Short Cut to Incremental Typed Functional Programs (Extended Abstract)

Akimasa Morihata
University of Tokyo, Komaba, Tokyo, Japan
morihata@graco.c.u-tokyo.ac.jp

1 Introduction

When an input is modified, *incremental computing* enables efficient computation of the output that corresponds to the modified input by using the output for the original input. Formally, for a computation f, the original input x, and the modified input $x \oplus dx$, where dx and $\oplus$ denote the modification and the application of the modification, respectively, incremental computing efficiently calculates $f(x \oplus dx)$ by using $f(x)$. Incremental computing has many applications, including structured editors [4,9,14], database processing [13], and developing dynamic data structures [3]. Readers who are interested in the literature can refer to an extensive survey [21].

Among other incremental computing methods, *incrementalizing lambda calculus (ILC)* [5] is a promising approach. Given function $f :: A \to B$, which is the subject of the incremental computing, ILC derives its *derivative*, $\partial f :: (A, \Delta A) \to \Delta B$, where ΔA and ΔB denote types of modifications on A and B, The derivative is characterized by the following equation, in which $(\oplus_A) :: (A \times \Delta A) \to A$ and $(\oplus_B) :: (B \times \Delta B) \to B$ denote application of modifications, respectively. .

$$f(x \oplus_A dx) = f(x) \oplus_B \partial f(x, dx) \tag{1}$$

That is, incremental computing is achieved by modifying the previous output, $f(x)$, by the result of the derivative, $\partial f(x, dx)$.

ILC has two good characteristics. First, the approach is purely based on program transformations and requires nothing on the side of the interpreter, compiler, and runtime system. Therefore, it can be immediately combined with other optimization methods. Second, the approach is applicable to expressions in the simply-typed lambda calculus, if each of its primitive functions has its derivative; hence, it is potentially applicable to complex and realistic programs.

Giarrusso et al. [11] proposed an improvement of ILC. Their method can deal with the fixed-point operator, which was the missing piece of applying ILC to realistic programs, and combines ILC with the cache-transfer transformation [18] that remembers the values calculated in the original computation. As Equation (1) indicates, the derivative should calculate the modification of the output without using any information from the computation of $f(x)$, except the final output. Thus, the derivative may recompute most of the computation of $f(x)$. The cache-transfer transformation enables us to reuse the intermediate results of $f(x)$ and thereby removes a major source of inefficiency.

Unfortunately, the method by Giarrusso et al. is rather complicated and difficult to apply to programs written in realistic languages. In particular, it requires input programs to be preprocessed by using A'-normalization and lambda lifting. Furthermore, their method is not applicable to typed languages because it introduces arbitrary recursively nested tuples of cached values.

This paper aims to make ILC practical by adopting a *semantic* approach. It reformulates ILC in terms of *parametricity* of polymorphic types [22,24] (Section 3.2). This reformulation relieves ILC from a particular source language: the correctness of the incremental program can be proved solely from a certain polymorphic type of the function that is the subject of incremental computing; incrementalization can be achieved by simply supplying the polymorphic function with some operators, which can be systematically obtained from their derivatives. The approach enables not only the original ILC but also an improvement based on caches (Section 3.3). It is ready to apply to realistic typed functional programs. Its potential is demonstrated by applications to Haskell programs (Section 2).

2 Incrementalizing Lambda Calculus at Work

2.1 Incrementalizing Lambda Calculus

First, we consider the following simple program. We use Haskell to describe programs, except that all binary operators are uncurried.

$$\begin{aligned}&\mathit{sqInc} :: \mathbb{R} \to \mathbb{R}\\&\mathit{sqInc}\ x = (x+1)\times(x+1)\end{aligned}$$

ILC [5, 11] is based on derivatives that calculate modifications on the output from modifications on inputs. We consider simple modifications by increasing or decreasing.

$$\begin{aligned}&\mathbf{type}\ \Delta\mathbb{R} = \mathbb{R}\\&(\oplus) :: (\mathbb{R},\Delta\mathbb{R}) \to \mathbb{R}\\&x \oplus dx = x + dx\end{aligned}$$

$\Delta\mathbb{R}$ is the type for modifications of numbers, and $\oplus$ denotes an application of a modification.

ILC assumes that every primitive operation has its derivative that propagates modifications on inputs to those on outputs. The following shows the derivatives.

$$\begin{aligned}&({}_{\partial}+) :: ((\mathbb{R},\mathbb{R}),(\Delta\mathbb{R},\Delta\mathbb{R})) \to \Delta\mathbb{R}\\&({}_{\partial}+)\ ((x,y),(dx,dy)) = dy + dx\\&({}_{\partial}\times) :: ((\mathbb{R},\mathbb{R}),(\Delta\mathbb{R},\Delta\mathbb{R})) \to \Delta\mathbb{R}\\&({}_{\partial}\times)\ ((x,y),(dx,dy)) = x\times dy + dx\times y + dx\times dy\\&\partial\mathit{const} :: \mathbb{R} \to \Delta\mathbb{R}\\&\partial\mathit{const}\ x = 0\end{aligned}$$

The derivative of $+$, namely ${}_{\partial}+$, simply propagates modifications. The derivative of $\times$, namely ${}_{\partial}\times$, multiplies the modification values by the original inputs. Constants are also regarded as primitive operations and are captured by $\mathit{const}\ x = x$. Accordingly, $x+1$ is regarded as $x + \mathit{const}\ 1$. Because constants cannot be modified, the derivative of *const* returns the “zero” modification.

