MPRI 2.4
Programmation fonctionnelle et systèmes de types
Programming project

François Pottier

2022–2023

1 Overview

The purpose of this project is to implement (selected parts of) a small compiler which performs **forward-mode and reverse-mode automatic differentiation** for a very small language of arithmetic expressions.

A good introduction to automatic differentiation (AD) can be found in the textbook by Griewank and Walther (2008). Chapters 1–3 are sufficient. Please email François Pottier for an electronic copy (not for redistribution).

The project is based on the paper "You Only Linearize Once: Tangents Transpose to Gradients" (Radul et al., 2023), which will be presented at the conference POPL 2023 in January 2023. The paper defines a calculus named **Linear** and presents three transformations of this calculus into itself:

- forward-mode AD (Section 5);
- unzipping (Section 6);
- transposition (Section 7).

The composition of the three transformations, in the order shown above, yields reverse-mode AD.

The project is organized in **two main tasks**. Task 1 is to implement forward-mode AD. Task 2 is to implement unzipping and transposition, thereby yielding an implementation of reverse-mode AD. A test infrastructure is provided for each of these two implementations of AD.

2 The Surface calculus

Surface is a simple calculus of arithmetic expressions. It is equipped with functions and tuples. The syntax of expressions is as follows:

uop	$::=$	$\cos \mid \sin \mid \exp \mid \ldots$	unary operations
bop	$::=$	$+ \mid - \mid \times \mid / \mid \ldots$	binary operations
e	$::=$	$x \mid \mathsf{let}\, x = e\, \mathsf{in}\, e$	variable; sequencing
		$\mid c \mid uop(u) \mid u\ bop\ u$	arithmetic operations of arity 0, 1, 2
		$\mid (\vec{x}) \mid \mathsf{let}\, (\vec{x}) = e\, \mathsf{in}\, e$	tuple construction and deconstruction
		$\mid f(\vec{x})$	function call
τ	$::=$	`real`	real numbers
		$\mid (\vec{\tau})$	tuples

A program is a sequence of function definitions. A function can take multiple arguments and returns a single result. Functions are not recursive. The language is statically typed. The types τ include the type `real` as well as tuple types $(\vec{\tau})$. See `Surface.ml` for more details.

A lexer and parser for Surface are provided (`SurfaceLexer.mll`, `SurfaceParser.mly`).

A type-checker, an interpreter, and a pretty-printer for Surface are provided (`SurfaceTypeChecker.mli`, `SurfaceInterpreter.mli`, `SurfacePrinter.mli`).

A translation from Surface to Linear is provided (`Surface2Linear.mli`). This transformation is trivial. A Surface variable x is transformed to an unrestricted Linear variable u.

A random generator of well-typed Surface programs is provided (`SurfaceGenerator.ml`). It is used in the test infrastructure.

3 The Linear calculus

Linear is also a calculus of arithmetic expressions. In this calculus, there is a distinction between two classes of variables, namely *unrestricted variables* u and *linear variables* l.

Furthermore, in Linear, an expression returns not just one result, but *a list of results*, which are further separated into *a list of unrestricted results* and *a list of linear results*. Each result is a value, whose type can be `real` or a tuple type. We stress that a list of results is not the same thing as a tuple: for instance, a function that returns one unrestricted result of type (`real`, `real`) is not the same thing as a function that returns two unrestricted results of type `real` and `real`.

It helps to think of an expression visually as a *box* with input and output wires. Each wire is either unrestricted or linear. An input wire is named: it corresponds to an unrestricted variable u or a linear variable l that occurs free in the expression. An output wire is not named, but is implicitly numbered. For instance, if an expression has two unrestricted results and one linear result, then it has two unrestricted output wires (implicitly numbered 0 and 1) and one linear output wire (implicitly numbered 0).

These aspects are made precise by the type discipline presented in Figure 4 of the paper. The source code of the Linear type-checker, in the file `LinearTypeChecker.ml`, can also serve as a reference.

