \oplus dy, c_2)$ must equal the result of the base function fC applied to the updated input $x_1 \oplus dx$, that is $(y_1 \oplus dy, c_2) = fC\ (x_1 \oplus dx)$.

For CTS-converted functions, the cache type FC is a tuple of intermediate results and caches of subcalls. For primitive functions like *div*, the cache type *DivC* could contain information needed for efficient computation of output changes. In the case of *div*, no additional information is needed. The definition of *divC* uses *div* and produces an empty cache, and the definition of *ddivC* follows the earlier definition for *ddiv*, except that we now pass along an empty cache.

```
data DivC = DivC
divC :: ℤ → ℤ → (ℤ, DivC)
divC a b = (div a b, DivC)
ddivC :: ℤ → Δℤ → ℤ → Δℤ → DivC → (Δℤ, DivC)
ddivC a da b db DivC = (div (a ⊕ da) (b ⊕ db) − div a b, DivC)
```

Finally, we can rewrite $average_3$ to incrementally compute y_2 and y_3:

$$
\begin{aligned}
&\mathit{ctsIncrementalAverage}_3 :: (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}) \\
&\mathit{ctsIncrementalAverage}_3 = \\
&\quad \mathbf{let}\ (y_1, c_1) = \mathit{averageC}\ \mathit{xs}_1; (dy_1, c_2) = \mathit{daverageC}\ \mathit{xs}_1\ \mathit{dxs}_1\ c_1 \\
&\qquad\quad y_2 = y_1 \oplus dy_1; (dy_2, c_3) = \mathit{daverageC}\ \mathit{xs}_2\ \mathit{dxs}_2\ c_2 \\
&\qquad\quad y_3 = y_2 \oplus dy_2 \\
&\quad \mathbf{in}\ \ (y_1, y_2, y_3)
\end{aligned}
$$

Since functions of the same type translate to CTS functions of different types, in a higher-order language CTS translation is not always type-preserving; however, this is not a problem for our case studies (Sect. 4); Sect. 4.1 shows how to map such functions, and we return to this problem in Sect. 4.4.

3 Formalization

We now formalize CTS-differentiation for an untyped Turing-complete λ-calculus, and formally prove it sound with respect to differentiation. We also give a novel proof of correctness for differentiation itself, since we cannot simply adapt Cai et al. [7]'s proof to the new syntax: Our language is untyped and Turing-complete, while Cai et al. [7]'s proof assumed a strongly normalizing simply-typed λ-calculus and relied on its naive set-theoretic denotational semantics. Our entire formalization is mechanized using Coq [26]. For reasons of space, some details are deferred to the appendix.

		Terms			***Closed values***
a_t ::=	$\mathbf{let}\, a_p = a_t\, \mathbf{in}\, a_t$	*Let*	a_v ::=	$a_E[\lambda \overline{a_p}.\, a_t]$	*Closure*
|	a_T	*Tuple*	|	$(\overline{a_v})$	*Tuple*
|	$f\, \overline{x}$	*Application*	|	ℓ	*Literal*
		Nested tuples	|	$\mathbf{p}$	*Primitive*
a_T ::=	x	*Variable*	|	$0_{\mathbf{p}}$	*Nil change for primitive*
|	$x \oplus dx$	*Update*	|	$!a_v$	*Replacement change*
|	$(\overline{a_T})$	*Tuple*			***Value environments***
		Patterns	a_E ::=	$\bullet$	*Empty*
a_p ::=	x	*Variable*	|	$a_E; x = a_v$	*Value binding*
|	$(\overline{a_p})$	*Tuple*	j, k, n ∈	$\mathbb{N}$	***Step indexes***

Fig. 1. Our language λ_L of lambda-lifted programs. Tuples can be nullary.

Transformations. We introduce and prove sound three term transformations, namely differentiation, CTS translation and CTS differentiation, that take a function to its corresponding (non-CTS) derivative, CTS function and CTS derivative. Each CTS function produces a base output and a cache from a base input, while each CTS derivative produces an output change and an updated cache from an input, an input change and a base cache.

Proof technique. To show soundness, we prove that CTS functions and derivatives simulate respectively non-CTS functions and derivatives. In turn, we formalize (non-CTS) differentiation as well, and we prove differentiation sound with respect to non-incremental evaluation. Overall, this shows that CTS functions and derivatives are sound relatively to non-incremental evaluation. Our presentation proceeds in the converse order: first, we present differentiation, formulated as a variant of Cai et al. [7]'s definition; then, we study CTS differentiation.

By using logical relations, we simplify significantly the setup of Cai et al. [7]. To handle an untyped language, we employ *step-indexed* logical relations. Besides, we conduct our development with big-step operational semantics because that choice simplifies the correctness proof for CTS conversion. Using big-step semantics for a Turing complete language restricts us to terminating computations. But that is not a problem: to show incrementalization is correct, we need only consider computations that terminate on both old and new inputs, following Acar et al. [3] (compared with in Sect. 5).

Structure of the formalization. Section 3.1 introduces the syntax of the language λ_L we consider in this development, and introduces its four sublanguages λ_{AL}, λ_{IAL}, λ_{CAL} and λ_{ICAL}. Section 3.2 presents the syntax and the semantics of λ_{AL}, the source language for our transformations. Section 3.3 defines differentiation and its target language λ_{IAL}, and Sect. 3.4 proves differentiation correct. Section 3.5 defines CTS conversion, comprising CTS translation and CTS differentiation, and their target languages λ_{CAL} and λ_{ICAL}. Section 3.6 presents the semantics of λ_{CAL}. Finally, Sect. 3.7 proves CTS conversion correct.

Notations. We write $\overline{X}$ for a sequence of X of some unspecified length $X_1, \ldots, X_m$.

3.1 Syntax for $\boldsymbol{\lambda_L}$

A superlanguage. To simplify our transformations, we require input programs to have been lambda-lifted [15] and converted to A'-normal form (A'NF). Lambda-lifted programs are convenient because they allow us to avoid a specific treatment for free variables in transformations. A'NF is a minor variant of ANF [24], where every result is bound to a variable before use; unlike ANF, we also bind the result of the tail call. Thus, every result can thus be stored in a cache by CTS conversion and reused later (as described in Sect. 2). This requirement is not onerous: A'NF is a minimal variant of ANF, and lambda-lifting and ANF conversion are routine in compilers for functional languages. Most examples we show are in this form.

In contrast, our transformation's outputs are lambda-lifted but not in A'NF. For instance, we restrict base functions to take exactly one argument—a base input. As shown in Sect. 2.1, CTS functions take instead two arguments—a base input and a cache—and CTS derivatives take three arguments—an input, an input change, and a cache. We could normalize transformation outputs to inhabit the source language and follow the same invariants, but this would complicate our proofs for little benefit. Hence, we do not *prescribe* transformation outputs

to satisfy the same invariants, and we rather *describe* transformation outputs through separate grammars.

As a result of this design choice, we consider languages for base programs, derivatives, CTS programs and CTS derivatives. In our Coq mechanization, we formalize those as four separate languages, saving us many proof steps to check the validity of required structural invariants. For simplicity, in this paper we define a single language called λ_L (for λ-Lifted). This language satisfies invariants common to all these languages (including some of the A'NF invariants). Then, we define *sublanguages* of λ_L. We describe the semantics of λ_L informally, and we only formalize the semantics of its sublanguages.

Syntax for terms. The λ_L language is a relatively conventional lambda-lifted λ-calculus with a limited form of pattern matching on tuples. The syntax for terms and values is presented in Fig. 1. We separate terms and values in two distinct syntactic classes because we use big-step operational semantics. Our **let**-bindings are non-recursive as usual, and support shadowing. Terms cannot contain λ-expressions directly, but only refer to closures through the environment, and similarly for literals and primitives; we elaborate on this in Sect. 3.2. We do not introduce case expressions, but only bindings that destructure tuples, both in **let**-bindings and λ-expressions of closures. Our semantics does not assign meaning to match failures, but pattern-matchings are only used in generated programs and our correctness proofs ensure that the matches always succeed. We allow tuples to contain terms of form $x \oplus dx$, which update base values x with changes in dx, because A'NF-converting these updates is not necessary to the transformations. We often inspect the result of a function call "$f\ x$", which is not a valid term in our syntax. Hence, we write "$@(f, x)$" as a syntactic sugar for "$\textbf{let}\ y = f\ x\ \textbf{in}\ y$" with y chosen fresh.

Syntax for closed values. A closed value is either a closure, a tuple of values, a literal, a primitive, a nil change for a primitive or a replacement change. A closure is a pair of an evaluation environment E and a λ-abstraction closed with respect to E. The set of available literals ℓ is left abstract. It may contain usual first-order literals like integers. We also leave abstract the primitives $\mathbf{p}$ like `if-then-else` or projections of tuple components. Each primitive $\mathbf{p}$ comes with a nil change, which is its derivative as explained in Sect. 2. A change value can also represent a replacement by some closed value a_v. Replacement changes are not produced by static differentiation but are useful for clients of derivatives: we include them in the formalization to make sure that they are not incompatible with our system. As usual, environments E map variables to closed values.