The original ILC by Cai et al. [5] provided a syntactic transformation to obtain a derivative of a program whose primitive operators have the derivatives. The method results in the following $\partial\mathit{sqInc}$ from *sqInc*.

$$\begin{aligned}&\partial\mathit{sqInc} :: (\mathbb{R},\Delta\mathbb{R}) \to \Delta\mathbb{R}\\&\partial\mathit{sqInc}\ (x,dx) = ({}_{\partial}\times)\ ((x+1,x+1),(({}_{\partial}+)\ ((x,1),(dx,\partial\mathit{const}\ 1)),({}_{\partial}+)\ ((x,1),(dx,\partial\mathit{const}\ 1))))\end{aligned}$$

The function, $\partial\mathit{sqInc}$, can be simplified as follows.

$$\begin{array}{rcl} \partial sqInc\ (x,dx) & = & \{ \text{ definitions of } {}_{\partial}+ \text{ and } \partial const \ \} \\ & & ({}_{\partial}\times)\ ((x+1,x+1),(dx,dx)) \\ & = & \{ \text{ definition of } {}_{\partial}\times \ \} \\ & & 2\times\ (x+1)\times dx + dx\times dx \end{array}$$

It is not difficult to see that $\partial sqInc$ is in fact a derivative of *sqInc*.

2.2 Short Cut to Incremental Numeric Computation

Instead of the syntactic transformation, our proposal uses a *template* obtained by abstracting all occurrences of the primitive operations.

$$\begin{array}{l} sqInc_{\mathrm{T}} :: \forall\alpha.\ ((\alpha,\alpha)\to\alpha)\to((\alpha,\alpha)\to\alpha)\to(\mathbb{R}\to\alpha)\to\alpha\to\alpha \\ sqInc_{\mathrm{T}}\ (\underline{+})\ (\underline{\times})\ \underline{const}\ x = (x\,\underline{+}\,\underline{const}\ 1)\ \underline{\times}\ (x\,\underline{+}\,\underline{const}\ 1) \end{array}$$

The template, $sqInc_{\mathrm{T}}$, has a polymorphic type, in which every occurrence of the type of modifiable values ($\mathbb{R}$, in this example) is abstracted[1]. Then, the corresponding derivative is obtained by supplying the following *derivative-associated* operators to the template. Each derivative-associated operator simultaneously applies both the original primitive operation and its derivative.

$$\begin{array}{l} \textbf{type}\ DA_{\mathbb{R}} = (\mathbb{R},\Delta\mathbb{R}) \\ (+^{\mathrm{DA}}) :: (DA_{\mathbb{R}},DA_{\mathbb{R}})\to DA_{\mathbb{R}} \\ (x,dx)+^{\mathrm{DA}}(y,dy) = (x+y,({}_{\partial}+)\ ((x,y),(dx,dy))) \\ (\times^{\mathrm{DA}}) :: (DA_{\mathbb{R}},DA_{\mathbb{R}})\to DA_{\mathbb{R}} \\ (x,dx)\times^{\mathrm{DA}}(y,dy) = (x\times y,({}_{\partial}\times)\ ((x,y),(dx,dy))) \\ const^{\mathrm{DA}} :: \mathbb{R}\to DA_{\mathbb{R}} \\ const^{\mathrm{DA}}\ x = (const\ x,\partial const\ x) \end{array}$$

Now define $\partial sqInc$ as follows.

$$\begin{array}{l} \partial sqInc :: DA_{\mathbb{R}}\to\Delta\mathbb{R} \\ \partial sqInc\ (x,dx) = \textbf{let}\ (y,dy) = sqInc_{\mathrm{T}}\ (+^{\mathrm{DA}})\ (\times^{\mathrm{DA}})\ const^{\mathrm{DA}}\ (x,dx)\ \textbf{in}\ dy \end{array}$$

It is not difficult to see that the function, $\partial sqInc$, is equivalent to the one obtained in Section 2.1.

One may consider $\partial sqInc$ unsatisfactory. $\partial sqInc$ contains the computation of *sqInc*; therefore, the incremental version is very similar to the naive recomputation of *sqInc*. This problem is essential in ILC. For each primitive operator, its derivative requires the original input passed to the operator; hence, recomputation is necessary. Giarrusso et al. [11] solved this problem by another transformation that derives programs for caching the calculated results.

Our approach solves this problem by borrowing an idea from the partial evaluation [16,23]. Consider $f :: A\to\Delta A$ and its derivative $\partial f :: (A,\Delta A)\to\Delta$. The type of a derivative, $(A,\Delta A)\to\Delta B$, is isomorphic to $A\to\Delta A\to\Delta B$. Therefore, by supplying the original input (of type A), we can get the function (of type $\Delta A\to\Delta B$) that translates a modification on the input to the corresponding one on the output. Formally, we would like to derive $f^{\mathrm{CDA}} :: A\to(A,\Delta A\to\Delta B)$ satisfying the following equation.

$$f^{\mathrm{CDA}}\ x = (f\ x,\lambda dx\to\partial f\ (x,dx))$$

When we hope to know $f\ x$, we can instead use $f^{\mathrm{CDA}}\ x$; moreover, as a "side effect", $f^{\mathrm{CDA}}\ x$ additionally yields a function that enables us to update the output of $f\ x$.

Unfortunately, the above definition of f^{CDA} cannot avoid recomputation because both f and ∂f traverse over x. We remove the multiple traversals by supplying the template with different operators, called *caching derivative-associated* operators.

The following is the caching derivative-associated module for *sqInc*.

