Rose: Composable Autodiff for the Interactive Web

Sam Estep
Software and Societal Systems Department, Carnegie Mellon University, Pittsburgh, PA, USA

Wode Ni
Software and Societal Systems Department, Carnegie Mellon University, Pittsburgh, PA, USA

Raven Rothkopf
Barnard College, Columbia University, New York, NY, USA

Joshua Sunshine
Software and Societal Systems Department, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Reverse-mode automatic differentiation (autodiff) has been popularized by deep learning, but its ability to compute gradients is also valuable for interactive use cases such as bidirectional computer-aided design, embedded physics simulations, visualizing causal inference, and more. Unfortunately, the web is ill-served by existing autodiff frameworks, which use autodiff strategies that perform poorly on dynamic scalar programs, and pull in heavy dependencies that would result in unacceptable webpage sizes. This work introduces Rose, a lightweight autodiff framework for the web using a new hybrid approach to reverse-mode autodiff, blending conventional tracing and transformation techniques in a way that uses the host language for metaprogramming while also allowing the programmer to explicitly define reusable functions that comprise a larger differentiable computation. We demonstrate the value of the Rose design by porting two differentiable physics simulations, and evaluate its performance on an optimization-based diagramming application, showing Rose outperforming the state-of-the-art in web-based autodiff by multiple orders of magnitude.

2012 ACM Subject Classification Software and its engineering → Compilers; Information systems → Web applications; Software and its engineering → Domain specific languages; Computing methodologies → Symbolic and algebraic manipulation; Software and its engineering → Formal language definitions; General and reference → Performance; Computing methodologies → Neural networks; General and reference → General conference proceedings

Keywords and phrases Automatic differentiation, differentiable programming, compilers, web

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.15

Related Version *Full Version*: `https://arxiv.org/abs/2402.17743` [10]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.10.2.7`
Software (Source Code): `https://github.com/rose-lang/rose`
archived at `swh:1:dir:bc091e3b381dd680e389e14729302848edd7d0aa`

Funding This material is based upon work supported by the Aqueduct Foundation and by National Science Foundation under Grant Numbers 1910264, 2119007, and 2150217.

Acknowledgements Thanks to Adam Paszke for corresponding about JAX and Dex. The Rose icons were created by Aaron Weiss; we use them via the CC BY 4.0 license.

1 Introduction

The web provides a platform for interactive experiences with a uniquely low barrier to usage, because the browser obviates the need for software installation by automatically downloading JavaScript code and running it securely on the client. Industry tools like Google Slides [13] and Figma [11], as well as experimental tools like Sketch-n-Sketch [17]

and Penrose [54], leverage this platform to enable authoring of visual media. Interactive explainers like Red Blob Games [34], Bartosz Ciechanowski's work [9], and Bret Victor's "Explorable Explanations" [50] use the web to help people understand complicated ideas in depth, building up a causal mental model by using sliders to manipulate values and immediately see the effects.

Many of these interactions are fairly simple: often the user just drags a slider back and forth, manipulating a parameter in, for instance, a small physical simulation. But there is room for much richer interactions. An early exploration was g9.js [52], which lets the user directly drag around visual shapes, and automatically propagates those changes backward to modify the underlying parameters driving the visualization. This idea of *bidirectional editing* or *bidirectional transformations* [3] is quite powerful. Some more recent work [8] has explored bidirectional editing in computer-aided design (CAD) via *automatic differentiation (autodiff)*, a technique for efficiently computing derivatives of numerical functions. Autodiff has become popularized over the past few years by machine learning (ML) frameworks such as TensorFlow [2], PyTorch [32], and JAX [12].

Autodiff engines built for ML are focused on high throughput for functions composed of a relatively small number of operations on relatively large tensors. They use *reverse-mode* autodiff to compute the gradient of a *loss function* in an iterative loop, using a numerical optimization algorithm like stochastic gradient descent or Adam [25] to update the parameters of the ML model until the loss value is sufficiently reduced. The loss function is usually chosen to be parallelizable on a GPU. These characteristics do not generally apply to other domains, which often involve *scalar programs* [22] on which overhead between operations would dominate any tensor-level attempts at parallelism.

For scalar programs, program transformation tools are far more appropriate; examples include Tapenade [16] for Fortran and C, Zygote [21] for Julia [7], and Enzyme [29] for LLVM [26]. These tools consume and emit code that deals directly with scalars, reducing expressiveness limitations and operation-level overhead at the expense of the parallelism that ML frameworks gain by specializing to tensor operations. They typically leverage heavy modern compiler technology, using various optimization passes on the program after (and sometimes before) differentiation.

But in the interactive web setting, none of these existing points in the design space are appropriate. We are operating in an environment that is

- **dynamic:** the goal is to let the user author content or build up their mental causal model, by (either implicitly through direct manipulation or explicitly through writing code) specifying a differentiable function themselves. The autodiff engine must operate *online*, differentiating functions directly inside of the user's browser.
- **bandwidth-constrained:** because of the no-install model described above, any JavaScript or WebAssembly code used for autodiff must be shipped over the network to the user's browser. Heavyweight components are unacceptable because their bandwidth requirements would exacerbate page load times beyond the user's patience.
- **latency-constrained:** the system must respond to the user's manipulation of the differentiable function definition, at interactive speed. What we care about is not just the performance of the synthesized gradient, but the sum of that latency with the latency to synthesize the gradient in the first place; *quantitative* differences like a slow "compilation" step result in *qualitative* differences in the kinds of interaction possible.

All existing autodiff tools, including web-focused tools like TensorFlow.js [46], fall short on at least one of these constraints: they impose large constant factors for scalar programs, or depend on giant codebases that are difficult to package for the web and result in large bundles, or are too slow to use in an interactive setting, or some combination of these.

To address this gap, we present **Rose**,[1] a scalar-focused autodiff engine for the web that achieves fast compilation time and high generated code performance in a small bundle. As we will describe in Section 4, Rose is a hybrid autodiff system [22] which blends together techniques from tracing and program transformation before emitting WebAssembly [15]. Unlike prior program transformation approaches that take advantage of heavyweight compiler optimization toolchains, we produce efficient Rose IR before differentiating by using JavaScript as a *metaprogramming environment*, somewhat similar to tracing in popular ML frameworks. But unlike prior tracing approaches that expand all operations into one large graph, we reduce generated code size and thus compilation time by allowing the user to explicitly define *composable functions* that can be nested and reused. Our primary contributions are as follows:

- We establish the importance of, and constraints imposed by, the interactive web setting, and articulate how those constraints translate to system requirements for autodiff in such a setting.
- We describe a novel system design that satisfies these requirements using a careful blend of tracing with program transformation.
- We present experiments demonstrating how each component of our design is key to achieving the requirements we have laid out.
- We publish Rose, an open-source software package implementing this design for others to consume and build upon.

The rest of this paper is structured as follows. In Section 2 we give relevant general mathematical background information about autodiff; we introduce a running example that we then implement in Section 3, which discusses Rose from a user perspective. Then Section 4 discusses the novel design of Rose, focusing on the high level because that is our more interesting contribution, but also describing some low-level details of autodiff for the curious reader. Section 5 describes the experiments we conducted with results showing why this design is key to achieving our design goals. Finally we discuss related work in Section 6, and conclude with future work in Section 7.

2 Background

To illustrate the basic ideas of reverse-mode autodiff, we'll walk through the classic example of using gradients to perform linear regression via least-squares optimization. Nothing in this section is new. Almost all the content here can be found in standard textbooks for calculus, linear algebra, and convex optimization; all the rest can be found in the research literature on autodiff [39].

Suppose we have n measurements $\mathbf{y} \in \mathbb{R}^n$ of a dependent variable, each corresponding to one of n data points $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^m$. These data can be assembled into a matrix $\mathbf{X} \in \mathbb{R}^{n \times (m+1)}$ defined by

$$\mathbf{X} = \begin{bmatrix} 1 & \mathbf{x}_1^\top \\ \vdots & \vdots \\ 1 & \mathbf{x}_n^\top \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{nm} \end{bmatrix}.$$

We would like to predict the dependent variable as a linear function $\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\beta}$ where the parameters $\boldsymbol{\beta} \in \mathbb{R}^{m+1}$ are chosen to minimize the sum of squares of the errors $\boldsymbol{\varepsilon} = \mathbf{y} - \hat{\mathbf{y}}$.

[1] Not to be confused with the ROSE (all caps) compiler infrastructure. [38]

That is, one would like to find an optimal solution to the optimization problem

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^{m+1}} f(\boldsymbol{\beta}) \quad \text{where} \quad f(\boldsymbol{\beta}) = \|\boldsymbol{\varepsilon}\|^2 = \|\mathbf{y}-\hat{\mathbf{y}}\|^2 = \|\mathbf{y}-\mathbf{X}\boldsymbol{\beta}\|^2.$$

Applying convex optimization theory here is standard so we won't belabor it, but because this particular f is differentiable, convex, and smooth, there exists a step size $\eta > 0$ such that if we start with any $\boldsymbol{\beta}_0 \in \mathbb{R}^{m+1}$ and iteratively compute $\boldsymbol{\beta}_{i+1} = \boldsymbol{\beta}_i - \eta\nabla f(\boldsymbol{\beta}_i)$, then

$$f(\boldsymbol{\beta}_{i+1}) \le f(\boldsymbol{\beta}_i)\ \forall i \in \mathbb{N}, \quad \text{and} \quad \lim_{i\to\infty} f(\boldsymbol{\beta}_i) \le f(\boldsymbol{\beta})\ \forall \boldsymbol{\beta} \in \mathbb{R}^{m+1}.$$

This is gradient descent. Crucially, it depends on being able to compute the gradient ∇f.

2.1 The vector-Jacobian product

As briefly mentioned in Section 1, reverse-mode autodiff is a general method for computing gradients, which takes in an algorithm to compute a function, and returns an algorithm to compute its gradient. Unlike other approaches to compute derivatives, the power of autodiff lies in its *compositionality* and *efficiency*: we naturally express functions by composing together smaller functions. If we possess an algorithm that computes a given function with a given time complexity, reverse-mode autodiff gives us an algorithm to compute its derivative with the same time complexity, in a way that can be directly composed with derivatives for other functions. For example, in least-squares we compose together three functions

$$\begin{aligned} \xi &: \mathbb{R}^{m+1} \to \mathbb{R}^n & \varphi &: \mathbb{R}^n \to \mathbb{R}^n & \psi &: \mathbb{R}^n \to \mathbb{R} \\ \xi(\boldsymbol{\beta}) &= \mathbf{X}\boldsymbol{\beta} & \varphi(\hat{\mathbf{y}}) &= \mathbf{y}-\hat{\mathbf{y}} & \psi(\boldsymbol{\varepsilon}) &= \|\boldsymbol{\varepsilon}\|^2 \end{aligned}$$

to form $f = \psi \circ \varphi \circ \xi$. Clearly, the gradient itself is insufficient to express ∇f compositionally, because ξ and φ are not scalar-valued and thus do not have gradients. So we must first have a compositional definition for the derivative.