Sublanguages of λ_L. The source language for all our transformations is a sublanguage of λ_L named λ_{AL}, where A stands for A'NF. To each transformation we associate a target language, which matches the transformation image. The target language for CTS conversion is named λ_{CAL}, where "C" stands for CTS. The target languages of differentiation and CTS differentiation are called, respectively, λ_{IAL} and λ_{ICAL}, where the "I" stands for incremental.

3.2 The Source Language $\boldsymbol{\lambda_{AL}}$

We show the syntax of λ_{AL} in Fig. 2. As said above, λ_{AL} is a sublanguage of λ_L denoting lambda-lifted base terms in A'NF. With no loss of generality, we assume that all bound variables in λ_{AL} programs and closures are distinct. The step-indexed big-step semantics (Fig. 3) for base terms is defined by the judgment written $E \vdash t \Downarrow_n v$ (where n can be omitted) and pronounced "Under environment E, base term t evaluates to closed value v in n steps." Intuitively, our step-indexes count the number of "nodes" of a big-step derivation.[1] As they are relatively standard, we defer the explanations of these rules to Appendix B.

Term differentiation $\boxed{dt = \mathcal{D}^\iota(t)}$

$$\mathcal{D}^\iota(x) = dx$$
$$\mathcal{D}^\iota(\textbf{let } y = f\ x \textbf{ in } t) = \textbf{let } y = f\ x, dy = df\ x\ dx \textbf{ in } \mathcal{D}^\iota(t)$$
$$\mathcal{D}^\iota(\textbf{let } y = (\overline{x}) \textbf{ in } t) = \textbf{let } y = (\overline{x}), dy = (\overline{dx}) \textbf{ in } \mathcal{D}^\iota(t)$$

Value differentiation $\boxed{dv = \mathcal{D}^\iota(v)}$

$$\mathcal{D}^\iota((\overline{v})) = (\overline{\mathcal{D}^\iota(v)})$$
$$\mathcal{D}^\iota(E_f[\lambda x.\, t]) = \mathcal{D}^\iota(E_f)[\lambda x\ dx.\, \mathcal{D}^\iota(t)]$$
$$\mathcal{D}^\iota(\ell) = \textbf{nil}\, \ell$$
$$\mathcal{D}^\iota(\mathbf{p}) = 0_{\mathbf{p}}$$

Environment differentiation $\boxed{dE = \mathcal{D}^\iota(E)}$

$$\mathcal{D}^\iota(\bullet) = \bullet$$
$$\mathcal{D}^\iota(E; x = v) = \mathcal{D}^\iota(E); x = v; dx = \mathcal{D}^\iota(v)$$

Base/updated environment $\boxed{E = \lfloor dE \rfloor_i}$

$$\lfloor \bullet \rfloor_i = \bullet \qquad \text{i} = 1, 2$$
$$\lfloor dE; x = v; dx = dv \rfloor_i = \lfloor dE \rfloor_i; x = v'$$
$$v' = v \text{ if } i = 1 \text{ or}$$
$$v' = v \oplus dv \text{ if } i = 2$$

λ_{IAL} **change terms**

$$dt ::= dx \mid \textbf{let } y = f\ x, dy = df\ x\ dx \textbf{ in } dt \mid \textbf{let } y = (\overline{x}), dy = (\overline{dx}) \textbf{ in } dt$$

λ_{IAL} **change values**

$$dv ::= (\overline{dv}) \mid dE[\lambda x\ dx.\, dt] \mid d\ell \mid 0_{\mathbf{p}} \mid !v$$

λ_{IAL} **change environments**

$$dE ::= \bullet \mid dE; x = v; dx = dv$$

λ_{AL} **base terms**

$$t ::= x \mid \textbf{let } y = f\ x \textbf{ in } t \mid \textbf{let } y = (\overline{x}) \textbf{ in } t$$

λ_{AL} **closed values**

$$v ::= (\overline{v}) \mid E[\lambda x.\, t] \mid \ell \mid \mathbf{p}$$

λ_{AL} **value environments**

$$E ::= \bullet \mid E; x = v$$

Fig. 2. Static differentiation $\mathcal{D}^\iota(-)$; syntax of its target language λ_{IAL}, tailored to the output of differentiation; syntax of its source language λ_{AL}. We assume that in λ_{IAL} the same **let** binds both y and dy and that α-renaming preserves this invariant. We also define the *base environment* $\lfloor dE \rfloor_1$ and the *updated environment* $\lfloor dE \rfloor_2$ of a change environment dE.

Expressiveness. A closure in the base environment can be used to represent a top-level definition. Since environment entries can point to primitives, we need no syntax to directly represent calls of primitives in the syntax of base terms. To encode in our syntax a program with top-level definitions and a term to be evaluated representing the entry point, one can produce a term t representing the

[1] It is more common to count instead small-step evaluation steps [3,4], but our choice simplifies some proofs and makes a minor difference in others.

$$\frac{}{E \vdash x \Downarrow_1 E(x)}\,[\text{SVar}] \qquad \frac{E; y = (E(\overline{x})) \vdash t \Downarrow_n v}{E \vdash \mathbf{let}\, y = (\overline{x})\, \mathbf{in}\, t \Downarrow_{n+1} v}\,[\text{STuple}] \qquad \frac{E(f) = \mathbf{p} \quad E; y = \delta_{\mathbf{p}}(E(x)) \vdash t \Downarrow_n v}{E \vdash \mathbf{let}\, y = f\, x\, \mathbf{in}\, t \Downarrow_{n+1} v}\,[\text{SPrimitiveCall}]$$

$$\frac{E(f) = E_f[\lambda x.\, t_f] \qquad E_f; x = E(x) \vdash t_f \Downarrow_m v_y \qquad E; y = v_y \vdash t \Downarrow_n v}{E \vdash \mathbf{let}\, y = f\, x\, \mathbf{in}\, t \Downarrow_{m+n+1} v}\,[\text{SClosureCall}]$$

Fig. 3. Step-indexed big-step semantics for base terms of source language λ_{AL}.

entry point together with an environment E containing as values any top-level definitions, primitives and literals used in the program. Semi-formally, given an environment E_0 mentioning needed primitives and literals, and a list of top-level function definitions $D = \overline{f = \lambda x.\, t}$ defined in terms of E_0, we can produce a base environment $E = \mathcal{L}(D)$, with $\mathcal{L}$ defined by:

$$\mathcal{L}(\bullet) = E_0 \text{ and } \mathcal{L}(D, f = \lambda x.\, t) = E, f = E[\lambda x.\, t] \text{ where } \mathcal{L}(D) = E$$

Correspondingly, we extend all our term transformations to values and environments to transform such encoded top-level definitions.

Our mechanization can encode n-ary functions "$\lambda(x_1, x_2, \ldots, x_n).\, t$" through unary functions that accept tuples; we encode partial application using a **curry** primitive such that, essentially, $\mathbf{curry}\, f\, x\, y = f\,(x, y)$; suspended partial applications are represented as closures. This encoding does not support currying efficiently, we further discuss this limitation in Sect. 4.4.

Control operators, like recursion combinators or branching, can be introduced as primitive operations as well. If the branching condition changes, expressing the output change in general requires replacement changes. Similarly to branching we can add tagged unions.

To check the assertions of the last two paragraphs, the Coq development contains the definition of a **curry** primitive as well as a primitive for a fixpoint combinator, allowing general recursion and recursive data structures as well.

3.3 Static Differentiation from $\boldsymbol{\lambda_{AL}}$ to $\boldsymbol{\lambda_{IAL}}$

Previous work [7] defines static differentiation for simply-typed λ-calculus terms. Figure 2 transposes differentiation as a transformation from λ_{AL} to λ_{IAL} and defines λ_{IAL}'s syntax.

Differentiating a base term t produces a change term $\mathcal{D}^{\iota}(t)$, its *derivative*. Differentiating final result variable x produces its change variable dx. Differentiation copies each binding of an intermediate result y to the output and adds a new binding for its change dy. If y is bound to tuple $(\overline{x})$, then dy will be bound to the change tuple $(\overline{dx})$. If y is bound to function application "$f\ x$", then dy will be bound to the application of function change df to input x and its change dx. We explain differentiation of environments $\mathcal{D}^{\iota}(E)$ later in this section.