[1]The input of *const* cannot be modified and thus is not abstracted.

$$
\begin{aligned}
&\textbf{type}\ \mathit{CDA}_{\Delta\mathbb{R},\mathbb{R}} = (\mathbb{R}, \Delta\mathbb{R} \to \Delta\mathbb{R})\\
&(+^{\mathrm{CDA}}) :: (\mathit{CDA}_{\Delta\mathbb{R},\mathbb{R}}, \mathit{CDA}_{\Delta\mathbb{R},\mathbb{R}}) \to \mathit{CDA}_{\Delta\mathbb{R},\mathbb{R}}\\
&(x, f_x) +^{\mathrm{DA}} (y, f_y) = (x+y, \lambda da \to (\partial{+})\ ((x,y),(f_x\ da, f_y\ da)))\\
&(\times^{\mathrm{CDA}}) :: (\mathit{CDA}_{\Delta\mathbb{R},\mathbb{R}}, \mathit{CDA}_{\Delta\mathbb{R},\mathbb{R}}) \to \mathit{CDA}_{\Delta\mathbb{R},\mathbb{R}}\\
&(x, f_x) \times^{\mathrm{DA}} (y, f_y) = (x \times y, \lambda da \to (\partial{\times})\ ((x,y),(f_x\ da, f_y\ da)))\\
&\mathit{const}^{\mathrm{CDA}} :: \mathbb{R} \to \mathit{CDA}_{\Delta\mathbb{R},\mathbb{R}}\\
&\mathit{const}^{\mathrm{CDA}}\ x = (\mathit{const}\ x, \lambda da \to \partial \mathit{const}\ x)
\end{aligned}
$$

The caching derivative-associated module constructs function closures that correspond to the computation of derivatives. The function closure will be invoked when the original input is modified (the modification is expressed by *da*).

Now define $\mathit{sqInc}^{\mathrm{CDA}}$ as follows, where $\mathit{id}\ x = x$ is the identify function.

$$
\begin{aligned}
&\mathit{sqInc}^{\mathrm{CDA}} :: \mathbb{R} \to \mathit{CDA}_{\Delta\mathbb{R},\mathbb{R}}\\
&\mathit{sqInc}^{\mathrm{CDA}}\ x = \mathit{sqInc}_{\mathrm{T}}\ (+^{\mathrm{CDA}})\ (\times^{\mathrm{CDA}})\ \mathit{const}^{\mathrm{CDA}}\ (x, \mathit{id})
\end{aligned}
$$

The function satisfies the following equations.

$$
\begin{aligned}
&(\textbf{let}\ (y, f) = \mathit{sqInc}^{\mathrm{CDA}}\ x\ \textbf{in}\ y) = \mathit{sqInc}\ x\\
&(\textbf{let}\ (y, f) = \mathit{sqInc}^{\mathrm{CDA}}\ x\ \textbf{in}\ y \oplus f\ dx) = \mathit{sqInc}\ (x \oplus dx)
\end{aligned}
$$

The first equation shows that $\mathit{sqInc}^{\mathrm{CDA}}$ results in the same value as *sqInc*. The second one states that $\mathit{sqInc}^{\mathrm{CDA}}$ additionally yields a function that behaves as the derivative with avoiding recomputation of *sqInc*.

The limitation of $\mathit{sqInc}^{\mathrm{CDA}}$ is that it cannot deal with a series of modifications. This problem can be solved by supplying *recursively-caching* operators to the template. Currently, the function part results in the modification, $\Delta\mathbb{R}$; instead, the recursively-caching variant constructs a function that additionally results in a new function (of the same type as itself) for dealing with the next modification. Because of the space limitation, the construction is omitted.

2.3 Average for Multiset

Next, we consider the example discussed by Giarrusso et al. [11]: calculation of the average of a multiset (i.e., bag).

$$\mathit{average}\ x = \mathit{sum}\ x\ /\ \mathit{length}\ x$$

For simplicity of presentation, here we regard a list as a multiset. We consider reverse list concatenations[2] as modifications. We regard *sum*, *length*, and $(/)$ as primitives. The following shows their derivatives.

$$
\begin{aligned}
&\textbf{type}\ \mathit{Bag} = [\mathbb{R}]\\
&\textbf{type}\ \Delta \mathit{Bag} = \mathit{Bag}\\
&(\oplus) :: (\mathit{Bag}, \Delta\mathit{Bag}) \to \mathit{Bag}\\
&x \oplus dx = dx +\!\!+\, x\\[4pt]
&\partial \mathit{sum} :: (\mathit{Bag}, \Delta\mathit{Bag}) \to \Delta\mathbb{R}\\
&\partial \mathit{sum}\ (x, dx) = \mathit{sum}\ dx\\[4pt]
&\partial \mathit{length} :: (\mathit{Bag}, \Delta\mathit{Bag}) \to \Delta\mathbb{R}\\
&\partial \mathit{length}\ (x, dx) = \mathit{length}\ dx\\[4pt]
&(\partial/) :: ((\mathbb{R},\mathbb{R}),(\Delta\mathbb{R},\Delta\mathbb{R})) \to \Delta\mathbb{R}\\
&(\partial/)\ ((x,y),(dx,dy)) = (x+dx)\ /\ (y+dy) - x\ /\ y
\end{aligned}
$$

[2] Reverse concatenation is faster than usual concatenation for small modifications.

Table 1: Elapsed times for two implementations of average computing (in seconds).

#modifications	1,000	2,000	5,000	10,000	100,000	1,000,000	10,000,000
average	0.20	1.14	6.78	26.89	N/A	N/A	N/A
$average^{\mathrm{CDA}}$	0.00	0.00	0.00	0.02	0.06	0.53	7.54

The original ILC cannot derive efficient incremental computing program for this example. Because $(_{\partial}/)$ uses the original inputs, which are the results of *sum x* and *length x*, the derivative of *average* requires recomputation of *sum* and *length*. To avoid this inefficiency, we derive $average^{\mathrm{CDA}}$, which is the cashing incremental version of *average*.