The *Jacobian* of a function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ is the matrix-valued function $\mathbf{J_f} : \mathbb{R}^n \to \mathbb{R}^{m\times n}$ defined by

$$\mathbf{J_f}(\mathbf{x}) = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}.$$

From this, if we fix $\mathbf{x} \in \mathbb{R}^n$ then we can define a function $\mathrm{vjp}^{\mathbf{x}}_{\mathbf{f}} : \mathbb{R}^{1\times m} \to \mathbb{R}^{1\times n}$, called the *vector-Jacobian product (VJP)*, operating on row vectors called *adjoints* by $\mathrm{vjp}^{\mathbf{x}}_{\mathbf{f}}(\ddot{\mathbf{y}}) = \ddot{\mathbf{y}}\mathbf{J_f}(\mathbf{x})$. In the special case of $m = 1$ we can recover the gradient by $\nabla\mathbf{f}(\mathbf{x}) = \mathrm{vjp}^{\mathbf{x}}_{\mathbf{f}}(1)^\top$, but unlike the gradient, this notion of a derivative actually composes. For instance, if we also have $\mathbf{g} : \mathbb{R}^m \to \mathbb{R}^p$, then

$$\mathrm{vjp}^{\mathbf{x}}_{\mathbf{g}\circ\mathbf{f}} = \mathrm{vjp}^{\mathbf{x}}_{\mathbf{f}} \circ \mathrm{vjp}^{\mathbf{y}}_{\mathbf{g}} \quad \text{where} \quad \mathbf{y} = \mathbf{f}(\mathbf{x}).$$

This is the chain rule for reverse-mode autodiff, so-called because it composes $\mathrm{vjp}_{\mathbf{f}}$ and $\mathrm{vjp}_{\mathbf{g}}$ in the reverse order of how $\mathbf{f}$ and $\mathbf{g}$ themselves were originally composed. As shown here, computing the derivative $\mathrm{vjp}_{\mathbf{g}\circ\mathbf{f}}$ depends on computing the original function $\mathbf{f}$ itself, so in practice the term "VJP" is sometimes actually used to refer to the mapping

$$\begin{aligned} &\mathbb{R}^n \to \mathbb{R}^m \times (\mathbb{R}^{1\times m} \to \mathbb{R}^{1\times n}) \\ &\quad \mathbf{x} \mapsto (\mathbf{f}(\mathbf{x}), \mathrm{vjp}^{\mathbf{x}}_{\mathbf{f}}) \end{aligned}$$

that returns both the output of the original function – which we call a *primal* value to contrast it with the VJP's adjoints – and the VJP function.

We'll provide a more general set of composable VJPs in Sections 4.1 and 4.3, but for this example, we can derive the VJPs

$$\begin{aligned} &\operatorname{vjp}_\xi^{\boldsymbol{\beta}} : \mathbb{R}^{1\times n} \to \mathbb{R}^{1\times(m+1)} && \operatorname{vjp}_\varphi^{\hat{\mathbf{y}}} : \mathbb{R}^{1\times n} \to \mathbb{R}^{1\times n} && \operatorname{vjp}_\psi^{\boldsymbol{\varepsilon}} : \mathbb{R} \to \mathbb{R}^{1\times n} \\ &\quad \operatorname{vjp}_\xi^{\boldsymbol{\beta}}(\bar{\hat{\mathbf{y}}}) = \bar{\hat{\mathbf{y}}}\mathbf{X} && \quad \operatorname{vjp}_\varphi^{\hat{\mathbf{y}}}(\bar{\boldsymbol{\varepsilon}}) = -\bar{\boldsymbol{\varepsilon}} && \quad \operatorname{vjp}_\psi^{\boldsymbol{\varepsilon}}(\bar{\sigma}) = 2\bar{\sigma}\boldsymbol{\varepsilon}^\top \end{aligned}$$

of the functions we decomposed earlier. One key property to notice here is that, given algorithms to compute ξ, φ, and ψ, we immediately have algorithms to compute vjp_ξ, $\operatorname{vjp}_\varphi$, and vjp_ψ, respectively, with the same time complexities. For instance, ξ is $O(mn)$ with naïve matrix multiplication, as is vjp_ξ. This is not too surprising, since that is also the same time complexity as directly computing and multiplying by $\mathbf{J}_\xi$. But the time complexity for both φ and ψ is $O(n)$, as are the formulas given above for $\operatorname{vjp}_\varphi$ and vjp_ψ, in contrast to the $O(n^2)$ cost of naïvely computing $\mathbf{J}_\varphi$ or $\mathbf{J}_\psi$. This is because those Jacobians are *sparse*; the ability of reverse-mode autodiff to preserve time complexity in the presence of sparse Jacobians is called the *cheap gradient principle*.

In any case, from these simpler VJPs we can easily compose the gradient of f as

$$\begin{aligned} \nabla f(\boldsymbol{\beta}) = \operatorname{vjp}_f^{\boldsymbol{\beta}}(1)^\top = \operatorname{vjp}_\xi^{\boldsymbol{\beta}}(\operatorname{vjp}_\varphi^{\hat{\mathbf{y}}}(\operatorname{vjp}_\psi^{\boldsymbol{\varepsilon}}(1)))^\top = (-2\boldsymbol{\varepsilon}^\top\mathbf{X})^\top = 2\mathbf{X}^\top(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) \\ \text{where} \quad \hat{\mathbf{y}} = \varepsilon(\boldsymbol{\beta}) = \mathbf{X}\boldsymbol{\beta} \quad \text{and} \quad \boldsymbol{\varepsilon} = \varphi(\hat{\mathbf{y}}) = \mathbf{y} - \hat{\mathbf{y}}. \end{aligned}$$

2.2 The Jacobian-vector product

We've talked about the VJP used for reverse-mode autodiff, which is the more useful for optimization, but also the more challenging to implement and specify. Rose allows users to specify custom derivatives for reasons described in Section 3.2, so to reduce user burden, we allow those custom derivatives to be defined using the simpler *Jacobian-vector product (JVP)* instead of the VJP. In Section 4.1 we'll discuss the actual program transformation used to derive the VJP from the JVP [39], but here we first lay out the mathematical groundwork.

For $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, the JVP of $\mathbf{f}$ at $\mathbf{x} \in \mathbb{R}^n$ is a function $\operatorname{jvp}_\mathbf{f}^\mathbf{x} : \mathbb{R}^n \to \mathbb{R}^m$ that operates on column vectors called *tangents* by $\operatorname{jvp}_\mathbf{f}^\mathbf{x}(\dot{\mathbf{x}}) = \mathbf{J}_\mathbf{f}(\mathbf{x})\dot{\mathbf{x}}$. In the special case of $n = 1$ we can recover the ordinary derivative by $\mathbf{f}'(\mathbf{x}) = \operatorname{jvp}_\mathbf{f}^\mathbf{x}(1)$. But more generally, given $\mathbf{g} : \mathbb{R}^m \to \mathbb{R}^p$ we also have a chain rule

$$\operatorname{jvp}_{\mathbf{g}\circ\mathbf{f}}^\mathbf{x} = \operatorname{jvp}_\mathbf{g}^\mathbf{y} \circ \operatorname{jvp}_\mathbf{f}^\mathbf{x} \quad \text{where} \quad \mathbf{y} = \mathbf{f}(\mathbf{x})$$

that composes $\operatorname{jvp}_\mathbf{f}$ and $\operatorname{jvp}_\mathbf{g}$ in the same order as $\mathbf{f}$ and $\mathbf{g}$, hence the name "forward-mode." This is much simpler computationally, because while reverse-mode needed to compose together the VJPs themselves until the end when it could call them with the final adjoint value, forward-mode can simply use the mapping

$$\begin{aligned} &\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^m \\ &\quad (\mathbf{x}, \dot{\mathbf{x}}) \mapsto (\mathbf{f}(\mathbf{x}), \operatorname{jvp}_\mathbf{f}^\mathbf{x}(\dot{\mathbf{x}})) \end{aligned}$$

to package together the primal and tangent values.

The judgment is somewhat subjective, but we invite the reader to decide for themselves whether the VJPs derived earlier or these JVPs

$$\begin{aligned} &\operatorname{jvp}_\xi^{\boldsymbol{\beta}} : \mathbb{R}^{m+1} \to \mathbb{R}^n && \operatorname{jvp}_\varphi^{\hat{\mathbf{y}}} : \mathbb{R}^n \to \mathbb{R}^n && \operatorname{jvp}_\psi^{\boldsymbol{\varepsilon}} : \mathbb{R}^n \to \mathbb{R} \\ &\quad \operatorname{jvp}_\xi^{\boldsymbol{\beta}}(\dot{\boldsymbol{\beta}}) = \mathbf{X}\dot{\boldsymbol{\beta}} && \quad \operatorname{jvp}_\varphi^{\hat{\mathbf{y}}}(\dot{\hat{\mathbf{y}}}) = -\dot{\hat{\mathbf{y}}} && \quad \operatorname{jvp}_\psi^{\boldsymbol{\varepsilon}}(\dot{\boldsymbol{\varepsilon}}) = 2\boldsymbol{\varepsilon}^\top\dot{\boldsymbol{\varepsilon}} \end{aligned}$$

are closer to the original definitions of ξ, φ, and ψ.

When packaging together $\mathbf{f}$ with $\text{jvp}_{\mathbf{f}}$, it is convenient to represent the pair $(\mathbf{x}, \dot{\mathbf{x}}) \in \mathbb{R}^n \times \mathbb{R}^n$ as the single vector $\bar{\mathbf{x}} = \mathbf{x} + \dot{\mathbf{x}}\varepsilon \in \mathbb{D}^n$, making use of the infinitesimal element ε in the *dual numbers* defined by the commutative algebra $\mathbb{D} = \{a + b\varepsilon \mid a, b \in \mathbb{R}\}$ where $\varepsilon^2 = 0$. For dual numbers $\bar{x} = x + \dot{x}\varepsilon \in \mathbb{D}$ and $\bar{y} = y + \dot{y}\varepsilon \in \mathbb{D}$, the arithmetic operations

$$\begin{aligned}
\bar{x} + \bar{y} &= x + y + (\dot{x} + \dot{y})\varepsilon \\
\bar{x} - \bar{y} &= x - y + (\dot{x} - \dot{y})\varepsilon \\
\bar{x}\bar{y} &= xy + (\dot{x}y + x\dot{y})\varepsilon \\
\frac{\bar{x}}{\bar{y}} &= \frac{x}{y} + \frac{\dot{x}y - x\dot{y}}{y^2}\varepsilon \qquad \text{where} \quad y \neq 0
\end{aligned}$$

correspond directly to the JVPs of the corresponding arithmetic operations on real numbers. This allows us to define what we'll call the *dual JVP*

$$\begin{aligned}
\overline{\text{jvp}}_{\mathbf{f}} &: \mathbb{D}^n \to \mathbb{D}^m \\
\overline{\text{jvp}}_{\mathbf{f}}(\mathbf{x} + \dot{\mathbf{x}}\varepsilon) &= \mathbf{f}(\mathbf{x}) + \text{jvp}^{\mathbf{x}}_{\mathbf{f}}(\dot{\mathbf{x}})\varepsilon
\end{aligned}$$

which operates on column vectors of dual numbers. In this framing, specifying the JVPs of our three functions

$$\begin{aligned}
\overline{\text{jvp}}_{\xi} &: \mathbb{D}^{m+1} \to \mathbb{D}^n \\
\overline{\text{jvp}}_{\xi}(\bar{\boldsymbol{\beta}}) &= \mathbf{X}\bar{\boldsymbol{\beta}}
\end{aligned}
\qquad
\begin{aligned}
\overline{\text{jvp}}_{\varphi} &: \mathbb{D}^n \to \mathbb{D}^n \\
\overline{\text{jvp}}_{\varphi}(\bar{\hat{\mathbf{y}}}) &= \mathbf{y} - \bar{\hat{\mathbf{y}}}
\end{aligned}
\qquad
\begin{aligned}
\overline{\text{jvp}}_{\psi} &: \mathbb{D}^n \to \mathbb{D} \\
\overline{\text{jvp}}_{\psi}(\bar{\boldsymbol{\varepsilon}}) &= \bar{\boldsymbol{\varepsilon}}^\top \bar{\boldsymbol{\varepsilon}}
\end{aligned}$$

becomes almost trivial. So when we refer to the JVP in Rose, we're always talking about this dual JVP, not the raw JVP which would operate on real numbers.