The syntax of Linear expressions is as follows. As a warm-up exercise, we suggest drawing the boxes that correspond to each form of expression, as well as their input and output wires.

uop	$::=$	$\cos \mid \sin \mid \exp \mid \ldots$	unrestricted unary operations
bop	$::=$	$+ \mid - \mid \times \mid / \mid \ldots$	unrestricted binary operations
l			linear variables
u			unrestricted variables
e	$::=$	$(\vec{u}; \vec{l}) \mid \mathsf{let}\ (\vec{u}; \vec{l}) = e\ \mathsf{in}\ e$	return; sequencing
		$\mid c \mid uop(u) \mid u\ bop\ u$	unrestricted fragment
		$\mid 0 \mid l +. l \mid u \times. l \mid \mathsf{drop}\, l \mid \mathsf{dup}\, l$	linear fragment
		$\mid (\vec{u}) \mid \mathsf{let}\ (\vec{u}) = u\ \mathsf{in}\ e$	unrestricted tuples
		$\mid (\vec{l}) \mid \mathsf{let}\ (\vec{l}) = l\ \mathsf{in}\ e$	linear tuples
		$\mid f(\vec{u}; \vec{l})$	function call

As is evident in this syntax, linear variables take part in linear computations only: the linear addition operator $l +. l$ takes two linear variables as arguments, and the scaling operator $u \times. l$ takes a linear variables as its second argument. Addition and scaling are the only two arithmetic operations that take linear variables as arguments. This ensures that *every expression denotes a linear function of its linear inputs to its linear outputs*.

Furthermore, in Linear, every linear variable must be used exactly once. This is the reason why the expressions $\mathsf{drop}\, l$ and $\mathsf{dup}\, l$ are provided. The expression $\mathsf{drop}\, l$ has one input wire, namely l, and zero output wire. The expression $\mathsf{dup}\, l$ has one input wire, namely l, and two output wires. It is usually known as a "fan-out" box, because a single wire is split into two wires.

Thus, linear variables are "linear" both in the sense of mathematics (linear variables serve as inputs of linear functions) and in the sense of computer science (linear variables are used exactly once).

Sometimes, ensuring that every linear variable is used exactly once can be painful. We allow this discipline to be temporarily violated and repair it afterwards (`DupDropInsertion.mli`). For example, forward-mode AD is allowed to produce code where this discipline is violated.

A type-checker, an interpreter, and a pretty-printer for Linear are provided (`LinearTypeChecker.mli`, `LinearInterpreter.mli`, `LinearPrinter.mli`).

Two transformations that hoist and simplify let bindings are provided (`Normalize.mli`, `Simplify.mli`). A transformation that renames every variable to a fresh name is provided (`Freshen.mli`).

A transformation that transforms all linear variables into unrestricted variables is provided (`Forget.mli`). It serves as a first step in the translation of Linear back to Surface (`Linear2Surface.mli`).

Many useful auxiliary functions are provided in the file `LinearHelp.ml`. It is worth spending some time to understand what these functions do; you may need to use some or all of them.

4 Forward-mode AD

Task 1, the implementation of forward-mode AD, takes place in the file `ForwardMode.ml`. The intended effect of this program transformation is informally described in the file `ForwardMode.mli`. See also §6.1 for an example.

This transformation is expected to produce code that computes a Jacobian-Vector Product (JVP). The function `test_forward_mode` in the file `Test.ml`, and the comment that precedes this function, explain what property is expected.

An example is given in §6.1.

One way to test is via the following command:

```
dune exec src/Main.exe -- --forward-mode --test-forward
```

Or, you can edit `main.sh` to test just the forward mode and then type "`make test`". More details about testing are given in §7.1.

5 Reverse-mode AD

Task 2, the implementation of reverse-mode AD, is split in two transformations, unzipping and transposition, which take place in the files `Unzip.ml` and `Transpose.ml`.

The composition of forward-mode AD, unzipping, and transposition is expected to produce code that computes a Vector-Jacobian Product (VJP). The function `test_reverse_mode` in the file `Test.ml`, and the comment that precedes this function, explain what property is expected.

An example is given in §6.2.

One way to test is via the following command:

```
dune exec src/Main.exe -- --reverse-mode --test-reverse
```

Or, you can edit `main.sh` to test the reverse mode and then type "`make test`". These commands test the *combination* of forward-mode AD, unzipping, and transposition, so they cannot be used until all three passes have been implemented.