First, we prepare a template for *average*.

$$\begin{aligned}
&average_{\mathrm{T}} :: \forall \alpha, \beta.\, (\alpha \to \beta) \to (\alpha \to \beta) \to ((\beta, \beta) \to \beta) \to \alpha \to \beta \\
&average_{\mathrm{T}}\ \underline{sum}\ \underline{length}\ (\underline{/})\ x = \underline{sum}\ x \mathbin{\underline{/}} \underline{length}\ x
\end{aligned}$$

Note that the polymorphic type correctly captures types of modifiable values, $\mathbb{R}$ and *Bag*.

Second, we derive caching derivative-associated versions of the primitive operators.

$$\begin{aligned}
&\textbf{type}\ CDA_{\Delta Bag, Bag} = (Bag, \Delta Bag \to \Delta Bag) \\
&\textbf{type}\ CDA_{\Delta Bag, \mathbb{R}} = (\mathbb{R}, \Delta Bag \to \Delta \mathbb{R}) \\
&sum^{\mathrm{CDA}} :: CDA_{\Delta Bag, Bag} \to CDA_{\Delta Bag, \mathbb{R}} \\
&sum^{\mathrm{CDA}}\ (x, f_x) = (sum\ x, \lambda da \to \partial sum\ (x, f_x\ da)) \\
&length^{\mathrm{CDA}} :: CDA_{\Delta Bag, Bag} \to CDA_{\Delta Bag, \mathbb{R}} \\
&length^{\mathrm{CDA}}\ (x, f_x) = (length\ x, \lambda da \to \partial length\ (x, f_x\ da)) \\
&(/^{\mathrm{CDA}}) :: (CDA_{\Delta Bag, \mathbb{R}}, CDA_{\Delta Bag, \mathbb{R}}) \to CDA_{\Delta Bag, \mathbb{R}} \\
&(x, f_x) /^{\mathrm{CDA}} (y, f_y) = (x / y, \lambda da \to (_{\partial}/)\ ((x, y), (f_x\ da, f_y\ da)))
\end{aligned}$$

Finally, we obtain the incremental program by supplying these operators to the template.

$$\begin{aligned}
&average^{\mathrm{CDA}} :: Bag \to CDA_{\Delta Bag, \mathbb{R}} \\
&average^{\mathrm{CDA}}\ x = average_{\mathrm{T}}\ sum^{\mathrm{CDA}}\ length^{\mathrm{CDA}}\ (/^{\mathrm{CDA}})\ (x, id)
\end{aligned}$$

To evaluate the effectiveness of the proposed approach, *average* and $average^{\mathrm{CDA}}$ were compared[3]. The initial input was a list of 10^4 double-precision numbers. Each modification concatenated a list of 100 double-precision numbers. For each modification, the new average was calculated. The environment of the experiment consisted of Intel Core i7-7600U CPU (2.80 GHz), 16 GB memory, GHC 8.0.2, and Ubuntu 18.04 on Windows Subsystem Linux. The optimization flag was `-O2`. Elapsed times (including the time for input generation) were measured by using the runtime option `-s`. Table 1 summarizes the results of the experiment. $average^{\mathrm{CDA}}$ was significantly faster. While *average* needs time quadratic to the number of modifications, $average^{\mathrm{CDA}}$ takes roughly constant time for each modification.

3 Background Theory

This section formulates the incremental computing method presented in the previous section.

[3]Actually, the recursively-caching variant discussed in Section 2.2 is used.

3.1 Modification, Functor, and Module

Given a type A, ΔA denotes the type of its modifications. $(\oplus_A) :: (A \times \Delta A) \to A$ applies a modification to a value. Subscripts may be omitted if they are apparent from their context. For function $f :: A \to B$, its *derivative* is a function $\partial f :: (A, \Delta A) \to \Delta B$ that satisfies the following equality.

$$f\ (x \oplus_A dx) = f\ x \oplus_B \partial f\ (x, dx)$$

To capture the variation of function definitions, such as multi-input and multi-output functions, we borrow the notion of *functors* from the category theory. Functors are similar to type constructors. For functor F and type A, $\mathsf{F}A$ is a type that contains A inside the "structure" of F. Functors can be applied to functions as well: for functor F and function $f :: A \to B$, the function $\mathsf{F}f :: \mathsf{F}A \to \mathsf{F}B$ applies f to every "occurrence" of A in $\mathsf{F}A$. Each functor F should satisfy $\mathsf{F}id_A = id_{\mathsf{F}A}$, where id_T is the identity function for type T, and $\mathsf{F}(f \circ g) = \mathsf{F}f \circ \mathsf{F}g$. Examples of functors include the identity functor, $\mathsf{I}A = A$ and $\mathsf{I}f\ a = f\ a$, the diagonal functor, $\mathsf{D}A = (A, A)$ and $\mathsf{D}f\ (a_1, a_2) = (f\ a_1, f\ a_2)$, and a constant functor, $\mathsf{K}_{\mathbb{R}}A = \mathbb{R}$ and $\mathsf{K}_{\mathbb{R}}f\ a = a$. In what follows, F and G are used as metavariables that range over functors.