2.3 The Hessian

While gradient descent is a first-order method that only makes use of the gradient, other optimization techniques such as Newton's method also need the *Hessian*, which turns $f : \mathbb{R}^n \to \mathbb{R}$ into a matrix-valued function $\mathbf{H}_f : \mathbb{R}^n \to \mathbb{R}^{n \times n}$ yielding all the second-order partial derivatives of f at a given point. The JVP and VJP can be used together to define the Hessian, which is actually just the Jacobian of the gradient; that is, $\mathbf{H}_f = \mathbf{J}_{\nabla f}$. We already know how to use the VJP of a function to compute its gradient. To compute the Jacobian, we just need to observe that the i^{th} row of the Jacobian is equal to the JVP applied to the i^{th} basis element $\mathbf{e}_i$ of $\mathbb{R}^n$:

$$\mathbf{H}_f(\mathbf{x}) = \begin{bmatrix} \text{jvp}^{\mathbf{x}}_{\nabla f}(\mathbf{e}_1) & \cdots & \text{jvp}^{\mathbf{x}}_{\nabla f}(\mathbf{e}_n) \end{bmatrix} \quad \text{where} \quad \nabla f(\mathbf{x}) = \text{vjp}^{\mathbf{x}}_{f}(1)^\top$$

3 Using Rose

Now that we've discussed the mathematical ideas behind Rose, in this section we'll describe how programmers use Rose to understand how it fits into the context described in Section 1. Then in Section 4 we'll describe our design that blends together tracing with program transformation to fit into this context.

Listing 1 shows a comprehensive end-to-end example using Rose to perform gradient descent to solve the linear regression problem laid out in Section 2. This entire example is JavaScript code, which makes use of Rose as a library; lines 1–2 use standard JavaScript `import` statements to pull in definitions from the Rose library. Lines 3–10 use arithmetic primitives from Rose to implement the loss function from Section 2:

$$f(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{m} x_{ij}\beta_j \right)^2$$

```
1  import { Real, Vec, compile, fn, struct, vjp } from "rose";
2  import { add, mul, sub, sum } from "rose";
3  const sqr = (x) => mul(x, x);
4  const leastSquares = ({ m, n }) => fn([{
5    x: Vec(n, Vec(m, Real)), y: Vec(n, Real),
6    b0: Real, b: Vec(m, Real),
7  }], Real, ({ x, y, b0, b }) => sum(n, (i) => {
8    const yHat = add(b0, sum(m, (j) => mul(x[i][j], b[j])));
9    return sqr(sub(y[i], yHat));
10 }));
11 const linearRegression = async ({ x, y, eta }) => {
12   const [n, m] = [y.length, x[0].length];
13   const Beta = struct({ b0: Real, b: Vec(m, Real) });
14   const f = leastSquares({ m, n });
15   const g = fn([Beta], Real, ({ b0, b }) => f({ x, y, b0, b }));
16   const h = fn([Beta], Beta, (beta) => vjp(g)(beta).grad(1));
17   const grad = await compile(h);
18   let b0 = 0; let b = Array(m).fill(0);
19   for (;;) {
20     const beta = grad({ b0, b });
21     const bb0 = b0 - eta * beta.b0;
22     const bb = b.map((bi, i) => bi - eta * beta.b[i]);
23     if (bb0 === b0 && bb.every((bi, i) => bi === b[i])) break;
24     b0 = bb0; b = bb;
25   }
26   return { b0, b };
27 };
28 console.log(await linearRegression({ eta: 1e-4,
29   x: [[10],[8],[13],[9],[11],[14],[6],[4],[12],[7],[5]],
30   y: [8.04,6.95,7.58,8.81,8.33,9.96,7.24,4.26,10.84,4.82,5.68],
31 }));
```

Listing 1 Using Rose to do linear regression on the first dataset in Anscombe's quartet [4].

Lines 4–7 define the type of the `leastSquares` function, which takes as input $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^m$, as well as $\mathbf{y} \in \mathbb{R}^n$ and $\boldsymbol{\beta} \in \mathbb{R}^{m+1}$, and returns a scalar. Lines 11–27 wrap around that low-level function to provide a high-level method to perform linear regression, which is then used by lines 28–31. More specifically, line 13 uses Rose to define the type of $\boldsymbol{\beta} \in \mathbb{R}^{m+1}$, a vector with m elements plus an additional scalar bias. Line 14 uses the earlier `leastSquares` definition to get a Rose function for the specific $m, n \in \mathbb{N}$ needed, and line 15 partially applies that function using the provided $\mathbf{x}$ and $\mathbf{y}$ values as constants. Line 16 uses Rose's builtin `vjp` function to take the gradient of that partially applied loss function. While mathematically it can be useful to distinguish column and row vectors, the Rose library does not, so the VJP directly produces the gradient. Line 17 compiles that gradient function to WebAssembly, producing a function that can be called with concrete standard JavaScript values instead of abstract Rose values. Finally, lines 18–26 perform gradient descent by calling the compiled `grad` function.

As demonstrated by the above examples, Rose works by letting the user define *differentiable functions* of the form `fn(paramTypes, returnType, body)`. We'll discuss this more in Section 4, but the high-level idea from a user perspective is that normal JavaScript functions like the one defined on line 3 of Listing 1 roughly correspond to what one might think of as *macros* that get expanded on demand to produce code, while Rose functions defined using `fn` correspond to traditional functions, and must be well-typed. One could define that `sqr` "macro" as a function instead as

```
const sqr = fn([Real], Real, (x) => mul(x, x));
```

where the difference is that the body of this function would then be traced only *once* immediately when it is defined, as opposed to the `sqr` "macro" defined in Listing 1 which gets expanded/traced every time it is called (which in this case happens to only be on line 9). This ability for users to choose between these two ways to define functions is a key feature in the novel design of Rose, and we will see in Section 5 that it is crucial to achieving the design goals for interactive differentiable web applications that we laid out in Section 1.

3.1 Opaque functions

The above example works well using only builtin arithmetic functions, but it's not interactive; let's look at an interactive example that takes advantage of Rose's ability to call predefined JavaScript functions and define custom derivatives for them. The Rose project website[2] has an interactive widget displaying the local quadratic approximation to the function $(x, y) \mapsto x^y$, allowing a user to drag the point around to see how the shape of the local quadratic approximation shifts; see the full version of this paper [10] for a screenshot. The page also allows the user to modify the mathematical expression defining the function, causing Rose to immediately re-derive the gradient and Hessian, and compile the new function to WebAssembly. For brevity we omit the code to generate the user interface, and instead focus on how one would use Rose to calculate the first and second derivatives used to visualize the quadratic approximation.

Listing 2 shows a Rose program calculating the value, gradient, and Hessian of the power function at $x = 2$ and $y = 3$. Line 2 imports a power function with a custom derivative, as we'll describe shortly. Lines 3–4 define type aliases for $\mathbb{R}^2$ and $\mathbb{R}^{2\times 2}$, respectively. Rose types are simply JavaScript values, so type aliases are defined using `const` in the same way as any other JavaScript value.

Recall that the vector-Jacobian product (VJP) introduced in Section 2.1 swaps the domain and codomain from the original function. In addition, JavaScript only allows functions to return one argument. Therefore to take the VJP of a Rose function, that function must have only have one parameter. So, line 5 wraps the `pow` function to take a single vector argument rather than two scalar arguments, allowing it to be passed to Rose's `vjp` function. Just as we discussed in Section 2.1, we compute the gradient by passing in a value of `1`.

Lines 7–10 then use the gradient `g` of `f` to compute its Hessian by differentiating once more. Line 8 runs the forward pass for the Hessian just once and saves all necessary intermediate values, after which line 9 runs the backward pass twice with the two basis vectors to compute the full Hessian matrix. Lines 11–14 wrap these three functions into a single function that calls all three and returns the results in a structured form. Finally, line 15 compiles that function to WebAssembly, and line 16 calls it at the point $(2, 3)$.

[2] `https://rosejs.dev/`

```
import { Real, Vec, compile, fn, vjp } from "rose";
import { pow } from "./pow.js";
const R2 = Vec(2, Real);
const R22 = Vec(2, R2);
const f = fn([R2], Real, ([x, y]) => pow(x, y));
const g = fn([R2], R2, (v) => vjp(f)(v).grad(1));
const h = fn([R2], R22, (v) => {
  const { grad } = vjp(g)(v);
  return [grad([1, 0]), grad([0, 1])];
});
const all = fn([Real, Real], { z: Real, g: R2, h: R22 }, (x, y) => {
  const v = [x, y];
  return { z: f(v), g: g(v), h: h(v) };
});
const compiled = await compile(all);
console.log(compiled(2, 3));
```

Listing 2 An example Rose program.

```
import { Dual, Real, add, div, mul, fn, opaque } from "rose";
const log = opaque([Real], Real, Math.log);
log.jvp = fn([Dual], Dual, ({re:x,du:dx}) => {
  return { re: log(x), du: div(dx, x) };
});
export const pow = opaque([Real, Real], Real, Math.pow);
pow.jvp = fn([Dual, Dual], Dual, ({re:x,du:dx}, {re:y,du:dy}) => {
  const z = pow(x, y);
  const dw = add(mul(dx, div(y, x)), mul(dy, log(x)));
  return { re: z, du: mul(dw, z) };
});
```

Listing 3 The contents of `pow.js` defining a differentiable power function.