To test unzipping in isolation, before transposition is working, the following command can be used:

```
dune exec src/Main.exe -- --reverse-mode --test-unzip
```

(For this command to succeed, transposition must not fail. You may need to define `Transpose.transform` as the identity function so as to avoid hitting the exception `NOT YET IMPLEMENTED`.)

More details about testing are given in §7.1.

6 Example

6.1 Forward-mode AD

As a simple example, let us consider the function $f : \mathbb{R}^2 \to \mathbb{R}$ defined by $f(x_0, x_1) = x_0 \times \cos(x_1) + x_0$. This function is defined in Surface as follows:

```
let f (x0 : real, x1 : real) =
  x0 * cos(x1) + x0
```

This program is translated down to Linear as follows:

```
let f (hu0 : real, hu1 : real;) =
  let (hu3 : real;) = cos(hu1) in
  let (hu2 : real;) = hu0 * hu3 in
  hu2 + hu0
```

Then, forward-mode AD (followed with insertion of `dup` and `drop`, normalization, simplification, freshening) produces the following Linear program:

```
let f (hu0 : real, hu1 : real;) =
  let (hu2 : real;) = cos(hu1) in
  let (hu3 : real;) = hu0 * hu2 in
  hu3 + hu0

let df (hu0 : real, hu1 : real; hl0 : real, hl1 : real) =
  let (hu6 : real;) = cos(hu1) in
  let (hu8 : real;) = -1.00 in
  let (hu9 : real;) = sin(hu1) in
  let (hu7 : real;) = hu8 * hu9 in
  let (; hl10 : real) = hu7 *. hl1 in
  let (; hl3 : real, hl4 : real) = dup(hl0) in
  let (hu5 : real;) = hu0 * hu6 in
  let (; hl8 : real) = hu0 *. hl10 in
  let (; hl9 : real) = hu6 *. hl3 in
  let (; hl7 : real) = hl8 +. hl9 in
  let (hu4 : real;) = hu5 + hu0 in
  let (; hl6 : real) = hl7 +. hl4 in
  (hu4; hl6)
```

The function `f` has been preserved, and a new function `df`, which computes both `f` and the differential of `f`, has been constructed.

By translating this Linear program back to Surface, and by performing a compression step (which transforms `let x = e1 in e2` to `e2[e1/x]` when there is at most one occurrence of `x` in `e2`), one obtains the following code:

```
let f (hu0 : real, hu1 : real) =
  hu0 * cos(hu1) + hu0

let df (hu0 : real, hu1 : real, hl0 : real, hl1 : real) =
  let hu6 = cos(hu1) in
  [hu0 * hu6 + hu0, hu0 * (-1.00 * sin(hu1) * hl1) + hu6 * hl0 + hl0]
```

It is not too difficult to check that the results computed by `df` are correct: they are the original result of `f` and the differential of `f`. This correctness property can be expressed in the following manner: if

$$[y, dy] = df(x_0, x_1, dx_0, dx_1)$$

then

$$y = f(x_0, x_1)$$

and

$$dy = Jf(x_0, x_1) \cdot \begin{bmatrix} dx_0 \\ dx_1 \end{bmatrix}$$

where $Jf(x_0, x_1)$ is the Jacobian matrix of f at (x_0, x_1). Here, it is the matrix $[\frac{\partial f}{\partial x_0}(x_0, x_1) \quad \frac{\partial f}{\partial x_1}(x_0, x_1)]$.

It is worth noting that the intermediate result `hu6` is shared. Work is never duplicated, so, up to a constant factor, the cost of `df` is the cost of `f`.

By design of the language Linear, a linear result can depend on an unrestricted variable, but an unrestricted result cannot depend on a linear variable. Thus, there is a one-way dependency between the unrestricted fragment and the linear fragment of a computation. This is exploited during unzipping.