We impose two assumptions on functors. First, each functor F is associated with $zip_{\mathsf{F}} :: (\mathsf{F}A, \mathsf{F}B) \to \mathsf{F}(A, B)$, which pairs the corresponding elements of two operands. It should satisfy the following laws: $\mathsf{F}fst\ (zip_{\mathsf{F}}\ (a, b)) = a$ and $\mathsf{F}snd\ (zip_{\mathsf{F}}\ (a, b)) = b$, where $fst\ (a, b) = a$ and $snd\ (a, b) = b$. Note that $unzip_{\mathsf{F}}\ x = (\mathsf{F}fst\ x, \mathsf{F}snd\ x)$ is the inverse of zip_{F}. The zip_{F} function is necessary only for aligning original values and modifications, i.e., identifying the corresponding pair of an A value and a ΔA modification in $\mathsf{F}A$ and $\mathsf{F}(\Delta A)$; therefore, it is safe to assume that zip_{F} takes two structures of exactly the same shape. Second, structures expressed by functors cannot be modified. In particular, we assume that $\Delta(\mathsf{F}A) \simeq \mathsf{F}(\Delta A)$ and $(\oplus_{\mathsf{F}A}) \circ unzip_{\mathsf{F}} = \mathsf{F}(\oplus_A)$, i.e., modifications on $\mathsf{F}A$ are specified by the modifications on the A parts. For example, $\Delta(\mathsf{D}A) \simeq \mathsf{D}(\Delta A) = (\Delta A, \Delta A)$ and $(x, y) \oplus_{\mathsf{D}A} (dx, dy) = (x \oplus_A dx, y \oplus_A dy)$. These assumptions hold for many functors including those constructed by products (tuples), coproducts (tagged sum), constants, and arrows (functions).

We capture a set of primitive functions by a *module* and its *signature*. A module is a set of functions, and its signature is a set of the types of functions. For example, $(+)$, $(\times)$, and *const* considered in Section 2.2 forms a module $((+), (\times), const)$, and its signature is $((\mathbb{R}, \mathbb{R}) \to \mathbb{R}, (\mathbb{R}, \mathbb{R}) \to \mathbb{R}, \mathbb{R} \to \mathbb{R})$. Module $\mathscr{M}$ whose signature is $\mathscr{S}$ is denoted by $\mathscr{M} :: \mathscr{S}$. To focus on a module that manipulates a particular type of values, we usually express signatures by using type parameters. For example, the above-mentioned signature may be written by $\mathscr{S} = \mathsf{S}\mathbb{R}$, where $\mathsf{S}X = ((X, X) \to X, (X, X) \to X, \mathbb{R} \to X)$. This approach can be naturally extended to modules that deal with more than one type of values, by either considering tagged sums of these types, or considering multi-parameter functors.

3.2 Incremental Computing without Cache

Now we reformulate the original ILC by templates and modules. A template is a polymorphic function that takes a module as a parameter. We derive its derivative by supplying the *derivative-associated* module. Correctness of this approach can be proved by parametricity [22, 24]. Parametricity helps us to understand the relationship between two instances of a polymorphic function. Here, we derive the relationship between two instances of the template from the relationship between two modules, namely, the original and the derivative-associated.

Definition 1. *For a module $\mathscr{M} = (f_i)_{i \in I} :: \mathsf{S}A = (\mathsf{F}_i A \to \mathsf{G}_i A)_{i \in I}$, its* derivative-associated *module is $\mathscr{M}^{\mathrm{DA}} = (f_i^{\mathrm{DA}})_{i \in I} :: \mathsf{S}(DA_A)$ where $DA_A = (A, \Delta A)$ and each f_i^{DA} $(i \in I)$ is defined as follows:*

$$
\begin{array}{l}
f_i^{\mathrm{DA}} :: \mathsf{F}_i(DA_A) \to \mathsf{G}_i(DA_A) \\
f_i^{\mathrm{DA}}\ x = \mathbf{let}\ (x',dx) = unzip_{\mathsf{F}_i}\ x\ \mathbf{in}\ zip_{\mathsf{G}_i}\ (f_i\ x', \partial f_i\ (x',dx)).
\end{array}
$$

Section 2.2 already introduced an instance of a derivative-associated module. For example, the definition of $(+^{\mathrm{DA}})$ is obtained by instantiating $\mathsf{F}_i = \mathsf{D}$ and $\mathsf{G}_i = \mathsf{I}$:

$$
\begin{array}{cl}
 & (+^{\mathrm{DA}})\ xy \\
= & \{ \text{ definition above } \} \\
 & zip_{\mathsf{I}}\ ((+)\ (\mathsf{D}fst\ xy), (\partial+)\ (unzip_{\mathsf{D}}\ xy)) \\
= & \{ \text{ let } xy = ((x,dx),(y,dy)); \text{ definitions of } zip_{\mathsf{I}},\ \mathsf{D}fst \text{ and } unzip_{\mathsf{D}} \ \} \\
 & ((+)\ (x,y), (\partial+)\ ((x,y),(dx,dy))).
\end{array}
$$

The following theorem derives derivatives.

Theorem 2. *Let $\mathscr{M}^{\mathrm{DA}} :: \mathsf{S}(DA_A)$ be the derivative-associated module of $\mathscr{M} :: \mathsf{S}A$. Then, for any polymorphic function $g :: \forall\alpha.\ \mathsf{S}\alpha \to \mathsf{F}\alpha \to \mathsf{G}\alpha$, the following two properties hold:*

- $(\mathsf{G}fst \circ g\ \mathscr{M}^{\mathrm{DA}} \circ zip_{\mathsf{F}})\ (x,dx) = g\ \mathscr{M}\ x$,
- $\mathsf{G}snd \circ g\ \mathscr{M}^{\mathrm{DA}} \circ zip_{\mathsf{F}}$ *is a derivative of* $g\ \mathscr{M}$.