Listing 3 shows how the `pow` function can be defined to call JavaScript's existing `Math.pow` function. Because Rose cannot see the definition of this `opaque` function, it must be given a definition for its derivative. Lines 3–5 use Rose define the logarithm's dual JVP, which is automatically transposed to produce a VJP as we'll describe in Section 4. Specifically, the signature of this function takes the original `log` function and maps every instance of the `Real` numbers to become the `Dual` numbers we introduced in Section 2.2. In this case, the returned tangent is given by the familiar rule $\frac{\mathrm{d}}{\mathrm{d}x} \ln x = \frac{1}{x}$ from calculus.

Similarly, lines 6–11 define the power function along with its derivative. Note that, while these two functions use `opaque` to define their bodies, they define their derivatives via `fn`, the same as the Rose functions we discussed in earlier sections. This means that only the first forward derivative needs to be provided. Since the body of this first derivative is transparent to Rose, the reverse derivative and any higher derivatives can be computed automatically.

```
import { Dual, Real, fn, mul, neg, opaque } from "rose";
const sin = opaque([Real], Real, Math.sin);
const cos = opaque([Real], Real, Math.cos);
sin.jvp = fn([Dual], Dual, ({ re: x, du: dx }) => {
  return { re: sin(x), du: mul(dx, cos(x)) };
});
cos.jvp = fn([Dual], Dual, ({ re: x, du: dx }) => {
  return { re: cos(x), du: mul(dx, neg(sin(x))) };
});
```

Listing 4 Definitions of sine and cosine functions with custom derivatives.

```
import { Dual, Real, fn, opaque } from "rose";
const print = opaque([Real], Real, (x) => {
  console.log(x);
  return x;
});
print.jvp = fn([Dual], Dual, (z) => {
  print(z.re);
  return z;
});
```

Listing 5 A custom Rose function for print debugging.

3.2 Custom derivatives

Rose lets users define custom derivatives for functions that depend on each other, as in Listing 4. The user can also define their own functions to use with `opaque`. For instance, one might want to define a `print` function for debugging purposes as in Listing 5, but Rose cannot look inside the definition of `print`; by setting `print.jvp`, the programmer can tell Rose that the derivative of this function should similarly perform its side effect and otherwise act like the identity function.

The other situation is when Rose has automatically constructed a derivative for a function, but that derivative is unstable or otherwise exhibits some undesirable property. Rose allows the user to set a custom derivative for any function, not just `opaque` ones. For instance, by default the derivative of the square root function tends to infinity as the argument approaches zero, which causes problems if it is ever called with a zero argument. To prevent this exploding-gradient problem, we sometimes use a square root with a clamped derivative, as in Listing 6.

In all of these examples, notice that the user only needs to specify the JVP, and not the VJP; this is true even if they later decide to use any of these functions in a VJP context, because Rose uses transposition (described in Section 4.3) to automatically construct a VJP from the JVP. A large part of the value of autodiff is that it ensures that the derivative remains in sync with the primal function by construction. Similarly, if we can also assist in keeping the forward-mode and reverse-mode derivatives in sync when one of them must be manually specified, this is a significant benefit for user ergonomics and maintainability.

```
import * as rose from "rose";
import { Dual, Real, div, fn, gt, mul, select } from "rose";

const max = (x: Real, y: Real) =>
  select(gt(x, y), Real, x, y);

const sqrt = fn([Real], Real, (x) => rose.sqrt(x));
sqrt.jvp = fn([Dual], Dual, ({ re: x, du: dx }) => {
  const y = sqrt(x);
  const dy = mul(dx, div(1 / 2, max(1e-5, y)));
  return { re: y, du: dy };
});
```

Listing 6 A custom derivative of the square root function to avoid exploding gradients.

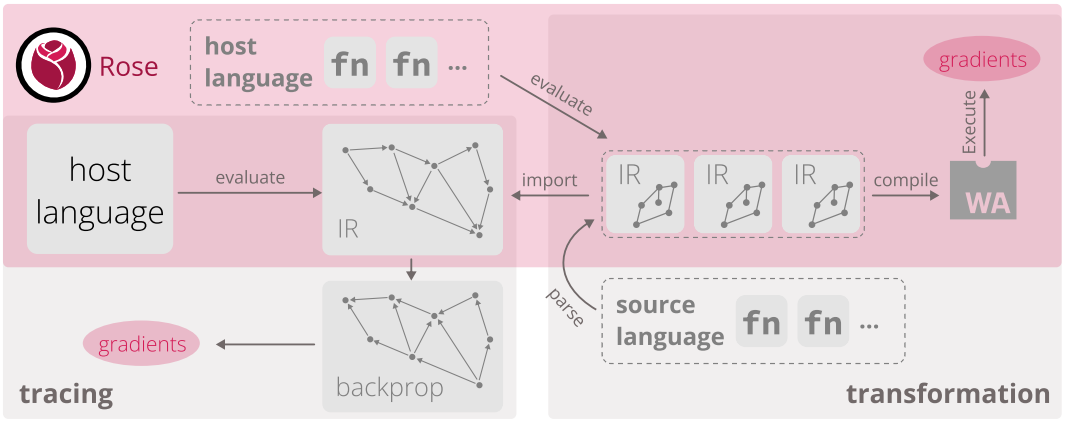

Figure 1 With **Rose**, the programmer uses the host language for *metaprogramming* like in **tracing** autodiff, and defines composable *functions* like in **transformation** autodiff.

4 Design

The previous section described a user experience that hints at the design of Rose; this section makes that design explicit. Autodiff frameworks typically fall into the two categories of *tracing* and *transformation*, with some *hybrid* frameworks combining aspects from the two extremes [22]. Rose chooses a specific point in the space of possible hybrid approaches, as diagrammed in Figure 1. To highlight the novel aspects of this design, we will first briefly describe tracing and transformation autodiff; then we will explain how Rose uses parts of both approaches to provide an autodiff engine for the setting described in Section 1.

In tracing autodiff, the programmer writes code in what we call the *host language* (e.g. Python [2, 32]). They use an autodiff library to construct differentiable scalars, vectors, matrices, and other tensors. Each such tensor can be thought of as a single node in a large *computation graph*. Then the programmer calls functions from that autodiff library which take in differentiable tensors and produce more differentiable tensors. Each such function call creates edges in the computation graph from the arguments to the return value. Eventually, a final differentiable scalar value is produced. The programmer calls a special procedure from the autodiff framework, passing in this final scalar value. The autodiff framework traces

backward through the computation graph in reverse topological order, attaching gradient values to every node as it goes. The programmer can then use the autodiff framework to access the gradient value attached to any node as they please.

In transformation autodiff, the programmer writes code in what we will call the *source language* (e.g. Fortran [16] or Julia [20]). The compiler frontend parses and typechecks this source language to convert it to an *intermediate representation (IR)*. This first step typically preserves most of the structure of what the programmer wrote, modulo source formatting. In particular, function definitions and calls in the source text are typically represented as a call graph in an imperative IR, or as lambda terms in a functional IR. Then, the autodiff framework takes in this IR to compute the function the programmer wrote, and emits transformed IR to compute that function along with its gradient. Crucially, this autodiff transformation preserves the asymptotic size of the original program: if the IR representation of the original program has size n, then the size of the transformed program to compute gradients is $O(n)$. Then, the compiler backend converts the IR to an executable binary like normal, which can be run to compute the desired gradients.

Figure 1 shows how Rose combines these two approaches. Like tracing, Rose lets the programmer write in a host language they are familiar with: JavaScript, in this case. And like tracing, the programmer is free to use all the features of the host language to describe the shape of their computation. But unlike tracing, and more like transformation, Rose also allows the programmer to explicitly define multiple functions that can be composed together to form a larger computation. Unlike transformation, the programmer's code does not get directly parsed and typechecked to produce the IR; the IR is instead produced by symbolic evaluation like in tracing. But like transformation, the IR can include control flow and function calls, which get explicitly transformed by autodiff rather than being effectively erased as in tracing. And like transformation, the resulting differentiated IR is compiled to WebAssembly [15] that can then be repeatedly executed to compute gradients for the same program.

By restricting our IR to not allow recursive functions, we are able to use a compilation strategy similar to destination-passing style [40] to avoid the cost of sophisticated memory management, increasing performance. This strategy not to deallocate memory while computing gradients is justified by the way that reverse-mode autodiff generally needs to retain intermediate values, as described in Section 4.3. Importantly, while Rose IR does not allow recursion, the programmer can freely express recursive computations by using the host language for metaprogramming, as we will later discuss in Sections 4.4 and 5.4.2.

In the following subsections, we will discuss the Rose IR at a theoretical level and explain the autodiff and transposition [39] program transformations which we use to compute gradients; then in Section 4.4 we will step back again to discuss how the programmer interacts with this IR indirectly through Rose as a library. All inference rules can be found in the full version of this paper [10].

4.1 Rose intermediate representation

Figure 2 shows the abstract syntax for the Rose IR. It is a first-order functional language with non-mutable array types, and a "reference" or "accumulator" type constructor [33], written $\&\tau$. The full version of this paper [10] gives the typing rules for the Rose IR. These are all fairly standard, except for the rules for type constraints κ. We will explain these less common features of the language in the context of a specific example. Section 4.3 will demonstrate the need for accumulators in more detail, but at a high level, they arise naturally because the backward pass of reverse-mode autodiff essentially runs the program backward in

$$
\begin{array}{rcl}
m, n & \in & \mathbb{Z}_{\geq 0} \\
c & \in & \mathbb{R} \\
\kappa & ::= & \texttt{Type} \mid \texttt{Value} \mid \texttt{Index} \\
\tau & ::= & t \mid \texttt{()} \mid \texttt{Bool} \mid \texttt{Real} \mid n \mid \texttt{\&}\tau \mid \texttt{[}\tau\texttt{]}\tau \mid \texttt{(}\tau\texttt{,}\ \tau\texttt{)} \mid \texttt{<}\underline{t\texttt{:}\ \kappa}\texttt{>(}\underline{\tau}\texttt{)}\ \texttt{->}\ \tau \\
\ominus & ::= & \neg \mid - \mid \texttt{abs} \mid \texttt{sgn} \mid \texttt{ceil} \mid \texttt{floor} \mid \texttt{trunc} \mid \texttt{sqrt} \\
\oplus & ::= & \wedge \mid \vee \mid \texttt{iff} \mid \texttt{xor} \mid \neq \mid < \mid \leq \mid = \mid > \mid \geq \mid + \mid - \mid \times \mid \div \\
e & ::= & \texttt{()} \mid \texttt{true} \mid \texttt{false} \mid c \mid n \mid \texttt{[}\underline{x}\texttt{]} \mid \texttt{(}x\texttt{,}\ x\texttt{)} \mid \ominus x \mid x \oplus x \mid x\ \texttt{?}\ x\ \texttt{:}\ x \mid x\ \texttt{+=}\ x \\
& & \mid x\texttt{[}x\texttt{]} \mid \texttt{fst}\ x \mid \texttt{snd}\ x \mid \texttt{\&}x\texttt{[}x\texttt{]} \mid \texttt{\&fst}\ x \mid \texttt{\&snd}\ x \mid f\texttt{<}\underline{\tau}\texttt{>(}\underline{x}\texttt{)} \mid \texttt{[for}\ x\texttt{:}\ \tau\texttt{,}\ b\texttt{]} \\
& & \mid \texttt{accum}\ x\ \texttt{from}\ x\ \texttt{in}\ b \\
b & ::= & x \mid \texttt{let}\ x\texttt{:}\ \tau\ \texttt{=}\ e\ \texttt{in}\ b \\
d & ::= & \texttt{def}\ f\texttt{<}\underline{t\texttt{:}\ \kappa}\texttt{>(}\underline{x\texttt{:}\ \tau}\texttt{):}\ \tau\ \texttt{=}\ b
\end{array}
$$

Figure 2 Abstract syntax for Rose IR.

```
1  def sum<n: Index>(v: [n]Real): Real =
2    let z: Real = 0.0 in
3    let t: (Real, [n]()) =
4      accum a from z in
5        [for i: n,
6          let x: Real = v[i] in
7          let u: () = a += x in
8          u
9        ]
10     in
11   let y: Real = fst t in
12   y
```

Listing 7 A Rose IR function to compute the sum of a vector of real numbers.

time, so reads become accumulation, and writes become reads. Having a type which restricts mutation in this way makes it easier to ensure correctness without introducing additional data dependencies that hinder compiler optimization.