[SDVar]
$$\frac{}{dE \vdash dx \Downarrow_1 dE(dx)}$$

[SDTuple]
$$\frac{dE(\overline{x}, \overline{dx}) = \overline{v}_x, \overline{dv}_x \qquad dE; y = (\overline{v}_x); dy = (\overline{dv}_x) \vdash dt \Downarrow_n dv}{dE \vdash \mathbf{let}\ y = (\overline{x}), dy = (\overline{dx})\ \mathbf{in}\ dt \Downarrow_{n+1} dv}$$

[SDReplaceCall]
$$\frac{\lfloor dE \rfloor_1 \vdash @(f, x) \Downarrow_m v_y \qquad \lfloor dE \rfloor_2 \vdash @(f, x) \Downarrow_n v_y{}' \qquad dE(df) = !v_f \qquad dE; y = v_y; dy = !v_y{}' \vdash dt \Downarrow_p dv}{dE \vdash \mathbf{let}\ y = f\ x, dy = df\ x\ dx\ \mathbf{in}\ dt \Downarrow_{m+n+p+1} dv}$$

[SDPrimitiveNil]
$$\frac{dE(f, df) = \mathbf{p}, 0_{\mathbf{p}} \qquad dE(x, dx) = v_x, dv_x \qquad dE; y = \delta_{\mathbf{p}}(v_x); dy = \Delta_{\mathbf{p}}(v_x, dv_x) \vdash dt \Downarrow_n dv}{dE \vdash \mathbf{let}\ y = f\ x, dy = df\ x\ dx\ \mathbf{in}\ dt \Downarrow_{n+1} dv}$$

[SDClosureChange]
$$\frac{\begin{array}{c} dE(f, df) = E_f[\lambda x.\, t_f], dE_f[\lambda x\ dx.\, dt_f] \\ dE(x, dx) = v_x, dv_x \qquad E_f; x = v_x \vdash t_f \Downarrow_m v_y \\ dE_f; x = v_x; dx = dv_x \vdash dt_f \Downarrow_n dv_y \qquad dE; y = v_y; dy = dv_y \vdash dt \Downarrow_p dv \end{array}}{dE \vdash \mathbf{let}\ y = f\ x, dy = df\ x\ dx\ \mathbf{in}\ dt \Downarrow_{m+n+p+1} dv}$$

Fig. 4. Step-indexed big-step semantics for the change terms of λ_{IAL}.

Evaluating $\mathcal{D}^{\iota}(t)$ recomputes all intermediate results computed by t. This recomputation will be avoided through cache-transfer style in Sect. 3.5. A comparison with the original static differentiation [7] can be found in Appendix A.

Semantics for λ_{IAL}. We move on to define how λ_{IAL} change terms evaluate to change values. We start by defining necessary definitions and operations on changes, such as define *change values* dv, *change environments* dE, and the *update operator* $\oplus$.

Closed change values dv are particular λ_L values a_v. They are either a closure change, a tuple change, a literal change, a replacement change or a primitive nil change. A closure change is a closure containing a change environment dE and a λ-abstraction expecting a value and a change value as arguments to evaluate a change term into an output change value. An evaluation environment dE follows the same structure as **let**-bindings of change terms: it binds variables to closed values and each variable x is immediately followed by a binding for its associated change variable dx. As with **let**-bindings of change terms, α-renamings in an environment dE must rename dx into dy if x is renamed into y. We define the *update operator* $\oplus$ to update a value with a change. This operator is a partial function written "$v \oplus dv$", defined as follows:

$$\begin{aligned}
v_1 \oplus\, !v_2 &= v_2 \\
\ell \oplus d\ell &= \delta_\oplus(\ell, d\ell) \\
E[\lambda x.\, t] \oplus dE[\lambda x\ dx.\, dt] &= (E \oplus dE)[\lambda x.\, t] \\
(v_1, \ldots, v_n) \oplus (dv_1, \ldots, dv_n) &= (v_1 \oplus dv_1, \ldots, v_n \oplus dv_n) \\
\mathbf{p} \oplus 0_{\mathbf{p}} &= \mathbf{p}
\end{aligned}$$

where $(E; x = v) \oplus (dE; x = v; dx = dv) = ((E \oplus dE); x = (v \oplus dv))$.

Replacement changes can be used to update all values (literals, tuples, primitives and closures), while tuple changes can only update tuples, literal changes can only update literals, primitive nil can only update primitives and closure changes can only update closures. A replacement change overrides the current value v with a new one v'. On literals, $\oplus$ is defined via some interpretation function $\delta_\oplus$, which takes a literal and a literal change to produce an updated literal. Change update for a closure ignores dt instead of computing something like $dE[t \oplus dt]$. This may seem surprising, but we only need $\oplus$ to behave well for valid changes (as shown by Theorem 3.1): for valid closure changes, dt must behave anyway similarly to $\mathcal{D}^\iota(t)$, which Cai et al. [7] show to be a nil change. Hence, $t \oplus \mathcal{D}^\iota(t)$ and $t \oplus dt$ both behave like t, so $\oplus$ can ignore dt and only consider environment updates. This definition also avoids having to modify terms at runtime, which would be difficult to implement safely. We could also implement $f \oplus df$ as a function that invokes both f and df on its argument, as done by Cai et al. [7], but we believe that would be less efficient when $\oplus$ is used at runtime. As we discuss in Sect. 3.4, we restrict validity to avoid this runtime overhead.

Having given these definitions, we show in Fig. 4 a step-indexed big-step semantics for change terms, defined through judgment $dE \vdash dt \Downarrow_n dv$ (where n can be omitted). This judgment is pronounced "Under the environment dE, the change term dt evaluates into the closed change value dv in n steps." Rules [SDVar] and [SDTuple] are unsurprising. To evaluate function calls in **let**-bindings "$\mathbf{let}\ y = f\ x, dy = df\ x\ dx\ \mathbf{in}\ dt$" we have three rules, depending on the shape of $dE(df)$. These rules all recompute the value v_y of y in the original environment, but compute differently the change dy to y. If $dE(df)$ replaces the value of f, [SDReplaceCall] recomputes $v'_y = f\ x$ from scratch in the new environment, and bind dy to $!v'_y$ when evaluating the **let** body. If $dE(df)$ is the nil change for primitive $\mathbf{p}$, [SDPrimitiveNil] computes dy by running $\mathbf{p}$'s derivative through function $\Delta_{\mathbf{p}}(-)$. If $dE(df)$ is a closure change, [SDClosureChange] invokes it normally to compute its change dv_y. As we show, if the closure change is valid, its body behaves like f's derivative, hence incrementalizes f correctly.

Closure changes with non-nil environment changes represent partial application of derivatives to non-nil changes; for instance, if f takes a pair and dx is a non-nil change, $0_{\mathbf{curry}}\ f\ df\ x\ dx$ constructs a closure change containing dx, using the derivative of **curry** mentioned in Sect. 3.2. In general, such closure changes do not arise from the rules we show, only from derivatives of primitives.

3.4 A New Soundness Proof for Static Differentiation

In this section, we show that static differentiation is sound (Theorem 3.3) and that Eq. (1) holds:

$$f\ a_2 = f\ a_1 \oplus \mathcal{D}^{\iota}(f)\ a_1\ da \tag{1}$$

whenever da is a valid change from a_1 to a_2 (as defined later). One might want to prove this equation assuming only that $a_1 \oplus da = a_2$, but this is false in general. A direct proof by induction on terms fails in the case for application (ultimately because $f_1 \oplus df = f_2$ and $a_1 \oplus da = a_2$ do not imply that $f_1\ a_1 \oplus df\ a_1\ da = f_2\ a_2$). As usual, this can be fixed by introducing a logical relation. We call ours *validity*: a function change is valid if it turns valid input changes into valid output changes.

- $d\ell \rhd_n \ell \hookrightarrow \delta_{\oplus}(\ell, d\ell)$ • $!v_2 \rhd_n v_1 \hookrightarrow v_2$ • $0_{\mathbf{p}} \rhd_n \mathbf{p} \hookrightarrow \mathbf{p}$
- $(dv_1, \ldots, dv_m) \rhd_n (v_1, \ldots, v_m) \hookrightarrow (v'_1, \ldots, v'_m)$
  if and only if $(v_1, \ldots, v_m) \oplus (dv_1, \ldots, dv_m) = (v'_1, \ldots, v'_m)$
  and $\forall k < n,\ \forall i \in [1 \ldots m],\ dv_i \rhd_k v_i \hookrightarrow v'_i$
- $dE[\lambda x\ dx.\ dt] \rhd_n E_1[\lambda x.\ t] \hookrightarrow E_2[\lambda x.\ t]$
  if and only if $E_2 = E_1 \oplus dE$ and
  $\forall k < n, v_1, dv, v_2,$
  if $dv \rhd_k v_1 \hookrightarrow v_2$ then
  $(dE; x = v_1; dx = dv \vdash dt) \blacktriangleright_k (E_1; x = v_1 \vdash t) \hookrightarrow (E_2; x = v_2 \vdash t)$
- $(dE \vdash dt) \blacktriangleright_n (E_1 \vdash t_1) \hookrightarrow (E_2 \vdash t_2)$
  if and only if $\forall k < n, v_1, v_2,$
  $E_1 \vdash t_1 \Downarrow_k v_1$ and $E_2 \vdash t_2 \Downarrow v_2$ implies that
  $\exists dv, dE \vdash dt \Downarrow dv \wedge dv \rhd_{n-k} v_1 \hookrightarrow v_2$

Fig. 5. Step-indexed validity, through judgments for values and for terms.