Proof. The proof is based on parametricity [22, 24]. For the first property, it is sufficient to show $\mathsf{G}_i fst \circ f_i^{\mathrm{DA}} = f_i \circ \mathsf{F}_i fst$ for every $f_i :: \mathsf{F}_i A \to \mathsf{G}_i A \in \mathscr{M}$, which is apparent from the definition of f_i^{DA}. The second one can be proved by showing $(\oplus_{\mathsf{G}A}) \circ unzip_{\mathsf{G}} \circ g\ \mathscr{M}^{\mathrm{DA}} \circ zip_{\mathsf{F}} = g\ \mathscr{M} \circ (\oplus_{\mathsf{F}A})$. This equation follows from $(\oplus_{\mathsf{G}_i A}) \circ unzip_{\mathsf{G}_i} \circ f_i^{\mathrm{DA}} = f_i \circ (\oplus_{\mathsf{F}_i A}) \circ unzip_{\mathsf{F}_i}$ for every $f_i \in \mathscr{M}$, which is justified as follows:

$$
\begin{array}{cl}
 & ((\oplus_{\mathsf{G}_i A}) \circ unzip_{\mathsf{G}_i} \circ f_i^{\mathrm{DA}})\ x \\
= & \{ \text{ definition of } f_i^{\mathrm{DA}} \text{ and canceling } unzip_{\mathsf{G}_i} \text{ with } zip_{\mathsf{G}_i} \ \} \\
 & \mathbf{let}\ (x',dx) = unzip_{\mathsf{F}_i}\ x\ \mathbf{in}\ f_i\ x' \oplus_{\mathsf{G}_i A} \partial f_i\ (x',dx) \\
= & \{ \ \partial f_i \text{ is the derivative of } f_i \ \} \\
 & \mathbf{let}\ (x',dx) = unzip_{\mathsf{F}_i}\ x\ \mathbf{in}\ f_i\ (x' \oplus_{\mathsf{F}_i A} dx).
\end{array}
$$

□

3.3 Incremental Computing with Cache

Caches can be introduced similarly, by supplying a template with a *caching derivative-associated* module. However, the situation is subtly different. We need $zip'_{\mathsf{F}} :: (\mathsf{F}A, B \to \mathsf{F}C) \to \mathsf{F}(A, B \to C)$, which is a variant of zip'_{F}. The function, zip'_{F} should satisfy $\mathsf{F}fst\ (zip'_{\mathsf{F}}\ (a,f)) = a$ and $\mathsf{F}(@b)\ (\mathsf{F}snd\ (zip'_{\mathsf{F}}\ (a,f))) = f\ b$, where $(@b)\ f = f\ b$. Such zip'_{F} exists for functors constructed from products, coproducts, constants, and arrows. Let $CDA_{B,A} = (A, B \to \Delta A)$.

Definition 3. *For a module $\mathscr{M} = (f_i)_{i \in I} :: \mathsf{S}A = (\mathsf{F}_i A \to \mathsf{G}_i A)_{i \in I}$, its* caching derivative-associated *module is $\mathscr{M}^{\mathrm{CDA}} = (f_i^{\mathrm{CDA}})_{i \in I} :: \forall\delta.\ \mathsf{S}(CDA_{\delta,A})$ where each f_i^{CDA} $(i \in I)$ is defined as follows:*

$$
\begin{array}{l}
f_i^{\mathrm{CDA}} :: \forall\delta.\ \mathsf{F}_i(CDA_{\delta,A}) \to \mathsf{G}_i(CDA_{\delta,A}) \\
f_i^{\mathrm{CDA}}\ x = \mathbf{let}\ (y,g) = unzip_{\mathsf{F}_i}\ x\ \mathbf{in}\ zip'_{\mathsf{G}_i}\ (f_i\ y, \lambda da \to \partial f_i\ (y, \mathsf{F}_i(@da)\ g)).
\end{array}
$$

Sections 2.2 and 2.3 contain instances of caching derivative-associated modules. For example, the definition of $(+^{\mathrm{CDA}})$ is obtained as follows.

$$
\begin{array}{cl}
 & (+^{\mathrm{CDA}})\ ((x,f_x),(y,f_y)) \\
= & \{ \text{ definition above, where } unzip_{\mathsf{D}}\ ((x,f_x),(y,f_y)) = ((x,y),(f_x,f_y)) \ \} \\
 & zip'_{\mathsf{I}}\ (x+y, \lambda da \to (\partial+)\ ((x,y),(f_x\ da, f_y\ da))) \\
= & \{ \text{ definition of } zip'_{\mathsf{I}} \ \} \\
 & (x+y, \lambda da \to (\partial+)\ ((x,y),(f_x\ da, f_y\ da)))
\end{array}
$$

The following theorem proves the correctness of the incrementalization.

Theorem 4. *Let $\mathscr{M}^{\text{CDA}} :: \forall\delta.\, \mathsf{S}(CDA_{\delta,A})$ be the caching derivative-associated module of $\mathscr{M} :: \mathsf{S}A$. Then, for any polymorphic function $g :: \forall\alpha.\, \mathsf{S}\alpha \to \mathsf{F}\alpha \to \mathsf{G}\alpha$, the following two equations hold, where $g^{\text{CDA}}\ x = g\ \mathscr{M}^{\text{CDA}}\ (zip_{\mathsf{F}}\ (x, \mathsf{F}id))$:*

- $\mathsf{G}fst\ (g^{\text{CDA}}\ x) = g\ \mathscr{M}\ x$,
- $(\mathbf{let}\ (r,f) = unzip_{\mathsf{G}}\ (g^{\text{CDA}}\ x)\ \mathbf{in}\ r \oplus_{\mathsf{G}A} \mathsf{G}(@dz)\ f) = g\ \mathscr{M}\ (x \oplus_{\mathsf{F}A} dz)$.