6.2 Reverse-mode AD

Now, let us start again from the Linear program that is produced by forward-mode AD:

```
let f (hu0 : real, hu1 : real;) =
  let (hu2 : real;) = cos(hu1) in
  let (hu3 : real;) = hu0 * hu2 in
  hu3 + hu0

let df (hu0 : real, hu1 : real; hl0 : real, hl1 : real) =
  let (hu6 : real;) = cos(hu1) in
  let (hu8 : real;) = -1.00 in
  let (hu9 : real;) = sin(hu1) in
  let (hu7 : real;) = hu8 * hu9 in
  let (; hl10 : real) = hu7 *. hl1 in
  let (; hl3 : real, hl4 : real) = dup(hl0) in
  let (hu5 : real;) = hu0 * hu6 in
  let (; hl8 : real) = hu0 *. hl10 in
  let (; hl9 : real) = hu6 *. hl3 in
  let (; hl7 : real) = hl8 +. hl9 in
  let (hu4 : real;) = hu5 + hu0 in
  let (; hl6 : real) = hl7 +. hl4 in
  (hu4; hl6)
```

Unzipping transforms the function `df` into two functions, `udf` and `ldf`, which represent the unrestricted fragment and the linear fragment of the computation performed by `df`. Furthermore, the little function `cdf` is a composition of `udf` and `ldf`. The (normalized, simplified) Linear program produced by unzipping is as follows:

```
let f (hu0 : real, hu1 : real;) =
  let (hu2 : real;) = cos(hu1) in
  let (hu3 : real;) = hu0 * hu2 in
  hu3 + hu0

let udf (hu0 : real, hu1 : real;) =
  let (hu2 : real;) = cos(hu1) in
```

```
  let (hu3 : real;) = -1.00 in
  let (hu4 : real;) = sin(hu1) in
  let (hu5 : real;) = hu3 * hu4 in
  let (hu6 : real;) = hu0 * hu2 in
  let (hu7 : real;) = hu6 + hu0 in
  (hu7, hu0, hu2, hu5;)

let ldf (hu0 : real, hu1 : real, hu2 : real; hl0 : real, hl1 : real) =
  let (; hl2 : real) = hu2 *. hl1 in
  let (; hl3 : real, hl4 : real) = dup(hl0) in
  let (; hl5 : real) = hu0 *. hl2 in
  let (; hl6 : real) = hu1 *. hl3 in
  let (; hl7 : real) = hl5 +. hl6 in
  let (; hl8 : real) = hl7 +. hl4 in
  (; hl8)

let cdf (hu0 : real, hu1 : real; hl0 : real, hl1 : real) =
  let (hu2 : real, hu3 : real, hu4 : real, hu5 : real;) = udf(hu0, hu1;) in
  ldf(hu3, hu4, hu5; hl0, hl1)
```

The functions `ldf` and `cdf` have zero unrestricted outputs. Thus, when their unrestricted inputs are fixed, these functions denote linear maps of their linear inputs to their linear outputs.

These functions can be transposed. When an expression is viewed as a matrix, this transformation corresponds to matrix transposition. When an expression is viewed as a data flow network (composed of boxes and wires), this corresponds to reversing the direction of the wires, transforming zero into drop (and vice-versa), and transforming addition into dup (and vice-versa).

The result is the following Linear program:

```
let f (hu0 : real, hu1 : real;) =
  let (hu2 : real;) = cos(hu1) in
  let (hu3 : real;) = hu0 * hu2 in
  hu3 + hu0

let udf (hu0 : real, hu1 : real;) =
  let (hu2 : real;) = cos(hu1) in
  let (hu3 : real;) = -1.00 in
  let (hu4 : real;) = sin(hu1) in
  let (hu5 : real;) = hu3 * hu4 in
  let (hu6 : real;) = hu0 * hu2 in
  let (hu7 : real;) = hu6 + hu0 in
  (hu7, hu0, hu2, hu5;)

let ldf (hu0 : real, hu1 : real, hu2 : real; hl0 : real, hl1 : real) =
  let (; hl2 : real) = hu2 *. hl1 in
  let (; hl3 : real, hl4 : real) = dup(hl0) in
  let (; hl5 : real) = hu0 *. hl2 in
  let (; hl6 : real) = hu1 *. hl3 in
  let (; hl7 : real) = hl5 +. hl6 in
  let (; hl8 : real) = hl7 +. hl4 in
  (; hl8)

let tldf (hu0 : real, hu1 : real, hu2 : real; hl0 : real) =
  let (; hl16 : real, hl17 : real) = dup(hl0) in
  let (; hl18 : real, hl19 : real) = dup(hl17) in
```

```
  let (; hl15 : real) = hu1 *. hl19 in
  let (; hl11 : real) = hu0 *. hl18 in
  let (; hl7 : real) = hl15 +. hl16 in
  let (; hl3 : real) = hu2 *. hl11 in
  (; hl7, hl3)

let cdf (hu0 : real, hu1 : real; hl0 : real, hl1 : real) =
  let (hu2 : real, hu3 : real, hu4 : real, hu5 : real;) = udf(hu0, hu1;) in
  ldf(hu3, hu4, hu5; hl0, hl1)

let tcdf (hu0 : real, hu1 : real; hl0 : real) =
  let (hu2 : real, hu3 : real, hu4 : real, hu5 : real;) = udf(hu0, hu1;) in
  tldf(hu3, hu4, hu5; hl0)
```