Static differentiation is only sound on input changes that are *valid*. Cai et al. [7] show soundness for a strongly normalizing simply-typed λ-calculus using denotational semantics. Using an operational semantics, we generalize this result to an untyped and Turing-complete language, so we must turn to a *step-indexed* logical relation [3, 4].

Validity as a step-indexed logical relation. We say that "dv is a valid change from v_1 to v_2, up to k steps" and write

$$dv \rhd_k v_1 \hookrightarrow v_2$$

to mean that dv is a change from v_1 to v_2 and that dv is a *valid* description of the differences between v_1 and v_2, with validity tested with up to k steps. This relation *approximates* validity; if a change dv is valid at all approximations, it is simply valid (between v_1 and v_2); we write then $dv \rhd v_1 \hookrightarrow v_2$ (omitting the step-index k) to mean that validity holds at all step-indexes. We similarly omit step-indexes k from other step-indexed relations when they hold for all k.

To justify this intuition of validity, we show that a valid change from v_1 to v_2 goes indeed from v_1 to v_2 (Theorem 3.1), and that if a change is valid up to k steps, it is also valid up to fewer steps (Lemma 3.2).

Theorem 3.1 ($\oplus$ agrees with validity)
If $dv \rhd_k v_1 \hookrightarrow v_2$ holds for all $k > 0$, then $v_1 \oplus dv = v_2$.

Lemma 3.2 (Downward-closure)
If $N \geq n$, then $dv \rhd_N v_1 \hookrightarrow v_2$ implies $dv \rhd_n v_1 \hookrightarrow v_2$.

Crucially, Theorem 3.1 enables (a) computing v_2 from a valid change and its source, and (b) showing Eq. (1) through validity. As discussed, $\oplus$ ignores changes to closure bodies to be faster, which is only sound if those changes are nil; to ensure Theorem 3.1 still holds, validity on closure changes must be adapted accordingly and forbid non-nil changes to closure bodies. This choice, while unusual, does not affect our results: if input changes do not modify closure bodies, intermediate changes will not modify closure bodies either. Logical relation experts might regard this as a domain-specific invariant we add to our relation. Alternatives are discussed by Giarrusso [10, Appendix C].

As usual with step-indexing, validity is defined by well-founded induction over naturals ordered by $<$; to show well-foundedness we observe that evaluation always takes at least one step.

Validity for values, terms and environments is formally defined by cases in Fig. 5. First, a literal change $d\ell$ is a valid change from ℓ to $\ell \oplus d\ell = \delta_{\oplus}(\ell, d\ell)$. Since the function $\delta_{\oplus}$ is partial, the relation only holds for the literal changes $d\ell$ which are valid changes for ℓ. Second, a replacement change $!v_2$ is always a valid change from any value v_1 to v_2. Third, a primitive nil change is a valid change between any primitive and itself. Fourth, a tuple change is valid up to step n, if each of its components is valid up to any step strictly less than n. Fifth, we define validity for closure changes. Roughly speaking, this statement means that a closure change is valid if (i) its environment change dE is valid for the original closure environment E_1 and for the new closure environment E_2; and (ii) when applied to related values, the closure *bodies* t are related by dt, as defined by the auxiliary judgment $(dE \vdash dt) \blacktriangleright_n (E_1 \vdash t_1) \hookrightarrow (E_2 \vdash t_2)$ for validity between terms under related environments (defined in Appendix C). As usual with step-indexed logical relations, in the definition for this judgment about terms, the number k of steps required to evaluate the term t_1 is subtracted from the number of steps n that can be used to relate the outcomes of the term evaluations.

Soundness of differentiation. We can state a soundness theorem for differentiation without mentioning step-indexes; thanks to this theorem, we can compute the updated result v_2 not by rerunning a computation, but by updating the base result v_1 with the result change dv that we compute through a derivative on the input change. A corollary shows Eq. (1).

Theorem 3.3 (Soundness of differentiation in λ_{AL}). *If dE is a valid change environment from base environment E_1 to updated environment E_2, that is $dE \rhd E_1 \hookrightarrow E_2$, and if t converges both in the base and updated environment, that is $E_1 \vdash t \Downarrow v_1$ and $E_2 \vdash t \Downarrow v_2$, then $\mathcal{D}^\iota(t)$ evaluates under the change environment dE to a valid change dv between base result v_1 and updated result v_2, that is $dE \vdash \mathcal{D}^\iota(t) \Downarrow dv$, $dv \rhd v_1 \hookrightarrow v_2$ and $v_1 \oplus dv = v_2$.*

We must first show that derivatives map input changes valid up to k steps to output changes valid up to k steps, that is, the *fundamental property* of our step-indexed logical relation:

Lemma 3.4 (Fundamental Property)
For each n, if $dE \rhd_n E_1 \hookrightarrow E_2$ then $(dE \vdash \mathcal{D}^\iota(t)) \blacktriangleright_n (E_1 \vdash t) \hookrightarrow (E_2 \vdash t)$.

Translation of terms $\boxed{M = \mathcal{T}_t(t')}$

$$\begin{aligned}
\mathcal{T}_t(\mathbf{let}\, y = f\, x\, \mathbf{in}\, t') &= \mathbf{let}\, y, c^y_{fx} = f\, x\, \mathbf{in}\, \mathcal{T}_t(t') \\
\mathcal{T}_t(\mathbf{let}\, y = (\overline{x})\, \mathbf{in}\, t') &= \mathbf{let}\, y = (\overline{x})\, \mathbf{in}\, \mathcal{T}_t(t') \\
\mathcal{T}_t(x) &= (x, \mathcal{C}(t))
\end{aligned}$$

Cache of a term $\boxed{C = \mathcal{C}(t)}$

$$\begin{aligned}
\mathcal{C}(\mathbf{let}\, y = f\, x\, \mathbf{in}\, t) &= ((\mathcal{C}(t), y), c^y_{fx}) \\
\mathcal{C}(\mathbf{let}\, y = (\overline{x})\, \mathbf{in}\, t) &= (\mathcal{C}(t), y) \\
\mathcal{C}(x) &= ()
\end{aligned}$$

Translation of values $\boxed{V = \mathcal{T}(v)}$

$$\begin{aligned}
\mathcal{T}((\overline{v})) &= (\overline{\mathcal{T}(v)}) \\
\mathcal{T}(E[\lambda x.\, t]) &= \mathcal{T}(E)[\lambda x.\, \mathcal{T}_t(t)] \\
\mathcal{T}(\ell) &= \ell \\
\mathcal{T}(\mathbf{p}) &= \mathbf{p}
\end{aligned}$$

Base terms

$$\begin{aligned}
M ::=\ & \mathbf{let}\, y, c^y_{fx} = f\, x\, \mathbf{in}\, M \\
\mid\ & \mathbf{let}\, y = (\overline{x})\, \mathbf{in}\, M \\
\mid\ & (x, C)
\end{aligned}$$

Cache terms/patterns

$$C ::= (C, c^y_{fx}) \mid (C, x) \mid ()$$

Closed values

$$V ::= (\overline{V}) \mid F[\lambda x.\, M] \mid \ell \mid \mathbf{p}$$

Cache values

$$V_c ::= () \mid (V_c, V_c) \mid (V_c, V)$$

Evaluation environments

$$F ::= \bullet \mid F; D_v$$

Base environment entries

$$D_v ::= x = V \mid c^y_{fx} = V_c$$

Fig. 6. Cache-Transfer Style translation and syntax of its target language λ_{CAL}.

3.5 CTS Conversion

Figures 6 and 7 define both the syntax of λ_{CAL} and λ_{ICAL} and CTS conversion. The latter comprises CTS differentiation $\mathcal{D}(-)$, from λ_{AL} to λ_{ICAL}, and CTS translation $\mathcal{T}(-)$, from λ_{AL} to λ_{CAL}.

Syntax definitions for the target languages λ_{CAL} *and* λ_{ICAL}. Terms of λ_{CAL} follow again λ-lifted A'NF, like λ_{AL}, except that a **let**-binding for a function application "$f\ x$" now binds an extra *cache identifier* c^y_{fx} besides output y. Cache identifiers have non-standard syntax: it can be seen as a triple that refers to the value identifiers f, x and y. Hence, an α-renaming of one of these three identifiers must refresh the cache identifier accordingly. Result terms explicitly

return cache C through syntax (x, C). Caches are encoded through nested tuples, but they are in fact a tree-like data structure that is isomorphic to an execution trace. This trace contains both immediate values and the execution traces of nested function calls.