Rose users write JavaScript, so prior examples have been written in JavaScript. However, in this subsection, and Sections 4.2 and 4.3, we describe the IR and so the examples are written in Rose IR. To make this distinction clear, we will format IR examples differently by putting them inside grey boxes.

Listing 7 computes the sum of a vector of real numbers. Line 1 says that the function is generic over the size of the array, where the size is represented as a type `n` with the `Index` constraint. The three type constraints in Rose IR are ordered as a hierarchy `Index` <: `Value` <: `Type`. Semantically, `Type` refers to any type that can be stored in a variable; the only types that don't satisfy `Type` are function types, since Rose IR is first-order. `Value` is contrasted with reference types: that is, types τ : `Type` satisfy the `Value` constraint unless they are of the form $\tau = \texttt{\&}\tau'$. Only `Value` types can be used in other type constructors, so for example, a reference cannot be stored as an element of an array. Finally, the `Index` constraint marks types that can be used as the index type of an array; the only Rose IR types that satisfy this constraint are those of the form $n \in \mathbb{N}$, which correspond to the value set $\{0, \ldots, n-1\}$.

Line 2 defines a local called `z` of type `Real` with the value `0.0`. This is used by the `accum` block on lines 4 to 10. This block serves as the scope for the variable `a` of type `&Real`, because `z` is of type `Real`. The variable `a` is in scope for lines 5 to 9 and goes out of scope on line 10. As mentioned above, this is an *accumulator* type: it represents a container holding a value of type `Real`, but that value cannot be read, and can only be accumulated. In other words, there is no operation of type `&Real` → `Real`. But, given a value of type `Real`, one can use the `+=` operation to accumulate into the value contained in `a`.

With this accumulator in hand, lines 5 to 9 execute. These lines are an *array constructor* with index type `n`, as shown on line 5. The value type of this array constructor is the unit type `()` so its resulting array type `[n]()` holds no data; its sole importance comes from the side effect it performs on line 7. The result is that every element of `v` gets accumulated into the value of `a`, so after this `for` block executes, the inner value of `a` is equal to the sum of all the values from `v`. Again, though, `a` cannot actually be read.

Finally, after the body of the `accum` block executes, it returns the inner value from `a`, along with the value that was returned from the `accum` block body itself, which is of type `[n]()` as mentioned before. These two items are packaged together into a tuple `t` of type `(Real, [n]())`. Because no accumulator type can be part of a tuple, this prevents `a` itself from escaping from the `accum` body, so it is guaranteed to be inaccessible after the `accum` block executes. Thus, every accumulator of type `&`τ starts as accumulate-only, and then when it goes out of scope, its value is guaranteed to be inaccessible except as a "decayed" read-only value of type τ. These semantics may seem unintuitive, but they turn out to perfectly model the mapping from forward-mode to reverse-mode autodiff described in Section 4.3.

That function definition was quite verbose, as it strictly adhered to the syntax from Figure 2 for clarity. In the remainder of this section, we will allow ourselves syntactic sugar to write expressions in places where the strict syntax requires variable names, with the understanding that these could be desugared by introducing intermediate `let` bindings:

```
def sum<n: Index>(v: [n]Real): Real =
  fst (accum a from 0.0 in [for i: n, a += v[i]])
```

4.2 Forward-mode autodiff

As we showed in Section 2.2, the forward-mode JVP can often be easier to specify than the reverse-mode VJP, which is why we allow the user to specify custom derivatives using the JVP as described in Section 3.2. So, we decompose reverse-mode autodiff into two parts [39], first applying forward-mode autodiff as we will describe shortly, and then applying a second transformation which we will describe in Section 4.3. The full version of this paper [10] lists inference rules for forward-mode autodiff of Rose IR. Specifically, these rules can be used to transform a function f into a function f' that computes the *dual JVP* of f, where the dual numbers are represented in Rose IR by the tuple type `Dual` = `(Real, Real)`.

Consider this function, which assumes that `sin` : `<>(Real) -> Real` already exists:

```
def f(u: Real): Real =
  let v: Real = sin(u) in
  let w: Real = -v in
  w
```

We assume that we are already given a dual JVP for `sin`. For instance, if the function `cos` : `<>(Real) -> Real` also exists, then $\overline{\text{jvp}}_{\texttt{sin}}$: `<>(Dual) -> Dual` might be:

```
def jvp_sin((x, dx): Dual): Dual = (sin(x), dx * cos(x))
```

Then, by applying the inference rules we have laid out, we get $\overline{\text{jvp}}_{\texttt{f}}$: `<>(Dual) -> Dual`:

```
def jvp_f(u: Dual): Dual =
  let v: Dual = jvp_sin(u) in
  let (v_re, v_du) = v in
  let w: Dual = (-v_re, -v_du) in
  w
```

All the rules for forward-mode autodiff are straightforward and quite standard, so we will not dwell on them here. Now we move on to the more complicated transformation, which maps from forward-mode autodiff to reverse-mode autodiff.

4.3 Transposition

The name "transposition" comes from the fact that the JVP and VJP can be thought of as transposes of each other, in the sense of transposing a matrix. Recall that, for $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ and $\mathbf{x} \in \mathbb{R}^n$, we have

$$\begin{aligned} \text{jvp}^{\mathbf{x}}_{\mathbf{f}} &: \mathbb{R}^n \to \mathbb{R}^m & \qquad \text{vjp}^{\mathbf{x}}_{\mathbf{f}} &: \mathbb{R}^{1\times m} \to \mathbb{R}^{1\times n} \\ \text{jvp}^{\mathbf{x}}_{\mathbf{f}}(\dot{\mathbf{x}}) &= \mathbf{J}_{\mathbf{f}}(\mathbf{x})\dot{\mathbf{x}} & \qquad \text{vjp}^{\mathbf{x}}_{\mathbf{f}}(\ddot{\mathbf{y}}) &= \ddot{\mathbf{y}}\mathbf{J}_{\mathbf{f}}(\mathbf{x}) \end{aligned}$$

which both make use of the Jacobian matrix $\mathbf{J}_{\mathbf{f}}(\mathbf{x}) : \mathbb{R}^{m\times n}$. So, $\mathbf{J}_{\mathbf{f}}(\mathbf{x})$ is the matrix of the linear transformation $\text{jvp}^{\mathbf{x}}_{\mathbf{f}}$. But then, observe that $\mathbf{J}_{\mathbf{f}}(\mathbf{x})^\top : \mathbb{R}^{n\times m}$ is the matrix of the linear transformation

$$\begin{aligned} &\mathbb{R}^m \to \mathbb{R}^n \\ &\quad \dot{\mathbf{y}} \mapsto \mathbf{J}_{\mathbf{f}}(\mathbf{x})^\top \dot{\mathbf{y}} = (\dot{\mathbf{y}}^\top \mathbf{J}_{\mathbf{f}}(\mathbf{x}))^\top = \text{vjp}^{\mathbf{x}}_{\mathbf{f}}(\dot{\mathbf{y}}^\top)^\top. \end{aligned}$$

If we had an explicit dense representation of $\mathbf{J}_{\mathbf{f}}$ then it would be trivial to transpose. But we don't; instead, we have an implicit, potentially sparse, representation of $\mathbf{J}_{\mathbf{f}}$ in the form of the dual JVP function $\overline{\text{jvp}}_{\mathbf{f}}$. The full version of this paper [10] lists inference rules to transpose the linear derivative represented by the dual JVP in Rose IR. This transformation is considerably more complicated than the initial transformation to do forward-mode autodiff. We will walk through this transformation using the example from Section 4.2, leaving a more systematic exposition to the prior literature [33, 39] on which our approach was based.

Now, back to the example. By strictly following the inference rules we have laid out, we end up with Listing 8. As is common with program transformations like this, the resulting code is highly redundant; via a straightforward optimization pass that we omit here for brevity, we obtain the following:

```
def fwd_f((u, _): Dual): (Dual, Tape_sin) =
  let ((v, _), t) = fwd_sin((u, 0)) in
  let w = -v in
  ((w, 0), t)
def bwd_f(ddu: &Dual, (_, dw): Dual, t: Tape_sin): () =
  let dv = -dw in
  bwd_sin(ddu, (0, dv), t)
```

This example hints at the fact that the infinitesimal part is always zero for dual numbers representing primal values, and the real part is always zero for dual numbers representing adjoint values. Thus, in our actual system, in the aforementioned optimization pass we also translate the `Dual` type back to `Real`, essentially reversing our replacement of `Real` with `Dual` from Section 4.2:

```
type Tf = (Dual,(Dual,(Tape_sin,(Real,(Real,(Dual,()))))))
def fwd_f(u: Dual): (Dual, Tf) =
  let (v, t0) = fwd_sin(u) in
  let (v_re, v_du) = v in
  let w_re = -v_re in
  let w_du = 0 in
  let w = (w_re, v_re) in
  (w, (v, (t0, (w_re, (w_du, (w, ()))))))
def bwd_f(ddu: &Dual, dw0: Dual, t: Tf): () =
  let (v, (t0, t1)) = t in
  let (dv, ()) = accum ddv from v in (
    let v_re = fst v in
    let ddv_re = &fst ddv in
    let v_du = snd v in
    let ddv_du = &snd ddv in
    let (w_re, t2) = t1 in
    let (dw_re, ()) = accum ddw_re from w_re in (
      let (w_du, t3) = t2 in
      let (dw_du, ()) = accum ddw_du from w_du in (
        let (w, t4) = t3 in
        let (dw1, ()) = accum ddw from w in (
          ddw += dw0
        ) in
        let (dw1_re, dw1_du) = dw1 in
        ddw_re += dw1_re ;
        ddw_du += dw1_du
      ) in
      ddv_du += -dw_du
    ) in
    ()
  ) in
  bwd_sin(ddu, dv, t0)
```

Listing 8 Strict transposition of the function `f` from Section 4.2.

```
def fwd_f(u: Real): (Real, Tape_sin) =
  let (v, t) = fwd_sin(u) in
  let w = -v in
  (w, t)
def bwd_f(ddu: &Real, dw: Real, t: Tape_sin): () =
  let dv = -dw in
  bwd_sin(ddu, dv, t)
```

Thus concludes the transposition of `f`. Similarly we can also transpose `jvp_sin`:

```
def fwd_sin(x: Real): (Real, Real) =
  (sin(x), cos(x))
def bwd_sin(ddx: &Real, dy: Real, z: Tape_sin): () =
  ddx += dy * z
```

```
import { Real, add, div, fn } from "rose";
const f = (x) =>
  fn([Real], Real, (y) => div(x, y));
const g = (y) =>
  fn([Real], Real, (x) => div(x, y));
let [f0, f1, f2] = [f(5), f(7), f(5)];
fn([Real], Real, (z) => add(f1(z), f1(z)));
```

```
def f0(y: Real): Real =
  5 / y
def f1(x: Real): Real =
  7 / y
def f2(x: Real): Real =
  5 / y
def h(z: Real): Real =
  f1(z) + f1(z)
```

Figure 3 JavaScript code (left) that produces Rose IR (right) when evaluated.