Proof. The proof is based on parametricity [22, 24]. The first equation follows from $\mathsf{G}_i fst \circ f_i^{\text{CDA}} = f_i \circ \mathsf{F}_i fst$ for every $f_i :: \mathsf{F}_i A \to \mathsf{G}_i A \in \mathscr{M}$, which is apparent from the definition of f_i^{CDA}. For the second equation, we first reason about $f_i :: \mathsf{F}_i A \to \mathsf{G}_i A \in \mathscr{M}$.

$$
\begin{aligned}
& \mathbf{let}\ (r,f) = unzip_{\mathsf{G}_i}\ (f_i^{\text{CDA}}\ x)\ \mathbf{in}\ r \oplus_{\mathsf{G}_i A} \mathsf{G}_i(@dz)\ f \\
= \ & \{\ \text{definition of } f_i^{\text{CDA}}\ \} \\
& \mathbf{let}\ (y,g) = unzip_{\mathsf{F}_i}\ x \\
& \quad\ (r,f) = unzip_{\mathsf{G}_i}\ (zip'_{\mathsf{G}_i}\ (f_i\ y, \lambda dz \to \partial f_i\ (y, \mathsf{F}_i(@dz)\ g))) \\
& \mathbf{in}\ r \oplus_{\mathsf{G}_i A} \mathsf{G}_i(@dz)\ f \\
= \ & \{\ \text{property of } zip' \text{ and } unzip\ \} \\
& \mathbf{let}\ (y,g) = unzip_{\mathsf{F}_i}\ x\ \mathbf{in}\ f_i\ y \oplus_{\mathsf{G}_i A} \partial f_i\ (y, \mathsf{F}_i(@dz)\ g) \\
= \ & \{\ \text{property of derivative}\ \} \\
& \mathbf{let}\ (y,g) = unzip_{\mathsf{F}_i}\ x\ \mathbf{in}\ f_i\ (y \oplus_{\mathsf{F}_i A} \mathsf{F}_i(@dz)\ g)
\end{aligned}
$$

Now the second equation follows from parametricity.

$$
\begin{aligned}
& \mathbf{let}\ (r,f) = unzip_{\mathsf{G}}\ (g^{\text{CDA}}\ x)\ \mathbf{in}\ r \oplus_{\mathsf{G}A} \mathsf{G}(@dz)\ f \\
= \ & \{\ \text{parametricity: because of the reasoning above}\ \} \\
& g\ \mathscr{M}\ (\mathbf{let}\ (r,f) = unzip_{\mathsf{F}}\ (zip_{\mathsf{F}}\ (x, \mathsf{F}id))\ \mathbf{in}\ r \oplus_{\mathsf{F}A} \mathsf{F}(@dz)\ f) \\
= \ & \{\ \text{property of } zip,\ unzip, \text{ and } \mathsf{F}id\ \} \\
& g\ \mathscr{M}\ (x \oplus_{\mathsf{F}A} dz)
\end{aligned}
$$

□

4 Discussions

This paper reformulated ILC [5, 11] by using parametricity. ILC has two major characteristics. First, it is applicable to general higher-order functional programs. The original ILC by Cai et al. is applicable to simply-typed lambda calculus. The one by Giarrusso et al. is applicable to a Turing-complete, higher-order functional language. The current approach is applicable to any typed functional languages that satisfy parametricity. Roughly speaking, any reasonably defined typed pure functional language satisfies parametricity, except for some corner cases such as nontermination and selective strictness [15, 24]. The current approach clarified that the proof of the correctness ILC can be largely simplified if the underlying language satisfies parametricity. Second, ILC is a pure program transformation and therefore requires none of special interpreter, compiler, nor runtime system. This is a important characteristics for incrementalizing programs written in practical languages. Unfortunately, the ILC by Giarrusso et al. is not satisfactory from this perspective because it requires A'-normalization and lambda lifting as preprocessing. The current approach only needs the templates obtained by a variant of lambda lifting and avoids A'-normalization. Note that existing methods [8, 17, 26] can automate the derivation of templates.

Programs derived by Theorem 2 are essentially the same as those obtained by the ILC by Cai et al. Programs derived by Theorem 4 are different from those obtained by the ILC by Giarrusso et al. Their caching method is based on recursively nested tuples, which is unavailable in typed language. Instead, the current approach developed a function-based method for caching.

Sundaresh and Hudak [23] proposed an incrementalization method based on partial evaluation [16]. Teitelbaum et al. [18, 20] developed an incrementalization method based on caching. All of them introduce program transformations applicable to general-purpose programming languages but do not use derivatives. The current proposal combines partial evaluation and tupling [7, 12] with ILC. Tupling is a variant of caching and enables simultaneous evaluation of multiple functions on the same input. It is used for developing the derivative-associated and caching derivative-associated modules.

There are many other program incrementalization methods that are based on different design decision. Those that consider domain-specific programs include [4, 9, 13, 14, 19, 20, 25]. Those that require modifications on runtime system include [1–3, 6].

The proposed approach of using derivative-associated modules instead of standard modules is similar to the one by Elliott [10]. The objective of Elliott's method is not incrementalization but automatic differentiation, i.e., differentiating numeric programs as mathematics. Studying the relationship between the proposed method and Elliott's method may open up a new perspective on how to connect program incrementalization and mathematical differentiation.

Acknowledgement The author is grateful to anonymous reviewers of WPTE 2020 for their comments useful for improving presentation. The author acknowledges the discussion with Zhenjian Hu and that in the meeting of the Cooperative Research Project of the Research Institute of Electrical Communication, Tohoku University. This research is partially supported by JSPS KAKENHI 19K11896 and ROIS NII Open Collaborative Research 19FA01.

References

[1] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper & Kanat Tangwongsan (2009): *An experimental analysis of self-adjusting computation*. *ACM Trans. Program. Lang. Syst.* 32(1), pp. 3:1–3:53, doi:10.1145/1596527.1596530.

[2] Umut A. Acar, Guy E. Blelloch & Robert Harper (2006): *Adaptive functional programming*. *ACM Trans. Program. Lang. Syst.* 28(6), pp. 990–1034. doi:10.1145/1186634.