The syntax for λ_{ICAL} matches the image of the CTS derivative and witnesses the CTS discipline followed by the derivatives: to determine dy, the derivative of f evaluated at point x with change dx expects the cache produced by evaluating y in the base term. The derivative returns the updated cache which contains the intermediate results that would be gathered by the evaluation of $f\ (x \oplus dx)$. The result term of every change term returns the computed change and a cache update dC, where each value identifier x of the input cache is updated with its corresponding change dx.

Differentiation of terms $\boxed{dM = \mathcal{D}_t(t')}$

$$\begin{aligned}
\mathcal{D}_t(\mathbf{let}\ y = f\ x\ \mathbf{in}\ t') &= \mathbf{let}\ dy, c^y_{fx} = df\ x\ dx\ c^y_{fx}\ \mathbf{in} \\
&\quad\ \mathcal{D}_t(t') \\
\mathcal{D}_t(\mathbf{let}\ y = (\overline{x})\ \mathbf{in}\ t') &= \mathbf{let}\ dy = (\overline{dx})\ \mathbf{in}\ \mathcal{D}_t(M') \\
\mathcal{D}_t(x) &= (dx, \mathcal{U}(t))
\end{aligned}$$

Cache update of a term $\boxed{dC = \mathcal{U}(t)}$

$$\begin{aligned}
\mathcal{U}(\mathbf{let}\ y = f\ x\ \mathbf{in}\ t) &= ((\mathcal{U}(t), y \oplus dy), c^y_{fx}) \\
\mathcal{U}(\mathbf{let}\ y = (\overline{x})\ \mathbf{in}\ t) &= (\mathcal{U}(t), y \oplus dy) \\
\mathcal{U}(x) &= ()
\end{aligned}$$

Differentiation of change values $\boxed{dV = \mathcal{T}(dv)}$

$$\begin{aligned}
\mathcal{T}((\overline{dv})) &= (\overline{\mathcal{T}(dv)}) \\
\mathcal{T}(dE[\lambda x\ dx.\ \mathcal{D}^\iota(t)]) &= \mathcal{T}(dE)[\lambda x\ dx\ (\mathcal{C}(t)).\ \mathcal{D}_t(t)] \\
\mathcal{T}(!v) &= !\mathcal{T}(v) \\
\mathcal{T}(d\ell) &= d\ell \\
\mathcal{T}(0_{\mathbf{p}}) &= 0_{\mathbf{p}}
\end{aligned}$$

Change terms

$$\begin{aligned}
dM ::= &\ \mathbf{let}\ dy, c^y_{fx} = df\ x\ dx\ c^y_{fx} \\
&\ \mathbf{in}\ dM \\
\mid &\ \mathbf{let}\ dy = (\overline{dx})\ \mathbf{in}\ dM \\
\mid &\ (dx, dC)
\end{aligned}$$

Cache updates

$$dC ::= (dC, c^y_{fx}) \mid (dC, x \oplus dx) \mid ()$$

Change values

$$dV ::= (\overline{dV}) \mid dF[\lambda x\ dx\ C.\ dM] \mid d\ell \mid 0_{\mathbf{p}} \mid !V$$

Change environments

$$dF ::= \bullet \mid dF; dD_v$$

Change environment entries

$$dD_v ::= D_v \mid dx = dV$$

Fig. 7. CTS differentiation and syntax of its target language λ_{ICAL}. Beware $\mathcal{T}(dE[\lambda x\ dx.\ \mathcal{D}^\iota(t)])$ applies a left-inverse of $\mathcal{D}^\iota(t)$ during pattern matching.

CTS conversion and differentiation. These translations use two auxiliary functions: $\mathcal{C}(t)$ which computes the cache term of a λ_{AL} term t, and $\mathcal{U}(t)$, which computes the cache update of t's derivative.

CTS translation on terms, $\mathcal{T}_t(t')$, accepts as inputs a *global* term t and a subterm t' of t. In tail position ($t' = x$), the translation generates code to return both the result x and the cache $\mathcal{C}(t)$ of the global term t. When the transformation visits **let**-bindings, it outputs extra bindings for caches c^y_{fx} on function calls and visits the **let**-body.

Similarly to $\mathcal{T}_t(t')$, CTS derivation $\mathcal{D}_t(t')$ accepts a global term t and a subterm t' of t. In tail position, the translation returns both the result change dx and the cache update $\mathcal{U}(t)$. On **let**-bindings, it *does not* output bindings for y but for dy, it outputs extra bindings for c^y_{fx} as in the previous case and visits the **let**-body.

To handle function definitions, we transform the base environment E through $\mathcal{T}(E)$ and $\mathcal{T}(\mathcal{D}^{\iota}(E))$ (translations of environments are done pointwise, see Appendix D). Since $\mathcal{D}^{\iota}(E)$ includes E, we describe $\mathcal{T}(\mathcal{D}^{\iota}(E))$ to also cover $\mathcal{T}(E)$. Overall, $\mathcal{T}(\mathcal{D}^{\iota}(E))$ CTS-converts each source closure $f = E[\lambda x.\, t]$ to a CTS-translated function, with body $\mathcal{T}_t(t)$, and to the CTS derivative df of f. This CTS derivative pattern matches on its input cache using cache pattern $\mathcal{C}(t)$. That way, we make sure that the shape of the cache expected by df is consistent with the shape of the cache produced by f. The body of derivative df is computed by CTS-deriving f's body via $\mathcal{D}_t(t)$.

3.6 Semantics of λ_{CAL} and λ_{ICAL}

An evaluation environment F of λ_{CAL} contains both values and cache values. Values V resemble λ_{AL} values v, cache values V_c match cache terms C and change values dV match λ_{IAL} change values dv. Evaluation environments dF for change terms must also bind change values, so functions in change closures take not just a base input x and an input change dx, like in λ_{IAL}, but also an input cache C. By abuse of notation, we reuse the same syntax C to both deconstruct and construct caches.

Base terms of the language are evaluated using a conventional big-step semantics, consisting of two judgments. Judgment "$F \vdash M \Downarrow (V, V_c)$" is read "Under evaluation environment F, base term M evaluates to value V and cache V_c". The semantics follows the one of λ_{AL}; since terms include extra code to produce and carry caches along the computation, the semantics evaluates that code as well. For space reasons, we defer semantic rules to Appendix E. Auxiliary judgment "$F \vdash C \Downarrow V_c$" evaluates cache terms into cache values: It traverses a cache term and looks up the environment for the values to be cached.

Change terms of λ_{ICAL} are also evaluated using a big-step semantics, which resembles the semantics of λ_{IAL} and λ_{CAL}. Unlike those semantics, evaluating cache updates $(dC, x \oplus dx)$ is evaluated using the $\oplus$ operator (overloaded on λ_{CAL} values and λ_{ICAL} changes). By lack of space, its rules are deferred to Appendix E. This semantics relies on three judgments. Judgment "$dF \vdash dM \Downarrow (dV, V_c)$" is read "Under evaluation environment F, change term dM evaluates to change value dV and updated cache V_c". The first auxiliary judgment "$dF \vdash dC \Downarrow V_c$" defines evaluation of cache update terms. The final auxiliary judgment "$V_c \sim C \rightarrow dF$" describes a limited form of pattern matching used by CTS derivatives: namely, how a cache pattern C matches a cache value V_c to produce a change environment dF.

3.7 Soundness of CTS Conversion

The proof is based on a simulation in lock-step, but two subtle points emerge. First, we must relate λ_{AL} environments that do not contain caches, with λ_{CAL} environments that do. Second, while evaluating CTS derivatives, the evaluation environment mixes caches from the base computation and updated caches computed by the derivatives.

Theorem 3.7 follows because differentiation is sound (Theorem 3.3) and evaluation commutes with CTS conversion; this last point requires two lemmas. First, CTS translation of base terms commutes with our semantics:

Lemma 3.5 (Commutation for base evaluations)
For all E, t *and* v, *if* $E \vdash t \Downarrow v$, *there exists* V_c, $\mathcal{T}(E) \vdash \mathcal{T}_t(t) \Downarrow (\mathcal{T}(v), V_c)$.

Second, we need a corresponding lemma for CTS translation of differentiation results: intuitively, evaluating a derivative and CTS translating the resulting change value must give the same result as evaluating the CTS derivative. But to formalize this, we must specify which environments are used for evaluation, and this requires two technicalities.

Assume derivative $\mathcal{D}^\iota(t)$ evaluates correctly in some environment dE. Evaluating CTS derivative $\mathcal{D}_t(t)$ requires cache values from the base computation, but they are not in $\mathcal{T}(dE)$! Therefore, we must introduce a judgment to complete a CTS-translated environment with the appropriate caches (see Appendix F).