4.4 Metaprogramming

We have described the right-hand side of Figure 1; now all that remains in this section is to describe the "evaluate" step on the left-hand side. Figure 3 shows a simple example of how evaluating JavaScript code produces Rose IR. In this example, we have two JavaScript functions `f` and `g`, each of which uses `fn` to construct and return a Rose function when called. We call `f` three times and never call `g`, so three Rose IR functions resulted from the single instance of `fn` in the source of `f`, and zero Rose IR functions resulted from the instance of `fn` in the source of `g`. Then, we call `fn` one more time, and in the body of that `fn`, we call `f1` twice. Note that those calls to `f1` remain in the resulting IR; Rose does not inline calls, in contrast to standard tracing autodiff frameworks.

5 Evaluation

As discussed in Section 4, Rose is characterized by three primary design choices:
D1 Allow users to define and compose custom functions using `fn`.
D2 Use program transformation to compile to WebAssembly.
D3 Use tracing to allow metaprogramming in JavaScript.

These design choices are motivated by Rose's role as a toolkit for building interactive, differentiable web applications. The dynamic bandwidth- and latency-constrained environment of web browsers poses significant constraints on the size and speed of Rose. In addition to good performance, Rose also needs to be expressive and flexible enough for the end user to build web applications in a myriad of domains.

In this section, we evaluate Rose both quantitatively and qualitatively by integrating Rose into three web-based applications for diagramming, physical simulation, and reinforcement learning (Section 5.1). In subsequent subsections, we report on Rose's performance (Sections 5.2 and 5.3) and discuss qualitative observations of how Rose's design choices impact its expressiveness and flexibility (Section 5.4).

5.1 Benchmark and applications

To the best of our knowledge, there are no benchmark suites for evaluating autodiff performance generally [41], let alone web-based autodiff of scalar programs. Further, measuring performance on just a benchmark would limit our ability to qualitatively evaluate the expressiveness of Rose in real-world applications. Therefore, we opted to find applications in which autodiff plays a central role, and re-implement the autodiff module or the entire application using Rose. We believe this approach gives us better ecological validity (i.e. the realism of our evaluation setup) and potentially leads to a rich source of examples. Our

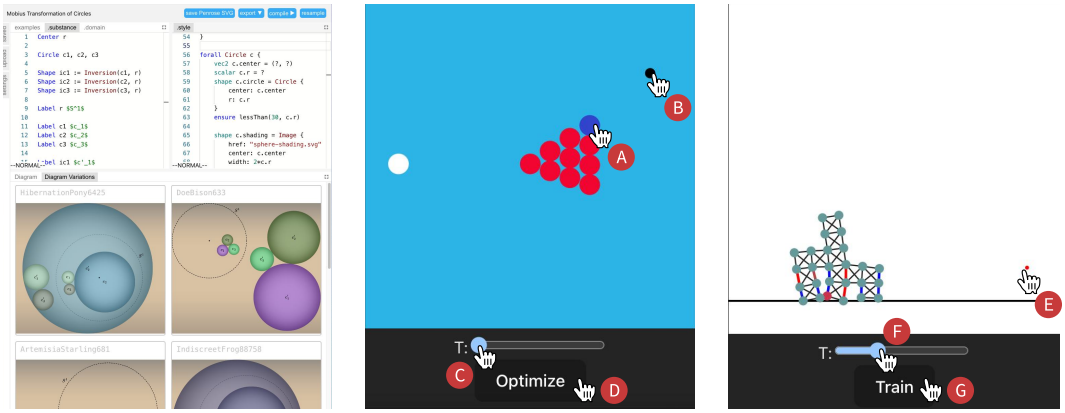

Figure 4 Three web-based applications re-implemented with Rose. **Left:** the Penrose IDE. **Middle:** billiards simulator that optimizes D for cue ball angle and speed such that the object ball A reaches the target B. **Right:** mass-spring robot controlled by a neural net trained G with a designated goal D. Both simulations can be replayed by dragging the sliders at any point D and G.

search resulted in two frameworks that are rich sources of applications and benchmarks: **Penrose** [54], a web-based diagramming framework; and **Taichi** [19], a Python library for high-performance parallel programming. The two frameworks also have sufficiently different settings that together they illustrate the flexibility of Rose.

Penrose allows the user to specify a diagram by constructing a custom numerical optimization problem in a DSL called Style, then runs a numerical solver to rearrange the shapes in the diagram until it finds a local minimum. Optimizing the layout of these diagrams involves defining and differentiating a wide range of mathematical operations on scalars, from simple operations like finding the distance between points to more sophisticated calculations like Minkowski addition, KL divergence, and silhouette points. The main application of Penrose is a web-based IDE (Figure 4, left), where users live-edit programs to produce layout-optimized diagrams. Importantly, the framework uses TensorFlow.js for autodiff and ships with 173 "registry" diagrams for performance testing, each of which was compiled to a unique differentiable computation. Therefore, we deemed it as a suitable target for performance comparison with TensorFlow.js, the closest baseline to which we can compare Rose's performance. This set of registry diagrams is quite diverse, comprising a total of 97 unique Style programs which are preprocessed by the Penrose compiler frontend before being passed to Rose.

Many applications of Taichi involve differentiable programming and DiffTaichi [18] presents a few example Python applications that combines physical simulation and neural networks. We used Rose to implement and augment two such differentiable simulations from Taichi: `billiards` and `robot` (Figure 4, middle and right). The `billiards` example is a differentiable simulation of pool combination shots. The program simulates rigid body collisions between a cue ball and object balls. Leveraging the differentiability of the simulation, a gradient descent optimizer solves for the initial position and velocity of the cue ball to send a designated object ball to a target position. The `robot` example simulates a robot made of a mass-spring system, where springs are actuated to move the robot towards a goal position. A neural network controller is trained on simulator gradients to update the spring actuation magnitude over time. In both cases, the simulation is run to completion, remembering intermediate computations along the way, and then autodiff is used to run back through the simulation in reverse to compute gradients for the initial state.

All three applications were implemented using Rose's JavaScript binding. They all run in major browsers such as Safari and Chrome. To showcase the benefits of running in the web browser, we added interactive features to the Taichi applications (Figure 4). For instance, the Taichi version of `billiards` is a command-line application that outputs a series of static images based on hard-coded parameters for the choice of the object ball and goal position. The Rose version allows the user to interactively explore the simulator by selecting the object ball (Figure 4A), moving the goal position (Figure 4B), optimizing the cue ball position (Figure 4D), and re-playing the simulation (Figure 4C).

5.2 Size

Similar to TensorFlow.js, Rose is a client-side JavaScript package that is typically bundled with the rest of a web application and delivered over the internet. To run in web browsers, Rose needs to be comparable or smaller then similar packages such as TensorFlow.js. As a baseline, a common TensorFlow.js distribution, `@tensorflow/tfjs-core` version 4.18.0 (latest at time of writing), is 479.98 kB after minification (85.88 kB after gzip). We publish Rose via the npm package `rose`,[3] which is at version 0.4.10 at time of writing. The Rose WebAssembly binary size [5] is 168.51 kB (63.97 kB after gzip), and the JavaScript wrapper layer is 31.77 kB after minification (8.74 kB after gzip). For a more extreme comparison, there are projects that package heavy compiler infrastructure like LLVM to WebAssembly [47], but those produce binaries on the order of a hundred megabytes, causing unacceptable load times for end users. Another reference point is the Taichi package on PyPI, which is about 50–80 megabytes depending on the platform. It is unclear how difficult it would be to package Taichi to run in the browser using Pyodide [37], which is 6.4 megabytes and whose authors claim to be 3×–5× slower than native Python.

5.3 Performance

To compare with the TensorFlow.js baseline for execution performance, we replaced the Penrose TensorFlow.js-based autodiff engine with one written in Rose and ran both versions on the benchmark of 173 diagrams (Section 5.1). We measured the amount of time it took for each autodiff engine to perform any necessary compilation, plus the time taken by the Penrose L-BFGS [27] optimization engine to converge on each diagram. We specifically include the time it takes for Rose to do autodiff, transposition, and WebAssembly compilation, despite the fact that TensorFlow.js does not have an analogous compilation step. On the surface this puts Rose at a disadvantage, but fast compilation time is essential when constructing Rose functions dynamically in a user-facing web application, as Penrose does.

Figure 5 shows the results.[4] The quartiles for the ratio of TensorFlow.js optimization time to Rose optimization time were 37×, 173×, and 598×. These results show that Rose provides an enormous advantage over TensorFlow.js (the state-of-the-art for autodiff on the web) for scalar programs like those found in Penrose diagrams. Because these numbers include both compile time and optimization time, the results demonstrate the end-to-end performance of Rose.

[3] `https://www.npmjs.com/package/rose`
[4] All numbers we report in this section were measured in the V8 JavaScript engine (used in both Chrome and Node.js) on a 2020 MacBook Pro with M1 chip.