[3] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes & Shan Leung Maverick Woo (2004): *Dynamizing static algorithms, with applications to dynamic trees and history independence*. In J. Ian Munro, editor: *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*, SIAM, pp. 531–540. doi:10.5555/982792.982871

[4] Henk Alblas (1991): *Incremental Attribute Evaluation*. In Henk Alblas & Borivoj Melichar, editors: *Attribute Grammars, Applications and Systems, International Summer School SAGA, Prague, Czechoslovakia, June 4-13, 1991, Proceedings*, *Lecture Notes in Computer Science* 545, Springer, pp. 215–233. doi:10.1007/3-540-54572-7_8

[5] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel & Klaus Ostermann (2014): *A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation*. In Michael F. P. O'Boyle & Keshav Pingali, editors: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, ACM, pp. 145–155. doi:10.1145/2594291.2594304

[6] Yan Chen, Joshua Dunfield & Umut A. Acar (2012): *Type-directed automatic incrementalization*. In Jan Vitek, Haibo Lin & Frank Tip, editors: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, ACM, pp. 299–310. doi:10.1145/2345156.2254100

[7] Wei-Ngan Chin, Siau-Cheng Khoo & Neil Jones (2006): *Redundant Call Elimination via Tupling*. *Fundamenta Informaticae* 69(1-2), pp. 1–37.

[8] Olaf Chitil (1999): *Type Inference Builds a Short Cut to Deforestation*. In: *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming, ICFP'99, Paris, France, September 27-29, 1999*, ACM, pp. 249–260. doi:10.1145/317765.317907

[9] Alan J. Demers, Thomas W. Reps & Tim Teitelbaum (1981): *Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors*. In John White, Richard J. Lipton & Patricia C. Goldberg, editors: *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, Williamsburg, Virginia, USA, January 1981*, ACM Press, pp. 105–116. doi:10.1145/567532.567544

[10] Conal M. Elliott (2009): *Beautiful differentiation*. In Graham Hutton & Andrew P. Tolmach, editors: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, ACM, pp. 191–202. doi:10.1145/1631687.1596579

[11] Paolo G. Giarrusso, Yann Régis-Gianas & Philipp Schuster (2019): *Incremental Lambda-Calculus in Cache-Transfer Style - Static Memoization by Program Transformation*. In Luís Caires, editor: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, *Lecture Notes in Computer Science* 11423, Springer, pp. 553–580. doi:10.1007/978-3-030-17184-1_20

[12] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi & Akihiko Takano (1997): *Tupling Calculation Eliminates Multiple Data Traversals*. In: *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming, ICFP'97, Amsterdam, The Netherlands, June 9-11, 1997*, ACM Press, pp. 164–175. doi:10.1145/258948.258964

[13] Rudi Horn, Roly Perera & James Cheney (2018): *Incremental relational lenses*. *PACMPL* 2(ICFP), pp. 74:1–74:30. doi:10.1145/3236769

[14] Scott E. Hudson (1991): *Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update*. *ACM Trans. Program. Lang. Syst.* 13(3), pp. 315–341. doi:10.1145/117009.117012.

[15] Patricia Johann & Janis Voigtländer (2006): *The Impact of seq on Free Theorems-Based Program Transformations*. *Fundam. Inform.* 69(1-2), pp. 63–102.

[16] Neil D. Jones (1996): *An Introduction to Partial Evaluation*. *ACM Computing Surveys* 28(3), pp. 480–503. doi:10.1145/243439.243447

[17] John Launchbury & Tim Sheard (1995): *Warm Fusion: Deriving Build-Catas from Recursive Definitions*. In: *Conference Record of FPCA'95 SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture. La Jolla, CA, USA, 25-28 June 1995*, ACM, pp. 314–323. doi:10.1145/224164.224223

[18] Yanhong A. Liu, Scott D. Stoller & Tim Teitelbaum (1998): *Static Caching for Incremental Computation*. *ACM Trans. Program. Lang. Syst.* 20(3), pp. 546–585. doi:10.1145/291889.291895.

[19] Akimasa Morihata (2018): *Incremental computing with data structures*. *Sci. Comput. Program.* 164, pp. 18–36. doi:10.1016/j.scico.2017.04.001

[20] William Pugh & Tim Teitelbaum (1989): *Incremental Computation via Function Caching*. In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, ACM Press, pp. 315–328. doi:10.1145/75277.75305

[21] G. Ramalingam & Thomas W. Reps (1993): *A Categorized Bibliography on Incremental Computation*. In Mary S. Van Deusen & Bernard Lang, editors: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, ACM Press, pp. 502–510. doi:10.1145/158511.158710.

[22] John C. Reynolds (1983): *Types, Abstraction and Parametric Polymorphism*. *Information Processing* 83, pp. 513–523.

[23] R. S. Sundaresh & Paul Hudak (1991): *Incremental Compilation via Partial Evaluation*. In David S. Wise, editor: *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, ACM Press, pp. 1–13. doi:10.5555/170544

[24] Philip Wadler (1989): *Theorems for Free!* In: *FPCA'89 Conference on Functional Programming Languages and Computer Architecture. Imperial College, London, England, 11-13 September 1989*, ACM, New York, pp. 347–359. doi:10.1145/99370.99404

[25] Daniel M. Yellin & Robert E. Strom (1991): *INC: A Language for Incremental Computations*. *ACM Trans. Program. Lang. Syst.* 13(2), pp. 211–236. doi:10.1145/103135.103137.

[26] Tetsuo Yokoyama, Zhenjiang Hu & Masato Takeichi (2005): *Calculation Rules for Warming-up in Fusion Transformation*. In: *6th Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*, pp. 399–412.