Once translated back to Surface (and compressed), the program is:

```
let f (hu0 : real, hu1 : real) =
  hu0 * cos(hu1) + hu0

let udf (hu0 : real, hu1 : real) =
  let hu2 = cos(hu1) in
  [hu0 * hu2 + hu0, hu0, hu2, -1.00 * sin(hu1)]

let ldf (hu0 : real, hu1 : real, hu2 : real, hl0 : real, hl1 : real) =
  hu0 * (hu2 * hl1) + hu1 * hl0 + hl0

let tldf (hu0 : real, hu1 : real, hu2 : real, hl0 : real) =
  [hu1 * hl0 + hl0, hu2 * (hu0 * hl0)]

let cdf (hu0 : real, hu1 : real, hl0 : real, hl1 : real) =
  let [hu2, hu3, hu4, hu5] = udf(hu0, hu1) in
  ldf(hu3, hu4, hu5, hl0, hl1)

let tcdf (hu0 : real, hu1 : real, hl0 : real) =
  let [hu2, hu3, hu4, hu5] = udf(hu0, hu1) in
  tldf(hu3, hu4, hu5, hl0)
```

The correctness of the function `tcdf` can be expressed as follows. If $[dx_0, dx_1] = tcdf(x_0, x_1, dy)$ then

$$[dx_0, dx_1] = [dy] \cdot Jf(x_0, x_1)$$

or, equivalently,

$$\begin{bmatrix} dx_0 \\ dx_1 \end{bmatrix} = Jf(x_0, x_1)^\top \cdot [dy].$$

7 Practical details

7.1 Building and using the compiler

The directory `src/` contains the source files. The file `Main.ml` contains the compiler's main program. Depending on the command line arguments, this compiler can be applied either to a specific Surface program or to a large number of randomly-generated Surface programs. Furthermore, it is optionally capable of running tests to check that the transformed code seems to be correct.

Typing "`make`" in the root directory produces the executable program `_build/default/src/Main.exe`. By default, this program expects to receive on its command line the name of a file that contains a Surface program.

The tiny script `main.sh` builds and runs this executable program. For instance, typing "`./main.sh test/inputs/simple.s`" runs the compiler on the Surface program stored in `test/inputs/simple.s`. You can edit this script in order to add or remove some command line arguments. For instance, in the beginning, you will want to remove `--test-forward` and `--test-reverse`. You can restore them once you are ready to test your implementations of forward-mode AD and reverse-mode AD.