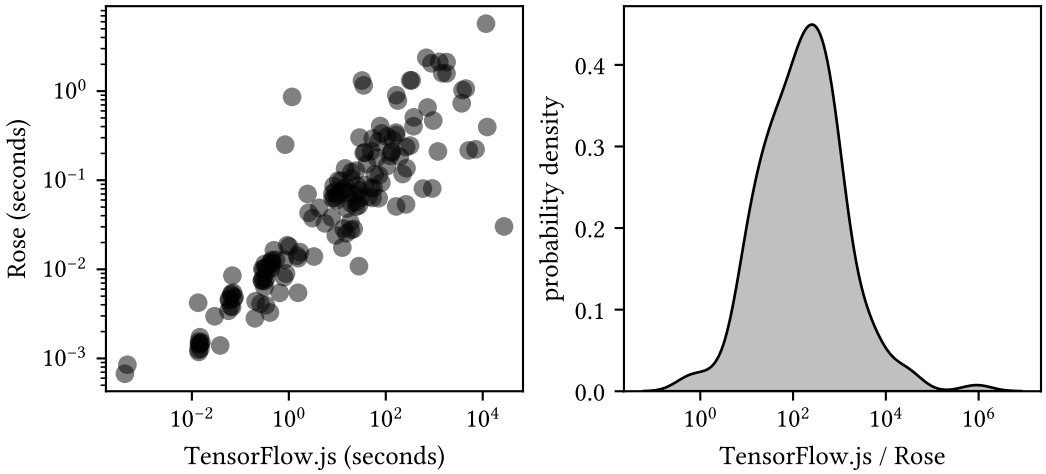

Figure 5 Left: Log-log scatterplot of Penrose diagram optimization time with TensorFlow.js versus Rose. **Right:** Log-scale kernel density estimate (KDE) plot of the optimization time of TensorFlow.js to Rose.

We omitted 10 of the 173 diagrams from our data analysis:

- **9 NaN failures:** Penrose aborts if it detects a "not-a-number" (NaN) value in the gradient as it is optimizing. This occurred in the TensorFlow.js version of Penrose for nine diagrams. *The Rose version of Penrose did not encounter NaNs for these programs.*
- **1 timeout:** For one diagram, we stopped the TensorFlow.js version of Penrose after it had run for over 24 hours. *The Rose version of Penrose took 42 milliseconds to compile and optimize this diagram.*

Tensorflow.js runs on both CPU and GPU. We used the "`cpu`" backend in our comparison because we found that, for scalar programs, it was faster than their GPU backend. To double-check this, we took the 88 diagrams (over half) that were quickest to run with TensorFlow.js, and also ran them with `@tensorflow/tfjs-node` and `@tensorflow/tfjs-node-gpu`, which they claim are faster than the "`cpu`" backend. We found that the Node backend is *79% slower* (median ratio) than the "`cpu`" backend, and the Node GPU backend is *75% slower* (median ratio) than the "`cpu`" backend. Also, those backends are unable to run in a browser, unlike the "`cpu`" backend, so they would be inappropriate for a direct comparison to Rose.

As we will discuss in Section 5.4.1, Rose's ability to define separate functions in a graph (rather than just a single graph of scalar or tensor values) is crucial to producing small enough WebAssembly binaries to feed to the browser. To investigate whether WebAssembly brought significant performance gains in the first place to be worth facing that challenge, we compared against a modified version of Rose which emits JavaScript code instead of WebAssembly. This experiment gave quartile slowdowns of 10%, 49%, and 100% for optimization of Penrose diagrams, showing that WebAssembly provides a significant advantage over JavaScript as a compilation target for Rose.

For the two Taichi applications (Figure 4, middle and right), we compare the running time of training/optimizing with the Python command-line counterparts. On average (of 10 runs), Rose is on par with the native performance of the Python versions: for the default initial condition in `billiards`, Rose completed the optimization in $22.7\text{s} \pm 0.2\text{s}$ while Taichi completed it in $20.6\text{s} \pm 0.8\text{s}$; for the default condition in `robot`, Rose finished 100 iterations of learning in $32.1\text{s} \pm 0.2\text{s}$ while Taichi took $31.3\text{s} \pm 1.1\text{s}$.

In the Penrose IDE (Figure 4, left), the main interaction that will trigger differentiation is compiling the DSL to diagrams. As reported in the previous section, Rose-based Penrose is significantly faster than the TensorFlow.js version, often leading to visible reduction in diagram layout optimization time in the user interface.

5.4 Qualitative observations

Our implementation effort to port both Penrose and Taichi applications to Rose spanned thousands of lines of code, including replacing the Penrose autodiff engine and function library and rewriting both `billiards` and `robot` for scratch. In this process, we have written a wide variety of differentiable programs using Rose, and had a chance to observe how Rose's main design choices (D1, D2, D3) impact the way programs are written. In this section, we report on our qualitative observations using Rose in these real-world settings, highlighting how these design choices interact with each other to form a coherent system.

5.4.1 Writing scalar programs as composable functions

The original versions of Penrose, `billiards`, and `robot` are naturally written as scalar programs. In Penrose, `bboxCircle` (line 10 of Listing 9) computes the bounding box by performing arithmetic on scalar values for the center and radius of a circle. In Taichi, both `billiards` and `robot` involve hand-crafted scalar programs for differentiable simulations. For instance, `apply_spring_force` (Figure 6) loops through individual springs in the robot, computing the force on the spring based on scalar-valued parameters, and scatter forces to end points of springs.

Because Rose is designed for writing scalar programs, translating both Penrose and Taichi source programs to Rose is straightforward and largely preserves the structures of the programs. For instance, when translating the Python programs from Taichi into TypeScript and Rose, as shown in Figure 6, Taichi kernels can be translated one-to-one to Rose functions.

Reflecting on the design choices, the combination of transformation to WebAssembly (D2) and the basic building block of composable functions (D1) gives the user both performance gains and an ergonomic programming interface. In the case of Taichi, the Rose abstraction of `fn` is not only useful for one-to-one translation from Taichi, but also necessary for running the simulator in browsers. Major WebAssembly engines have limits on WebAssembly binary size and on the number of local variables in each function. While it is possible to encapsulate much of the simulation code of `billiards` and `robot` in bigger JavaScript functions, the compiled size and local counts of these functions would quickly exceed these limits and would not run in the browser. Therefore, segmenting the source into functional units of `fn`s effectively reduces the size of emitted WebAssembly functions and modules, avoiding these errors and reducing compile times.

5.4.2 Metaprogramming and function dynamism

The Rose IR is designed to be performant and easy to compile to WebAssembly (D2) and therefore has limited expressiveness (Section 4). Metaprogramming using JavaScript enables the user to dynamically generate complex computation graphs that are impossible to specify with the Rose IR alone (D3). For instance, the `bboxGroup` function in Listing 9 computes the bounding box of a `Group` in Penrose, a recursive collection of shapes. For non-collection shape types such as `Circle`, we ported the TensorFlow.js implementation to Rose easily, e.g. `bboxCircle`. However, `bboxGroup` needs to recurse over the `Group` data structure to find out

```python
@ti.kernel
def apply_spring_force(t: ti.i32):
    for i in range(n_springs):
        a = spring_anchor_a[i]
        b = spring_anchor_b[i]
        pos_a = x[t, a]
        pos_b = x[t, b]
        dist = pos_a - pos_b
        length = dist.norm() + 1e-4
        target_length = spring_length[i] *
            (1.0 + spring_actuation[i] * act[t, i])
        impulse = dt * (length - target_length) *
            spring_stiffness[i] / length * dist

        ti.atomic_add(v_inc[t + 1, a], -impulse)
        ti.atomic_add(v_inc[t + 1, b], impulse)
```

```javascript
const apply_spring_force = fn(
  [Objects, Act], Objects, (x, act) => {
      const v_inc = [];
      for (let i = 0; i < n_objects; i++)
        v_inc.push([0, 0]);
      for (let i = 0; i < n_springs; i++) {
        const spring = robot.springs[i];
        const a = spring.object1;
        const b = spring.object2;
        const pos_a = x[a];
        const pos_b = x[b];
        const dist = vsub2(pos_a, pos_b);
        const length = add(norm(dist), 1e-4);
        const target_length = mul(spring.length,
          add(1, mul(act[i], spring.actuation))
        );
        const impulse =
          vmul(div(mul(dt * spring.stiffness,
                       sub(length, target_length)),
                   length),
               dist);
        v_inc[a] = vsub2(v_inc[a], impulse);
        v_inc[b] = vadd2(v_inc[b], impulse);
      }
      return v_inc;
});
```

Figure 6 A function that applies spring actuation on the mass-spring robot model in the `robot` example, written in Taichi (**Left**) and Rose (**Right**). The translation from Taichi to Rose is straightforward.

the bounding boxes of individual shapes before aggregating them into the final bounding box. This requires conditional dispatch of (1) Rose functions based on a discrete tag (`shape.kind`) and (2) recursive calls to `bboxGroup` to handle nested groups.

Figure 7 shows an example of calling `bboxGroup` on nested groups of shapes. The diagram in Figure 7 (left) has 1 group containing the whole diagram, and 3 subgroups of molecules that contain shapes such as `Text` and `Circle`. Figure 7 shows how `bboxGroup` uses JavaScript language features to compose Rose functions into a computation graph, denoting JavaScript constructs in gray and Rose functions in red. First, for each member shape, we `switch` on `shape.kind` to determine whether to (a) call individual Rose bounding box functions like `bboxCircle` or (b) recurse to call `bboxGroup`. Then, after all the child bounding boxes are computed, we use JavaScript `map` and `reduce` to aggregate via Rose `min` and `max` functions.

In the case of Penrose, metaprogramming actually helped us reduce the lines of code to refactor, because many plain JavaScript functions can stay the same and we only had to refactor functions that involve actual computation. We also speculate that by reducing the size of Rose-specific constructs, new users can learn a smaller API easier and experience a smoother learning curve.

6 Related work

Autodiff first started being seriously studied a few decades ago [48], with Griewank and Walther's book [14] consolidating research on the topic up until its publication. Some tools were developed, such as Tapenade [16] which operates over C and Fortran. The machine learning community developed an interest in autodiff over the past decade, resulting in popular tools including TensorFlow [2], PyTorch [32], and JAX [12] as mentioned in Section 1. JAX is particularly interesting because in a way it blends together tracing and transformation like we do here, but unlike Rose, JAX is not scalar-friendly and does not allow the programmer to explicitly define functions to serve as boundaries for tracing. Other autodiff systems include Zygote [21] for Julia, and Enzyme [29, 30, 31] for LLVM IR [26]. For graphics programming, Aδ [53] and Dr.Jit [22] can be used to differentiate shaders.

```
const bboxGroup = (shapes) => {
  const bboxes = shapes.map(bbox);
  const left = bboxes.map((b) => b.left).reduce(min);
  const right = bboxes.map((b) => b.right).reduce(max);
  const bottom = bboxes.map((b) => b.bottom).reduce(min);
  const top = bboxes.map((b) => b.top).reduce(max);
  return { left, right, bottom, top };
};

const bboxCircle = fn([Circle], Rectangle,
  ({ center: [x, y], radius: r }) => {
    const left = sub(x, r);
    const right = add(x, r);
    const bottom = sub(y, r);
    const top = add(y, r);
    return { left, right, bottom, top };
  },
);

const bbox = (shape) => {
  switch (shape.kind) {
    case "Rectangle": return shape.value;
    case "Circle": return bboxCircle(shape.value);
    case "Group": return bboxGroup(shape.value);
  }
};
```

Listing 9 Examples of JavaScript metaprogramming to construct Rose functions for recursive data structures.