Next, consider evaluating a change term of the form $dM = \mathbb{C}[dM']$, where $\mathbb{C}$ is a standard single-hole change-term context—that is, for λ_{ICAL}, a sequence of **let**-bindings. When evaluating dM, we eventually evaluate dM' in a change environment dF updated by $\mathbb{C}$: the change environment dF contains both the updated caches coming from the evaluation of $\mathbb{C}$ and the caches coming from the base computation (which will be updated by the evaluation of dM). Again, a new judgment, given in Appendix F, is required to model this process.

With these two judgments, the second key Lemma stating the commutation between evaluation of derivatives and evaluation of CTS derivatives can be stated. We give here an informal version of this Lemma, the actual formal version can be found in Appendix F.

Lemma 3.6 (Commutation for derivatives evaluation)
If the evaluation of $\mathcal{D}^\iota(t)$ *leads to an environment* dE_0 *when it reaches the differentiated context* $\mathcal{D}^\iota(\mathbb{C})$ *where* $t = \mathbb{C}[t']$, *and if the CTS conversion of* t *under this environment completed with base (resp. changed) caches evaluates into a base value* $\mathcal{T}(v)$ *(resp. a changed value* $\mathcal{T}(v')$*) and a base cache value* V_c *(resp. an updated cache value* V_c'*), then under an environment containing the caches already updated by the evaluation of* $\mathcal{D}^\iota(\mathbb{C})$ *and the base caches to be updated, the CTS derivative of* t' *evaluates to* $\mathcal{T}(dv)$ *such that* $v \oplus dv = v'$ *and to the updated cache* V_c'.

Finally, we can state soundness of CTS differentiation. This theorem says that CTS derivatives not only produce valid changes for incrementalization but that they also correctly consume and update caches.

Theorem 3.7 (Soundness of CTS differentiation)
If the following hypotheses hold:

1. $dE \triangleright E \hookrightarrow E'$
2. $E \vdash t \Downarrow v$
3. $E' \vdash t \Downarrow v'$

then there exists dv, V_c, V'_c *and* F_0 *such that:*

1. $\mathcal{T}(E) \vdash \mathcal{T}(t) \Downarrow (\mathcal{T}(v), V_c)$
2. $\mathcal{T}(E') \vdash \mathcal{T}(t) \Downarrow (\mathcal{T}(v'), V'_c)$
3. $\mathcal{C}(t) \sim V_c \rightarrow F_0$
4. $\mathcal{T}(dE); F_0 \vdash \mathcal{D}_t(t) \Downarrow (\mathcal{T}(dv), V'_c)$
5. $v \oplus dv = v'$

4 Incrementalization Case Studies

In this section, we investigate two questions: whether our transformations can target a typed language like Haskell and whether automatically transformed programs can perform well. We implement by hand primitives on sequences, bags and maps in Haskell. The input terms in all case studies are written in a deep embedding of λ_{AL} into Haskell. The transformations generate Haskell code that uses our primitives and their derivatives.

We run the transformations on three case studies: a computation of the average value of a bag of integers, a nested loop over two sequences and a more involved example inspired by Koch et al. [17]'s work on incrementalizing database queries. For each case study, we make sure that results are consistent between from scratch recomputation and incremental evaluation; we measure the execution time for from scratch recomputation and incremental computation as well as the space consumption of caches. We obtain efficient incremental programs, that is ones for which incremental computation is faster than from scratch recomputation. The measurements indicate that we do get the expected asymptotic improvement in time of incremental computation over from scratch recomputation by a linear factor while the caches grows in a similar linear factor.

Our benchmarks were compiled by GHC 8.2.2 and run on a 2.20 GHz hexa core Intel(R) Xeon(R) CPU E5-2420 v2 with 32 GB of RAM running Ubuntu 14.04. We use the *criterion* [21] benchmarking library.

4.1 Averaging Bags of Integers

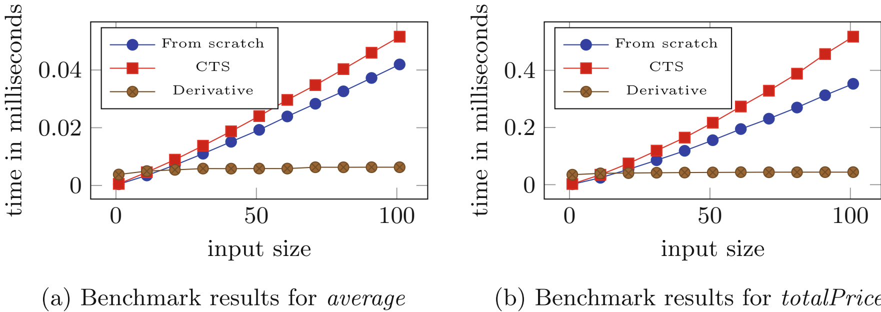
Section 2.1 motivates our transformation with a running example of computing the average over a bag of integers. We represent bags as maps from elements to (possibly negative) multiplicities. Earlier work [7,17] represents bag changes as bags of removed and added elements. We use a different representation of bag changes that takes advantage of the changes to elements and provide primitives on bags and their derivatives. The CTS variant of *map*, that we call *mapC*, takes a function *fC* in CTS and a bag *as* and produces a bag and a cache. The cache stores for each invocation of *fC*, and therefore for each distinct element in *as*, the result of *fC* of type b and the cache of type c.

Inspired by Rossberg et al. [23], all higher-order functions (and typically, also their caches) are parametric over cache types of their function arguments. Here, functions *mapC* and *dmapC* and cache type *MapC* are parametric over the cache type c of *fC* and *dfC*.

$map :: (a \rightarrow b) \rightarrow Bag\ a \rightarrow Bag\ b$
$\textbf{data}\ MapC\ a\ b\ c = MapC\ (Map\ a\ (b, c))$
$mapC :: (a \rightarrow (b, c)) \rightarrow Bag\ a \rightarrow (Bag\ b, MapC\ a\ b\ c)$
$dmapC :: (a \rightarrow (b, c)) \rightarrow (a \rightarrow \Delta a \rightarrow c \rightarrow (\Delta b, c)) \rightarrow Bag\ a \rightarrow \Delta(Bag\ a) \rightarrow$
$\quad MapC\ a\ b\ c \rightarrow (\Delta(Bag\ b), MapC\ a\ b\ c)$

We wrote the *length* and *sum* functions used in our benchmarks in terms of primitives *map* and *foldGroup* and had their CTS function and CTS derivative generated automatically.

We evaluate whether we can produce an updated result with *daverageC* shown in Sect. 2.1 faster than by from scratch recomputation with *average*. We expect the speedup of *daverageC* to depend on the size of the input bag n. We fix an input bag of size n as the bag containing the numbers from 1 to n. We define a change that inserts the integer 1 into the bag. To measure execution time of from scratch recomputation, we apply *average* to the input bag updated with the change. To measure execution time of the CTS function *averageC*, we apply *averageC* to the input bag updated with the change. To measure execution time of the CTS derivative *daverageC*, we apply *daverageC* to the input bag, the change and the cache produced by *averageC* when applied to the input bag. In all three cases we ensure that all results and caches are fully forced so as to not hide any computational cost behind laziness.



(a) Benchmark results for *average*

(b) Benchmark results for *totalPrice*

Fig. 8. Benchmark results for *average* and *totalPrice*

The plot in Fig. 8a shows execution time versus the size n of the base input. To produce the base result and cache, the CTS transformed function *averageC* takes longer than the original *average* function takes to produce just the result. Producing the updated result incrementally is slower than from scratch recomputation for small input sizes, but because of the difference in time complexity becomes faster as the input size grows. The size of the cache grows linearly with the size of the input, which is not optimal for this example. We leave optimizing the space usage of examples like this to future work.

4.2 Nested Loops over Two Sequences

Next, we consider CTS differentiation on a higher-order example. To incrementalize this example efficiently, we have to enable detecting nil function changes at runtime by representing function changes as closures that can be inspected by incremental programs. Our example here is the Cartesian product of two sequences computed in terms of functions *map* and *concat*.

$$\begin{array}{l} \mathit{cartesianProduct} :: \mathit{Sequence}\ a \rightarrow \mathit{Sequence}\ b \rightarrow \mathit{Sequence}\ (a, b) \\ \mathit{cartesianProduct}\ \mathit{xs}\ \mathit{ys} = \mathit{concatMap}\ (\lambda x \rightarrow \mathit{map}\ (\lambda y \rightarrow (x, y))\ \mathit{ys})\ \mathit{xs} \\ \mathit{concatMap} :: (a \rightarrow \mathit{Sequence}\ b) \rightarrow \mathit{Sequence}\ a \rightarrow \mathit{Sequence}\ b \\ \mathit{concatMap}\ f\ \mathit{xs} = \mathit{concat}\ (\mathit{map}\ f\ \mathit{xs}) \end{array}$$

We implemented incremental sequences and related primitives following Firsov and Jeltsch [9]: our change operations and first-order operations (such as *concat*) reuse their implementation. On the other hand, we must extend higher-order operations such as *map* to handle non-nil function changes and caching. A correct and efficient CTS derivative *dmapC* has to work differently depending on whether the given function change is nil or not: For a non-nil function change it has to go over the input sequence; for a nil function change it has to avoid that.