Typing "`./main.sh --help`" shows the command line options accepted by the compiler. In particular,

- `--forward-mode` causes the compiler to stop after performing forward-mode AD. You should use this option in the beginning and until your implementation of reverse-mode AD is ready.
- `--reverse-mode` causes the compiler to perform forward-mode AD, unzipping, and transposition, thereby obtaining reverse-mode AD. It is enabled by default.
- `--show-surface` causes the compiler to print the Surface program (on the standard output channel) at various stages of the transformation. This can be useful while debugging.
- `--show-linear` causes the compiler to print the Linear program (on the standard output channel) at various stages of the transformation. This can be useful while debugging.
- `--test-forward` causes the compiler to test the Surface program that is obtained as a result of forward-mode AD. The program is tested by checking that the equations given at the end of §6.1 appear to hold. These equations are evaluated at randomly-chosen input values. All computations are performed using the arbitrary-precision real numbers provided by the library creal. Because division is not defined (therefore not differentiable) at 0, if a test involves a division by a number that seems close to 0, then this test is aborted and ignored.
- `--test-reverse` causes the compiler to test the Surface program that is obtained as a result of reverse-mode AD. The program is tested by checking that the equations given at the end of §6.2 appear to hold.
- `--test-unzip` causes the compiler to test the Linear program that is obtained as a result of unzipping.

When invoked without a file name, the compiler automatically applies itself to a number of randomly-generated Surface programs of increasing sizes, up to a certain size, which is specified on the command line via `--budget <int>`. The command "`make auto`" exploits this to perform random tests up to size 100.

The command "`make human`" applies the compiler to each of the Surface programs stored in the directory `test/inputs/`. This directory currently contains only a few Surface programs. You are encouraged to write more. Writing several very small Surface programs that exercise various features, one at a time, should help you debug your compiler.

The command "`make test`" combines "`make human`" and "`make auto`".

Task 1 is to complete the file `ForwardMode.ml` so that "`make test`" succeeds, with `--forward-mode` and `--test-forward` enabled.

Task 2 is to additionally complete the files `Unzip.ml` and `Transpose.ml` so that "`make test`" succeeds, with `--reverse-mode` and `--test-forward` and `--test-reverse` enabled.

8 Required software

To use the sources that we provide, you need OCaml 4.14 and Menhir. You also need the library `pprint`. Assuming that you have installed OCaml via `opam`, these components can be installed by typing "`opam update && opam install menhir pprint`".

9 Evaluation

Assignments will be evaluated by a combination of:

- **Testing**. Your compiler will be tested with the input programs that we provide (make sure that "`make test`" succeeds!) and with additional input programs.
- **Reading**. We will browse through your source code and evaluate its correctness and elegance.

The **correctness** of your code matters; its performance does not. It is acceptable to favor clarity over efficiency: for instance, it is permitted to call the functions `flv` or `fuv` (which compute the free variables of a term) without worrying about the cost of these calls, even if (as a result of these calls) the complexity of a program transformation becomes quadratic instead of linear.

10 Extra credit

For extra credit, or just for fun, you may wish to go beyond what is strictly requested. What to do is up to you. Here are some suggestions of things to do. Beware: we do not know exactly how difficult or time-consuming these extensions are.

- Extend Surface and Linear with more primitive arithmetic operations.
- Implement a measure of the runtime cost of an expression, and check (experimentally) that each transformation preserves the cost (perhaps up to certain constant factor), as claimed in Radul *et al.*'s paper.
- Test each of the three main transformations (forward-mode AD; unzipping; transposition) in isolation. This requires a generator of Linear programs.
- At the level of the Surface language, perform algebraic simplifications by identifying and simplifying expressions such as `0 + e`, `0 * e`, `1 * e`, and so on.
- At the level of the Surface language, implement a form of common subexpression elimination (CSE).

11 What to send

When you are done, please send to François Pottier a `.tar.gz` archive containing your completed programming project. The archive should contain a single directory `mpri-2.4-projet-2022-2023`.

Please include a file **README.md** to describe what you have achieved (task 1; task 2; extra tasks). You are welcome to provide explanations (in French or in English) about your solution or about the difficulties that you have encountered.

12 Deadline

Please send your project on or before **Monday, February 13, 2023**.

References

Andreas Griewank and Andrea Walther. *Evaluating derivatives – principles and techniques of algorithmic differentiation, Second Edition*. SIAM, 2008. URL https://doi.org/10.1137/1.9780898717761.

Alexey Radul, Adam Paszke, Roy Frostig, Matthew J. Johnson, and Dougal Maclaurin. You only linearize once: Tangents transpose to gradients. *Proceedings of the ACM on Programming Languages*, 7(POPL), January 2023. URL https://arxiv.org/abs/2204.10923.