The programming languages community has also taken an interest in autodiff [36], producing proofs of correctness [1], program transformations for SSA [20] and CPS [51], and more recently, reformulations of reverse-mode autodiff in terms of dual numbers [44], as well as a new autodiff formulation called CHAD [49, 45]. Some work also attempts to bridge the gap between programming language theory and the machine learning domain by facilitating automatic parallelization [6, 33]. The latter work also resulted in a formalization of function transposition [39] which directly inspired the low-level design of autodiff in Rose.

Rose supports higher-order derivatives because its core IR is closed under differentiation and transposition. A more sophisticated approach we don't explore here would be derivative towers [23, 35], sometimes called "Taylor towers" because they use Taylor expansions instead of the chain rule. We would be interested to see how derivative towers can be combined with our approach in future work, while avoiding the pitfalls of perturbation confusion [42, 28]. Another optimization that becomes crucial when scaling up autodiff is checkpointing, which cuts down drastically on memory requirements; for instance, while the semantics of Rose can in general result in keeping around arbitrarily many intermediate results, a recursive "divide-and-conquer" checkpointing scheme [43] reduces the memory impact of reverse-mode autodiff to a logarithmic factor; the cost, though, is that the asymptotic running time would also suffer a logarithmic factor.

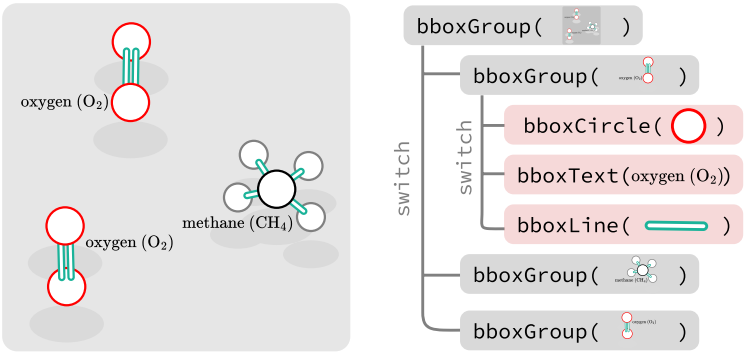

Figure 7 In Penrose, we used JavaScript to programmatically generate Rose functions. **Left:** a figure comprised of a top-level group containing all molecules and sub-groups for each molecule. **Right:** the `bboxGroup` function conditionally generates Rose functions or recursively calls itself based on the shape type.

7 Conclusion and future work

This paper introduced Rose, an embedded domain-specific language for automatic differentiation of interactive programs on the web, which blends together the two primary autodiff techniques of tracing and transformation. Currently Rose targets WebAssembly, which runs on the CPU; as we showed in Section 5.3, this already provides an enormous performance advantage for scalar programs when compared to the state-of-the-art for autodiff on the web. In future work, we would also like to pursue further performance gains by implementing a backend that targets WebGPU [24]. We have already laid the groundwork for this by drawing inspiration for the Rose IR from Dex [33] to be friendly to automatic parallelization, such as the `for` construct and accumulate-only reference types. In general, we plan to continue this line of work to open up new modes of differentiable interactivity.

References

1 Martín Abadi and Gordon D. Plotkin. A simple differentiable programming language. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. `doi:10.1145/3371106`.

2 Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems, 2016. `arXiv:1603.04467`.

3 Anthony Anjorin, Li-yao Xia, and Vadim Zaytsev. Bidirectional transformations wiki, 2011. URL: `http://bx-community.wikidot.com/`.

4 Francis J. Anscombe. Graphs in statistical analysis. *The American Statistician*, 27(1):17–21, 1973.

5 Hudson Ayers, Evan Laufer, Paul Mure, Jaehyeon Park, Eduardo Rodelo, Thea Rossman, Andrey Pronin, Philip Levis, and Johnathan Van Why. Tighten Rust's belt: Shrinking embedded Rust binaries. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2022, pages 121–132, New York, NY, USA, 2022. Association for Computing Machinery. `doi:10.1145/3519941.3535075`.

6 Gilbert Bernstein, Michael Mara, Tzu-Mao Li, Dougal Maclaurin, and Jonathan Ragan-Kelley. Differentiating a tensor language, 2020. arXiv:2008.11256.

7 Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing, 2012. arXiv:1209.5145.

8 Dan Cascaval, Mira Shalah, Phillip Quinn, Rastislav Bodik, Maneesh Agrawala, and Adriana Schulz. Differentiable 3D CAD programs for bidirectional editing. *Computer Graphics Forum*, 41(2):309–323, 2022. doi:10.1111/cgf.14476.

9 Bartosz Ciechanowski, 2014. URL: https://ciechanow.ski/.

10 Sam Estep, Wode Ni, Raven Rothkopf, and Joshua Sunshine. Rose: Composable autodiff for the interactive web, 2024. arXiv:2402.17743.

11 Figma, Inc. Figma, 2016. URL: https://figma.com/.

12 Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018.

13 Google LLC. Google Slides, 2006. URL: https://google.com/slides.

14 Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.

15 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3062341.3062363.

16 Laurent Hascoet and Valérie Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39(3), May 2013. doi:10.1145/2450153.2450158.

17 Brian Hempel, Justin Lubin, and Ravi Chugh. Sketch-n-Sketch: Output-directed programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, pages 281–292, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3332165.3347925.

18 Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. DiffTaichi: Differentiable programming for physical simulation, 2020. arXiv:1910.00935.

19 Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph.*, 38(6), November 2019. doi:10.1145/3355089.3356506.

20 Michael Innes. Don't unroll adjoint: Differentiating SSA-form programs, 2019. arXiv:1810.07951.

21 Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckas, Elliot Saba, Viral B. Shah, and Will Tebbutt. ∂P: A differentiable programming system to bridge machine learning and scientific computing, 2019. arXiv:1907.07587.

22 Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. Dr.Jit: A just-in-time compiler for differentiable rendering. *ACM Trans. Graph.*, 41(4), July 2022. doi:10.1145/3528223.3530099.

23 Jerzy Karczmarczuk. Functional differentiation of computer programs. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 195–203, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/289423.289442.

24 Benjamin Kenwright. Introduction to the WebGPU API. In *ACM SIGGRAPH 2022 Courses*, SIGGRAPH '22, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3532720.3535625.

25 Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. arXiv:1412.6980.

26 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi:10.1109/CGO.2004.1281665.

27 Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

28 Oleksandr Manzyuk, Barak A. Pearlmutter, Alexey Andreyevich Radul, David R. Rush, and Jeffrey Mark Siskind. Perturbation confusion in forward automatic differentiation of higher-order functions. *Journal of Functional Programming*, 29:e12, 2019. doi:10.1017/S095679681900008X.

29 William Moses and Valentin Churavy. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 12472–12485. Curran Associates, Inc., 2020. URL: https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf.

30 William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. Reverse-mode automatic differentiation and optimization of GPU kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3458817.3476165.

31 William S. Moses, Sri Hari Krishna Narayanan, Ludger Paehler, Valentin Churavy, Michel Schanen, Jan Hückelheim, Johannes Doerfert, and Paul Hovland. Scalable automatic differentiation of multiple parallel paradigms through compiler augmentation. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

32 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

33 Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. Getting to the point: Index sets and parallelism-preserving autodiff for pointful array programming. *Proc. ACM Program. Lang.*, 5(ICFP), August 2021. doi:10.1145/3473593.

34 Amit Patel. Red Blob Games, 2013. URL: https://www.redblobgames.com/.

35 Barak A. Pearlmutter and Jeffrey Mark Siskind. Lazy multivariate higher-order forward-mode AD. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 155–160, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1190216.1190242.

36 Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.*, 30(2), March 2008. doi:10.1145/1330017.1330018.

37 Pyodide contributors and Mozilla. Pyodide, 2019. URL: https://pyodide.org/.

38 Dan Quinlan and Chunhua Liao. The ROSE source-to-source compiler infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT*, volume 2011, page 1. Citeseer, 2011.

39 Alexey Radul, Adam Paszke, Roy Frostig, Matthew J. Johnson, and Dougal Maclaurin. You only linearize once: Tangents transpose to gradients. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571236.

40 Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, pages 12–23, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3122948.3122949.

41 Xipeng Shen, Guoqiang Zhang, Irene Dea, Samantha Andow, Emilio Arroyo-Fang, Neal Gafter, Johann George, Melissa Grueter, Erik Meijer, Olin Grigsby Shivers, Steffi Stumpos, Alanna Tempest, Christy Warden, and Shannon Yang. Coarsening optimization for differentiable programming. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485507.

42 Jeffrey Mark Siskind and Barak A Pearlmutter. Nesting forward-mode AD in a functional framework. *Higher-Order and Symbolic Computation*, 21(4):361–376, 2008.

43 Jeffrey Mark Siskind and Barak A. Pearlmutter. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software*, 33(4-6):1288–1330, 2018. doi:10.1080/10556788.2018.1459621.

44 Tom J. Smeding and Matthijs I. L. Vákár. Efficient dual-numbers reverse AD via well-known program transformations. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571247.

45 Tom J. Smeding and Matthijs I. L. Vákár. Efficient CHAD. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi:10.1145/3632878.

46 Daniel Smilkov, Nikhil Thorat, Yannick Assogba, Charles Nicholson, Nick Kreeger, Ping Yu, Shanqing Cai, Eric Nielsen, David Soegel, Stan Bileschi, Michael Terry, Ann Yuan, Kangyi Zhang, Sandeep Gupta, Sarah Sirajuddin, D Sculley, Rajat Monga, Greg Corrado, Fernanda Viegas, and Martin M Wattenberg. TensorFlow.js: Machine learning for the web and beyond. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 309–321, 2019. URL: https://proceedings.mlsys.org/paper_files/paper/2019/file/acd593d2db87a799a8d3da5a860c028e-Paper.pdf.

47 Bobbie Soedirgo. Compile and run LLVM IR in the browser, October 2023. original-date: 2021-02-24T14:29:16Z. URL: https://github.com/soedirgo/llvm-wasm.

48 Bert Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, 1980. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-02-19. URL: https://www.proquest.com/dissertations-theses/compiling-fast-partial-derivatives-functions/docview/302969224/se-2.

49 Matthijs Vákár and Tom Smeding. CHAD: Combinatory homomorphic automatic differentiation. *ACM Trans. Program. Lang. Syst.*, 44(3), August 2022. doi:10.1145/3527634.

50 Bret Victor. Explorable explanations, 2011. URL: https://worrydream.com/ExplorableExplanations/.

51 Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi:10.1145/3341700.

52 Guillermo Webster. g9: Automatically interactive graphics, 2016. URL: https://omrelli.ug/g9/.

53 Yuting Yang, Connelly Barnes, Andrew Adams, and Adam Finkelstein. Aδ: Autodiff for discontinuous programs – Applied to shaders. *ACM Trans. Graph.*, 41(4), July 2022. doi:10.1145/3528223.3530125.

54 Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. Penrose: From mathematical notation to beautiful diagrams. *ACM Trans. Graph.*, 39(4), August 2020. doi:10.1145/3386569.3392375.