Cai et al. [7] use static analysis to conservatively approximate nil function changes as changes to terms that are closed in the original program. But in this example the function argument $(\lambda y \rightarrow (x, y))$ to *map* in *cartesianProduct* is not a closed term. It is, however, crucial for the asymptotic improvement that we avoid looping over the inner sequence when the change to the free variable x in the change environment is $\mathbf{0}_x$.

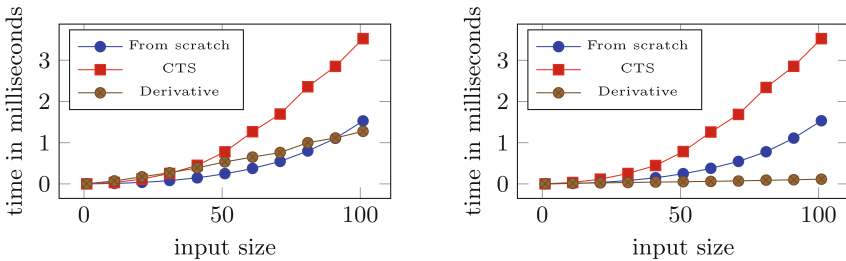
To enable runtime nil change detection, we apply closure conversion to the original program and explicitly construct closures and changes to closures. While the only valid change for closed functions is their nil change, for closures we can have non-nil function changes. A function change *df*, represented as a closure change, is nil exactly when all changes it closes over are nil.

We represent closed functions and closures as variants of the same type. Correspondingly we represent changes to a closed function and changes to a closure as variants of the same type of function changes. We inspect this representation at runtime to find out if a function change is a nil change.

$$\begin{array}{l} \textbf{data}\ \mathit{Fun}\ a\ b\ c\ \textbf{where} \\ \quad \mathit{Closed} :: (a \rightarrow (b, c)) \rightarrow \mathit{Fun}\ a\ b\ c \\ \quad \mathit{Closure} :: (e \rightarrow a \rightarrow (b, c)) \rightarrow e \rightarrow \mathit{Fun}\ a\ b\ c \\ \textbf{data}\ \Delta(\mathit{Fun}\ a\ b\ c)\ \textbf{where} \\ \quad \mathit{DClosed} :: (a \rightarrow \Delta a \rightarrow c \rightarrow (\Delta b, c)) \rightarrow \Delta(\mathit{Fun}\ a\ b\ c) \\ \quad \mathit{DClosure} :: (e \rightarrow \Delta e \rightarrow a \rightarrow \Delta a \rightarrow c \rightarrow (\Delta b, c)) \rightarrow e \rightarrow \Delta e \rightarrow \Delta(\mathit{Fun}\ a\ b\ c) \end{array}$$

We use the same benchmark setup as in the benchmark for the average computation on bags. The input of size n is a pair of sequences $(\mathit{xs}, \mathit{ys})$. Each sequence

initially contains the integers from 1 to n. Updating the result in reaction to a change dxs to the outer sequence xs takes less time than updating the result in reaction to a change dys to the inner sequence ys. While a change to the outer sequence xs results in an easily located change in the output sequence, a change for the inner sequence ys results in a change that needs a lot more calculation to find the elements it affects. We benchmark changes to the outer sequence xs and the inner sequence ys separately where the change to one sequence is the insertion of a single integer 1 at position 1 and the change for the other one is the nil change.



(a) Benchmark results for Cartesian product changing *inner* sequence.

(b) Benchmark results for Cartesian product changing *outer* sequence.

Fig. 9. Benchmark results for *cartesianProduct*

Figure 9 shows execution time versus input size. In this example again preparing the cache takes longer than from scratch recomputation alone. The speedup of incremental computation over from scratch recomputation increases with the size of the base input sequences because of the difference in time complexity. Eventually we do get speedups for both kinds of changes (to the inner and to the outer sequence), but for changes to the outer sequence we get a speedup earlier, at a smaller input size. The size of the cache grows super linearly in this example.

4.3 Indexed Joins of Two Bags

Our goal is to show that we can compose primitive functions into larger and more complex programs and apply CTS differentiation to get a fast incremental program. We use an example inspired from the DBToaster literature [17]. In this example we have a bag of orders and a bag of line items. An order is a pair of an order key and an exchange rate. A line item is a pair of an order key and a price. We build an index mapping each order key to the sum of all exchange rates of the orders with this key and an index from order key to the sum of the prices of all line items with this key. We then merge the two maps by key, multiplying corresponding sums of exchange rates and sums of prices. We compute the total price of the orders and line items as the sum of those products.

```
type Order = (ℤ, ℤ)
type LineItem = (ℤ, ℤ)
totalPrice :: Bag Order → Bag LineItem → ℤ
totalPrice orders lineItems = let
  orderIndex = groupBy fst orders
  orderSumIndex = Map.map (Bag.foldMapGroup snd) orderIndex
  lineItemIndex = groupBy fst lineItems
  lineItemSumIndex = Map.map (Bag.foldMapGroup snd) lineItemIndex
  merged = Map.merge orderSumIndex lineItemSumIndex
  total = Map.foldMapGroup multiply merged
  in total
groupBy :: (a → k) → Bag a → Map k (Bag a)
groupBy keyOf bag =
  Bag.foldMapGroup (λa → Map.singleton (keyOf a) (Bag.singleton a)) bag
```

Unlike DBToaster, we assume our program is already transformed to explicitly use indexes, as above. Because our indexes are maps, we implemented a change structure, CTS primitives and their CTS derivatives for maps.

To build the indexes, we use a *groupBy* function built from primitive functions *foldMapGroup* on bags and *singleton* for bags and maps respectively. The CTS function *groupByC* and the CTS derivative *dgroupByC* are automatically generated. While computing the indexes with *groupBy* is self-maintainable, merging them is not. We need to cache and incrementally update the intermediately created indexes to avoid recomputing them.

We evaluate the performance in the same way we did in the other case studies. The input of size n is a pair of bags where both contain the pairs (i, i) for i between 1 and n. The change is an insertion of the order $(1, 1)$ into the orders bag. For sufficiently large inputs, our CTS derivative of the original program produces updated results much faster than from scratch recomputation, again because of a difference in time complexity as indicated by Fig. 8b. The size of the cache grows linearly with the size of the input in this example. This is unavoidable, because we need to keep the indexes.

4.4 Limitations and Future Work

Typing of CTS programs. Functions of the same type $f_1, f_2 :: A \to B$ can be transformed to CTS functions $f_1 :: A \to (B, C_1), f_2 :: A \to (B, C_2)$ with different cache types C_1, C_2, since cache types depend on the implementation. This heterogeneous typing of translated functions poses difficult typing issues, e.g. what is the translated type of a *list* $(A \to B)$? We cannot hide cache types behind existential quantifiers because they would be too abstract for derivatives, which only work on very specific cache types. We can fix this problem with some runtime overhead by using a single type *Cache*, defined as a tagged union of all cache types or, maybe with more sophisticated type systems—like first-class translucent sums, open existentials or Typed Adapton's refinement types [12]—that could be able to correctly track down cache types properly.

In any case, we believe that these machineries would add a lot of complexity without helping much with the proof of correctness. Indeed, the simulation relation is more handy here because it maintains a global invariant about the whole evaluations (typically the consistency of cache types between base computations and derivatives), not many local invariants about values as types would.

One might wonder why caches could not be totally hidden from the programmer by embedding them in the derivatives themselves; or in other words, why we did not simply translate functions of type $A \to B$ into functions of type $A \to B \times (\Delta A \to \Delta B)$. We tried this as well; but unlike automatic differentiation, we must remember and update caches according to input changes (especially when receiving a sequence of such changes as in Sect. 2.1). Returning the updated cache to the caller works; we tried closing over the caches in the derivative, but this ultimately fails (because we could receive function changes to the original function, but those would need access to such caches).

Comprehensive performance evaluation. This paper focuses on theory and we leave benchmarking in comparison to other implementations of incremental computation to future work. The examples in our case study were rather simple (except perhaps for the indexed join). Nevertheless, the results were encouraging and we expect them to carry over to more complex examples, but not to all programs. A comparison to other work would also include a comparison of space usage for auxiliary data structure, in our case the caches.

Cache pruning via absence analysis. To reduce memory usage and runtime overhead, it should be possible to automatically remove from transformed programs any caches or cache fragments that are not used (directly or indirectly) to compute outputs. Liu [19] performs this transformation on CTS programs by using *absence analysis*, which was later extended to higher-order languages by Sergey et al. [25]. In lazy languages, absence analysis removes thunks that are not needed to compute the output. We conjecture that the analysis could remove unused caches or inputs, if it is extended to *not* treat caches as part of the output.

Unary vs n-ary abstraction. We only show our transformation correct for unary functions and tuples. But many languages provide efficient support for applying curried functions such as $div :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$. Naively transforming such a curried function to CTS would produce a function $divC$ of type $\mathbb{Z} \to (\mathbb{Z} \to (\mathbb{Z}, DivC_2)), DivC_1$ with $DivC_1 = ()$, which adds excessive overhead. In Sect. 2 and our evaluation we use curried functions and never need to use this naive encoding, but only because we always invoke functions of known arity.

5 Related Work

Cache-transfer-style. Liu [19]'s work has been the fundamental inspiration to this work, but her approach has no correctness proof and is restricted to a first-order untyped language. Moreover, while the idea of cache-transfer-style is similar,

it's unclear if her approach to incrementalization would extend to higher-order programs. Firsov and Jeltsch [9] also approach incrementalization by code transformation, but their approach does not deal with changes to functions. Instead of transforming functions written in terms of primitives, they provide combinators to write CTS functions and derivatives together. On the other hand, they extend their approach to support mutable caches, while restricting to immutable ones as we do might lead to a logarithmic slowdown.

Finite differencing. Incremental computation on collections or databases by finite differencing has a long tradition [6,22]. The most recent and impressive line of work is the one on DBToaster [16,17], which is a highly efficient approach to incrementalize queries over bags by combining iterated finite differencing with other program transformations. They show asymptotic speedups both in theory and through experimental evaluations. Changes are only allowed for datatypes that form groups (such as bags or certain maps), but not for instance for lists or sets. Similar ideas were recently extended to higher-order and nested computation [18], though only for datatypes that can be turned into groups. Koch et al. [18] emphasize that iterated differentiation is necessary to obtain efficient derivatives; however, ANF conversion and remembering intermediate results appear to address the same problem, similarly to the field of automatic differentiation [27].

Logical relations. To study correctness of incremental programs we use a logical relation among base values v_1, updated values v_2 and changes dv. To define a logical relation for an untyped λ-calculus we use a *step-indexed* logical relation, following Ahmed [4], Appel and McAllester [5]; in particular, our definitions are closest to the ones by Acar et al. [3], who also work with an untyped language, big-step semantics and (a different form of) incremental computation. However, they do not consider first-class changes. Technically, we use environments rather than substitution, and index our big-step semantics differently.

Dynamic incrementalization. The approaches to incremental computation with the widest applicability are in the family of self-adjusting computation [1,2], including its descendant Adapton [14]. These approaches incrementalize programs by combining memoization and change propagation: after creating a trace of base computations, updated inputs are compared with old ones in $O(1)$ to find corresponding outputs, which are updated to account for input modifications. Compared to self-adjusting computation, Adapton only updates results that are demanded. As usual, incrementalization is not efficient on arbitrary programs, but only on programs designed so that input changes produce small changes to the computation trace; refinement type systems have been designed to assist in this task [8,12]. To identify matching inputs, Nominal Adapton [13] replaces input comparisons by pointer equality with first-class labels, enabling more reuse.

6 Conclusion

We have presented a program transformation which turns a functional program into its derivative and efficiently shares redundant computations between them thanks to a statically computed cache.

Although our first practical case studies show promising results, this paper focused on putting CTS differentiation on solid theoretical ground. For the moment, we only have scratched the surface of the incrementalization opportunities opened by CTS primitives and their CTS derivatives: in our opinion, exploring the design space for cache data structures will lead to interesting new results in purely functional incremental programming.

Acknowledgments. We are grateful to anonymous reviewers: they made important suggestions to help us improve our technical presentation. We also thank Cai Yufei, Tillmann Rendel, Lourdes del Carmen González Huesca, Klaus Ostermann, Sebastian Erdweg for helpful discussions on this project. This work was partially supported by DFG project 282458149 and by SNF grant No. 200021_166154.

References

1. Acar, U.A.: Self-adjusting computation. Ph.D. thesis, Carnegie Mellon University (2005)
2. Acar, U.A.: Self-adjusting computation: (an overview). In: PEPM, pp. 1–6. ACM (2009)
3. Acar, U.A., Ahmed, A., Blume, M.: Imperative self-adjusting computation. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 309–322. ACM, New York (2008). https://doi.acm.org/10.1145/1328438.1328476
4. Ahmed, A.: Step-indexed syntactic logical relations for recursive and quantified types. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 69–83. Springer, Heidelberg (2006). https://doi.org/10.1007/11693024_6
5. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. Program. Lang. Syst. **23**(5), 657–683 (2001). https://doi.acm.org/10.1145/504709.504712
6. Blakeley, J.A., Larson, P.A., Tompa, F.W.: Efficiently updating materialized views. In: SIGMOD, pp. 61–71. ACM (1986)
7. Cai, Y., Giarrusso, P.G., Rendel, T., Ostermann, K.: A theory of changes for higher-order languages—incrementalizing λ-calculi by static differentiation. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 145–155. ACM, New York (2014). https://doi.acm.org/10.1145/2594291.2594304
8. Çiçek, E., Paraskevopoulou, Z., Garg, D.: A type theory for incremental computational complexity with control flow changes. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, pp. 132–145. ACM, New York (2016)

9. Firsov, D., Jeltsch, W.: Purely functional incremental computing. In: Castor, F., Liu, Y.D. (eds.) SBLP 2016. LNCS, vol. 9889, pp. 62–77. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45279-1_5
10. Giarrusso, P.G.: Optimizing and incrementalizing higher-order collection queries by AST transformation. Ph.D. thesis, University of Tübingen (2018). Defended. http://inc-lc.github.io/
11. Gupta, A., Mumick, I.S.: Maintenance of materialized views: problems, techniques, and applications. In: Gupta, A., Mumick, I.S. (eds.) Materialized Views, pp. 145–157. MIT Press (1999)
12. Hammer, M.A., Dunfield, J., Economou, D.J., Narasimhamurthy, M.: Typed adapton: refinement types for incremental computations with precise names. October 2016 arXiv:1610.00097 [cs]
13. Hammer, M.A., et al.: Incremental computation with names. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, pp. 748–766. ACM, New York (2015). https://doi.acm.org/10.1145/2814270.2814305
14. Hammer, M.A., Phang, K.Y., Hicks, M., Foster, J.S.: Adapton: composable, demand-driven incremental computation. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 156–166. ACM, New York (2014)
15. Johnsson, T.: Lambda lifting: transforming programs to recursive equations. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 190–203. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-15975-4_37
16. Koch, C.: Incremental query evaluation in a ring of databases. In: Symposium Principles of Database Systems (PODS), pp. 87–98. ACM (2010)
17. Koch, C., et al.: DBToaster: higher-order delta processing for dynamic, frequently fresh views. VLDB J. **23**(2), 253–278 (2014). https://doi.org/10.1007/s00778-013-0348-4
18. Koch, C., Lupei, D., Tannen, V.: Incremental view maintenance for collection programming. In: Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, pp. 75–90. ACM, New York (2016)
19. Liu, Y.A.: Efficiency by incrementalization: an introduction. HOSC **13**(4), 289–313 (2000)
20. Liu, Y.A., Teitelbaum, T.: Caching intermediate results for program improvement. In: Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 1995, pp. 190–201. ACM, New York (1995). https://doi.acm.org/10.1145/215465.215590
21. O'Sullivan, B.: criterion: a Haskell microbenchmarking library (2014). http://www.serpentine.com/criterion/
22. Paige, R., Koenig, S.: Finite differencing of computable expressions. TOPLAS **4**(3), 402–454 (1982)
23. Rossberg, A., Russo, C.V., Dreyer, D.: F-ing modules. In: Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation, TLDI 2010, pp. 89–102. ACM, New York (2010)
24. Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. LISP Symb. Comput. **6**(3–4), 289–360 (1993)
25. Sergey, I., Vytiniotis, D., Peyton Jones, S.: Modular, higher-order cardinality analysis in theory and practice. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, pp. 335–347. ACM, New York (2014)

26. The Coq Development Team: The Coq proof assistant reference manual, version 8.8 (2018). http://coq.inria.fr
27. Wang, F., Wu, X., Essertel, G., Decker, J., Rompf, T.: Demystifying differentiable programming: shift/reset the penultimate backpropagator. Technical report (2018). https://arxiv.org/abs/1803.10228

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.